UNIVERSITAT AUTÒNOMA

DE

BARCELONA

FACULTAT DE CIÈNCIES

Departament d'Informàtica

Estrategias de asignación de programas en computadores paralelos

Memoria presentada por Miquel Angel Senar Rosell para optar al grado de Doctor en Informàtica

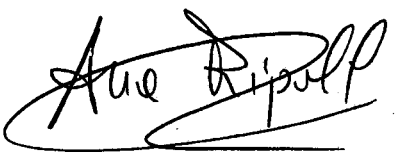
Barcelona, Octubre 1996

Estrategias de asignación de programas en computadores paralelos

Memoria presentada por Miquel Angel Senar Rosell para optar al grado de Doctor en Informática por la Universitat Autònoma de Barcelona. Trabajo realizado en el Departamento de Informàtica de la Facultad de Ciencias de la Universitat Autònoma de Barcelona, bajo la dirección de la Dra. Ana Ripoll Aracil,

Bellaterra, Octubre de 1996

Vo. Bo. Directora Tesis



Fdo. Ana Ripoll Aracil

A mi familia

Me quedo pasmado cuando termino algo. Me quedo pasmado y desolado. Mi instinto de perfección debería inhibirme de acabar; debería inhibirme de dar comienzo. Pero me distraigo y hago. Lo que consigo es un producto, en mí, no de la aplicación de la voluntad, sino de una cesión suya. Comienzo porque no tengo fuerza para pensar; acabo porque no tengo alma para suspender. Este libro es mi cobardía.

Fernando Pessoa

Agradecimientos

Esta tesis, fruto del trabajo personal durante varios años, no hubiese sido lo que ahora es sin la inestimable ayuda de un grupo de personas que han estado a mi lado durante todo este tiempo y para las que estas líneas pretenden ser un reflejo de mi más sincera gratitud hacia ellas.

En primer lugar, quiero expresar mi agradecimiento a la Dra. Ana Ripoll por su magistral dirección, sus constantes consejos, sus acertadas aportaciones y su actitud incansable al desaliento.

Al Dr. Emilio Luque por sus inestimables sugerencias y todas las enseñanzas que me ha brindado durante estos años y que espero haber sabido recoger y plasmar en este trabajo.

A los Drs. Joan Sorribes, Porfidio Hernández y Tomás Díez porque han sido desde mi entrada en el mundo universitario una fuente de experiencia y de constante estímulo.

A Tomás Margalef, Ana Cortés y Remo Suppi porque me han brindado su ayuda, apoyo técnico y su amistad en todo momento.

A la Dra. Dolores Isabel Rexachs y el resto de los miembros de dirección de la Escuela Universitaria de Sabadell, que me han apoyado incondicionalmente y perdonado mis ausencias durante los momentos álgidos de trabajo investigador.

También quiero agradecer a todos los "jóvenes" del equipo (Dani, Juan Carlos, Eduardo, Elisa, María, Josep Lluis, Carlos, Toni, Josep y Fina) su constante buen humor, su aliento y su infinita paciencia ante mis constantes preguntas sobre todos los paquetes ofimáticos que he utilizado.

Al grupo de Lérida formado por Concepció y Siscu, que han colaborado en la implementación de algunos programas usados en este trabajo.

A los miembros del instituto IMAG de Grenoble y, en especial, a Jacques, Pascal, Christophe y Marta, por todo el soporte técnico en el uso de sus herramientas y por su cordialidad y amabilidad durante mis estancias en Grenoble.

A los demás miembros de la Unidad de Arquitectura de Ordenadores y Sistemas Operativos y del Departamento de Infomática, por su constante apoyo.

Gracias también a mis padres, al resto de mi familia y a mis amigos ajenos a la Universidad por su comprensión, cariño y amistad, aunque sigan mostrándose sorprendidos porque alguien pueda hacer una tesis a base de "bolitas" y "palitos".

Espero que este trabajo sirva como una pequeña y sincera muestra de mi gratitud a todos ellos, y que sea reflejo de lo afortunado que soy por tenerlos junto a mí.

Índice

Prólogo

El paralelismo ha sido presentado tradicionalmente como una solución al problema de conseguir computadores que sean cada vez más rápidos y den respuesta de esa forma a las necesidades de cómputo que requieren muchas de las aplicaciones científicas de nuestros días. Ha sido, además, una solución que se ha ido extendiendo paulatinamente desde mediados de los años 80 como una alternativa atractiva, desde el punto de vista económico, al uso de sistemas monoprocesador de alta velocidad. La fabricación de componentes integrados cada vez más rápidos no se ha detenido en ningún momento y no parece que se haya alcanzado el límite de velocidad de los mismos. Sin embargo, esos límites físicos serán alcanzados tarde o temprano y entonces el paralelismo continuará siendo una de las alternativas más viables para el aumento de prestaciones en los computadores venideros.

A pesar de esas perspectivas prometedoras para el paralelismo, nos encontramos hoy en día con una situación que se caracteriza por un uso restringido del mismo. A esta situación ha contribuido, lógicamente, la evolución de los tradicionales sistemas monoprocesador y de sus entornos de programación. Por otra parte, el propio mundo del paralelismo se ha enfrentado con enemigos internos que han debilitado su expansión y generalización. La falta de estándares, la dificultad intrínseca en la resolución de un problema mediante un algoritmo paralelo o la aparición de problemas específicos al cómputo paralelo que limitan el rendimiento que pueden alcanzar las aplicaciones han sido algunos de esos factores que han limitado su crecimiento y difusión.

El siguiente trabajo se centra en uno de los problemas específicos del cómputo paralelo: la asignación de las tareas que componen un programa paralelo a los procesadores que forman parte del computador paralelo. Este problema ha recibido un tratamiento extenso en la literatura aunque las premisas de partida en su resolución, así como la efectividad y aplicabilidad de las soluciones propuestas no siempre han sido concordantes. Bajo denominaciones genéricas como "mapping", "allocation", "scheduling", "load balancing", "clustering", "partitioning" o "assignment", que no siempre se han usado con una significación coherente, nos encontramos toda una serie de propuestas relacionadas con el problema mencionado y que forman la base del presente trabajo.

Partiendo de un modelo de programa paralelo compuesto por un conjunto estático y persistente de procesos, de los que se conocen los valores relativos a su volumen de cómputo y de comunicación, en este trabajo se presentará una estrategia de asignación que es a la vez eficiente en la solución hallada y en el tiempo necesario para obtenerla. La estrategia mencionada ha sido contrastada con otras alternativas presentes en la literatura, comparándose con ellas en términos de eficiencia en la búsqueda de soluciones y en términos de velocidad computacional.

Para la realización de la fase experimental se ha utilizado un conjunto de grafos que ha cubierto un amplio rango de características relacionadas tanto a su estructura como a sus valores. Muchos de los grafos utilizados además correspondían a ejemplos derivados de aplicaciones reales.

La bondad de la estrategia se ha demostrado, en primer lugar, para aquellas arquitecturas paralelas con una topología totalmente interconectada. Partiendo de estos resultados se han introducido un conjunto de extensiones con objeto de aplicar la estrategia en sistemas con arquitecturas de tipo malla, hipercubo y anillo. En estos casos se ha desarrollado un modelo teórico capaz de evaluar los costes adicionales en que incurren los métodos propuestos cuando se considera la asignación sobre una arquitectura concreta.

La eficiencia de las distintas estrategias de asignación se mide tradicionalmente según el valor que adquiere una cierta función de coste. La suposición básica es que a menor valor de la función de coste, menor tiempo de ejecución presentará la aplicación. La importancia de la función de coste así como la influencia de la misma en el diseño de una cierta estrategia han sido analizadas también, intentando con ello establecer cuál es la correlación que podemos esperar entre los valores obtenidos por la función de coste y los tiempos de ejecución del programa.

La presente memoria está organizada del siguiente modo:

- En el primer capítulo, se enmarcará el problema de la asignación dentro del cómputo paralelo y se presentarán los diferentes enfoques aparecidos en la literatura bajo los que se ha abordado su resolución. Se analizarán, de forma general, tanto las distintas soluciones como los supuestos de los que parten.

- En el segundo capítulo, se concentrará el estudio en un de los modelo clásicos adoptados para la resolución del problema de la asignación, que fue introducido en el capítulo anterior y que constituye el modelo en el que se basa el presente trabajo. Se estudiarán las implicaciones que conlleva la adopción de dicho modelo, cuáles han sido los trabajos más relevantes que han aparecido en la literatura basándose en él, y cuáles son las principales funciones de coste utilizadas para evaluar la bondad de las asignaciones generadas por cualquier estrategia.

- En el tercer capítulo, se presentará la estrategia propuesta en este trabajo, justificando su diseño a partir de las características que presentan otras estrategias de la literatura. Se realizará también una primera evaluación de la bondad de la estrategia comparando sus resultados con los obtenidos mediante un método óptimo.

- En el cuarto capítulo, se extenderá la fase experimental utilizando ejemplos de mayor tamaño para comparar la estrategia propuesta con un conjunto representativo de otros métodos aparecidos en la literatura. Dicha comparación se centrará en el estudio de la calidad de las soluciones obtenidas y en su complejidad temporal.

- En el quinto capítulo, se extenderá la estrategia básica de forma que pueda ser aplicable en sistemas que no presentan una topología totalmente interconectada. Se hará un estudio comparativo de las posibles extensiones y se presentará un modelo teórico capaz de predecir el coste adicional debido a tales extensiones. Por

último, se realizará un estudio encaminado a determinar la correlación existente entre tiempo de ejecución de un programa paralelo asignado mediante una cierta estrategia y el correspondiente valor proporcionado por la función de coste.

- A continuación, se presentarán las principales conclusiones que se derivan del presente trabajo y las líneas de trabajo más importantes que quedan abiertas a partir de los resultados obtenidos.

- En el apartado de apéndices se han incluido todos los resultados obtenidos en la realización de los diferentes fases experimentales.

Por último, mencionemos que en la presente memoria se ha pretendido traducir al máximo los términos en inglés. No obstante, aparecen algunos no aceptados aún en castellano o cuya traducción resulta poco familiar y que hemos creído conveniente utilizarlos en su forma original.

CAPÍTULO 1.

El problema de la asignación en el cómputo paralelo

Desde la aparición en los años cuarenta y cincuenta de los primeros computadores siguiendo los postulados de J. von Neumann, J. Presper Eckert, J. Maulchly, A.W.Burks, H. H. Goldstein y M. V. Wilkes, la evolución que han experimentado los mismos hasta nuestros días se ha basado, en buena medida, en su aumento de prestaciones. Tal evolución ha intentado aumentar la velocidad de los computadores de forma que se traduzca, desde el punto de vista del usuario, en una reducción del tiempo empleado en solucionar aquellos problemas resueltos mediante el computador.

Los computadores paralelos constituyen hoy en día una de las propuestas más prometedora desde el punto de vista arquitectónico para satisfacer la creciente demanda de velocidad de cómputo. La gran cantidad de factores que intervienen en el desarrollo de aplicaciones paralela, sus complejas interrelaciones y su mayor o menor influencia sobre el rendimiento final que pueden conseguir las aplicaciones son las razones por las que existen múltiples aspectos del procesamiento paralelo que actualmente son objeto de investigación.

El presente trabajo se centra en el estudio y resolución de uno de los problema involucrados en el uso de computadores paralelos: la asignación de tareas a procesadores.

En este capítulo, se hará un estudio general de los distintos factores que condicionan la resolución del mencionado problema y se revisarán las principales aportaciones que existen en la literatura para su resolución.

1.1 Conceptos generales sobre la programación paralela

La construcción de computadores más veloces, y por ello, más potentes, ha venido acompañada tradicionalmente de avances tecnológicos significativos que van desde el uso de válvulas de vacío hasta los circuitos integrados de alta densidad de integración (VLSI). Tal ha sido la importancia de dichos avances tecnológicos que se suele clasificar a los computadores en cuatro generaciones, asociadas cada una de ellas a un avance tecnológico. Sin embargo, es fácilmente constatable que la potencia de los computadores actuales no sólo es debida a una simple mejora tecnológica que ha permitido aumentar la velocidad de sus componentes. La disciplina que hoy se conoce como Arquitectura de Computadores ha sido responsable de la introducción de una serie de aportaciones en el diseño de los computadores tradicionales que han permitido aumentar sus prestaciones en una medida equiparable a la conseguida por las mejoras tecnológicas. Entre las aportaciones más destacables cabe mencionar el uso de la memoria cache, el procesamiento segmentado (pipeline) y el uso de unidades funcionales múltiples. Mediante soluciones como las mencionadas se ha conseguido reducir el tiempo de ejecución de los programas bien porque se reducía el tiempo empleado en la búsqueda de las instrucciones y datos de un programa, o porque se reducía el tiempo de ejecución de las instrucciones.

Los computadores paralelos son una de las aportaciones arquitectónicas que desde inicios de los años 80 han ido concentrando paulatinamente la atención de los diseñadores de computadores de altas prestaciones y se presentan como una de las alternativas más viables para conseguir elevar su potencia de cómputo más allá de las limitaciones físicas que puede sufrir la tecnología; ofreciendo, además, una relación coste/prestaciones especialmente favorable pues resulta más económico conseguir potencial bruto de cómputo mediante la ejecución concurrente sobre procesadores de prestaciones medias-altas, que obtenerlo mediante el procesamiento secuencial sobre un procesador de altísimas prestaciones.

El uso de varias unidades de procesamiento puede hacerse bajo dos perspectivas diferenciadas:

- Los procesadores que forman parte del sistema son utilizados por varios programas de usuario simultáneamente sin que exista ningún tipo de colaboración entre los mismos. Este sistema sería equivalente a disponer de computadores separados para cada usuario. Se aumenta la productividad global del sistema pero un usuario individual no va a ver mejorado el tiempo de ejecución de su aplicación por el hecho de que existan procesadores desocupados en el sistema.

- Los procesadores que forman el sistema están dedicados a la ejecución de una única aplicación y, por lo tanto, existe una cooperación entre ellos y entre los procesos que ejecuta cada uno de ellos. En este sistema se pretende mejorar el tiempo de ejecución de un programa concreto acercándolo al límite teórico que supondría obtener un tiempo igual al tiempo empleado en la ejecución secuencial dividido por el número de procesadores. En la práctica, esta

ganancia nunca se alcanza debido a diversos factores: la aplicación no tiene porqué ser perfectamente paralelizable y puede contener partes secuenciales, y, en general, existirán relaciones de dependencia entre los componentes del programa paralelo que obligará a que los mismos se sincronicen y se comuniquen, introduciendo con ello un "overhead" que no existe en las aplicaciones secuenciales.

Nosotros nos centraremos en el segundo tipo de sistemas donde el objetivo del paralelismo se centra en minimizar el tiempo de ejecución de una determinada tarea. Consideraremos sistemas de propósito general dedicados a cualquier tipo de aplicación y donde no van a existir aplicaciones que requieran el cumplimiento de restricciones temporales en su finalización (*deadlines*). Esta consideración no excluye, sin embargo, la aplicación de nuestras propuestas en sistemas empotrados (*embedded*) o dedicados.

Nuestro trabajo se concentrará en el estudio del problema de la asignación que surge en las aplicaciones propias de sistemas que suelen incluirse bajo la categoría de MIMD (*Multiple Instruction stream Multiple Data stream*). Son sistemas formados por un conjunto de procesadores con su memoria local. No existe memoria global y los procesadores se comunican por paso de mensajes. Los procesadores pueden ejecutar los programas de forma asíncrona. La latencia de comunicación entre procesadores es relativamente pequeña y el ancho de banda es relativamente alto. En este sentido, distinguiremos nuestros sistemas paralelos de los sistemas conocidos tradicionalmente como *sistemas distribuidos*.

1.1.1 Una aproximación metodológica al desarrollo de aplicaciones paralelas

Partiendo de las premisas enunciadas en el punto anterior, el desarrollo de una aplicación paralela va a seguir una serie de pasos que podríamos considerar habituales y que no difieren substancialmente de los seguidos en el desarrollo de aplicaciones secuenciales [Pan90]:

- esbozo del dominio del problema y selección de una estrategia de resolución para el mismo.
- formulación de una solución algorítmica.
- implementación usando un lenguaje de programación
- traducción del programa a un formato ejecutable
- ejecución del programa

Este conjunto de pasos puede caracterizarse como cuatro subsistemas que operan a diferentes niveles de abstracción (figura 1.1) Cada uno de esos subsistemas define su propia colección de objetos, un conjunto de operaciones o manipulaciones aplicables a esos objetos y dominios que representan los valores que cada objeto puede tomar.

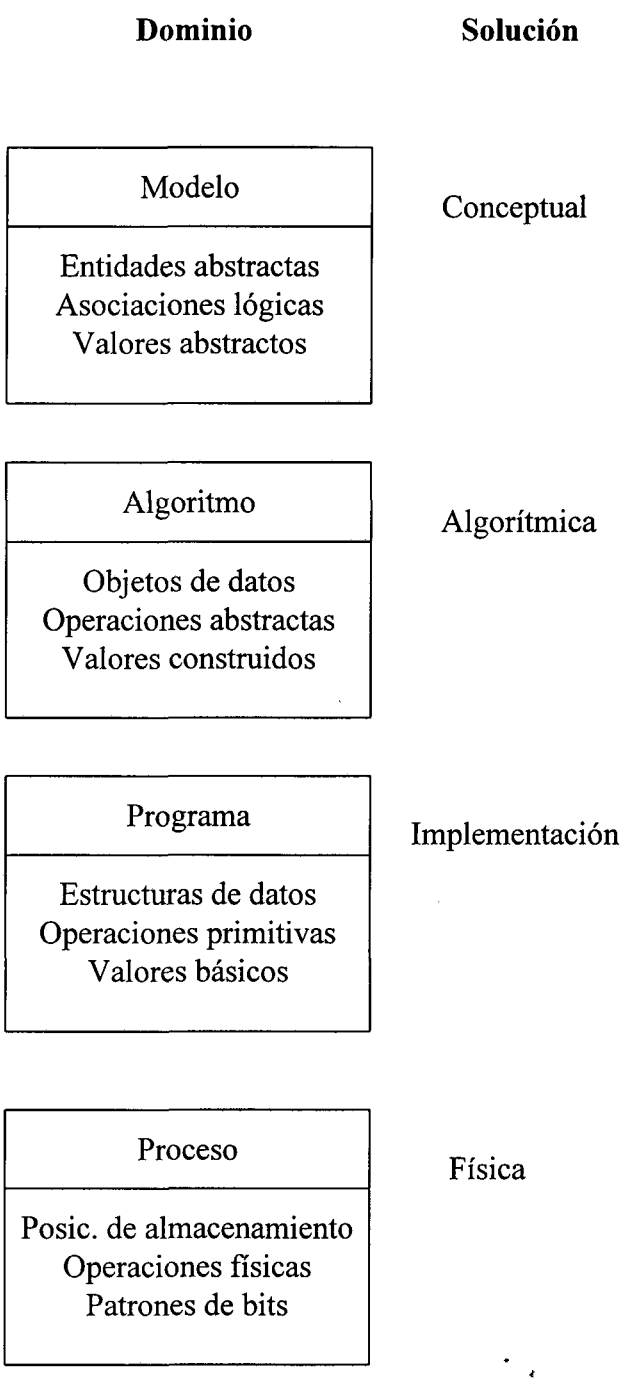


Figura 1.1 Metodología de desarrollo de aplicaciones paralelas

El nivel *conceptual* ocupa el nivel más alto. Aquí, el problema se expresa en los términos abstractos del razonamiento humano y de nuestra percepción de las leyes que gobiernan la naturaleza. La solución puede esbozarse en términos de entidades abstractas, asociaciones lógicas entre dichas entidades y la atribución de valores abstractos a las mismas. La estrategia de solución se diseña en términos muy generales sin considerar las capacidades del sistema en el cual va a ser implementada. Y su descripción suele hacerse en el propio lenguaje natural o diagramas. En este nivel es donde el usuario puede determinar qué porciones de la solución son candidatas para su paralelización. Aunque el grado de paralelismo estará limitado al número de procesadores disponible en el sistema, esta restricción no tiene porqué reflejarse en la

mayoría de soluciones conceptuales. Por el contrario, el programador identifica aquí el máximo grado de paralelismo que parece "natural".

Bajo el nivel conceptual se encuentra el nivel *algorítmico* que ya define los pasos específicos que se requieren para resolver el problema. Aunque las operaciones pueden tener todavía un cierto nivel de abstracción, se aplican sobre objetos y datos con rangos específicos de valores. La naturaleza de la especificación algorítmica refleja el hecho de que la solución del problema va a llevarse a cabo mediante un computador. La elección de subalgoritmos puede estar influenciada por consideraciones como el número de procesadores físicos o si la memoria va a ser común o distribuida, aunque, en general, la descripción suele ser independiente de la máquina, o sea, no va a estar ligada a ninguna arquitectura particular o sistema operativo. Cada uno de los subalgoritmos estará descrito como una única secuencia de pasos. Si el paralelismo aparece a este nivel se limitará a la noción de que dos subalgoritmos pueden estar ejecutándose concurrentemente.

Entre el nivel *algorítmico* y el *físico* nos encontramos con el nivel de *implementación* que va a cubrir el *gap* entre la representación del problema como un conjunto de manipulaciones abstractas y las operaciones físicas que realizan esas manipulaciones. En el nivel que nos ocupa, el problema se expresa en términos de un determinado lenguaje de programación con sus estructuras de datos concretas, sus operaciones primitivas y sus valores básicos. El propósito de esta descripción es permitir una traducción automática a código máquina y, por ello, el lenguaje de programación impone un formalismo riguroso. Al mismo tiempo, el lenguaje puede estar alejado de las capacidades funcionales del sistema físico con objeto de proporcionar mayor poder de expresividad, generalidad y portabilidad. Esta es una fase donde el programador va a verse obligado a expandir muchos de los pasos descritos en los subalgoritmos; donde habían ecuaciones u operaciones de alto nivel, ahora será necesario especificarlas en detalle mediante sentencias y estructuras de control. En este nivel la descripción puede ser dependiente del sistema dado que muchos compiladores suelen extender los lenguajes de programación para aprovechar características de la arquitectura física.

Por último, nos encontraríamos con el nivel *físico* donde ya se ejecutaría el programa una vez traducido. Este es, de hecho, un nivel en el que no existe control directo del programador.

Los pasos enunciados anteriormente obedecen a una división teórica ideal que, en la práctica, suelen verse entremezclados y mútuamente influidos y limitados por una serie de factores. Básicamente, y actuando en un sentido opuesto al que hemos enunciado los diferentes niveles, la arquitectura del sistema y los lenguajes de programación disponibles van a ser los factores que en la práctica guíen e impongan restricciones a las dos primeras fases donde el nivel de abstracción es mayor. Así, por ejemplo, el uso de un determinado lenguaje de programación puede obligar al programador a expresar y controlar el paralelismo de forma explícita o puede permitir al programador que use algoritmos secuenciales a partir de los cuales el compilador hará una extracción del paralelismo implícito. La elección además va a verse influenciada también por aspectos adicionales como la efectividad del proceso de traducción del

nivel de implementación al nivel físico y cómo esa transformación limita el rendimiento que puede conseguirse del programa final [Pan90].

1.1.2 Factores que limitan el crecimiento del paralelismo

Si en el inicio de este trabajo habíamos presentado el paralelismo como una alternativa económicamente muy efectiva para la obtención de mayor rendimiento frente a los computadores secuenciales, el apartado anterior nos ha esbozado algunos factores que han limitado de algún modo la difusión y expansión del paralelismo. A continuación, veremos con mayor extensión cuáles han sido los factores fundamentales que han sido esgrimidos a la hora de justificar la lenta difusión del paralelismo [Pan91][Cha91][Lew92].

Uno de los factores es la considerable inversión de dinero que se ha hecho y se sigue haciendo en programación secuencial. Puede costar muchísimo convertir los programas desarrollados para computadores secuenciales para que se ejecuten eficientemente en máquinas paralelas. Los compiladores que reestructuran automáticamente los programas secuenciales para reducir, o eliminar, el coste de la transformación de secuencial a paralelo no han tenido todavía un éxito aceptable universalmente.

Por otra parte, el manejo del paralelismo directamente puede ser difícil. Desarrollar programas eficientes para computadores paralelos es más duro que para computadores secuenciales. Explotar los computadores paralelos puede significar el desarrollo de algoritmos completamente nuevos y la mayoría de programadores han sido educados para el desarrollo de algoritmos secuenciales.

Existe también una inversión considerable en herramientas de programación secuencial para ayudar a la verificación de los programas, al análisis (*profiling*) de la ejecución y a la depuración interactiva. Dado que existen muchas menos herramientas de soporte para la programación paralela, el coste de desarrollo de estos programas es más elevado que el equivalente en programas secuenciales.

Otro problema es la falta de una sola y predominante arquitectura paralela. Muchos tipos distintos de computadores paralelos están disponibles hoy en día y puede que todavía puedan aparecer nuevas arquitecturas. Teniendo en cuenta que la vida de un programa puede alargarse durante muchos años, una gran inversión en el desarrollo de los mismos que sólo se ejecute bien en un tipo de arquitectura puede ser arriesgada. Por contra, la arquitectura secuencial basada en la máquina de von Neumann se ha mantenido inalterable durante décadas. Los programadores están aislados de las variaciones introducidas en los computadores secuenciales por compiladores de alto nivel y sistemas operativos estandarizados. Esta uniformidad todavía no existe para los computadores paralelos, lo que los hace menos atractivos como plataformas para el desarrollo de aplicaciones.

Por último, existen todavía determinados aspectos que son propios del mundo de la programación paralela y que, habitualmente, tienen un impacto importante en el

rendimiento de las aplicaciones. Podemos clasificarlos en dos grandes grupos: los dependientes del problema (o de la aplicación) y los dependientes del sistema.

1. ***Factores dependientes del problema***. En este apartado figuran aquellos factores que son propios del problema que se pretende resolver y del algoritmo utilizado en su resolución. Los más importantes son:

   - las dependencias entre los distintos cómputos (procesos), que pueden provocar que la ejecución de un determinado grupo de instrucciones deba esperar la finalización de otro.
   - la proporción existente entre las partes paralelas y las partes secuenciales. En general, no todos los problemas van a admitir una paralelización completa en los algoritmos que lo resuelven y en muchos casos suelen coexistir partes de cómputo paralelo y partes de cómputo secuencial. El rendimiento que se podrá conseguir estará, por lo tanto, limitado al tamaño de las porciones paralelas de que conste el programa.
   - las necesidades de comunicación y sincronización entre procesos. A lo largo de la ejecución de un programa paralelo, los distintos procesos que lo componen necesitarán habitualmente intercambiar información y/o coordinarse (sincronizarse) al llegar a determinados puntos del cómputo. Estas necesidades repercutirán en la aparición de una serie de retrasos inexistentes en las aplicaciones secuenciales.
   - la proporción existente entre la cantidad de cómputo que ejecuta un proceso y la cantidad de información que debe comunicar ("granularidad" del proceso). Esta proporción está ligada con la eficiencia con la que se ejecuta el cómputo y se transmite la información en un determinado sistema. Si, por ejemplo, los mecanismos de comunicación no son muy eficientes y una aplicación presenta un elevado número de comunicaciones, los tiempos finales de la ejecución pueden llegar a ser incluso más mayores que los obtenidos en un monoprocesador.
   - la distribución de las distintas partes de un programa paralelo sobre los procesadores del sistema (problema del *mapping*). La forma en que se haga esta distribución puede, por ejemplo, acrecentar o paliar los retrasos debidos a la existencia de dependencias o comunicaciones.

2. ***Factores dependientes del sistema***. Aquí se encontrarían aquellos factores que viene fijados tanto por el *hardware* del sistema como por el conjunto de programas que integrarían parte del *software* del sistema. Los más importantes son:

   - el número de procesadores disponibles.
   - el *overhead* involucrado en la gestión de las tareas asignadas a un cierto procesador (creación, cambio de contextos, etc.).
   - el *overhead* en el que se incurre al utilizar mecanismos de comunicación y de sincronización (que incluye la necesidad de realizar llamadas a funciones del sistema y el cómputo que se realiza en ellas).

- la pérdida de eficiencia en el uso de recursos del sistema debido a la inclusión de un mayor número de procesos en cada procesador (por ejemplo, en el uso de la memoria *cache*).

En este trabajo abordaremos específicamente el problema conocido como problema del *mapping*, aportando una solución de tipo general que permita resolverlo de forma automática y que permita aprovechar al máximo las prestaciones del sistema.

1.2 El problema del *mapping*

Llegados a este punto, podemos suponer que disponemos ya de una aplicación que ha sido diseñada de forma que está compuesta por un cierto número de módulos que pueden ser asignados a distintos procesadores en el computador paralelo. Puede suceder que el número de tales módulos sea superior al número de procesadores del sistema o que la interconexión de los mismos difiera de la interconexión que une a los procesadores. En ambos casos se justifica la necesidad de calcular cuál es la mejor asignación que satisface unos de determinados criterios. Además, puede ser necesario establecer una planificación para cada elemento de procesamiento, o sea, un orden en el que se deben ejecutar los módulos que han sido asignados a un cierto procesador.

El problema de asignación adoptado en este trabajo se centra en lo que se entiende como *paralelismo funcional*, es decir, aquel que implica la creación de tareas diferenciadas que van a realizar diferentes operaciones sobre diferentes conjuntos de datos o diferentes operaciones sobre el mismo conjunto de datos. Este tipo de paralelismo se distingue, por lo tanto, del conocido como *paralelismo de datos* que implica la creación de tareas paralelas que operan sobre diferentes partes de los datos. En este caso, todas las tareas realizan el mismo conjunto de operaciones pero sobre diferentes conjunto de datos. Un ejemplo de este segundo caso lo constituye el paralelismo a nivel de lazos, donde cada iteración de un lazo puede verse como una tarea separada y el problema de la asignación consiste en distribuir los datos con los que va a operar cada procesador.

Cuando hablemos del problema de encontrar la asignación de tareas a procesadores en el supuesto de paralelismo funcional nos referiremos a él como "problema de la asignación" (*mapping problem*). Si existen más tareas que procesadores hablaremos de un *mapping* N a 1, si el número de tareas es igual al de procesadores hablaremos de *mapping* 1 a 1. Adicionalmente, nos referiremos al problema de la planificación (*scheduling problem*) en aquellos casos donde además de obtener una asignación sea necesario determinar el orden de ejecución dentro de cada procesador. Cuando hablemos de forma genérica de algún problema relacionado con la asignación de tareas a procesadores usaremos, normalmente, el término *mapping* por ser una acepción más general.

Encontrar una solución óptima a un cierto problema de *mapping* es conceptual y computacionalmente difícil. La mayor parte de las distintas versiones del problema que

suelen plantearse pertenecen a la categoría de problemas NP-completos [Gar79]. En consecuencia, ha existido y existe todavía una gran interés por desarrollar técnicas generales de optimización que puedan ayudar al programador de aplicaciones paralelas a encontrar asignaciones razonablemente eficientes. Por ello, la mayoría de autores se ha concentrado en el desarrollo de técnicas eficientes que obtengan soluciones cercanas al óptimo en un tiempo razonable, aunque existen también otros trabajos más teóricos destinados a delimitar la frontera de los casos especiales del problema que pueden ser computacionalmente factibles y para los que es posible encontrar el valor óptimo absoluto.

Cualquier aproximación al problema del *mapping* parte (implícita o explícitamente) de unos modelos abstractos que caracterizan tanto el computador paralelo como la aplicación paralela. Lógicamente, estas abstracciones enfatizarán algunas características de esos elementos a expensas de minimizar o excluir otras. Por lo tanto, al hablar de que determinada estrategia de asignación produce unos resultados que se acercan en un cierto porcentaje del óptimo, debe tenerse en cuenta que tal afirmación sólo se aplica a una formulación concreta del problema que presupone un cierto tipo de arquitectura y un cierto tipo de aplicación. La siguiente sección esboza cuáles son los modelos clásicos que aparecen en la literatura.

1.2.1 Modelos usados en el problema del *mapping*

El problema del *mapping*, planteado como un problema de optimización va a basarse en una serie de modelos abstractos que le permitan representar el sistema paralelo real. Fundamentalmente, los modelos corresponden a tres elementos básicos: por una parte, la aplicación o programa paralelo con sus módulos y sus patrones de comunicación, en segundo lugar, el computador o sistema paralelo compuesto por procesadores y facilidades de comunicación, y, por último, la función que va a ser usada para medir el coste de una determinada asignación[Nor93].

- ***Modelos de los módulos del programa***: se suele considerar un conjunto de módulos que constituyen la aplicación. Un módulo es una unidad de cómputo que se ejecuta secuencialmente y suelen ser unidades algorítmicas (funciones, en el caso de una descomposición funcional). De forma alternativa, en las aplicaciones de paralelismo de datos los módulos pueden corresponder al cómputo asociado con divisiones del espacio de datos. En el caso de los módulos de cómputo se suelen seguir tres tipos básicos:

    - Modelos que ejecutan los módulos de forma independiente y donde no ocurre comunicación entre los mismos.
    - Modelos basados en tareas. Consisten en tareas dispuestas en un grafo dirigido acíclico (*DAG: Directed Acyclic Graph*), donde un arco entre una pareja de nodos corresponde a una relación de precedencia entre los correspondientes módulos del programa y a un evento asociado de comunicación entre ellos. Estos modelos suelen ser usados por investigadores interesados en problemas de *scheduling* y tienen su

base tradicional en los sistemas multiprocesadores de memoria compartida.

- Modelos basados en procesos. Consisten en procesos dispuestos en un grafo no dirigido (*TIG: Task Interaction Graph*) donde un arco corresponde a un cierto volumen de comunicación entre los módulos. En este caso, suelen ser los modelos usados más frecuentemente en la asignación explícita de programas paralelos y acostumbran a tener su base en los sistemas distribuidos (aquellos donde los elementos de procesamiento constituyen computadores independientes que están unidos por redes tipo *ethernet*: las redes de estaciones de trabajo serían el ejemplo más típico).

- ***Modelos del sistema paralelo***: suele hablarse en este punto de un conjunto de procesadores que conforman el sistema paralelo y se considera que pueden ser de tres tipos: idénticos, uniformes o inconexos. El sistema está formado por procesadores idénticos cuando cualquiera de ellos procesa cualquier módulo del programa a la misma velocidad que los demás procesadores. Cuando los procesadores son uniformes el tiempo de cualquiera de ellos para procesar cualquier módulo es una constante entera multiplicada por una velocidad unitaria. Por último, en los procesadores inconexos un procesador p puede ser más rápido que otro q al computar un módulo x, pero más lento que éste al computar un módulo z. Este último modelo se correspondería con el caso más general de sistemas paralelos heterogéneos [Sto77][Lo88a].
  Por otra parte, el sistema estará caracterizado también por una serie de facilidades de comunicación entre los procesadores que van a implicar la introducción de un cierto coste de comunicación entre los módulos asignados a los distintos procesadores del sistema. En este caso nos podremos encontrar diferentes opciones como las siguientes:

  - El coste de comunicación entre módulos del programa es independiente del procesador al que estén asignados.

  - El coste de comunicación entre los módulos depende sólo de si han sido asignados a un mismo procesador o no.

  - El coste de comunicación entre los módulos depende de los procesadores a los que han sido asignados. Esto lleva a considerar, en primer lugar que el coste de las comunicaciones entre módulos asignados a un par de procesadores depende de su distancia. Si la arquitectura se modela como un grafo no dirigido donde los procesadores son los nodos y las conexiones físicas entre los procesadores son los arcos, se entiende como distancia el número de arcos que hay que atravesar para ir desde un procesador origen al procesador destino. En segundo lugar, el coste de una comunicación puede depender de la distancia entre los procesadores y de la contención encontrada en los caminos del grafo de procesadores entre

esa comunicación y otras comunicaciones que puedan estar circulando en cada momento.

- ***Modelos de coste***: existen distintas formas para formular la "optimización" en el problema del *mapping*. Por ejemplo, desde el punto de vista de un sistema operativo es útil construir la optimización en términos de maximizar el *throughput* de trabajos [Bac90]. En otros casos, el interés puede venir fijado por minimizar el número de procesadores que pueden garantizar un cierto nivel de rendimiento (los sistemas de tiempo real son un ejemplo de ello) [Fer73][AlM90][Hou90]. Sin embargo, el interés de nuestro trabajo se va a centrar en aquellas formulaciones donde dado un número fijo de procesadores se va a optimizar el rendimiento de una aplicación.

A pesar de la diferenciación que se ha hecho en el uso tradicional entre DAGs y TIGs, su aplicación práctica resulta más ambigua y, así por ejemplo, podemos encontrar políticas de *scheduling* usando DAGs diseñadas para sistemas multiprocesador de memoria distribuida.

1.2.2 Clasificación jerárquica de las políticas de *mapping*

La clasificación en la que vamos a basarnos a continuación se debe a Casavant y Kuhl [Cas88], compartida también por El-Rewini[ElR94] en sus categorías básicas, y es, por lo tanto, una de las más comúnmente aceptadas para ordenar las distintas estrategias de *mapping* que han aparecido en la literatura. La clasificación, que se muestra en la figura 1.2, utiliza un criterio que considera fundamentalmente el tipo de algoritmo usado en la resolución del problema para agrupar las distintas estrategias en una categoría concreta. En la siguiente sección se presentará una segunda clasificación que obedece más a la forma en que se modela el problema y que, en definitiva, va a suponer la adopción de una abstracción concreta de los elementos que intervienen en un computador paralelo.

A - Mapping local versus Mapping global

El primer nivel de la jerarquía distingue por un lado aquellas estrategias que tratan de resolver el problema de asignar los ciclos de un procesador entre distintos procesos en un entorno monoprocesador[Cof73][Tsi74] o bien resolver la asignación de trabajos independientes en entornos distribuidos (*job-shop scheduling*) [Con67][Eva84]. Frente a estas estrategias que se centran en la resolución del mapping local están las que se centran en el problema del mapping global. En este caso se trata de decidir *dónde* ejecutar un proceso y, eventualmente, decidir el orden de ejecución local, aunque este segundo factor puede delegarse en el planificador de CPU del procesador en cuestión. En este tipo de mapping las tareas forman parte de un único programa, siendo objetivo de la estrategia de asignación la minimización del tiempo de ejecución de la tarea.

En nuestro trabajo nos centraremos en el problema del mapping global.

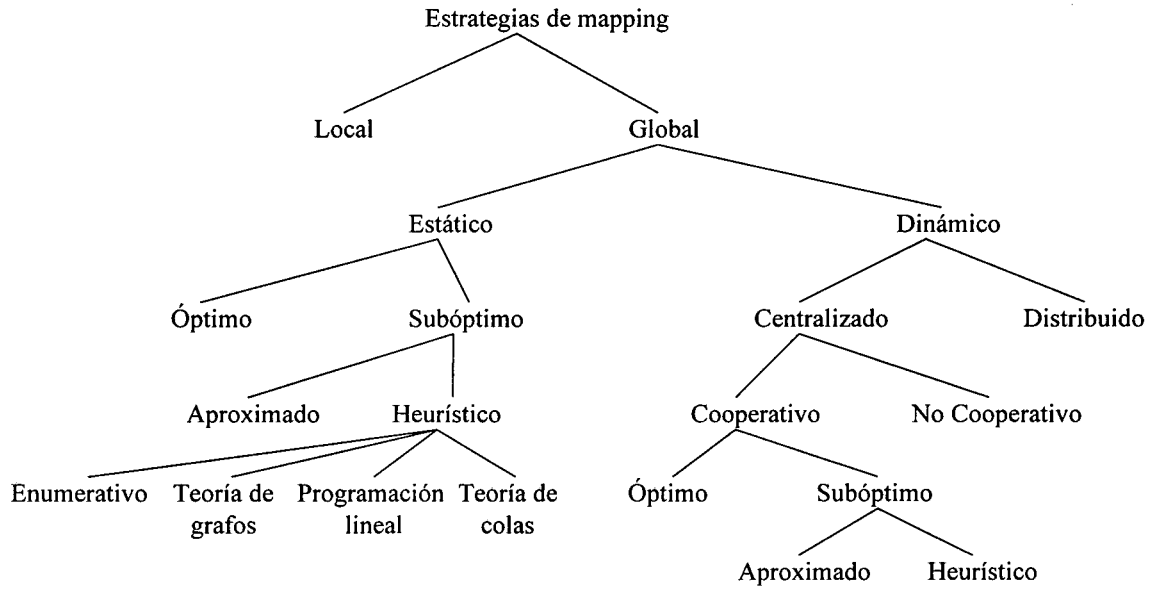


Figura 1.2 Clasificación jerárquica de las políticas de mapping

B - Estático versus dinámico

El siguiente nivel en la jerarquía distingue entre mapping estático y dinámico. Su diferencia radica en el momento en el cual se toma la decisión sobre la asignación y en la posibilidad de modificar la asignación realizada previamente.

En el caso del mapping estático, la información relativa al total de procesos de la aplicación se supone conocido y disponible en tiempo de compilación y antes de la ejecución de la misma. De este modo, se puede calcular la asignación *off-line* sin introducir ningún *overhead* extra en la ejecución de la aplicación. La eficiencia de estas políticas de asignación va a depender en gran medida de la bondad de los parámetros conocidos del programa que van a usarse para predecir su comportamiento durante la ejecución.

Por su parte, el *mapping* dinámico se realiza en tiempo de ejecución, sin requerir para ello de un conocimiento preciso del programa como en el caso anterior. En este caso las tareas pueden ser transferidas de un procesador a otro durante la ejecución del programa atendiendo a criterios de balanceo de carga entre los procesadores del sistema. Se pretende con ello distribuir el cómputo equitativamente entre todos los procesadores evitando la existencia de procesadores inactivos por falta de tareas listas mientras otros procesadores tienen más de una tarea lista [Liv82][Car85][Wan85][Cor92][Luq95]. Por el hecho de realizarse al mismo tiempo que la ejecución del programa se va a introducir un *overhead* por el intrusismo de la propia política monitorizando la situación del sistema y tomando decisiones, y por el movimiento posterior de tareas entre procesadores como consecuencia de las decisiones de la política.

A continuación analizaremos cada una de estas dos ramas de la clasificación.

C - Óptimos versus Sub-óptimos

En el caso de que toda la información referente al estado del sistema así como a los recursos que necesita la aplicación sean conocidos, se puede obtener una asignación óptima en base a un determinado criterio de optimización o función de coste. Ejemplos de criterios de optimización pueden ser la minimización del tiempo de respuesta de la aplicación, la maximización del uso de los recursos del sistema o la minimización del tiempo de ejecución [Ram72]. Si la resolución del problema del *mapping* se convierte en intratable desde un punto de vista computacional, su solución puede realizarse mediante las técnicas del siguiente nivel en la jerarquía: aproximadas o heurísticas.

D - Aproximadas versus Heurísticas

Las técnicas aproximadas utilizan el mismo modelo computacional para el algoritmo que las óptimas, pero en vez de hacer una búsqueda exhaustiva por el espacio de soluciones, se conforman con encontrar una "buena" solución dentro de unos rangos fijados a priori.

Por su parte, las técnicas heurísticas son las que utilizan unas suposiciones más realistas referentes al conocimiento a priori que se tiene del sistema y del algoritmo paralelo. También representan el grupo de estrategias que requieren unos tiempos de ejecución más razonables en comparación con las estrategias óptimas o aproximadas. También suelen hacer uso de parámetros especiales que afectan al sistema de modo indirecto. En la mayoría de las ocasiones, su planteamiento surge de ideas puramente intuitivas sin que exista ninguna prueba matemática de que exista una relación efectiva entre los mecanismos propuestos y la bondad de los resultados obtenidos.

E - Técnicas óptimas y subóptimas aproximadas

Tanto si las soluciones al problema del *mapping* se obtienen mediante técnicas óptimas como aproximadas se puede establecer una clasificación adicional de las mismas en cuatro grupos.

- *Métodos enumerativos*: las estrategias que proporcionan unos mejores resultados son los algoritmos basados en técnicas de enumeración implícita del espacio de soluciones (p. ej., métodos *Branch and Bound* y programación dinámica) [Sch70][Kho74]. Su grave inconveniente es que este tipo de algoritmos no poseen una complejidad polinómica. El análisis de todas las posibles soluciones no va a ser realizable y, por tanto, la bondad de la solución obtenida dependerá fuertemente del tiempo de ejecución que se dedique al algoritmo y del tamaño original de la aplicación que se quiere asignar.

- *Métodos basados en teoría de grafos*: en estas estrategias suelen englobarse los métodos basados, por ejemplo, en algoritmos de *max-flow min-cut* (corte mínimo con máximo flujo) de un grafo que representa al programa paralelo, de tal forma que se consiga una asignación en la que se minimice la comunicación entre procesos [Bok81][Sto77][Sto78][Lo88a]. Debido al tipo

involucrado en la toma de decisiones esté distribuido físicamente entre varios procesadores [Ens78]. La primera alternativa, dada su naturaleza centralizada, suele obtener mejores resultados en la distribución del trabajo entre los procesadores a costa de convertir el procesador encargado de ejecutar la política en un cuello de botella para el sistema, que será por ello menos tolerante a fallos y menos escalable. Sacrificando parcialmente la efectividad obtenida por las estrategias centralizadas, los métodos distribuidos suelen ofrecer mejores perspectivas por lo que a fiabilidad y escalabilidad se refiere. Este grupo de estrategias se divide a su vez en dos categorías que nos permiten descender un nuevo tramo dentro del árbol de la clasificación.

G - Cooperativo versus no cooperativo

Las políticas cooperativas, como su propio nombre indica, son aquellas donde existen mecanismos que implican la existencia de mecanismos de colaboración entre componentes distribuidos, mientras que en las no cooperativas cada procesador individual toma sus propias decisiones de forma independiente a las acciones de los otros procesadores del sistema. La cuestión fundamental en este caso radica en el grado de autonomía que cada procesador tiene a la hora de determinar cómo se usan sus recursos propios. En el caso no cooperativo cada procesador actúa solo, como una entidad independiente, y toma decisiones sobre el uso de sus recursos con independencia del efecto que esas decisiones pueden tener sobre el resto del sistema. En el caso cooperativo cada procesador tiene la responsabilidad de llevar a cabo su parte del trabajo de *mapping* pero todos los procesadores están trabajando conjuntamente para alcanzar un objetivo común que va a afectar al rendimiento de todo el sistema.

Como en el caso estático, las políticas dinámicas se pueden subdividir a su vez en óptimas, sub-óptimas aproximadas y heurísticas, y nos encontramos, por lo tanto, con una discusión equivalente a la que ya se hizo en el caso estático.

H - Otros factores

La clasificación jerárquica presentada no contempla, sin embargo, algunas características diferenciales entre las políticas de *mapping* que no son privativas de una sola rama de la clasificación sino que, en cierto modo, pueden ser comunes a diferentes políticas pertenecientes a distintas ramas.

- *Adaptativos versus no adaptativos*: una solución adaptativa al problema del *mapping* es aquella en la que los algoritmos y los parámetros usados para implementar la política cambian dinámicamente de acuerdo con el comportamiento previo y actual del sistema en respuesta a las decisiones previas realizadas por la política de *mapping* [Sta84]. Las soluciones no adaptativas son aquellas que no varían necesariamente su mecanismo básico de control en base a la historia de la actividad del sistema.

- *Apropiativo (preemptive) versus no apropiativo (non-preemptive)*: en este caso la clasificación atiende a cómo ejecuta sus tareas cada procesador local. Un *scheduling* apropiativo es aquel en el cual se permite que un procesador que está ejecutando una tarea deje de ejecutarla antes de que ésta haya sido

completada y pase a ejecutar otra tarea de prioridad mayor [Mun69]. Los *schedulings* no apropiativos, por contra, son aquellos donde no se permite que un procesador deje de ejecutar una tarea una vez que ha iniciado su ejecución [Cof76]. De este modo, se evitan los continuos cambios de contexto y los retardos que ello introduce. Sin embargo, los resultados teóricos de este tipo de *scheduling* no son tan satisfactorios como los obtenidos en los apropiativos. Para los *schedulings* no apropiativos pueden existir determinadas circunstancias donde resulte mejor dejar un procesador inactivo aunque existan tareas listas para su ejecución. Desgraciadamente, el análisis de este tipo de situaciones complican de modo desmesurado estas políticas y las hacen difíciles de integrar en sistemas reales. Por este motivo, generalmente no se permite que procesadores libres permanezcan inactivos si existen tareas listas, aunque esta decisión pueda perjudicar el rendimiento final obtenido [Gra79].

- *Sistema con única aplicación versus múltiples aplicaciones*: en este caso se está distinguiendo el tipo de entidad que debe asignarse. Si las entidades son tareas que comprenden un programa paralelo, podemos encontrar dos casos distintos. El primero es aquel en el que sólo una aplicación puede ejecutarse a la vez. La aplicación consistiría en un conjunto de tareas que cooperarían y se comunicarían entre sí, y el objetivo de la asignación sería, en este caso, minimizar el tiempo de ejecución de dicha aplicación. En el segundo caso, varias aplicaciones paralelas pueden estar ejecutándose a la vez, por ejemplo, en un entorno de tiempo compartido como el Sequent Symmetry. El objetivo en este segundo caso será minimizar el tiempo de respuesta y el tiempo promedio de finalización por aplicación.

  Una variante del caso propuesto lo constituyen aquellos problemas de asignación donde se deben asignar módulos de un programa paralelo que a su vez son programas paralelos. Esta complicación extra debido a los dos niveles de granularidad en el paralelismo del programa no es gratuita si pensamos en aquellas situaciones en las que el programador construye sus aplicaciones haciendo uso de determinadas funciones de biblioteca que son algoritmos ya paralelizados y para los cuales puede existir una estrategia de asignación fija, y que puede resultar óptima para una determinada familia de algoritmos. Ante estas situaciones puede resultar atractivo el uso de entornos de programación que hagan uso de estrategias de asignación preestablecidas y aplicables a partes concretas de una aplicación en función de las características particulares de cada una de esas partes [Lo92][Wan92].

1.2.3 Clasificación de las políticas de *mapping* según los modelos de cómputo adoptados

En el apartado anterior se ha visto una posible clasificación de las políticas de *mapping* en base a la estrategia adoptada para abordar y resolver el problema. Existe, además, un segundo tipo de clasificación donde se hace hincapié, no tanto en el algoritmo de resolución, sino en los modelos subyacentes que utiliza dicho algoritmo.

Esta clasificación se basa fundamentalmente en los modelos tradicionales usados para abstraer los módulos de un programa paralelo [Nor93].

A - Módulos independientes

El modelo más simple y más tratable desde el punto de vista computacional es aquel en el que el cómputo se modela como módulos, cada uno de ellos se ejecuta secuencialmente en un sólo procesador y no existe comunicación entre ellos. Este problema suele ser conocido como problema de *scheduling*, pero, realmente, la última característica mencionada en el modelo implica que éstos puedan ejecutarse en cualquier orden; por lo tanto, el problema es simplemente un problema de asignación donde el objetivo se centra en equilibrar la carga de cómputo de los distintos procesadores del sistema.

Este modelo correspondería a la ejecución de módulos independientes en un modo que algunos autores denominan como *embarassingly parallel* [Fox89]. Por ejemplo en las implementaciones paralelas de algoritmos de *ray-tracing* el cómputo suele dividirse en módulos que corresponden al cálculo de una rejillas cuadradas que juntas construyen toda la imagen. Los módulos se pueden ejecutar independientemente y el tiempo de cómputo de cada uno de ellos puede variar respecto a los demás.

En la figura 1.3 se muestra un ejemplo de este modelo. Existe un conjunto de tareas con su correspondiente tiempo de ejecución que son asignadas a un sistema con dos procesadores. El tiempo resultante al realizar la ejecución con dicha asignación es de 58 milisegundos (tiempo del segundo procesador).

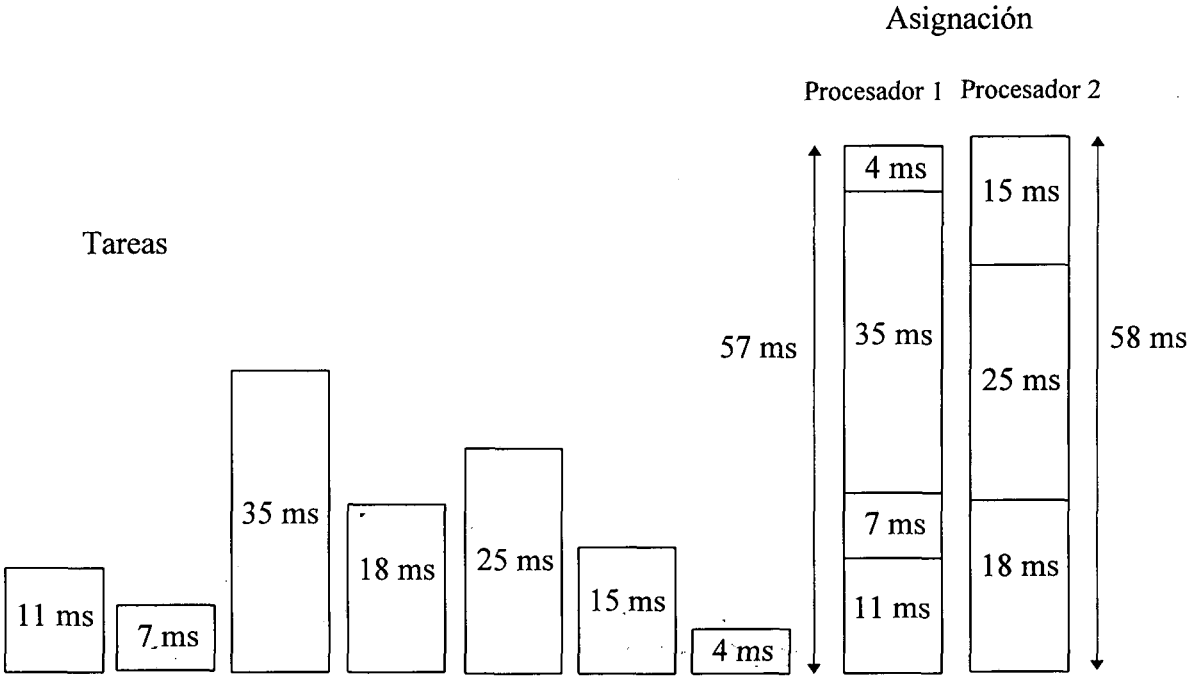


Figura 1.3. Ejemplo de asignación de módulos independientes

B - Módulos con precedencia

En este modelo el programa se representa como un conjunto de tareas que se comunican los resultados entre sí una vez han acabado. La estructura del cómputo se representa como un "digrafo" en el que un arco dirigido conecta un par de tareas distintas sí y sólo sí la tarea destino requiere los resultados de la tarea fuente del arco.

Este tipo de grafo representa un orden parcial del conjunto tareas que componen el programa donde determinadas acciones deben hacerse antes de otras. En un grafo válido no se permite ningún tipo de ciclo o vuelta atrás, por lo que nos encontramos ante un grafo dirigido acíclico (DAG).

Al igual que en el caso anterior los nodos del grafo están caracterizados por un valor (normalmente entero) que representa de algún modo el volumen de cómputo involucrado en la ejecución de ese módulo del programa (y que, idealmente, se correspondería de forma directa con el tiempo de ejecución del mismo). En este caso, el problema del *mapping* va a consistir tanto en determinar la colocación de cada módulo en un procesador y en fijar un orden de ejecución dentro de cada procesador, sujeto a dos restricciones: ningún procesador ejecuta más de una tarea a la vez y las tareas que preceden a otra cierta tarea han terminado cuando empieza la ejecución de ésta.

En este modelo aparece el concepto de camino crítico, entendido como el camino más largo que va desde el nodo inicial del grafo (aquel sin predecesores) al nodo final (aquel sin sucesores). La importancia del camino crítico es que, independientemente del número de procesadores, la suma de los volúmenes de cómputo de los nodos de este camino representa el tiempo mínimo alcanzable en la ejecución del programa.

En la figura 1.4 se muestra un ejemplo de grafo DAG junto con el resultado de la ejecución de dicho grafo asignado sobre dos procesadores. Ese diagrama (conocido como diagrama de Gantt) muestra la actividad de los procesadores a lo largo del tiempo y también muestra (mediante un arco) la ocurrencia de comunicaciones entre procesos asignados a procesadores distintos. En este caso, el camino crítico es el constituido por las tareas asignadas al procesador 1.

De los métodos heurísticos propuestos para resolver este problema llamado del *scheduling* en este caso, destacan las políticas de lista como uno de los más efectivos [Kas84][Shi90]. La base de su funcionamiento consiste en dos pasos principales. Primero se ordenan las tareas del grafo del programa según una función de coste previamente seleccionada, generando con ello una lista de prioridad. Después se realiza un proceso de simulación de la ejecución del programa de forma que se van asignando tareas a medida que quedan procesadores desocupados y que éstas han sido activadas por la finalización de las tareas predecesoras. La lista de prioridad se utiliza en los casos en que los que existe más de una tarea elegible para ejecución en un momento determinado.

de algoritmo en el que se basan estas estrategias, sus resultados son óptimos cuando el sistema paralelo consiste en dos procesadores o en un array lineal de procesadores. En otros supuestos sus resultados serán, en general, subóptimos.

- *Métodos basados en programación matemática*: en estas estrategias se formula el problema de la asignación de tareas como un problema de optimización que se va a solucionar a través de una estrategia matemática de programación entera [Ma82][Gab82]. Los objetivos de las solución suelen ser la minimización de las comunicaciones entre procesadores, equilibrar el uso de todos los procesadores y satisfacer otros requerimientos como restricciones de memoria del sistema o restricciones de tiempo (*deadlines*). A partir de este planteamiento la asignación se representa por un conjunto de variables cero-uno y el coste de ejecución total de la aplicación se representa por la suma de todos los costes en los que incurrirá una determinada asignación. El problema se formula además de modo que se incluyan algunas de las restricciones mencionadas. De este modo, el problema puede resolverse usando una estrategia de programación no lineal 0-1 entera, que suelen realizar una búsqueda por el espacio de soluciones usando técnicas derivadas del método básico de *branch-and-bound*. Se les suele criticar a estas técnicas su necesidad de imponer un elevado número de restricciones para que la resolución de las ecuaciones resultantes sea razonable desde el punto de vista del tiempo de cómputo requerido.

- *Métodos basados en teoría de colas*: estos métodos no consideran la existencia de comunicaciones entre las tareas que componen un programa paralelo. Los objetivos de la soluciones suelen ser minimizar el tiempo de ejecución del programa completo con objeto de maximizar el rendimiento del sistema, y los algoritmos aplicados se derivan de resultados de las teorías de decisión de Markov [Kle81][Cho82].

Se han presentado hasta aquí el conjunto de estrategias que conformarían el grupo de métodos estáticos. A continuación se describirá la rama de la clasificación que engloba las estrategias dinámicas. En ellas se parte de la suposición de que existe muy poco conocimiento a priori sobre los recursos que va a necesitar una aplicación paralela y se desconoce también en qué situación se va a encontrar el sistema sobre el que se ejecute. En el caso estático las decisiones sobre el programa ejecutable se toman antes de que este sea cargado físicamente en los procesadores, mientras que en el caso dinámico no se toma ninguna decisión hasta que el programa empieza su funcionamiento en el entorno dinámico de un cierto sistema. Dado que será responsabilidad del sistema decidir dónde se ejecuta un determinado proceso, la primera subdivisión de las estrategias dinámicas se basa en el punto donde se toma tal decisión.

F - Distribuido versus no distribuido

Existen dos alternativas para realizar un *mapping* dinámico global: que la toma de decisiones se haga físicamente en un solo procesador [Ous80] o que el trabajo

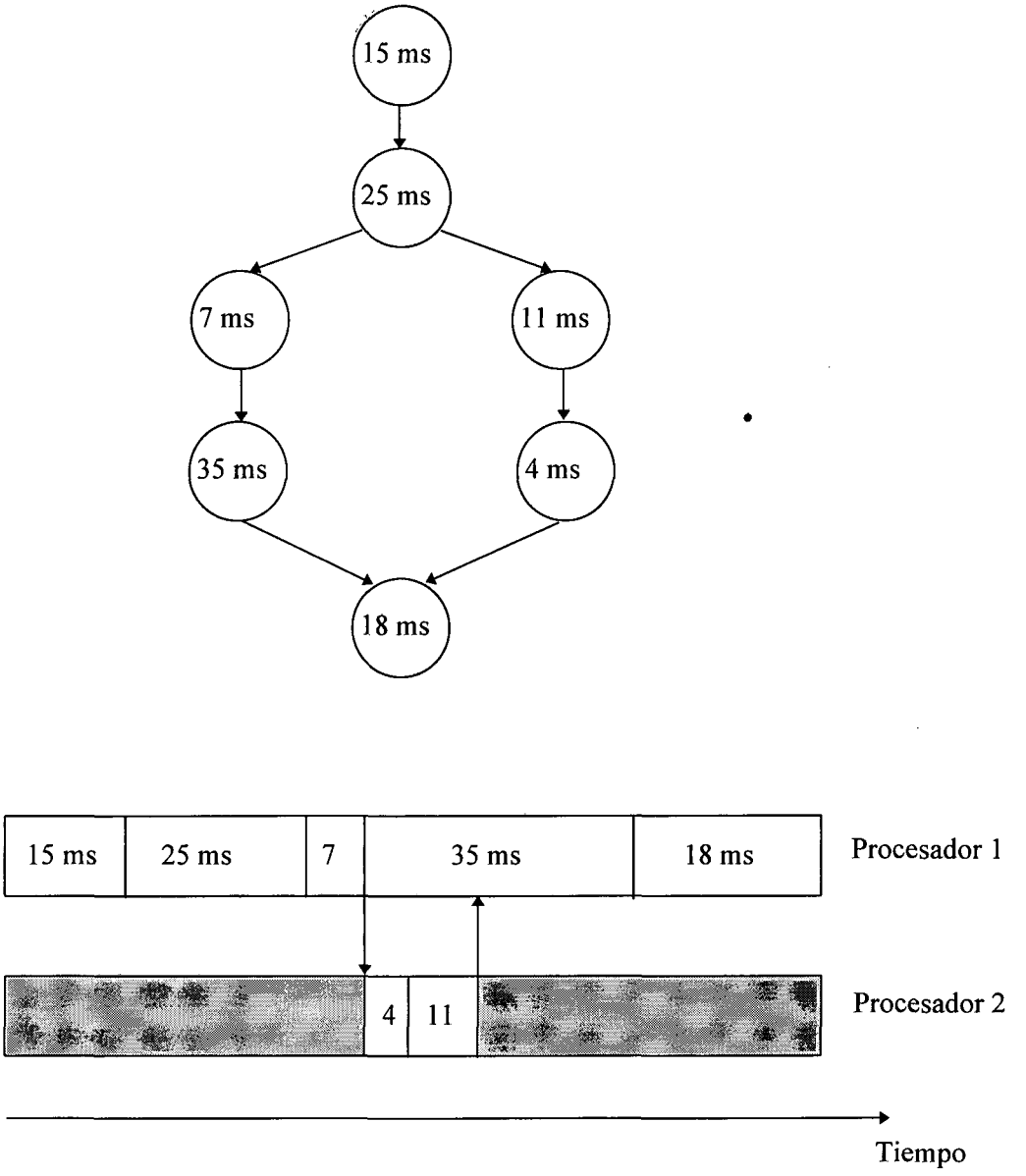


Figura 1.4 Ejemplo de grafo DAG y su traza de Gantt

La ejecución de los procesos es no apropiativa y no existe replicación de tareas. Sin embargo, extensiones del modelo básico donde sí se permite la replicación de nodos. Una primera extensión incluye nodos del grafo con tiempo de ejecución variable [Luq90][Her91], donde se estarían modelando la existencia de sentencias condicionales o iterativas en el interior de cada nodo. Una segunda ampliación del modelo permite considerar además la existencia de lazos estructurados en el grafo de tipo Do-serial, Do-all o Do-across [Luq93][Mar93].

C - Módulos con precedencia entre tareas y retardos de comunicación

Este modelo no es más que una extensión inmediata al modelo del apartado anterior introduciendo para ello el retardo involucrado en las comunicaciones entre las tareas del programa [Vel90]. En este caso, el grafo resultante contiene arcos etiquetados

con un valor entero que se correspondería, por ejemplo, con el número de octetos transmitidos entre una tarea origen y una tarea destino. Como en el caso anterior, el problema del *mapping* se divide en un problema de asignación y en un problema de planificación de CPU. La figura 1.5 muestra un ejemplo de este modelo. A diferencia del ejemplo de la figura anterior, en este caso la influencia de la comunicación retrasa el tiempo total de ejecución del programa.

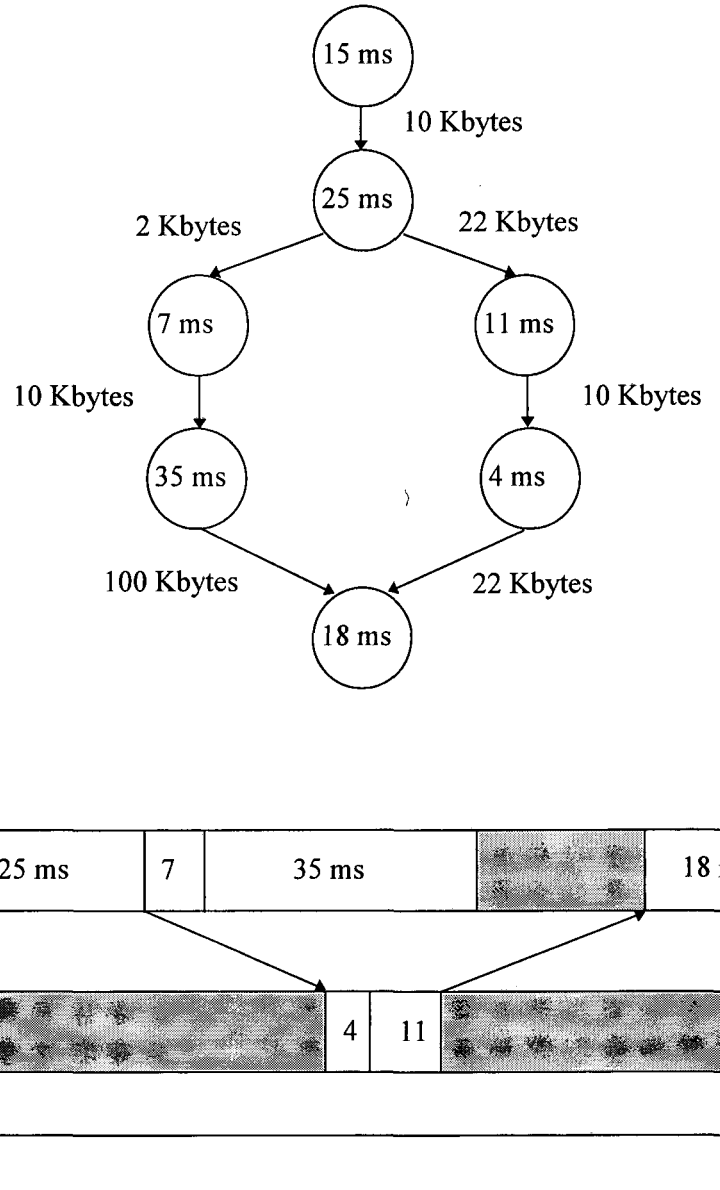


Figura 1.5. Ejemplo de grafo DAG con retardos de comunicación

En su formulación más simple se suele suponer que existe un retraso uniforme de comunicación desde que una tarea genera sus resultados hasta que estos son conocidos por los demás procesadores. En las formulaciones más habituales el retraso va a depender de un factor que mide la distancia entre los procesadores donde se

encuentran el nodo origen y el nodo destino, y en algún caso se han llegado a considerar los problemas de contención en la red de interconexión [ElR90].

Las soluciones propuestas en este caso incluyen, entre los más importantes, métodos que son extensiones a los métodos de lista mencionados en el apartado anterior [Hwa89], estrategias que contemplan la replicación de tareas (recomputando más de una vez una misma tarea) como mecanismo para evitar el retraso de las comunicaciones [Pap90], y métodos de dos etapas donde las tareas primero se asignan sobre un número ilimitado de procesadores y después se reasignan sobre un número fijo de procesadores [Yan94][Sar89][Thu92].

D - Procesos estáticos

En este modelo el programa se representa también mediante un grafo, aunque en este caso es un grafo no dirigido o TIG (del inglés, *Task Interaction Graph*). Los nodos del grafo siguen siendo módulos del programa caracterizados por un cierto volumen de cómputo como en el caso de los DAGs, pero los arcos ahora representan comunicaciones, que pueden ser bidireccionales, en vez de relaciones de precedencia y comunicaciones unidireccionales como ocurría en el caso anterior. Este modelo suele usarse para representar cómputos de granularidad elevada (*coarse-grain*), donde los módulos son procesos persistentes que existen durante todo el tiempo en el que se está ejecutando el programa y donde las comunicaciones siguen unos patrones estables a lo largo del tiempo. En muchos casos existe una equivalencia entre un modelo de cómputo basado en procesos y un grafo de dependencias probablemente cíclico y con bifurcaciones probabilísticas, pudiéndose a partir de éste derivar el TIG correspondiente [Chu84].

Los programas escritos mediante lenguajes de programación paralelos de tipo procedural [Bal89] o mediante lenguajes secuenciales ampliados con bibliotecas de gestión y comunicación de procesos como PVM o MPI suelen corresponder mayoritariamente a este tipo de modelo [And91]. Así, podemos encontrar, por ejemplo, programas compuestos por grupos de tareas idénticas que, a partir de unos valores de datos iniciales, calculan la solución mediante sucesivos pasos de refinamiento, cada uno de los cuales se caracteriza por un cierto cómputo del proceso seguido de una cierta comunicación con los procesos vecinos. Otro de los ejemplos característicos de aplicación lo constituyen los cómputos que siguen un modelo de *pipeline* donde los datos atraviesan una serie de capas de procesos funcionalmente especializados, dando lugar a una situación similar a una cadena de montaje.

Dado que este es el modelo adoptado en este trabajo para abordar el problema del *mapping,* en el siguiente capítulo se describirán con más detalle las estrategias de asignación correspondientes a este modelo que han servido de base más directa a nuestro trabajo. Sin embargo, mencionaremos aquí que dentro del propio modelo de TIG han existido diversas variantes en función de las características que se asumían en el modelo del grafo y de la arquitectura. Estas variantes incluyen la consideración de una arquitectura heterogénea [Sto77][Chu80][Lo88a], la consideración de grafos del programa con el mismo número de procesos que procesadores con arcos de coste de comunicación unitarios [Bok81] (*mapping* 1 a 1), la consideración de contención de la

red [Lee87][Ber87] y la consideración de grafos de cualquier número de nodos y arcos con costes no unitarios [Erc90][Wu94][Bou95] (*mapping* N a 1), que serán los que analizaremos. En la figura 1.6 se muestra un ejemplo del modelo de TIG, adoptado por Bokhari en [Bok1], y su correspondiente asignación a un *array* circular de procesadores. Se puede comprobar que tal asignación implica que la comunicación entre los procesos C y D no coincida con un enlace físico de la arquitectura.

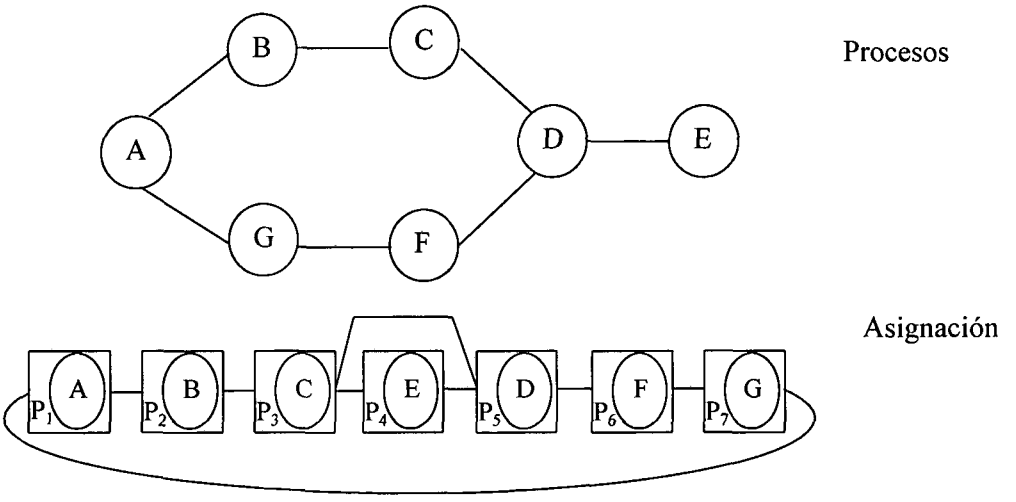


Figura 1.6 Ejemplo de grafo TIG y su asignación a un array de procesadores

1.3 Partición del programa y asignación

En los puntos anteriores se ha tratado de las distintas soluciones genéricas que se han propuesto al problema del *mapping* partiendo en todo momento de que el programa paralelo estaba dividido en una serie de módulos que debían ser asignadas. También se ha comentado que dichos módulos suelen identificarse normalmente con las tareas definidas a partir de un lenguaje de programación, sin embargo, ésta no es la única posibilidad y existen otras alternativas a partir de las cuales se generarán los módulos que deberán ser asignados. A continuación veremos cuáles son esas posibilidades.

Desde el punto de vista del programador de aplicaciones paralelas se pueden establecer tres fases en el desarrollo de una aplicación paralela por lo que concierne al problema del *mapping*:

1. identificación del paralelismo en el algoritmo propuesto para resolver determinado problema.
2. división del programa en tareas paralelas de acuerdo con la descomposición planteada en el punto anterior.
3. asignación y planificación de las tareas sobre el computador paralelo.

La primera de estas fases es claramente dependiente del problema que se pretende resolver y del programador que va a proponer la correspondiente solución. En definitiva, esta fase corresponde a la elaboración de un algoritmo de resolución de un determinado problema y ésta es una tarea para la que no existen soluciones automatizadas y se depende en buena medida de la habilidad y conocimientos del programador a la hora de encontrar una solución ingeniosa y eficiente. Aunque con lo dicho estamos presuponiendo la elaboración de algoritmos paralelos puros, cabría también la posibilidad de que el punto de partida fuese un algoritmo secuencial convenientemente paralelizado por un compilador que efectuase dicha tarea automáticamente; sin embargo, es sabido que un algoritmo paralelo eficiente para un determinado problema normalmente es bastante diferente a un algoritmo secuencial eficiente para el mismo problema.

La segunda de las fases se suele identificar con el problema del *particionamiento* (*partitioning*). El uso de un determinado lenguaje de programación va a permitir expresar el algoritmo ideado en la fase anterior como un programa paralelo a partir del cual se van a identificar toda una serie de unidades paralelas. Cómo son esas unidades paralelas y qué intervención tiene el programador en su definición y gestión son aspectos que dependen fuertemente del lenguaje de programación empleado. En los lenguajes paralelos procedurales (Ada, Parallel C, OCCAM,...) o en los lenguajes secuenciales tradicionales (C, FORTRAN) extendidos con bibliotecas de paso de mensajes (PVM, MPI), el programador está forzado a descomponer explícitamente el programa en tareas (unidades) y controlar su sincronización y comunicación. Sin embargo, en otro tipo de lenguajes paralelos, como los de tipo funcional o lógico, la expresión del paralelismo puede llegar a realizarse a nivel implícito o, aunque sea expresado también explícitamente, el compilador tiene una intervención importante en el momento de generar las unidades paralelas que, finalmente, pasarán a la tercera fase de la que ya hemos hecho mención en secciones anteriores.

Al hablar de la fase de partición se suele mencionar también el concepto de granularidad de un programa paralelo y granularidad de un computador paralelo, y que ya habíamos comentado brevemente al tratar los factores que limitaban la expansión del paralelismo. El primero de los conceptos hace referencia al tamaño promedio de una unidad secuencial de cómputo en el programa sin sincronizaciones o comunicaciones interprocesos. Para un multicomputador dado existe una granularidad mínima por debajo de la cual su rendimiento se degrada significativamente. Este límite es el que puede entenderse como granularidad del multicomputador. Para un multicomputador es deseable tener una granularidad fina, de forma que pueda soportar eficazmente un amplio rango de programas. Para un programa paralelo, por el contrario, es deseable que tenga una granularidad grande, de forma que pueda ser ejecutado eficientemente en un rango amplio de computadores paralelos.

Relacionado con el concepto de granularidad mencionado estaría la propiedad de la escalabilidad, como propiedad altamente deseable en un sistema. Podemos entender la escalabilidad como la capacidad de un sistema paralelo en incrementar linealmente su *speed-up* con un incremento lineal en el número de procesadores, suponiendo, claro está, que el programa tiene suficiente paralelismo y exhibe suficiente granularidad.

Existe un compromiso claro entre escalabilidad y granularidad: se consigue elevar la escalabilidad con granularidades mayores.

Con este planteamiento, el problema del particionamiento trata de especificar las unidades secuenciales de cómputo en el programa paralelo (o tareas, aunque no tienen que identificarse necesariamente con las características de una tarea de un sistema operativo). En este sentido, las tareas tienen una serie de propiedades interesantes para la resolución del mencionado problema:

1. el tiempo de ejecución secuencial de la tarea (o tamaño de la tarea).
2. el *overhead* total de la tarea, incluyendo el *overhead* debido a la planificación y el de comunicaciones de la tarea.
3. las limitaciones de precedencia de la tarea, que especifican el paralelismo potencial en el programa.

Lógicamente, el tiempo de ejecución de un programa va a depender de su partición así como de su *scheduling*. Una granularidad excesivamente pequeña comportará el excesivo *overhead* incurrido en la gestión de las tareas. El tiempo de ejecución se minimizará a una granularidad intermedia óptima que represente un compromiso entre la reducción de dichos *overhead* y la pérdida de paralelismo. El problema del particionamiento puede entenderse como el problema de encontrar dicho valor intermedio óptimo.

De lo dicho anteriormente se deduce una aproximación al problema del *mapping* como la propuesta por autores como Sarkar [Sar89] que consideran que existen dos fases en la asignación de procesos a procesadores: particionamiento, donde se forman "granos" a partir de tareas atómicas, y después la asignación propiamente dicha. La primera fase está precedida de la identificación del paralelismo que puede pensarse como una fase de compilación que extrae de un programa un conjunto de tareas paralelas atómicas que serán agrupadas en la fase de partición. Este proceso queda ilustrado en la parte izquierda de la figura 1.7.

Sarkar basa su propuesta en el lenguaje funcional Sisal, aunque, como ya se ha mencionado, el grafo de tareas atómicas podría ser obtenido a partir de un programa escrito en un lenguaje secuencial convencional. En este caso, sin embargo, hay que tener en cuenta que, en la práctica, lo que pueden hacer los compiladores paralelizantes actuales con los lenguajes secuenciales es normalmente traducir automáticamente algunas secciones del código mediante transformaciones que aprovechen el paralelismo de bajo nivel e incrementen la localidad de memoria [Zim90][Ban93]. En general, se pueden hallar dependencias cercanas pero el análisis interprocedural es todavía objeto de intensa investigación. Como resultado, el grafo de dependencias que va a generar el compilador sobreespecificará la secuencialidad del algoritmo secuencial subyacente. Es decir, en aquellos grupos de sentencias donde existan dependencias que no sean claramente paralelizables se ejecutarán de forma secuencial, aunque en la práctica los valores tomados por los datos permitiesen su ejecución paralela.

A partir del grafo de tareas atómicas se han presentado tres alternativas para la resolución del problema del mapping.

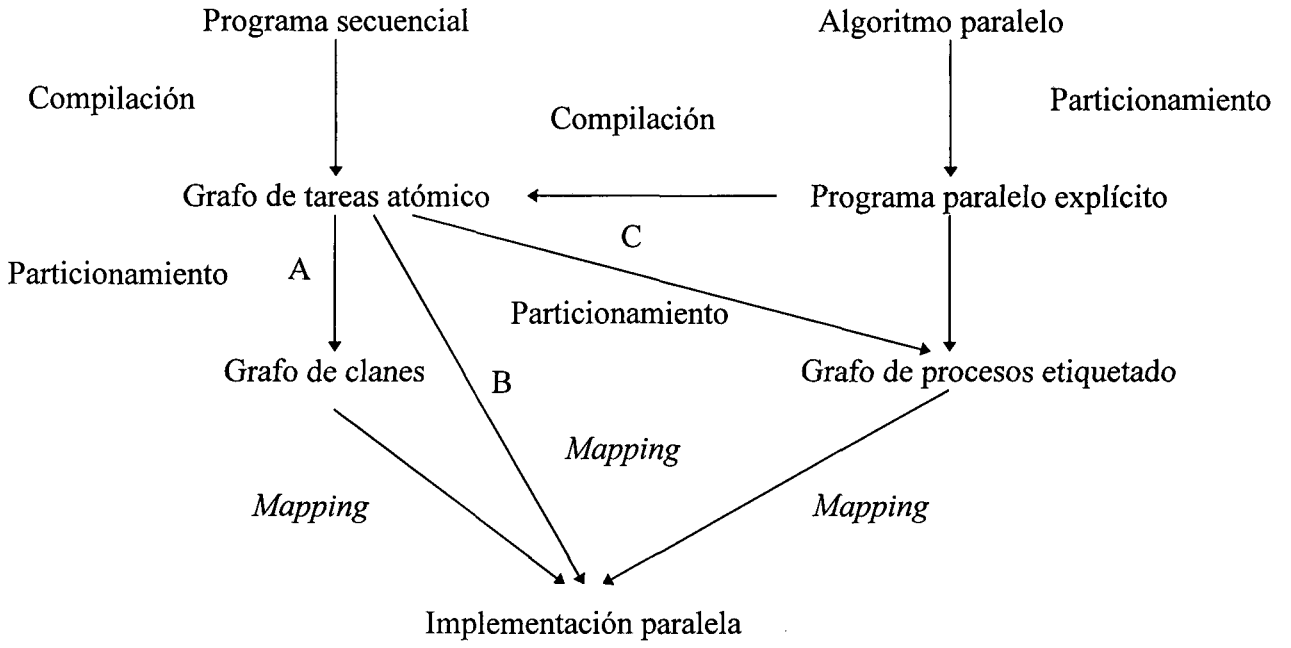


Figura 1.7 Relación particionamiento y mapping

1. La primera alternativa tiende a crear "granos" de mayor tamaño uniendo tareas atómicas y eliminado con ello sus comunicaciones internas (línea A en la figura 1.7). En este caso se restringe el problema del particionamiento a crear "clanes", o sea, subgrafos en los que todos sus nodos constituyentes compartan las mismas relaciones de ascendientes y descendientes[McG89]. El grafo resultante de la partición de un grafo dirigido acíclico es siempre un nuevo grafo dirigido acíclico. Dado que se puede suponer que los "clanes" empiezan su ejecución una vez hayan recibido todas las entradas de sus tareas atómicas constituyentes y que van a enviar sus salidas una vez hayan finalizado dichas tareas atómicas, a los grafos de "clanes" se les puede aplicar cualquiera de las técnicas de *scheduling* mencionadas al hablar de los modelos de DAGs.

2. La segunda alternativa consistiría en asignar directamente el grafo de tareas atómico [Kru88], de forma que se eviten algunos problemas debidos a que la optimalidad de las agrupaciones obtenidas por la fase de particionamiento sean dependientes del sistema paralelo (línea B en la figura 1.7). Además, si el particionamiento se restringe a la producción de grafos dirigidos acíclicos donde los granos sólo se comunican al final, esto puede llevar a una partición que no sea óptima. De hecho, parece tener poco sentido que las comunicaciones se limiten a producirse todas al final en un modo ráfaga, cuando las tareas atómicas han podido ir generando las mismas paulatinamente durante la ejecución del grano entero, con lo cual se estaría

generando un retraso innecesario en la activación de otras tareas que podrían pertenecer al camino crítico del programa.

3. La última alternativa consiste en permitir que los granos generados en la fase de particionamiento tengan comunicaciones a lo largo de su ejecución [Sar89][Yan94][Ger93] (línea C en la figura 1.7). Esto genera grafos de granos donde las relaciones de precedencia dirigidas pasan a ser volúmenes de comunicación no dirigidos y, por consiguiente, se obtiene un grafo de tipo TIG al que se le aplicarán las estrategias correspondientes.

Por su parte, la parte derecha de la figura 1.7 correspondería al problema del *mapping* desde el punto de vista del programador que expresa su solución como un conjunto de procesos secuenciales comunicantes mediante un lenguaje como Occam o C más unas extensiones de paso de mensajes (PVM o MPI). En este caso es el programador el que ha determinado el particionamiento; sin embargo, podemos pensar que lo que ha expresado el programador a este nivel es un grafo de procesos del que se puede extraer un grafo de tareas atómicas al que se le aplicaría alguna estrategia de las comentadas anteriormente.

Un ejemplo donde podría producirse una situación como la mencionada sería aquella donde el programador escribiese su aplicación siguiendo un formulismo de programación orientada a objetos. Los objetos son entidades modulares y el nivel natural de granularidad al que se haría la asignación sería el objeto o la clase de objetos. Sin embargo, si uno de los métodos asociados con uno de los objetos constituye la mayoría de cómputo del programa, sería ventajoso realizar una asignación a un nivel de granularidad más fino que el objeto para mejorar el rendimiento de la aplicación. Notemos en este caso que, aunque la flecha de la figura 1.7 nos lleva hacia el grafo de tareas atómico que, en un principio, habíamos dicho que se corresponde con un grafo dirigido acíclico, la estrategia de descomponer un cierto TIG en componentes más simples no tendría porqué restringirse a este caso y podría constituir un ejemplo de asignación multinivel de los que se comentó al hablar de los sistemas multiaplicaciones en la clasificación jerárquica de las estrategias de *mapping*.

De todo lo dicho anteriormente, y a raíz de lo que los propios autores indican en sus trabajos, se puede concluir que **asignación** y **partición** son dos problemas muy relacionados y que muchas veces la solución a uno de ellos también constituye una solución válida o, al menos aplicable, para el otro. Una estrategia de *mapping* puede ser sólo de *mapping* o puede verse como una estrategia de partición que reduce las tareas a una granularidad tal que sólo existen tantos clanes como procesadores. Esta ambivalencia es lo que ha llevado muchas veces a cierta confusión en la terminología utilizada en la literatura, que no hace más que reflejar la ambigüedad que presenta la distinción entre ambos problemas.

1.4 Estimación de los parámetros de los programas

Al hablar de los distintos modelos utilizados en la resolución del problema del *mapping* se ha mencionado que los grafos utilizados tienen sus nodos y sus arcos etiquetados o anotados con ciertos valores que representan, respectivamente, los volúmenes de cómputo y los volúmenes de comunicación de un cierto proceso del programa. Estos valores además se suponen conocidos en tiempo de compilación y exactos. Esta exactitud debe entenderse no tanto en un sentido absoluto por lo que se refiere a los valores que caracterizan a un nodo, sino en un sentido relativo que relaciona los valores de un cierto nodo con los de otro nodo. Es decir, esos valores miden la importancia relativa de las tareas del programa. Así, para una política de asignación no va a ser determinante el hecho de que dos nodos tengan cómputo 10 y 20, respectivamente. Lo importante será que la relación entre estos cómputos es de 1/2, ya sea que esta relación se consiga con valores de 10 y 20, o de 20 y 40. Son las proporciones relativas entre valores de cómputo y de comunicaciones las que van a determinar la agrupación o no de determinadas tareas más que el valor absoluto de los mismos.

A grandes rasgos, establecer el coste (volumen) de cómputo de un cierto proceso depende de varios factores entre los cuales, los más importantes son:

1. El tamaño de los datos de entrada. Si el cómputo se usa para ordenar una tabla de n elementos, el coste de tiempo del cómputo se incrementa a medida que el tamaño de la tabla crece.
2. El algoritmo utilizado en el cómputo. Para ordenar una tabla, se puede usar un algoritmo de ordenación como el de la burbuja o uno como el *quicksort*. El uso de distintos algoritmos proporcionará distintos costes temporales.
3. Las estructuras de datos usadas en el cómputo. Una cadena puede representarse como un vector o como una lista enlazada. Diferentes representaciones necesitan procesamientos distintos y, por lo tanto, tendrán un impacto final en el comportamiento del coste de tiempo del programa.

Por su parte, determinar el volumen de comunicación de un cierto nodo está condicionado de forma importante a conocer su volumen de cómputo. De hecho, todo lo que se requiere es saber cuántas veces va a comunicarse un cierto proceso y cuál es el tamaño de los mensajes en cada ocasión. El segundo valor puede resolverse satisfactoriamente adoptando primitivas de paso de mensajes donde se especifique explícitamente el "tipo" de datos que conforma el mensaje. El conocimiento del primer valor está fuertemente ligado al conocimiento del volumen de cómputo global del nodo.

En general, la mayoría de trabajos de *mapping* obvian el problema de obtención de los parámetros, ofreciendo tan sólo un conjunto de métodologías aplicables para calcular esos parámetros. El método más simple delega tal tarea en el propio usuario, que deberá aportar una estimación de los volúmenes de cómputo y comunicación de cada elemento del programa a partir de su conocimiento de la aplicación y el valor típico de los datos que se manejan. Otra posibilidad consiste en la realización de estimaciones basadas en simulación. En tiempo de compilación se simula el

comportamiento del programa para aquellas instrucciones que afectan al flujo de control del programa. A partir de estas simulaciones se obtienen resultados estadísticos que miden la frecuencia de ejecución del conjunto de instrucciones del programa. Un tercer método propuesto se basa en la estimación de tiempos a partir de la monitorización de ejecuciones previas del código. Por último, se proponen también estrategias analíticas que realizan las estimaciones a partir de la asignación de probabilidades de ejecución a las distintas instrucciones que conforman el programa.

En la práctica, sin embargo, muy pocos son los autores que hagan referencia explícita a alguna herramienta automática capaz de derivar dichos parámetros.

Un ejemplo lo constituiría la aproximación adoptada por Chu en [Chu84] que propone una metodología basada en el uso de grafos de flujo de control y flujo de datos para hacer una estimación de los costes de cómputo y los costes de comunicación entre distintos módulos. Aunque no lo menciona explícitamente en su artículo, su propuesta tiene muchos puntos de coincidencia con los métodos de microanálisis que trataremos a continuación y que han sido desarrollados por autores no interesados inicialmente en el problema del *mapping*. Otra propuesta formulada por autores que sí están interesados directamente en el problema del *mapping* es la de Antonelli en [Ant91] donde derivan los costes de comunicación y de cómputo con la ayuda de una heurística que considera que las sentencias serán ejecutadas con una frecuencia que va a depender de la profundidad de anidación dentro de los bucles en los que se encuentren.

El estudio de metodologías para determinar cotas para el tiempo de ejecución de los programas ha sido un problema abordado por autores interesados, en la mayoría de casos, en su uso con finalidad predictiva en sistemas de tiempo real. Los tipos de análisis realizados para determinar el coste de tiempo de ejecución de un determinado programa han seguido dos grandes metodologías: el macroanálisis y el microanálisis. El macroanálisis consiste en escoger una operación dominante y expresar el coste de tiempo como una función del número de veces que se va a realizar esta operación. Durante este análisis, se ignora el impacto de las otras operaciones en el coste de tiempo final. El análisis de complejidad es uno de los ejemplos típicos correspondientes a macroanálisis [Qui87]. Por su parte, el microanálisis consiste en expresar el coste de tiempo de un determinado cómputo como función del tiempo necesario para realizar cada una de las operaciones involucradas en ese cómputo. En el microanálisis todos los factores que afectan al tiempo de un cómputo son considerados. Existen distintos trabajos que han abordado el microanálisis para cómputos secuenciales y paralelos, llegando incluso a contemplar la influencia de mecanismos propios del paralelismo como el paso de mensajes o el uso de variables compartidas en sistemas de memoria común [Qin91][Avr94][Sha90].

Estos trabajos no tienen como objetivo la obtención de valores que sean directamente utilizables en los distintos modelos utilizados en las estrategias de *mapping*; en determinados casos, porque el interés se centra en la búsqueda de cotas de tiempo y los modelos de *mapping* requieren valores medios "típicos"; en otros casos, porque la derivación de las cotas tiene en cuenta muchos más factores de los que estrictamente se requieren para obtener los parámetros interesantes desde el punto de vista del *mapping*. En cualquier caso, el objetivo no puede ser nunca la obtención de

valores exactos sobre el tiempo de cómputo porque, lógicamente, existen problemas que lo impiden; por ejemplo, los relacionados con la estimación de tiempos en las sentencias condicionales o iterativas que, en general, no va a poder obtenerse de forma determinística. O los relativos a la dependencia de datos creada por el paso de mensajes que puede provocar que las variables que controlan el número de veces que se va a ejecutar una iteración, por ejemplo, sea un valor recibido desde otro proceso, con lo que se deberían realizar análisis interprocedurales adicionales.

De todos modos, la existencia de estas metodologías para predecir estáticamente el coste de tiempo de un cierto programa nos permite asumir en nuestro trabajo que los parámetros relativos a volumen de cómputo y de comunicación son conocidos y tienen una correspondencia aceptable con los valores reales típicos que exhibe una cierta aplicación paralela durante su ejecución. Esta suposición, sin embargo, no nos hace olvidar la dependencia de la bondad de dichos valores que van a tener los resultados de una cierta estrategia de *mapping* y que en la literatura no existe ningún estudio que haya evaluado la incidencia que pueden tener las variaciones en los valores usados por la estrategia de *mapping* frente a los valores presentes durante la ejecución real, y que permitiese acotar la necesidad de disponer de datos de una determinada precisión.

1.5 Herramientas de *mapping*

Los trabajos sobre *mapping* no siempre han terminado en el desarrollo de herramientas integradas en lo que denominaríamos entornos de programación donde se solucionase dicho problema automáticamente. Realmente, la mayoría han sido trabajos básicamente teóricos cuya aplicación práctica ha sido restringido, en el mejor de los casos, a un, relativamente pequeño, número de aplicaciones reales. Sin embargo, en los últimos años han ido apareciendo todo un conjunto de herramientas relacionadas con este problema que han integrado algoritmos de la literatura y se ofrecen como herramientas (o entornos) de *mapping*. Nos referimos con este tipo de herramientas a todos aquellos entornos de desarrollo de programas paralelos que ayudan a los usuarios a partir y asignar sus programas sobre un determinado sistema paralelo. Estas herramientas pretenden ser amigables para el usuario y suelen, por ello, tener un alto componente visual. Todas ellas consisten en herramientas de tipo CASE que, en los casos más simples permiten al usuario experimentar con distintas políticas de *mapping* y observar sus resultados en una hipotética ejecución de un determinado programa, y en los casos más complejos llegan a la generación de código ejecutable para un cierto sistema paralelo.

1.5.1 Componentes habituales de las herramientas de *mapping*

A continuación describiremos los diferentes tipos de herramienta que suelen incluirse en los entonos de *mapping*:

A - Herramienta para la especificación de aplicaciones

Esta herramienta se utiliza para generar el equivalente gráfico de una aplicación especificada mediante un cierto lenguaje de programación. Este formato gráfico, ya sea un DAG o un TIG, se usa posteriormente para realizar las operaciones de partición y/o asignación. Existen distintas formas en la forma de especificar los grafos de entrada:

- Algunas herramientas, por ejemplo, PARSA [Shi94], usan un lenguaje funcional como SISAL para la codificación del programa. Esto permite utilizar compiladores ya existentes que pueden trasladar estos programas a un formato gráfico. En el caso de SISAL los programas se traducen al lenguaje intermedio IF1, que es el formato *dataflow* equivalente al código de entrada, a partir del cual se extrae fácilmente el DAG asociado.

- Algunas herramientas, como Prep-P [Ber87] y OREGAMI [Lo91] usan código especializado o extensiones de lenguajes existentes que le permiten al programador especificar la estructura del programa de entrada.

- Por último, muchas herramientas requieren simplemente que el usuario le proporcione el grafo equivalente del programa como entrada [Lew93].

B - Herramienta de particionamiento

Esta herramienta se usa para reducir las comunicaciones entre procesos al tiempo que se intenta mantener el paralelismo útil que asegure un buen rendimiento. Algunos entornos usan una herramienta separada de particionamiento, como paso previo a la asignación, y otras combinan los pasos de partición y *scheduling* en una misma herramienta [Shi94] [Yan92].

C - Herramienta de mapping

Esta herramienta realiza la asignación inicial del programa sobre la arquitectura escogida. Algunas herramientas usan un método específico de asignación mientras que otras incorporan un conjunto de estrategias distintas dando opción al usuario de experimentar con distintas alternativas [Ber87][Lo91][Shi94][Yan92][Lew93].

D - Herramienta de evaluación de rendimiento

La mayoría de herramientas de *mapping* disponen de un elemento que muestra visualmente el rendimiento esperado del programa asignado. Las métricas que suelen proporcionarse son la utilización de los procesadores, el tiempo de finalización (*make-span*) del programa y factores de *speed-up*. Muchas herramientas incluyen también ventanas de animación donde se muestra cuando los procesadores están ocupados o cuándo están teniendo lugar comunicaciones durante la ejecución [Lo91][Lew93][Shi94].

E - Otras herramientas

Las herramientas indicadas anteriormente son las que suelen disponer la mayoría de entornos a los que estamos haciendo referencia. Sin embargo, existen otro tipo de herramientas adicionales que no son tan comunes. Por ejemplo, herramientas para especificar la arquitectura [Pea91], la distribución de datos [Bop91], generación total o parcial de código ejecutable para un sistema concreto [Yan92], y herramientas de depuración [App89].

1.5.2 Ejemplos de herramientas de mapping

A continuación, describiremos las características principales de algunas de las herramientas de *mapping* más relevantes. Para elaborar la lista que figura a continuación nos hemos basado exclusivamente en aquellas herramientas que aplican estrategias de asignación contrastadas en la literatura que obedecen a criterios de mejora en el tiempo de ejecución de las aplicaciones, utilización de los procesadores, equilibrio de carga, reducción de comunicaciones, etc. Sirva esta puntualización para descartar de esta enumeración todos aquellas herramientas de programación que ofrecen un entorno amigable para el desarrollo de aplicaciones paralelas y que incorporan muchos de lo elementos que hemos comentado anteriormente, pero que basan la resolución del *mapping* no en métodos automatizados sino en una intervención directa por parte del usuario que debe especificar manualmente, en la mayoría de casos, los pares proceso-procesador.

- PARSA [Shi94]: Este entorno está orientado a la resolución de los problemas de partición y *scheduling* de programas paralelos en sistemas multiprocesador de memoria distribuida. Permite la especificación de la arquitectura del multiprocesador y la selección de varias estrategias de partición y asignación. Los programas paralelos se especifican mediante el lenguaje SISAL. Además, la herramienta proporciona información sobre el rendimiento obtenido por la aplicación concreta, lo que permite al usuario adaptar o modificar su programa de entrada intentando una sintonización fina con la arquitectura.

- Parallax [Lew93]: Como en el caso anterior, esta herramienta se basa en programas modelados como DAGs, aunque en este caso el usuario debe hacer la especificación directamente como una grafo de tareas, junto con la estimación de los volúmenes de cómputo y de comunicación. También proporciona varias estrategias de asignación y arquitecturas entre las cuales el usuario puede escoger. Los resultados de una asignación se muestran mediante diversos diagramas que reflejan el uso de los procesadores, la ocurrencia de comunicaciones, etc., llegándose a ofrecer una pantalla animada donde se muestra una ejecución simulada del programa.

- Parafrase-2 [Pol89]: Esta herramienta es un compilador vectorizante/paralelizante implementado como una herramienta de reestructuración de código fuente a código fuente (usando como lenguajes C, Pascal y FORTRAN). Permite la interacción del usuario en distintos puntos del proceso de compilación de forma que, por ejemplo, se le pueda

proporcionar al compilador información adicional sobre el programa que permita aumentar el grado de paralelización obtenible. La fase de partición se realiza en tiempo de compilación y el *scheduling* de las tareas se deja para que sea realizado dinámicamente en tiempo de ejecución.

- Prep-P [Ber94]: Esta herramienta está orientada a la resolución de todas las fases involucradas en el *mapping* a partir de un algoritmo de entrada que es un grafo no dirigido. Cada nodo del grafo describe un cierto cómputo y está escrito como proceso en el lenguaje denominado XX. La salida de Prep-P es código máquina ejecutable en el sistema paralelo CHiP. La herramienta ofrece distintas estrategias de asignación y se pueden especificar arquitecturas de distintos tamaños siempre basadas en el sistema CHiP.

- OREGAMI [Lo91]: Este entorno lo constituyen una serie de herramientas que incluyen un compilador de descripciones de grafos especificadas en el lenguaje LaRCS que permiten describir las aplicaciones paralelas en un modelo denominado grafo de comunicaciones temporal (que intenta ser una fusión entre un modelo TIG y un grafo temporal de Lamport [Lo92]). Junto con el compilador existe la herramienta MAPPER que incorpora toda una serie de bibliotecas de para la asignación de forma óptima de determinadas familias de grafos sobre ciertas arquitecturas junto con estrategias más generales que permiten la asignación de grafos para los que no se posea una asignación de tipo óptimo. Por último, el entorno se completa con la herramienta METRICS diseñada para visualizar y analizar el rendimiento conseguido con una cierta asignación.

Además de las herramientas mencionadas anteriormente, cuyo diseño ha sido orientado especialmente al estudio y resolución del problema del *mapping*, existen también otras herramientas orientadas a otros propósitos pero que incorporan alguna estrategia de asignación automática. Un ejemplo de tal herramienta sería el entorno de simulación y evaluación de arquitecturas paralelas descrito en [Luq96] y [Sup96], que incorpora estrategias de asignación basadas en lista para DAGs o estrategias para TIGs.

1.6 Objetivos y desarrollo del trabajo

El objetivo del presente trabajo se centra en el estudio, diseño y evaluación de estrategias estáticas de asignación que permitan resolver el problema del *mapping* del tipo N a 1 manteniendo un compromiso entre la calidad de las soluciones encontradas y el tiempo de cómputo necesario para su obtención. Tales estrategias serán de "propósito general", entendiendo por tal definición que no se restringe a ningún tipo particular de aplicación paralela o arquitectura.

Las principales suposiciones básicas en las que se basa nuestro trabajo son:

- El sistema paralelo se encuentra dedicado a la ejecución de una única aplicación y el objetivo de la asignación pretende minimizar su tiempo de ejecución.

- El modelo de TIGs es el adoptado para caracterizar las aplicaciones paralelas. Ello implica que una aplicación está representada por un grafo no dirigido, donde los nodos corresponden a las tareas del programa paralelo y los arcos representan las comunicaciones existentes entre ellas. El grafo es estático y conocido en tiempo de compilación y durante la ejecución no se crean nuevas tareas. Los nodos y los arcos tienen asociados unos pesos que representan, respectivamente, el volumen de cómputo y de comunicación en que incurre una cierta tarea a lo largo de su ejecución. Cada arco y cada nodo tiene un peso asociado que es único, invariable y conocido (o estimado) en tiempo de compilación. Distintas tareas o arcos pueden tener, pesos distintos.

- El sistema es homogéneo, es decir , todos los procesadores tienen las mismas capacidades de cómputo y comunicación.

- La asignación de las tareas a procesadores es estática, es decir, se calculará antes de la ejecución de la aplicación.

En sucesivos capítulos se analizarán en detalle las implicaciones y posibles restricciones que ofrecen los modelos concretos adoptados en este trabajo, contrastándolos en todo momento con las soluciones existentes en la literatura.

Las estrategias propuesta se formularán, en primera instancia, como estrategias orientadas a arquitecturas totalmente interconectadas (*full-connected*). Se validará su eficiencia frente a un método óptimo para ejemplos de tamaño reducido y, posteriormente, serán comparadas con otras estrategias representativas utilizando ejemplos de mayores dimensiones.

Comprobada la bondad de los métodos propuestos, se incluirá y evaluará un conjunto de extensiones que permitirán adaptarlos a su uso para arquitecturas con topología punto-a-punto (no *full-connected*). En este caso se aplicarán técnicas de resolución del problema del *mapping* en su versión 1 a 1. También se formulará, y se validará experimentalmente, un modelo analítico capaz de acotar el coste adicional que va a suponer la consideración de una arquitectura frente al primer supuesto de la arquitectura *full-connected.*

Por último, se presentarán los resultados de un estudio encaminado a determinar el grado de correlación existente entre los valores de la función de coste obtenido en la asignación de un grafo y el tiempo de ejecución del correspondiente programa paralelo.

CAPÍTULO 2

Estrategias clásicas de asignación de tareas

En el capítulo precedente se ha dado una visión global del problema general del *mapping* y se ha visto que los trabajos existentes suelen caracterizarse por dos factores:

- Los modelos subyacentes usados para abstraer los parámetros correspondientes al programa y al sistema paralelo.
- El tipo de estrategia algorítmica utilizada en la resolución del problema.

En este capítulo vamos a dar una visión más detallada de aquellas estrategias más relevantes que existen en la literatura propuestas para resolver el problema de la asignación cuando se adopta el modelo basado en procesos estáticos (o TIGs: *Task Interaction Graph*).

En primer lugar, se planteará formalmente el problema del *mapping* cuando se adopta el modelo de TIG. A continuación, se analizarán las implicaciones que supone el modelo TIG cuando se usa para representar aplicaciones paralelas. Después se dará una visión amplia de los trabajos que existen en la literatura basados en este modelo.

Por último, después de analizar y clasificar las distintas soluciones al problema del *mapping* basadas en el modelo TIG se dedicará un punto final al análisis de las principales funciones de coste que han sido utilizadas como función objetivo.

2.1 Una definición formal del problema de la asignación

En primer lugar veamos una definición formal del problema de la asignación desde el punto de vista del modelo adoptado. Es necesario precisar, sin embargo, que las distintas soluciones que aparecen en la literatura que usan este mismo modelo no siempre se han ajustado estrictamente a esta definición y han usado versiones distintas que, normalmente, eran más reducidas. En cualquier caso, la definición que sigue puede considerarse como una de las más generales y comúnmente aceptadas.

El problema del *mapping* puede definirse como una 5-tupla $(\Delta, \Omega, \varphi, \theta, \delta)$, donde

1. $\Delta = (P, L)$ es un grafo no dirigido, donde P es un conjunto de K procesadores y L es un conjunto R arcos no dirigidos que corresponden a los enlaces entre los procesadores.

2. $\Omega = (T, V)$ es un grafo no dirigido, donde T es un conjunto de N tareas y V es un conjunto de M arcos no dirigidos correspondientes a la comunicación entre las tareas.

3. φ: $K \times Q \rightarrow Z_0$ es una función tal que $\varphi(t_i, p_j)$ devuelve el tiempo requerido para calcular la tarea t_i en el procesador p_j. En el caso de sistemas de procesadores homogéneos, esta función se reduce a φ: $K \rightarrow Z_0$, siendo $\varphi(t_i)$ el coste de cómputo asociado a la tarea t_i. Este valor también suele denominarse peso de la tarea t_i y se representará abreviadamente por w_i.

4. θ: $V \rightarrow Z_0$ es una función que devuelve el coste asociado en la comunicación entre los procesos si éstos son asignados en procesadores distintos. Cuando dos procesos se asignan al mismo procesador se supone un coste 0 debido a comunicación. Para cada elemento $v_{i,j} \in V$, $\theta(v_{i,j})$ devuelve el coste asociado a la comunicación entre el nodo t_i y el nodo t_j. Este valor se representará abreviadamente como $c_{i,j}$.[1]

5. δ: $P \times P \rightarrow Z_0$ es una función de coste que para $\delta(p_i,p_j)$ devuelve el coste asociado a enviar un mensaje entre los procesadores p_i y p_j. Para ello se define un camino R entre p_i y p_j como un conjunto de vértices

   $$\{r_1, r_2, ..., r_z\} \subseteq P - \{p_i, p_j\} \text{ tales que}$$

   $$\{p_i, r_1\}, \{r_1, r_2\}, ..., \{r_{z-1}, r_z\}, \{r_z, p_j\} \in L$$

   La longitud de este camino entre p_i y p_j es z y $d(p_i,p_j)$ se define como la longitud del camino más corto entre p_i y p_j en Q. Abreviadamente, la distancia entre los procesadores p_i y p_j se denotará $d_{i,j}$.

[1] Para evitar posibles problemas tipográficos en el manejo de subíndices, el arco entre los nodos t_i y t_j se designará tanto $v_{i,j}$ como (t_i,t_j).

Al formular un cierto problema de *mapping*, podemos considerar el conjunto F de funciones que asignan los elementos de T en P. Este conjunto puede pensarse como el conjunto de posibles funciones de asignación de tareas. Para cada $f \in F$ podemos definir una función asociada

$$g : P \rightarrow 2^T$$

que devuelve el conjunto de tareas asignadas a un cierto procesador de acuerdo con la función de asignación f.

A partir de estas definiciones, cualquier estrategia de asignación formulará una función de coste a optimizar, convirtiendo así el problema del *mapping* en un problema de optimización de una cierta función de coste.

En general, todas las estrategias que tratan de resolver el problema del *mapping* quieren mejorar el rendimiento del sistema paralelo de cómputo y, para ello, suelen plantearse dos objetivos genéricos:

1. minimizar la comunicación interprocesador porque esto va a ser fuente de retrasos en la ejecución del programa.
2. distribuir los costes de ejecución entre los distintos procesadores de forma que se reduzca el tiempo global de ejecución.

Estos dos objetivos pueden parecer contradictorios. Por una parte, tener todas las tareas en un procesador eliminará todos los posibles costes debidos a las comunicaciones, pero va a provocar una pobre utilización de los recursos del sistema y una pobre distribución del cómputo entre los procesadores. Por otro lado, una distribución equitativa de las tareas entre los procesadores maximizará el uso de éstos pero también va a incrementar los retrasos debidos a comunicaciones. Planteado de esta forma, el problema puede verse como un problema de tipo max-min: maximizar el uso de los procesadores al tiempo que se minimizan las comunicaciones, con lo que se conseguirá minimizar el tiempo de ejecución global.

Las estrategias que han abordado el problema han seguido las tres opciones que ya se mencionaron al comentar la taxonomía de Casavant, o sea, métodos basados en teoría de grafos [Sto77], en programación matemática [Chu80][Ma82] y métodos heurísticos. Los primeros aplican el algoritmo de *max-flow min-cut* sobre un grafo que representa el problema de obtener una asignación con mínima intercomunicación entre procesos. En la aproximación basada en programación matemática, el problema se formula como un problema de optimización para el que se aplican técnicas de programación matemática, aunque ello lleva a la imposición de una serie de supuestos que limitan su efectividad.

Aunque la mayoría de soluciones propuestas buscan la obtención de asignaciones cercanas al óptimo existen algunos trabajos que han hallado asignaciones óptimas para ciertas formulaciones del problema. En los siguientes apartados vamos a cubrir con cierta extensión cuáles han sido los resultados más significativos que

aparecen en la literatura tanto en las estrategias que buscan asignaciones óptimas como aquellas estrategias heurísticas que buscan soluciones cercanas al óptimo.

2.2 El modelo de grafo de programa TIG

En la definición formal del problema del *mapping* se han detallado los elementos externos que caracterizan a los grafos TIG, elementos que serán los que tendrán en cuenta las estrategias para calcular las posibles asignaciones. Junto con esa caracterización externa, la adopción del modelo de TIG implica también un conjunto de suposiciones referidas al comportamiento de los módulos que componen el programa paralelo. Las características más importantes subyacentes en el modelo TIG son las siguientes:

- Todas las tareas que componen el programa paralelo son persistentes en el tiempo, es decir, existen a lo largo de toda la ejecución del programa. Se supone también que todas se activan al inicio de la ejecución del programa.
- El número de tareas es fijo y conocido en tiempo de compilación. No hay, por consiguiente creación dinámica de procesos.
- El volumen de cómputo (o peso), w_i, de cada nodo del grafo TIG constituye una estimación de la cantidad total de instrucciones que típicamente ejecuta cada tarea a lo largo de su ejecución.
- Las operaciones de comunicación y sincronización entre las distintas tareas pueden ocurrir en cualquier momento durante la ejecución de la tarea. No es necesario que tales operaciones ocurran sólo al inicio y final de las tareas como ocurre en el modelo de DAGs. Es decir, las tareas pueden enviar y recibir mensajes en cualquier instante.
- Los arcos del grafo TIG corresponden a las operaciones de comunicación y sincronización entre las distintas tareas. El volumen de comunicación, $c_{i,j}$, asociado a cada arco mide el total de información enviado entre una pareja de tareas, tanto en un sentido como en otro.
- Se supone que las relaciones de precedencia son prácticamente despreciables entre cualquier pareja de tareas comunicantes. Es decir, todas las tareas se van ejecutando concurrentemente y en el momento de realizar un comunicación ni el emisor ni el receptor van a verse bloqueados durante un intervalo de tiempo considerable esperando la ocurrencia de esa comunicación.

A modo de ejemplo, mostraremos a continuación un algoritmo paralelo y la obtención del TIG correspondiente. Ya hemos comentado que este modelo se considera que es especialmente apropiado para modelar aplicaciones escritas en lenguajes clásicos de programación paralelo como Occam o Parallel C. En la figura 2.1 se muestra el código en pseudo-Occam de un proceso que efectúa la fusión de dos secuencias ordenadas de números. Las dos secuencias las irá recibiendo, respectivamente por los canales *leftin* y *rightin*. A partir de las dos de entrada, el proceso generará como resultado una secuencia ordenada de salida que irá enviando por el canal *out*. Los valores serán transmitidos de uno en uno. El final de la secuencia de números se indica mediante el valor especial EOS.

```
PROC Merge (leftin, rightin, out)
VAL INT EOS is -999
        INT v1, v2
        BOOL finish

        leftin ? v1
        rightin ? v2
        WHILE NOT finish
                IF (v1 ≠ EOS) AND (v2 ≠ EOS)
                        IF  (v1 ≤ v2)
                                out ! v1
                                leftin ? v2
                        | (v1 > v2)
                                out ! v2
                                rightin ? v2
                | (v1 ≠ EOS) AND (v2 = EOS)
                        out ! v1
                        leftin ? v1
                | (v1 = EOS) AND (v2 ≠ EOS)
                        out ! v2
                        rightin ? v2
                | (v1 = EOS) AND (v2 = EOS)
                        finish := TRUE
        out ! EOS
```

Figura 2.1 Pseudo-código de un proceso de fusión

Usando este proceso básico se puede crear un árbol de procesos de fusión [And91], conectando el canal *out* de un proceso con el canal *leftin* o *rightin* de un proceso de un nivel inferior. Ese efecto se conseguiría, por ejemplo, mediante el código mostrado en la figura 2.2. Gráficamente, la interconexión de los procesos es la que refleja la figura 2.3, donde c_i corresponde con el nombre de los distintos canales. El grafo de la figura 2.3 refleja, en definitiva, el TIG equivalente al árbol de fusión. Para completar la descripción del TIG se han incluido unos hipotéticos volúmenes de cómputo y comunicación (valores entre paréntesis) calculados suponiendo que los nodos del primer nivel del árbol reciben dos secuencias de 5 elementos y que el tiempo de cómputo para procesar cada elemento de salida es de una unidad. Por lo tanto, los nodos del primer nivel tendrán un volumen de cómputo de 10 unidades, los del segundo nivel, de 20 unidades y el del último, de 40 unidades. Los volúmenes de comunicación son en cada arco proporcionales al tamaño de la secuencia que se transmitirá entre la pareja de procesos del árbol. Lógicamente, el TIG de la figura debería completarse con nodos terminales que, por un lado, generasen las distintas secuencias de números y, por otro, recogiesen la secuencia resultante.

```
PROC main ()
        SEQ
        [16] CHAN OF INT c:
        PAR i = 1 FOR 7
                Merge (c[2*i], c[(2*i)+1], c[i])
```

Figura 2.2 Construcción del árbol de procesos de fusión

Si suponemos que las secuencias de números de entrada no siguen ningún patrón especial, el comportamiento del programa durante su ejecución va a semejarse al de un grupo de procesos conectados en forma de *pipeline*, de forma que todos los procesos estarán actuando sobre distintos datos que irán avanzando paulatinamente por los distintos niveles del árbol. En este sentido, serían correctas todas las suposiciones subyacentes al modelo de TIG que se han enunciado anteriormente.

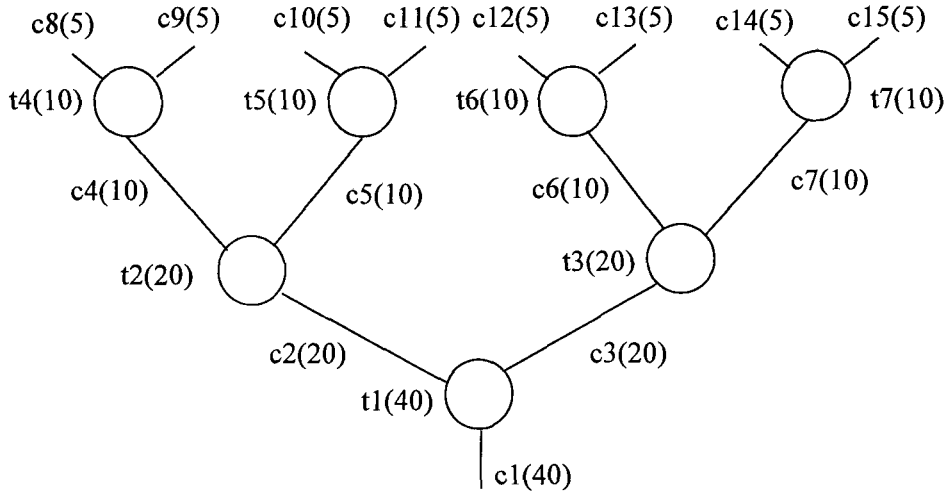


Figura 2.3 Árbol de procesos de fusión

2.3 Algoritmos óptimos

Tal y como se ha formulado el problema del *mapping* en el apartado anterior, existen dos criterios que son conflictivos entre sí: reducir el coste de las comunicaciones y reducir el coste de cómputo. Esta situación hace improbable que se pueda encontrar un algoritmo que encuentre la solución óptima para un caso general. Sin embargo, si se incluyen ciertas restricciones en el problema inicial, sí pueden encontrarse tales algoritmos. Existen dos casos principales en los que se pueden encontrar esas soluciones óptimas de forma eficiente: cuando el sistema está compuesto simplemente por dos procesadores [Sto77] y cuando el sistema se compone de un array lineal de cualquier número de procesadores [Lee92].

Asignación óptima en sistemas de dos procesadores

En este caso la función de coste que se plantea minimizar es la siguiente:

El coste de cómputo asociado a la función de *mapping* f viene dado por U_f

$$_f = \sum_{t_i \in T} \varphi\left(t_i, f(t_i)\right)$$

y el coste global de comunicaciones asociado a f es

$$C_f = \sum_{v_{i,j} \in V \mid g(t_i) \neq g(t_j)} c_{i,j}$$

El coste total de la asignación (función que quiere minimizarse) vendrá dado por

$$R_f = U_f + C_f$$

La solución al problema de asignación se obtiene construyendo, en primer lugar, un grafo con dos nodos terminales a partir del grafo original del programa de la siguiente forma:

1. Se añaden dos nodos etiquetados como S_1 y S_2 que representan a los procesadores p_1 y p_2, respectivamente. S_1 es el nodo fuente y S_2 el nodo destino del grafo visto como una red de dos terminales.
2. Para cada nodo t distinto a S_1 y S_2 añadir un arco entre t y S_1 y S_2. El peso del arco (t, S_1) corresponde al tiempo de ejecución de la tarea en el procesador p_2, mientras que el peso del arco (t, S_2) corresponde al tiempo equivalente en el caso de p_1.

La figura 2.4. muestra un ejemplo de grafo correspondiente a un programa y la tabla 2.1 contiene los tiempos de ejecución de cada una de las tareas sobre los dos procesadores.

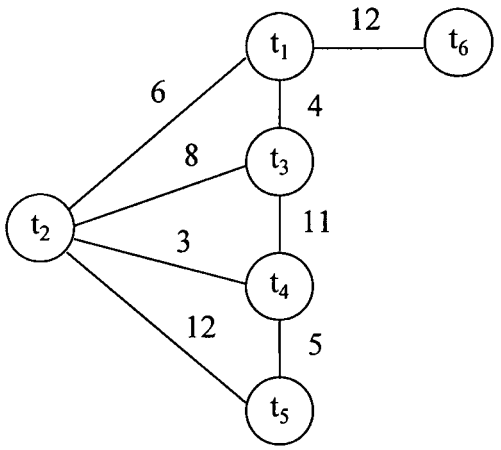


Figura 2.4 Ejemplo de TIG

Tabla 2.1 Tiempos de ejecución del ejemplo de la figura 2.4		
Tarea	p_1	p_2
t_1	5	10
t_2	2	∞
t_3	4	4
t_4	6	3
t_5	5	2
t_6	∞	4

La figura 2.5. muestra el grafo extendido construido según los criterios anteriormente mencionados.

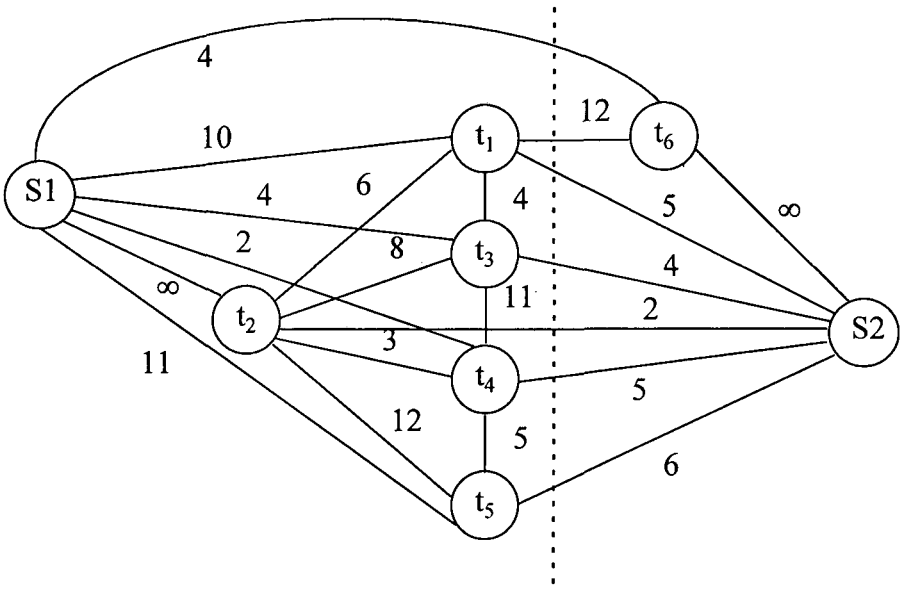


Figura 2.5 Grafo extendido correspondiente al grafo de la figura 2.4

A partir del nuevo grafo construido y utilizando un algoritmo de máximo flujo con mínimo corte (*max-flow min-cut*) se va a obtener la red dividida en dos mitades de corte mínimo. La solución óptima al problema de asignación a partir de dicho corte se obtiene si una tarea t se asigna al procesador p_i sí y sólo sí los correspondientes nodos t y S_i pertenecen al mismo corte obtenido en el grafo.

En el ejemplo que muestra la figura 2.5 el corte óptimo se obtiene al asignar las tareas t_1, t_2, t_3, t_4 y t_5 al procesador p_1, y la tarea t_6 al procesador p_2. En este caso, el coste de ejecución es igual a 5+2+4+6+5+4 = 26, y el coste de comunicación es igual a 12 (comunicación entre las tareas t_1 y t_6).

Asignación óptima en un array lineal de procesadores

Un array lineal de procesadores se compone de K procesadores p_1, p_2,..., p_K, y (K-1) enlaces de comunicación (p_h, p_{h+1}), $1 \leq h \leq K-1$. En este caso la definición de distancia entre dos procesadores p_r y p_s es $d_{r,s} = | s - r |$. Cuando dos procesadores no adyacentes p_r y p_s, $r < s -1$, se comunican entre sí, los procesadores intermedios, p_i's (r

$< i < s$), deben participar en la comunicación. Por consiguiente, coste de comunicación interprocesador por unidad de información transferida entre dos procesadores p_r y p_s se incrementa linealmente a medida que la distancia $d_{r,s}$ se incrementa.

El programa paralelo viene caracterizado de la misma forma que hemos visto en el punto anterior y la función de coste a optimizar se plantearía en los mismos términos, teniendo en cuenta, sin embargo, que en este caso el coste de comunicaciones para dos tareas t_i y t_j asignadas a procesadores distintos p_r y p_s se obtendrá multiplicando $c_{i,j}*d_{r,s}$.

La solución como en el caso anterior vuelve a pasar por modificar el grafo original del programa de forma que pueda aplicarse de nuevo el algoritmo de *max-flow min-cut*. El grafo con los dos nodos terminales se construye como sigue:

1. Para cada nodo t_i en el TIG se crean (K-1) nodos etiquetados como $u_{i,1}$, $u_{i,2}$,...$u_{i,K-1}$ respectivamente, y se añade un nodo fuente u_s y un nodo destino u_T (que para cada t_i también se les denomina $u_{i,0}$ y $u_{i,n}$).
2. Entre cada pareja de nodos $u_{i,j}$ y $u_{i,j+1}$ se añade un arco con un peso igual a la suma de $x_{i,j+1}$ y α, donde $x_{i,j+1}$ corresponde al coste de ejecución de t_i en p_{j+1} y α es un valor constante que debe ser mayor que la suma de todos costes de comunicación entre nodos del programa multiplicada por (K-1).
3. Entre cualquier par de nodos $u_{i,h}$ y $v_{j,h}$ se añade un arco con peso $c_{i,j}$ que corresponde a la coste de comunicación entre ambos procesos.

Por ejemplo, la figura 2.6. muestra un TIG compuesto por 6 nodos y en la tabla 2.2. se reflejan los costes de cómputo correspondientes a la ejecución de cada tarea sobre un sistema de cuatro procesadores. El grafo extendido resultante para este ejemplo (tomando $\alpha = 200$) es el que muestra la figura 2.7. Ahora un corte C del grafo extendido consiste en el conjunto de arcos que parten el conjunto de nodos U en dos conjuntos U_S y U_T de tal forma que $U_S \cap U_T = \varnothing$ y $U_S \cup U_T = U$, y tal que $u_s \in U_S$ y $u_T \in U_T$. Los conjuntos U_S y U_T se denominan conjunto fuente y conjunto destino, respectivamente.

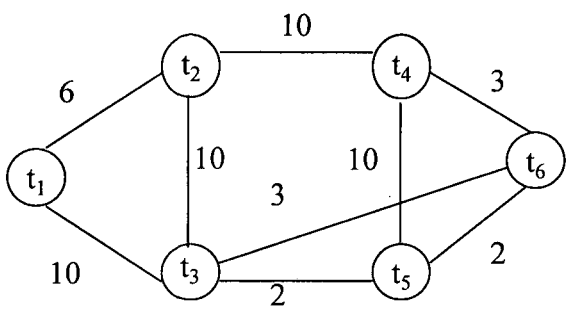


Figura 2.6 Ejemplo grafo TIG asignado a un array de procesadores

Tabla 2.3 Tiempos de ejecución del ejemplo de la figura 2.6				
Tarea	p_1	p_2	p_2	p_2
t_1	5	10	∞	7
t_2	2	∞	7	10
t_3	4	4	∞	∞
t_4	6	3	∞	10
t_5	5	2	∞	10
t_6	∞	4	15	4

Ahora cada corte C del grafo extendido corresponde a una asignación de las tareas del programa a los procesadores del sistema, y un corte mínimo del dicho grafo corresponde a la asignación óptima. Una tarea t_i se asigna al procesador p_{k+1} sí y sólo sí el corte C contiene el arco $(u_{i,k}, u_{i,k+1})$. En el ejemplo de la figura 2.7. la asignación resultante es la siguiente: tareas t_1 y t_3 asignadas al procesador p_2, tarea t_2 asignada al procesador p_3, tareas t_4 y t_5 asignadas al procesador p_4 y tarea t_6 asignada al procesador p_1.

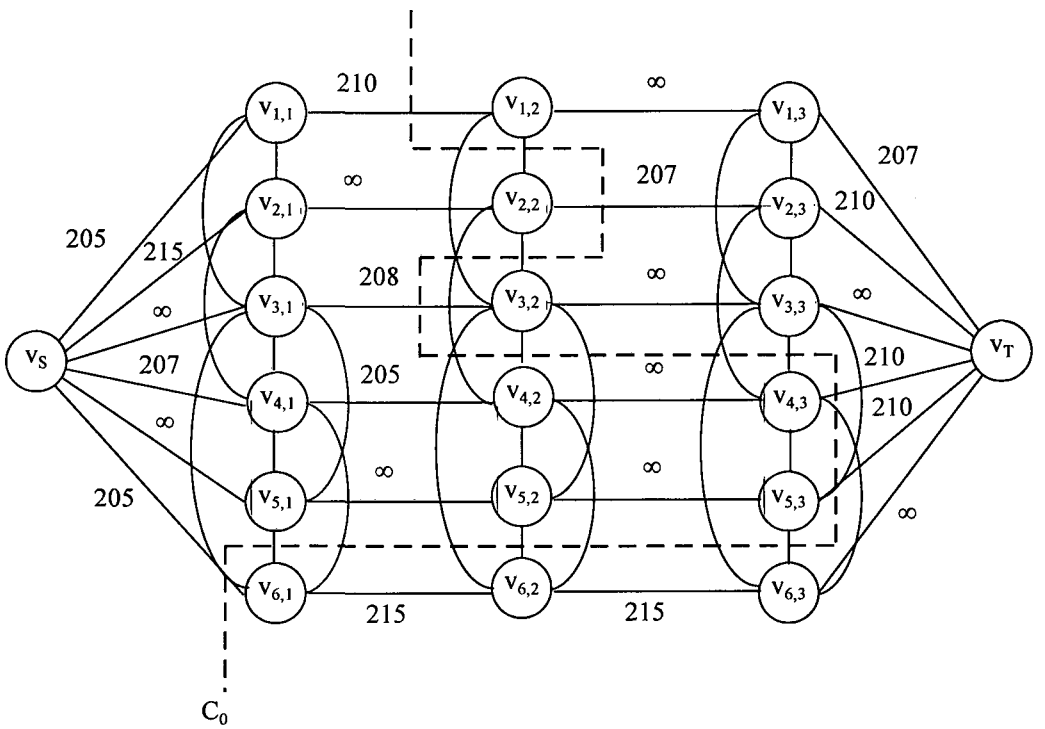


Figura 2.7 Grafo extendido y con el corte que determina la asignación

Estrategias de complejidad exponencial

Los dos métodos vistos anteriormente son los únicos casos en los cuales es posible encontrar una solución óptima al problema del *mapping* teniendo en cuenta además la restricciones impuestas en los modelos de arquitectura (sistema heterogéneo) y de función de coste (suma total de volúmenes de cómputo y comunicaciones). Tan sólo existe algunos casos donde se consiguen asignaciones óptimas aunque para ello se deban imponer fuertes restricciones. Por ejemplo en la propuesta de [Lo88b] se consigue una asignación óptima mediante un algoritmo de contracción de tiempo

polinomial siempre que el número de tareas sea inferior a dos veces el número de procesadores, que los volúmenes de cómputo sean iguales y que existan como máximo dos tareas en cada procesador.

Para los planteamientos generales del problema no se conoce solución óptima en tiempo polinomial y son problemas que se engloban en la categoría de los NP-completos. En estos casos se puede representar en forma de árbol la construcción de todas las soluciones posibles. Los algoritmos sistemáticos que construyan o recorran dicho árbol van a encontrar la solución óptima a costa de requerir un tiempo exponencial en su localización. De hecho, los algoritmos que adoptan esta opción parten de una situación donde no hay ninguna tarea asignada (raíz del árbol) y van construyendo paulatinamente al árbol hasta encontrar una solución con todas las tareas asignadas (hojas del árbol). El árbol puede construirse (recorrerse) de varias formas:

1. En profundidad prioritaria intentando construir lo más rápidamente posible una solución.
2. Desarrollando aquel nodo que tiene el valor más pequeño a partir de una cierta función de coste y de las tareas ya asignadas hasta el momento.
3. Desarrollando primero aquel nodo que tenga la esperanza mayor de ser la base de una solución óptima, teniendo en cuenta las tareas ya asignadas y utilizando alguna heurística que subestime las tareas que restan por asignar. Este es el procedimiento seguido por el algoritmo A* utilizado en muchas aplicaciones de inteligencia artificial. La heurística de subestimación debe dar valores que se aproximen lo máximo posible a la función de coste utilizada para expandir el árbol de forma que lleve rápidamente a una solución óptima [She85] [Bou95]. Desgraciadamente, en la mayoría de casos tal heurística es difícil de encontrar por lo que el proceso de búsqueda no deja de tener una complejidad exponencial.
4. Utilizando algún método de enumeración de las soluciones que vaya podando el árbol a medida que se encuentran soluciones parciales peores a la mejor encontrada hasta el momento. Esta metodología correspondería a los algoritmos de optimización de tipo *Branch-and-Bound* [Pap82][Gir88].

Cualquier algoritmo basado en el recorrido sistemático del árbol de soluciones puede servir para encontrar la solución óptima aunque la complejidad temporal va a ser de tipo exponencial en el peor de los casos, lo cual les va a convertir en métodos poco aplicables en la práctica. Es posible también diseñar alguna estrategia de este tipo a la que se le de un tiempo máximo de búsqueda una vez transcurrido el cual el algoritmo finalice y proporcione la mejor solución hallada hasta el momento y que puede resultar suficientemente satisfactoria.

A pesar del elevado tiempo que se necesita en la mayoría de casos para encontrar la solución óptima, el uso de alguno de los algoritmos mencionados podría justificarse para asignar alguna aplicación de pocas tareas de la que se sabe que va a usarse con muchísima frecuencia y para la que sería muy útil tener una asignación fija y óptima.

Métodos basados en programación matemática

Estas estrategias [Chu80][Ma82] se basan en formular el problema del *mapping* como un problema de optimización expresado por un conjunto de ecuaciones que van a resolverse mediante técnicas de programación matemática.

La función objetivo que se plantea es:

$$Cost(X)=\sum_k \sum_i \left\{ q_{i,k} x_{i,k} + \sum_{h<k} \sum_{j<i} \omega\, c_{i,j}\, d_{x,h}\, x_{i,k}\, x_{j,h} \right\} \qquad (1)$$

En esta ecuación, el primer término del sumatorio representa el coste de cómputo de cada módulo en el procesador al que ha sido asignado:

$q_{i,k}$ = coste del módulo t_i asignado en el procesador p_k
$x_{i,k}$ = 1 si el módulo t_i está asignado al procesador p_k; 0, en caso contrario

El segundo término se encarga de sumar los productos de volumen de comunicación por la distancia que separa a dos procesos asignados en procesadores distintos, usando una constante de normalización (ω) encargada de compensar en el sumatorio los pesos relativos a cómputo y los de comunicación, de tal forma que se adapten a las características de un cierto sistema real:

$c_{i,j}$ = volumen de comunicación entre los módulo t_i y t_j.
$d_{k,h}$ = distancia entre los procesadores p_k y p_h.

A partir de este punto, el equilibrio de cómputo entre todos los procesadores se consigue imponiendo una serie de restricciones. Fundamentalmente, se fijan restricciones en el tamaño de la memoria y restricciones en el tiempo de procesamiento.

La restricción de un sistema con memoria limitada se expresa como:

$$\sum s_i\, x_{i,k} \le R_k \qquad k=1,..,K \qquad (2)$$

donde s_i representa el tamaño de memoria requerido por el módulo t_i, y R_k representa la capacidad de memoria del procesador p_k. Esta ecuación indica que el tamaño de memoria consumido por todos los módulos asignados a un procesador no deben exceder su capacidad de memoria.

La restricción de tiempo viene fijada por:

$$\sum w_i\, x_{i,k} \le L_k \qquad k=1,..,K \qquad (3)$$

donde w_i representa el tiempo de procesamiento requerido por el módulo t_i y L_k representa el tiempo límite requerido para procesar los módulos asignados en el

procesador p_k. Esta ecuación indica que el tiempo total de cómputo de los procesos asignados a un cierto procesador no debe superar un cierto límite de tiempo real.

A partir de este punto, se puede obtener la asignación de tareas a procesadores minimizando la ecuación (1) sujeta a las restricciones fijadas por las ecuaciones (2) y (3). Esto puede resolverse como un problema de programación entera 0-1 no lineal. Además las ecuaciones 0-1 no lineales pueden linealizarse añadiendo restricciones adicionales que simplificarán el proceso de solución.

Según argumentan los propios autores, un problema de este tipo consistente en 8000 restricciones y 2500 incógnitas podía resolverse en los años 80 en pocos minutos usando un computador de la serie CDC6600. Esto representaba un problema de *mapping* correspondiente a 25 tareas y 15 procesadores. Estas cifras, junto con el número de restricciones que deben imponerse, resultan argumentos disuasorios en la práctica para abordar el problema del *mapping* mediante esta estrategia, lo que ha hecho que la gran mayoría de métodos propuestos posteriormente hayan seguido la filosofía de los métodos heurísticos.

2.4 Algoritmos heurísticos

Como se ha mencionado, la búsqueda de asignaciones suele requerir un tiempo de cómputo de tipo exponencial por lo que la resolución del problema del *mapping* ha solido realizarse mayoritariamente mediante alguna estrategia de tipo heurístico que proporcionase buenos resultados sin incurrir, por ello, en tiempos prohibitivos. Normalmente, no existen métodos formales que permitan determinar cuán buena es una heurística concreta en la obtención de valores sub-óptimos porque la base de la heurística se fundamenta en algún tipo de idea que intuitivamente parece prometedor o lógico a la hora de solucionar el problema planteado.

Por el mismo motivo, también resulta difícil valorar la bondad de las soluciones encontradas, lo que lleva a que el método tradicional de validación de las mismas sea la comparación con alguna estrategia de tipo óptimo con ejemplos pequeños o con otras heurísticas de probada eficiencia.

En general, las estrategias heurísticas pueden subdividirse en tres grandes grupos: métodos *greedy*, métodos iterativos y métodos mixtos. A continuación, describiremos las características diferenciales de cada uno de estos grupos y mencionaremos las aportaciones más relevantes en la literatura dentro de cada uno de ellos. Además de estos grupos, se han diferenciado también en un punto separado las técnicas basadas en algoritmos genéticos que, si estrictamente hablando podrían englobarse en el grupo de los métodos iterativos, en la práctica suelen presentarse como una categoría propia dada las peculiaridades en las que se basan dichos métodos.

2.4.1 Métodos *greedy*

En general, esta categoría incluye todos aquellas estrategias que permiten encontrar una asignación de forma rápida y la base de sus algoritmos consiste en construir la asignación paso a paso, sin reasignar ninguna tarea ya colocada. Así, la asignación de la tarea i-ésima en un procesador depende sólo de un cierto criterio determinado a partir de la asignación parcial de las (i-1) primeras tareas ya colocadas. Son, en este sentido, algoritmos sin vuelta atrás.

Dentro de los métodos *greedy* podríamos distinguir, a su vez, dos grupos de heurísticas: por una parte, están aquellos métodos que efectúan un procesamiento ligado a las propiedades estructurales (o topológicas) del grafo y, por otro lado, están aquellos métodos que sólo usan información cuantitativa del grafo. El primer grupo cuenta con estrategias que, sin dejar de ser heurísticas sin vueltas atrás, ejecutan un procesamiento de una complejidad significativamente superior al efectuado por las estrategias del segundo grupo. Esta complejidad se suele traducir, por ejemplo, en una serie de manipulaciones en el grafo o en su procesamiento siguiendo un orden dictado por las características topológicas del mismo.

Estrategias *greedy* topológicas

El primer ejemplo de este tipo de estrategias lo constituye la propuesta por Lo en [Lo88]. En su caso, la heurística propuesta es una generalización para K procesadores del algoritmo óptimo propuesto por Stone mencionado en la sección anterior. La base del algoritmo se reducía a aplicar el algoritmo de *min-cut* K veces convirtiendo en cada paso el grafo original en un grafo extendido con dos nodos terminales p_i y p_i'. El primer nodo p_i representa a uno de los procesadores del sistema y p_i' asume el papel del resto de (K-1) procesadores. Los arcos que conectan a estos dos nodos con los nodos del programa se definen mediante una fórmula compleja que pondera el coste de cómputo de una tarea en un procesador y su coste en los otros procesadores. Se aplica el algoritmo de *max-flow min-cut* a este grafo y se obtienen así los procesos asignados al procesador p_i. Repitiendo el proceso con todos los procesadores se alcanzará una situación en la que, idealmente, se habrán asignado todas las tareas. En caso contrario, se reconfigura el grafo extendido eliminando de él las tareas ya asignadas y se repite el proceso anterior con aquellas que quedaron sin asignar. Si no se consigue asignarlas, se recurre finalmente a una estrategia *greedy* del tipo cuantitativo.

Otro ejemplo de algoritmo heurístico de tipo estructural lo constituye el propuesto por Ali en [Ali93]. Como en el caso anterior, se supone un sistema formado por procesadores heterogéneos y la resolución del problema de la asignación se resuelve mediante la creación de un grafo extendido a partir del grafo del programa. El grafo extendido contiene una serie de nodos adicionales que representan los procesadores del sistema y que están unidos a cada uno de los nodos del grafo de programa por un arco que tiene un peso igual al coste de cómputo de dicho nodo en ese procesador. La figura 2.8 muestra un ejemplo de este tipo de grafo. En el ejemplo mostrado tenemos cuatro tareas: t_1, t_2, t_3 y t_4, y dos procesadores: p_1 y p_2.

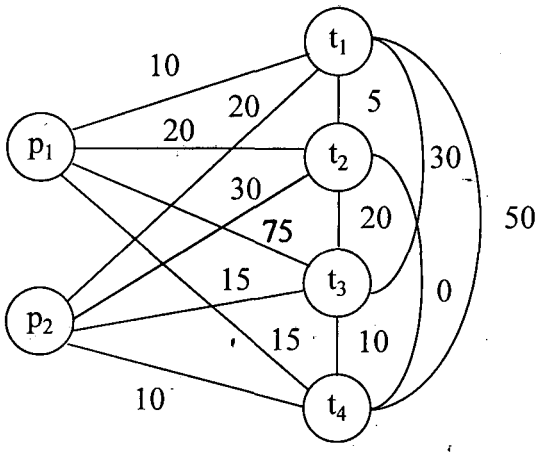


Figura 2.8 Ejemplo grafo extendido

Una vez construido este grafo extendido, el problema se formula en términos de obtener una división del grafo en clanes, donde cada clan contiene un único nodo procesador. La heurística propuesta va calculando dichos clanes en función de las conexiones que exhibe el grafo extendido. No reproducimos la heurística dada su extensión aunque en la figura 2.9 podemos ver su resultado una vez aplicada al ejemplo de la figura anterior: las tareas t_1 y t_4 han sido asignadas al procesador p_1, y las tareas t_2 y t_3 han sido asignadas al procesador p_2.

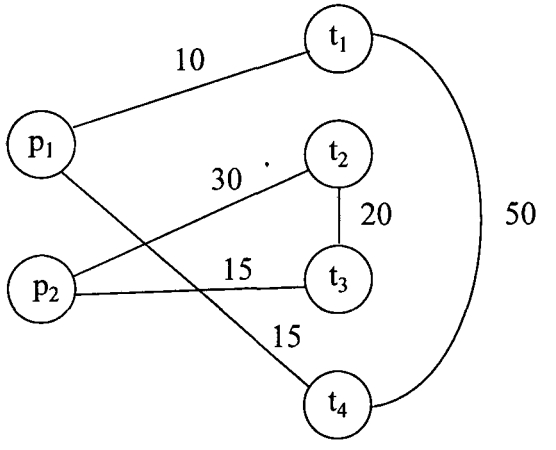


Figura 2.9 Obtención de clanes a partir del grafo de la figura 2.8

Por último, nos encontramos en este grupo aquellas estrategias que se basan en el concepto de contracción de un grafo, es decir, resolver el problema de asignación reduciendo el grafo del programa a un grafo de la misma familia que el grafo original aunque con un número de nodos igual o inferior al número de procesadores [Ber87] [Lo88b]. En ambos casos se intenta equilibrar el número de tareas por procesador al tiempo que se minimizan las comunicaciones. El método propuesto en [Ber87] se basa en la idea de truncamiento de árboles binarios completos. Sin embargo, es un método que no asegura la máxima reducción de las comunicaciones. Por su parte, en [Lo88b] se propone un método de contracción simétrica para programas formados por un conjunto de N tareas idénticas. Esta contracción simétrica asignará un mismo número de tareas a

cada procesador y consigue una asignación óptima cuando el número de tareas N cumple que $2N \leq K$ (siendo K el número de procesadores).

De todas las estrategias reseñadas en este apartado se puede concluir que constituyen de algún modo un grupo de heurísticas avanzadas que requieren la realización de un procesamiento de una complejidad que suele ser ligeramente superior a los métodos *greedy* de tipo cuantitativo. En general, todas tienen complejidades que están entre N^2 y N^3 (siendo N el número de tareas).

Estrategias *greedy* cuantitativas

A diferencia de las estrategias topológicas, las cuantitativas sólo se basan en parámetros cuantitativos de los grafos como volumen de cómputo de los nodos o volumen de comunicación de los arcos, y esa información va a usarse para ordenar bien sea los nodos o los arcos para realizar posteriormente un recorrido por esa lista ordenada y determinar de algún modo la asignación de cada nodo.

Claramente, la efectividad de estos métodos va a depender de la elección que se haga de la primera tarea y de las posteriores en cada paso sucesivo del algoritmo. Normalmente, esta situación va a provocar que, a medida que el algoritmo avanza, las decisiones que toma sean cada vez más limitadas, provocando con ello un empobrecimiento final de la asignación. Este es uno de sus inconvenientes principales: su propia naturaleza simple hace que siempre sea posible encontrar ejemplos donde la calidad de la asignación hallada sea baja. En contrapartida a este bajo rendimiento, su complejidad algorítmica suele ser muy favorable (típicamente, N o NlogN), que es la que se necesita para obtener una ordenación inicial de las tareas del grafo y poder realizar, a continuación, un recorrido lineal por ella.

Su bajo rendimiento hace que, habitualmente estos métodos se usen como primera fase de un método mixto, donde se ejecutará una segunda fase que mejorará la asignación encontrada previamente.

Un ejemplo de método *greedy* sería la estrategia LPTF (Largest Processing Time First), que va asignando las tareas en orden decreciente de su volumen de cómputo, sin considerar para nada los volúmenes de comunicación. Esa estrategia consigue, por lo tanto, un buen equilibrio de cómputo entre todos los procesadores, con la ventaja adicional que, en presencia de comunicaciones despreciables, una asignación M hallada por este método garantiza que:

$$M \leq (4/3 - 1/3K)\, M_{opt}$$

donde M_{opt} es la asignación de tiempo de ejecución óptimo y K es el número de procesadores [Gra69].

2.4.2 Métodos iterativos

Estos métodos parten de una asignación inicial completa (no necesariamente buena) a partir de la cual van a intentar mejorarla iterativamente. En la mayoría de los casos el proceso de mejora de basa en el intercambio de pares de tareas entre dos procesadores y, ocasionalmente, también se usa el movimiento individual de tareas. Los sucesivos pasos de movimiento y/o intercambio suelen llevar a estos algoritmos a situaciones en las que se alcanza un mínimo local en la función que evalúa la calidad de la asignación. Dicho fenómeno se puede observar en la fig. 2.10. Para superar estas situaciones los métodos iterativos suelen disponer de algún mecanismo de *hill-climbing* incorporado. Estos mecanismos pueden ser, por ejemplo, la aceptación en un determinado momento de asignaciones peores con la esperanza que permitan alcanzar una solución final más cercana al óptimo, o la realización de perturbaciones aleatorias una vez alcanzado un mínimo local. La naturaleza de estos métodos permite que obtengan asignaciones de mejor calidad que los métodos *greedy* a costa de aumentar su complejidad temporal.

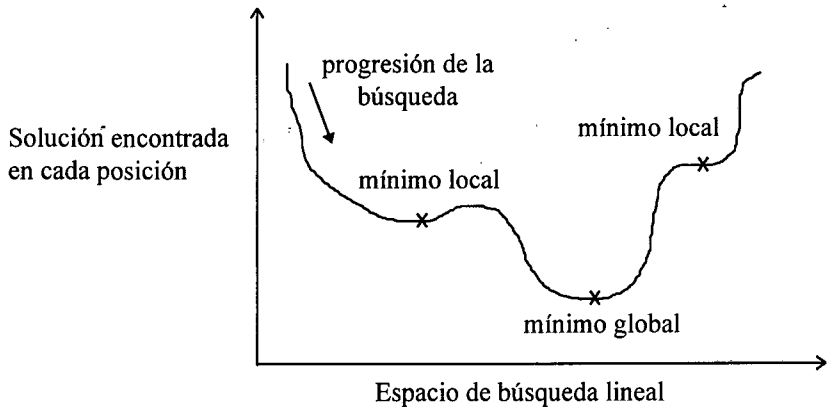


Figura 2.10 Aparición de mínimos locales en espacios de búsqueda

En este grupo podemos encontrar bastantes estrategias, algunas de las cuales pueden considerarse auténticos "clásicos" de la literatura. El primero de ellos es el propuesto por Bokhari [Bok81]. En su versión del problema del *mapping* suponía que se disponía de tantos procesadores como tareas a asignar, que las tareas tenían cómputos iguales y que los volúmenes de comunicación eran unitarios. A partir de estas premisas la función objetivo que intenta minimizar su algoritmo es

$$\sum_{i,j} c_{ij}\, d_{f(i),f(j)}$$

donde $c_{i,j}$ es el coste de comunicación entre las tareas t_i y t_j, y $d_{f(i),f(j)}$ indica la distancia entre los procesadores donde se han asignado ambas tareas. Dado que $c_{i,j}$ se considera 1 para todos los arcos del grafo, el problema de asignación se reducía a maximizar la cardinalidad de la asignación. Es decir, maximizar el número de arcos del grafo del programa cuyos nodos extremos estaban asignados en procesadores adyacentes en el multiprocesador.

A partir de estas premisas, la solución propuesta por Bokhari consiste en realizar una primera asignación aleatoria de los nodos y, a partir de este punto, repetir una iteración en la que se intentaban intercambiar de lugar cada nodo del grafo con sus vecinos. Una vez hecho esto, se realiza aquel intercambio que proporcionaba una mayor reducción en la cardinalidad del grafo y se vuelve a repetir el proceso hasta alcanzar una situación en la que no se mejorase dicho valor. Las situaciones de mínimos locales son superadas mediante le intercambio aleatorio de varios nodos. El algoritmo finaliza si después de realizar los intercambios aleatorios, la siguiente iteración que intenta intercambiar parejas de nodos no consigue ninguna mejora en la cardinalidad.

Aunque esta variante puede parecer discutible (máxime si tenemos en cuenta que los costes de comunicación considerados eran siempre unitarios), algunos autores han observado que un elevado porcentaje de aplicaciones paralelas pueden considerarse síncronas o débilmente síncronas, caracterizándose porque se ejecutan mediante una descomposición regular de datos con un proceso por procesador, cantidades de cómputo prácticamente iguales y periódicos intercambios de mensajes [Fox89]. Así pues, a partir de una situación inicial de equilibrio inherente de cómputo, el problema del *mapping* se resolvería minimizando el *overhead* de comunicaciones con una estrategia como la propuesta por Bokhari.

Otro método iterativo puro es el propuesto por Selvakumar en [Sel92]. En su propuesta parte de un problema inicial con un mayor número de procesos que procesadores y costes, tanto de cómputo como de comunicación, que pueden tener cualquier valor. Su solución empieza con una primera asignación aleatoria de las tareas, a partir de la cual intenta mejorarla mediante un conjunto de iteraciones donde se efectúan movimientos individuales de tareas de un procesador a otro, seguidos de un intercambio de dos tareas entre dos procesadores. Esta secuencia de movimientos seguidos de un intercambio se va realizando hasta que no se obtiene ninguna mejora en la función de coste, momento en el que se van a realizar varios movimientos aleatorios y se volverá a repetir el proceso de movimientos e intercambios. Sólo después de efectuar varios ciclos de movimientos e intercambio (que sus autores recomiendan que sean tres) sin obtener ninguna mejora, el algoritmo finaliza y proporciona la mejor solución hallada hasta el momento.

Un método iterativo clásico y ampliamente utilizado en la resolución de multitud de problemas de optimización es el método de *Simulated Annealing* [Laa87]. La idea del método surge de la observación de fenómenos físicos. Se basa en una analogía con la física estadística: cuando se desea obtener un metal que tenga una estructura lo más uniforme posible se utiliza la técnica denominada de recalentamiento. Para ello, se calienta el metal y después se va reduciendo lentamente su temperatura de forma que alcance el equilibrio durante el enfriamiento. Llegado a una temperatura suficientemente baja, el metal habrá alcanzado un estado de equilibrio correspondiente a la energía mínima. A altas temperaturas existe una elevada excitación térmica que puede localmente aumentar la energía del sistema. Este fenómeno se produce con un cierta probabilidad que disminuye con la temperatura. Esta característica se traducirá algorítmicamente a permitir la salida de un mínimo local en la función que se intenta optimizar.

La teoría demuestra que este método tiene una convergencia asintótica hacia la solución óptima. De hecho, usando un proceso de enfriamiento suficientemente lento, la ejecución del algoritmo se extiende hacia el infinito y permite recorrer todo el espacio de soluciones y hallar, por lo tanto, el óptimo global. En la práctica, las implementaciones deben tener un número de pasos acotado, lo que impedirá, normalmente, alcanzar ese óptimo global aunque se suelen obtener mínimos locales cercanos a él.

La aplicación de este método para resolver el problema del *mapping* se ha hecho normalmente de forma intuitiva. Es difícil justificarlo teóricamente sólo por una simple analogía y, sobre todo, de encontrar un significado concreto a parámetros tales como la temperatura del sistema. Sin embargo, varios autores han usado este método como estrategia de *mapping* [Ram88][Bou95].

En estos casos la resolución del problema parte de una asignación inicial con un coste asociado y se intentará mejorarla siguiendo un criterio local como el intercambio de pares de tareas entre dos procesadores. Se acepta dicho intercambio si la función de coste de la nueva asignación es menor y, bajo una cierta probabilidad que decrece a lo largo de la ejecución del algoritmo, se aceptan intercambios que provocan asignaciones peores. Esta última premisa permite evitar aquellas situaciones de mínimo local en la función que se quiere optimizar.

Las asignaciones generadas por este método son en promedio bastante buenas aunque presente varios inconvenientes como:

- un tiempo de cómputo elevado que se agudiza con el aumento del número de tareas.
- se basa en una asignación inicial aleatoria y sigue una sucesión de etapas donde se pasa por distintas asignaciones de forma también aleatoria (debido a la aceptación probabilística de malos movimientos). Esto hace que su comportamiento sea imprevisible y que después de cada ejecución se obtenga una asignación distinta para un mismo problema con distintos valores finales en la función de coste.
- el algoritmo depende de un cierto número de parámetros (temperatura inicial, temperatura final, velocidad de enfriamiento, etc.) que son difíciles de ajustar y pueden depender de la asignación que deba efectuarse.

Todos estos factores suelen llevar a la consideración del presente método como un método costoso y poco apropiado, en la práctica, para un uso habitual. Sin embargo, la experiencia acumulada después de muchos años usándolo en muy diversos problemas de optimización es considerable y eso ayuda de forma importante a la hora de fijar los valores de los parámetros mencionados en el tercer punto o de establecer reglas para su determinación. Además, la calidad de sus resultados lo han convertido tradicionalmente en un método de referencia frente al que se han cotejado los resultados obtenidos por otras estrategias menos costosas computacionalmente.

La idea de recalentamiento en la que se basa el algoritmo de *Simulated Annealing* (SA) la comparte también otra heurística reciente que se combina, sin

embargo, con el funcionamiento de una red neuronal de Hopfield en el momento de generar nuevas configuraciones [Bul92]. Esta estrategia de optimización recibe el nombre de *Mean Field Annealing* y, como en el caso del SA, es una técnica genérica aplicable a cualquier problema de optimización. Su gran ventaja frente a SA estriba en la propiedad de cómputo colectivo que caracteriza a las redes de Hopfield, lo que la hace especialmente atractiva para ser implementada de forma paralela, consiguiendo con ello una reducción importante en el tiempo de cómputo necesario.

La búsqueda Tabú es otro método puramente iterativo que, a partir de una solución inicial, busca la mejor solución a la que se puede llegar mediante una serie de transformaciones básicas (relaciones de vecindad). Vuelve a ser esta una técnica genérica de resolución de problemas de optimización que fue adaptada al problema del *mapping* por Bouvry [Bou95]. Los principales componentes de una búsqueda tabú son los siguientes:

- Una función de coste que permita estimar y comparar la calidad de las soluciones exploradas.
- Las relaciones de vecindad que permiten moverse por el espacio de soluciones. Este viaje debe garantizar que se pasará por el mínimo global o, al menos, por algún mínimo local.
- En cada iteración del algoritmo, se examina el conjunto de candidatos (soluciones vecinas) alcanzables. A mayor número de relaciones de vecindad, más oportunidades habrá de alcanzar buenas soluciones, pero al precio del coste de aumentar el tiempo necesario para evaluarlos.
- Se guarda de forma separada la mejor solución encontrada hasta el momento.
- Con objeto de evitar los ciclos (volver a una solución ya explorada), se memorizan las soluciones examinadas. De ahí el concepto de "lista tabú" que da nombre al método. La lista tabú tiene un tamaño limitado y cada elemento de ella tiene una serie de atributos que caracterizan la solución concreta o, más bien, que caracterizan los desplazamientos que deben hacerse de la situación actual a la precedente. Estos atributos deben escogerse con cautela porque de ellos dependerá parte de la eficiencia del algoritmo. Existen distintos criterios a seguir:

    a) guardar una descripción completa de la solución, lo que puede ser muy costoso desde el punto de vista del consumo de memoria,
    b) guardar ciertos atributos puede requerir la realización de costosos cálculos ulteriores para poder comparar la soluciones,
    c) en el caso de que problema exhiba soluciones que no puedan distinguirse (entre otros factores, por la función de coste), los atributos guardados deben ser suficientes para poder catalogar una familia de soluciones parecidas y no sólo una solución concreta.

- Un criterio de aspiración que debe permitir que se vuelva sobre una solución que no es admitida por la lista tabú. De hecho, la lista tabú sólo representa una vista parcial del recorrido efectuado por el espacio de soluciones y, por ejemplo, el hecho de mejorar el valor de la función de coste de la mejor

solución encontrada hasta el momento puede ser considerado como criterio de aspiración.

- Unos criterios de finalización, que pueden ser múltiples: la búsqueda puede pararse cuando se encuentre una solución suficientemente satisfactoria, o al cabo de un cierto tiempo, o si no se ha conseguido una mejora en la mejor solución después de un tiempo elevado.
- Unas funciones de intensificación y de diversificación que permiten acercarse, por un lado, y retardar o alejarse, por otro, de aquellos vecinos que parecen o que no parecen, respectivamente, interesantes.

Existen tres casos en los que una descripción considerada en su momento como tabú puede ser explorada de nuevo:

1. La solución ha salido de la lista tabú debido al tamaño finito de ésta. Esto puede ser síntoma de que la lista tabú es demasiado corta.
2. La solución aparece en la lista pero se satisface el criterio de aspiración.
3. Todos los vecinos visitables a partir de la situación actual están en la lista tabú. Esto puede ser síntoma de que las funciones de vecindad son demasiado restrictivas o de que los atributos registrados en la lista tabú son demasiado vagos y representan, por ello, demasiadas soluciones.

Las búsquedas tabú son, en general, más difíciles que las técnicas de *Simulated Annealing* por lo que a su puesta a punto se refiere, pero proporcionan unos resultados de calidad comparable. En este método, la experiencia y el conocimiento concreto del problema a optimizar permiten fijar buenos valores para los distintos parámetros. Las relaciones de vecindad se derivan del estudio del problema, mientras que el tamaño de la lista tabú será fijada arbitrariamente en función de la experimentación (se fija inicialmente un tamaño pequeño que, paulatinamente se va aumentando hasta que se evitan todos los ciclos).

Las informaciones almacenadas en la lista tabú no son siempre completas: nos podemos encontrar alguna situación que dicha lista considere como conocida aunque no lo sea realmente. En estos casos se impone la actuación del denominado criterio de aspiración. Un ejemplo simple consiste en examinar si la nueva situación mejora la función objetivo respecto a la situación precedente que se ha encontrado. Si lo hace, el vecino es aceptado; si no, se considera que se trata de una situación ya encontrada y no se acepta como vecino.

2.4.3 Métodos mixtos

Este grupo de heurísticas incluye todo un conjunto estrategias que buscan un compromiso entre la eficiencia computacional de los métodos puramente *greedy* y los métodos puramente iterativos. Para conseguir ese objetivo su diseño está condicionado, habitualmente por la existencia de dos fases diferenciadas: una primera que suele obedecer a la categoría de estrategia *greedy* y una segunda que se amolda al tipo de estrategia iterativa. A continuación, veremos las propuestas más importantes dentro de este grupo.

Históricamente, podemos considerar que la primera heurística de tipo mixto fue la propuesta por Efe en [Efe82]. En su versión del problema se partía de u sistema homogéneo de K procesadores y un programa paralelo de N procesos ($N \geq K$). Sin embargo, no todos los procesos eran iguales y existían algunos procesos catalogados como "ligados" (*attached*), que se caracterizaban porque sólo podían asignarse a ciertos procesadores específicos, limitando con ello la estrategia de asignación.

A partir del grafo original del programa, la primera fase obtiene agrupaciones de los nodos para configuraciones de 2, 3,..,K procesadores. Para formar cada una de estas agrupaciones se van uniendo repetidamente dos nodos siempre que el arco que los conecta tenga un volumen de comunicación mayor que los volúmenes que los unen al resto de nodos del programa. Este proceso de unión se va repitiendo hasta haber reducido el grafo, sucesivamente, a K, K-1,..,3 y 2 nodos. En todo momento, la unión de nodos está sujeta a la restricción de que dos procesos "ligados" a procesadores distintos nunca pueden unirse. Aunque Efe no propone explícitamente ninguna alternativa, podemos suponer que en el caso de existir arcos de igual volumen se van escogiendo los pares de nodos sin ningún criterio especial.

Una vez finalizada la primera fase, se escoge aquella agrupación que dio un número de agrupaciones que tenían el mejor equilibrio en sus volúmenes de cómputo y se pasa a la segunda fase donde se harán reasignaciones de tareas individuales entre las agrupaciones. Para ello, se calcula la carga promedio de todos los procesadores y se define un factor de tolerancia δ. Los procesadores tienen una carga aceptable si está en el intervalo definido por la carga promedio $\pm \delta$. Si están por encima, serán procesadores sobrecargados y si están por debajo, serán infracargados. El proceso de reasignación de tareas se produce modificando el grafo original aumentando el volumen de comunicación entre las tareas asignadas a un procesador sobrecargado y uno infracargado en un factor proporcional a la diferencia de carga de cómputo relativa entre ambos procesadores. Una vez hecho esto, se vuelve a aplicar el algoritmo de agrupación de la primera fase y se va repitiendo el proceso hasta encontrar una situación en la que todos los procesadores tengan una carga de cómputo aceptablemente equilibrada.

Como reconoce el propio Efe, este algoritmo no funciona de forma satisfactoria cuando la primera fase deja nodos asignados a distintos procesadores entre los cuales no hay ninguna comunicación (p. ejemplo, en el caso de programas con un grafo en forma de árbol), y también puede sufrir problemas de convergencia cuando el factor de tolerancia es demasiado pequeño. Sin embargo, el propio autor asegura que el método encontró una asignación óptima para un conjunto de ejemplos, aunque no se detalla ni cuántos ni cómo son.

Otra estrategia mixta es la propuesta inicialmente por Lee y Aggarwal en [Lee87] y que después fue ampliada y generalizada por Chaudhary y Aggarwal en [Chau93]. En la primera versión se partía de un programa con un número de nodos menor o igual al número de procesadores y se trataba de encontrar una asignación donde se minimizase sólo el coste debido a las comunicaciones. Para la evaluación del coste de comunicaciones se introducía el concepto de fases de comunicación: una fase es el intervalo de tiempo durante el cual tiene lugar una comunicación en un arco del grafo

del programa. El concepto de fase implica considerar que no todas las comunicaciones ocurren simultáneamente durante la ejecución del programa. Este parámetro venía justificado porque el tipo de aplicaciones que se utilizaban como ejemplo pertenecían al ámbito del procesamiento de imágenes y sus comunicaciones podían caracterizarse fácilmente usando dicho concepto.

Con la introducción de este nuevo parámetro se proponían 4 funciones de coste distintas (la suma de todas las comunicaciones, el valor máximo de las comunicaciones, la suma de los máximos de las distintas fases simultáneas o el máximo de los máximos de cada fase). Junto con este conjunto de funciones de coste se añade también unos procedimientos para la evaluación de las mismas teniendo en cuenta el tipo de comunicación seguido (síncrono o asíncrono) y la posibilidad de contenciones en la red de interconexión. Para realizar estos cálculos el modelo de programa usado no es el de TIG habitual sino que se corresponde con el modelo de grafo dirigido (equivalente a un DAG sin la restricción de que debe ser acíclico).

La primera fase se asignaba en primer lugar el proceso con mayor volumen de comunicación. Después iba escogiendo a cada paso aquél proceso que tuviera mayor volumen de comunicación con los ya asignados y lo colocaba en aquel procesador libre para el cual se minimizase la función objetivo. Una vez finalizada esta asignación preliminar, se pasaba a una fase de intercambio de procesos entre procesadores para intentar reducir el valor de la función de coste. La fase de intercambios finalizaba cuando se encontraba una asignación aceptable, criterio que dependía de la función de coste adoptada.

En la generalización de [Chau93], se amplía el problema de forma que se acepta un programa de tamaño mayor que el número de procesadores. El método de asignación no varía de lo expuesto en las dos fases anteriores aunque en la primera se incluye una serie de comprobaciones para asegurar que la disposición de un nodo sobre un cierto procesador no conlleva una asignación que no es factible. Un *mapping* es factible si "la ejecución del programa paralelo en la máquina paralela de acuerdo a ese *mapping* termina". Las situaciones en que esto puede no ocurrir son las ocasionadas por los posibles *deadlocks* producidos en las colas de mensajes de cada procesador y al orden de ejecución que se fijaba entre las tareas. La inclusión de esta comprobación eleva la complejidad de la primera fase a N^4 (siendo N el número de procesos) sin que realmente se justificase dicha comprobación porque el orden rígido de ejecución que marcaba la política de ejecución no se corresponde en la práctica con el modelo de ejecución bajo el que operan los multiprocesadores normales.

Otro grupo de estrategias mixtas corresponden a las que fueron proponiendo Sadayappan, Ercal y Ramanujam en [Sad87], [Sad90] y [Erc90], respectivamente.

La primera estrategia (*nearest-neighbor*) [Sad87] estaba orientada especialmente a la asignación de programas cuya topología fuese de tipo malla o, en su defecto, un grafo que exhibiese una elevada conectividad local entre sus nodos. Como computador paralelo se consideraba uno con una interconexión de tipo malla de dimensión k*h (k filas y h columnas). La primera fase agrupaba primero nodos vecinos en k grupos de tal forma que cada agrupación tenía un número de nodos parecido al de los otros. La

agrupación de nodos se conseguía recorriendo al grafo de "arriba a abajo", entendiendo los términos arriba y abajo desde el punto de visto topológico de un grafo en forma de malla. Después se repetía el proceso haciendo h agrupaciones y recorriendo el grafo del programa de "derecha a izquierda". La intersección de las agrupaciones hechas en los dos pasos proporcionaba un conjunto de grupos menor o igual que el número de procesadores y que tenían una asignación inmediata con los procesadores del sistema. La figura 2.11 muestra el resultado de esta primera fase para un grafo que va a asignarse a un sistema con 8 procesadores.

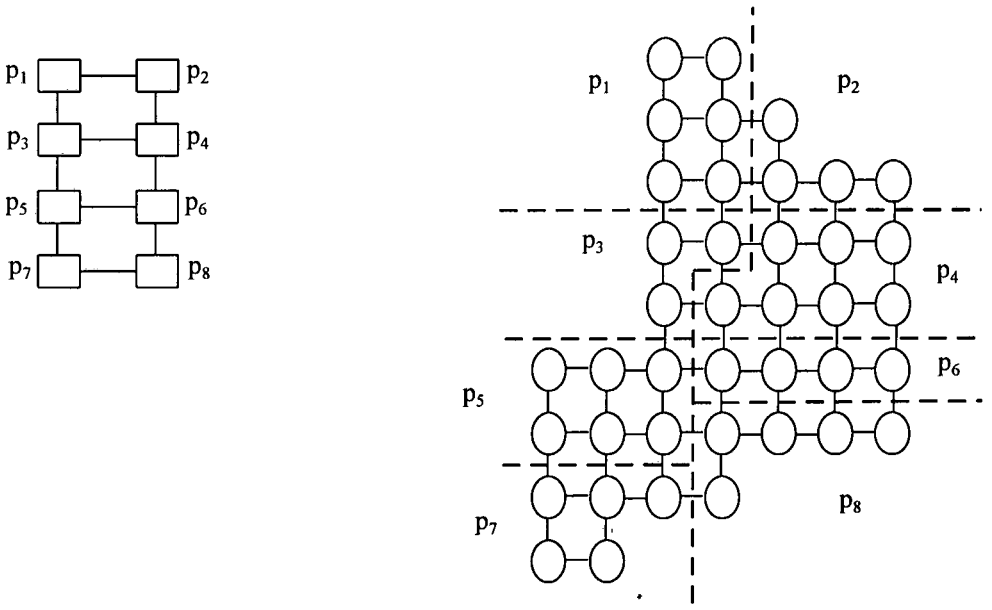


Figura 2.11 Ejemplo de partición generada por el algoritmo nearest-neighbor

La segunda fase intentaba mejorar la partición obtenida. En este caso se tenía en cuenta por una parte el equilibrio de cómputo entre las agrupaciones así como el volumen de comunicación entre ellas. Empezando por el procesador más cargado se van buscando tareas individuales que puedan ser transferidas a un procesador colindante con objeto de mejorar los criterios mencionados anteriormente de equilibrio de cómputo y volumen de comunicación. El proceso de refinamiento se va repitiendo hasta que no se encuentre ninguna mejora. La generalización del algoritmo para grafos que no tengan una topología parecida a una malla complica substancialmente el método debido a la necesidad de definir dos recorridos "perpendiculares" del grafo en la primera fase.

El segundo método propuesto por estos autores es el denominado de bipartición recursiva de mínimo corte, del que existen dos versiones [Sad90] y [Erc90]. La base de ambas versiones consiste en dividir a cada paso el grafo en dos mitades que sean equilibradas desde el punto de vista del coste de cómputo y tengan, a su vez, un coste de comunicación mínimo [Sad90]. Una vez hecha la primera división en dos mitades, el algoritmo continua recursivamente dividiento cada una de ellas hasta alcanzar un número de particiones iguales al número de procesadores. Obviamente, el número de procesadores debe ser necesariamente una potencia de 2 porque ese es el número de

particiones que es capaz de generar el algoritmo. La inclusión de este método en la categoría de los esquemas mixtos no se debe al comportamiento global del algoritmo sino a la realización de cada uno de los pasos individuales de partición, donde sí se usa una estrategia mixta. En primer lugar, las tareas del programa se van colocando una por una en una partición u otra de forma que ambas queden con un volumen de cómputo equilibrado, sin tener en cuenta ninguna información relativa a las comunicaciones. A continuación, se ejecuta un proceso de refinamiento de la partición obtenida anteriormente con objeto de mantener, en la medida de lo posible, el equilibrio de cómputo, pero minimizando las comunicaciones entre tareas asignadas a particiones distintas. Durante el proceso de creación inicial de las particiones, si el volumen de cómputo de las particiones en un determinado momento es igual, la siguiente tarea se coloca de forma aleatoria en cualquiera de ellas. Este comportamiento no determinista del método provoca que los resultados de la asignación para un mismo programa no siempre sean siempre iguales.

La segunda versión del método funciona prácticamente igual que la anterior, pero introduce una variación para conseguir una asignación especialmente ideada para una arquitectura de tipo hipercubo [Erc90]. En la versión anterior, después del proceso de partición las K particiones obtenidas deben asignarse finalmente a uno de los K procesadores físicos. En esta segunda versión, básicamente, se trata de ir efectuando una asignación parcial de todos los procesos a medida que se van haciendo las particiones. Así en el i-ésimo paso de partición del algoritmo se determina el i-ésimo bit del procesador al que estarán asignados los procesos de cada una de las particiones.

Finalmente, la última propuesta significativa de estrategia mixta que conocemos es la realizada por Wu en [Wu94]. En su propuesta existe una primera fase en la que se asignan inicialmente las K tareas con mayor coste de cómputo y comunicación a una de las K particiones del programa. A partir de este punto se escoge la partición menos cargada y se busca entre las restantes tareas no asignadas aquella que contribuya con una carga menor a dicha partición. Repitiendo iterativamente este proceso se consiguen formar K particiones del grafo. Posteriormente, se intentará mejorar las particiones halladas moviendo tareas entre la partición más cargada y cualquiera de las demás. La última fase del método consiste en la asignación las particiones a los procesadores físicos (en su caso un sistema basado en Transputers). Para ello, proponen un método de reconfiguración de la red de Transputers de forma que se adapte al grafo obtenido después de la partición, de forma que a aquellas particiones con más de 4 arcos se les haga coincidir sus arcos con mayor peso de comunicación con links de la arquitectura y sólo sea necesario rutear los otros arcos por algún procesador intermedio siguiendo un camino mínimo.

2.4.4 Algoritmos genéticos

Si el método SA se basaba en una analogía con el mundo de la física, este grupo de algoritmos se basan en una analogía con el mundo de la biología y, en concreto, con la evolución de las especies. Su fundamento podría resumirse en el concepto de supervivencia de los mejores ejemplares de la especie.

Su funcionamiento se basa en repetir un ciclo en el que se parte de una población sobre la que se aplica una evaluación para determinar cuales son los mejores ejemplares, a los que se aplicará, finalmente, una serie de operadores genéticos para generar nuevos individuos que pasarán a formar parte de la población. Este ciclo finaliza en el momento en el que se alcanza una población estable. La repetición de este ciclo de reproducción de la población es la característica que engloba este tipo de algoritmos en el grupo de los métodos iterativos. La complejidad de las operaciones realizadas en cada ciclo es lo que les convierte tradicionalmente en una categoría diferenciada.

A partir de este esquema básico, la implementación concreta de un algoritmo genético implica la resolución de varios problemas intrínsecos al uso de este tipo de estrategias. En primer lugar, se debe determinar una codificación de los individuos, que corresponderán a posibles soluciones del problema que se desea resolver (en el caso concreto del *mapping*, cada individuo es una posible asignación). Después debe determinarse el criterio de evaluación de los individuos (p. ej., mediante el uso de alguna función de coste). A continuación se debe fijar cuál es el criterio de selección de individuos para su posterior reproducción (un criterio habitual consiste en asignarle a un individuo que posee un valor γ_i en su función de evaluación un número de descendientes igual a γ_i/Γ, donde Γ es el valor medio de todos los valores de la evaluación de la población). Le sigue la aplicación de los operadores genéticos que suelen ser básicamente dos: cruzamiento de dos individuos para generar un descendiente (que contendrá parte de la solución codificada por cada uno de los padres) y mutación de los individuos (p. ej. alterando de algún modo la solución que cada uno codifica).

En su aplicación al problema del *mapping* las soluciones existentes se han restringido al problema de asignar un programa paralelo a un cierto multiprocesador cuando el número de procesos y procesadores es idéntico y se trata de obtener una colocación que minimice el coste en el que incurren las comunicaciones. Básicamente, es una versión del problema planteado al hablar del algoritmo de Bokhari en la que se elimina la restricción de las comunicaciones de coste unitario y se admiten costes variables [Tal93][Chan94].

Como sucede en el campo de las redes neuronales, el base del funcionamiento de estos algoritmos no se conoce perfectamente y, por lo tanto, todos los algoritmos aparecidos hasta el momento han sido fruto de las pruebas experimentales. De hecho, la analogía que se plantea con el mundo de la biología puede parecer tan superficial como la analogía comentada en el caso del SA pero, de todos modos, han demostrado en los casos mencionados su capacidad para obtener buenas soluciones. Sin embargo, la dificultad de puesta en marcha de un algoritmo genético debido a la gran cantidad de elementos que deben sintonizarse para su correcto funcionamiento ha hecho que cuenten con un elevado número de detractores.

2.5 Modelos de coste

La evaluación de la bondad de una cierta asignación se suele hacer por medio de lo que hemos denominado función de coste. En las distintas soluciones propuestas al problema del *mapping* no sólo los modelos han sido diversos sino que también las funciones de coste propuestas han sido distintas, complicando con ello aún más la posible comparación de los respectivos resultados. Aunque no existe un consenso universal sobre cual es la mejor función de coste, mayoritariamente han sido dos las más utilizadas: la ***minimax*** y la ***summed***.

A partir de la definición hecha en el apartado 2.1. una asignación es una función denominada f: T $\rightarrow$ P que asocia a cada tarea t_i un único procesador p_q, $f(t_i) = p_q$. Suponiendo que hay K procesadores el número de todas las posibles soluciones es N^K.

La agrupación de tareas de un procesador p_q, TC_q (*Task Cluster*), se puede definir como el grupo de tareas asignadas a él:

$$TC_q = \{ t_i \mid f(t_i) = p_q \}, q = 1,..,K$$

El coste de procesamiento del procesador p_q, PC_q (*Processing Cost*), es el peso total de los cómputos de todas las tareas asignadas a él

$$PC_q = \sum_{t_j \in TCq} w_j, \quad q = 1..K$$

siendo w_j el coste (peso) de cómputo de la tarea t_j.

El conjunto de comunicaciones del procesador p_q, CS_q (*Communication Set*), es el conjunto de arcos del grafo de comunicaciones (TIG) que van de él a cualquier otro procesador:

$$CS_q = \left\{ v_{i,j} \mid f(t_i) = p_q \quad y \quad f(t_j) \neq p_q \right\} \quad q = 1..K$$

El coste de comunicaciones del procesador p_q, CC_q (*Communication Cost)* es el peso total de los arcos del conjunto de comunicaciones:

$$CC_q = \sum_{v_{i,j} \in CS_q} c_{ij} \qquad q = 1..K$$

c_{ij} indica el peso de comunicación para el arco que une las tareas t_i y t_j.

El coste total para el procesador p_q, PWL_q (*Processor Work Load*: carga del procesador), es la suma de ambos términos: PC_q y CC_q.

$$PWL_q = PC_q + CC_q$$

Con el modelo de coste de la función ***minimax*** se estima el tiempo requerido por cada procesador (tiempo de ejecución + tiempo de comunicación) bajo un determinado *mapping* y el coste (tiempo) mayor entre todos los procesos es el valor a minimizar. Así la función de coste que evalúa la calidad de una determinada asignación es:

$$\cos t(f) = \max_{1 \le j \le K} (PWL_j)$$

De la función de coste anterior existen dos variantes que intentan reflejar alguna característica de la arquitectura del sistema.

La primera consiste en tener en cuenta los caminos físicos por los que van a circular las comunicaciones. Ese es un factor importante en las arquitecturas con una red punto a punto y que suele medirse como un conjunto de distancias entre los procesadores. La consecuencia de este factor es una redefinición del término CC_q:

$$CC_q = \sum_{v_{i,j} \in CSq} c_{i,j} * d_{f(t_i),f(t_j)} \qquad q = 1..K$$

donde $d_{M(i)M(j)}$ corresponde a la distancia física entre los procesadores donde han sido asignadas, respectivamente, las tareas t_i y t_j.

Adicionalmente, la segunda corrección tiene por objeto incorporar en el cálculo de PWL alguna característica dependiente de la arquitectura como puede ser la velocidad relativa de cómputo de los procesadores frente a la velocidad de comunicación de la red de interconexión. En este caso la corrección que suele incluirse en la fórmula de PWL es:

$$PWL_q = \alpha PC_q + \beta CC_q$$

donde α y β son dos parámetros dependientes de la arquitectura que sirven para ponderar el peso debido al cómputo y peso debido a las comunicaciones de forma que la suma final sea de elementos que hayan quedado expresados en unidades equivalentes.

Frente a este modelo de coste *minimax*, el modelo de coste basado en la función ***summed*** se puede justificar de la siguiente forma. Idealmente, la carga computacional de todos los procesadores debería estar igualmente distribuida entre todos los procesadores y no debería existir ningún coste adicional debido a comunicaciones. En la práctica, por supuesto, ninguna asignación va a lograr esta situación ideal. El mérito de una asignación se medirá, por lo tanto, en términos de desviación frente a esta situación ideal. Con respecto a la distribución de la carga de cómputo, ésta puede expresarse como la suma para todos los procesadores de la desviación (en valor absoluto) de la carga de cada uno de ellos frente al valor promedio ideal. Con respecto a la comunicación, dado que en el caso ideal no existiría ninguna comunicación en absoluto, la suma total de comunicaciones en el sistema servirá como buena medida de la desviación producida respecto al caso ideal. Así la función de coste *summed* puede expresarse como:

$$cost(f) = \left(\alpha' * \frac{1}{2} \sum_{i=1}^{K} CC_i + \beta' * \sum_{i=1}^{K} \left| PWL_i - \overline{PWL} \right| \right)$$

Donde el primer sumatorio corresponde a la suma total de comunicaciones del sistema (dividido por 2 para compensar la aparición de una misma comunicación $c_{k,h}$ en CC_i y CC_j. $\overline{PWL}$ corresponde a la carga promedio del sistema.

Los términos α' y β' vuelven a reflejar, como en el caso anterior, la penalización relativa que presenta la arquitectura para el coste de comunicaciones y cómputo, respectivamente. Mientras que en el caso de la función *minimax* estos dos términos podían capturar, por ejemplo, parámetros relativos a la latencia de comunicación entre procesadores por cada byte transmitido y el tiempo de ciclo de ejecución de una instrucción, no existe una interpretación física equivalente para los términos α' y β'. Probablemente, por ese motivo los autores que se decantan por esta función de coste suelen obviar la mención a estos términos.

El uso de una u otra función de coste tendrá, lógicamente, consecuencias a la hora de evaluar las posibles asignaciones encontradas por una estrategia de *mapping*. Por ejemplo, en la figura 2.12 se muestra un ejemplo con dos posibles asignaciones para un mismo grafo y su correspondiente valoración a partir de la función *summed* y la función *minimax*.

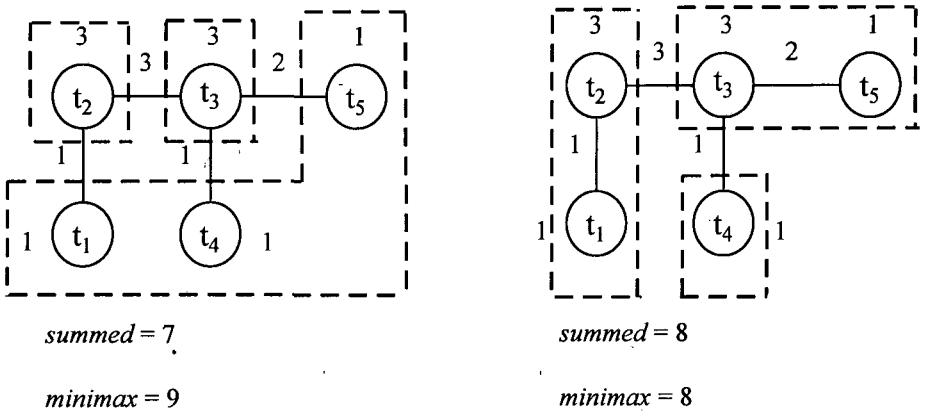


Figura 2.12 Comparación de las funciones de coste summed y minimax

Como muestra la figura 2.12, la valoración que cada función hace de distintas asignaciones puede llegar a ser significativamente diferente:

- la función *summed* puede provocar que se elija una asignación donde se prime el equilibrio de cómputo aunque se agrupen tareas sin ninguna comunicación entre ellas (en el ejemplo de la izquierda: $\{t_1, t_4, t_5\}$)
- la función *minimax*, por el contrario, tiende a valorar mejor aquellas asignaciones donde se agrupen nodos conectados mediante algún arco, aunque ello suponga desequilibrar el cómputo entre los procesadores (ejemplo de la derecha).

Entre estas dos alternativas para modelar la efectividad de una cierta asignación, parece claro que el primer modelo es el más preciso desde un punto de vista conceptual, como así reconocen diversos autores [Erc90].

Un aspecto destacable es que de forma implícita, la función *minimax* está asumiendo que en el sistema van a existir comunicaciones que van a tener lugar concurrentemente, al igual que el cómputo de los distintos procesadores. Sin embargo, en el modelo *summed* se considera implícitamente que el cómputo va a ser concurrente pero con la suma de todas las comunicaciones en un único término, se da la impresión más bien de que éstas vayan a ocurrir de forma secuencial.

Por esto, coincidimos con la opinión de Lo en [Lo88b] cuando afirma que la función de coste *summed* es preferible para sistemas paralelos del tipo que podría catalogarse como sistemas distribuidos, donde los elementos de cómputo (habitualmente estaciones de trabajo) están unidos por un medio de interconexión como una red ethernet. En estos sistemas la comunicación va a producirse efectivamente en forma secuencial porque sólo existe un camino físico por el que van a competir todos los módulos del sistema. Más aún, en este tipo de sistemas será irrelevante la noción de distancia entre procesadores mencionada anteriormente.

Por su parte, en todos aquellos sistemas donde la red de interconexión de los procesadores permite la ocurrencia simultáneamente de más de una comunicación, el modelo de función de coste más apropiado es el de la función *minimax*. Dicho modelo sería el adecuado tanto para sistemas donde exista una red de interconexión punto a punto que permita establecer distancias entre procesadores (sistemas tipo hipercubo, sistemas basados en Transputers, etc.), como aquellos sistemas donde los procesadores se ven todos ellos conectados a la misma distancia los unos de los otros (por ejemplo, sistemas tipo SP2 de IBM). A pesar de lo dicho, se han propuesto trabajos que se orientan a sistemas punto a punto pero han utilizado la función de coste *summed* [Ram88][Sad87][Erc90].

Además de las consideraciones referentes a la mayor adecuación teórica de una función de coste u otra para una determinada arquitectura paralela, la adopción de una determinada función también puede afectar al diseño de la estrategia de asignación concreta. Es decir, la función de coste puede influir en el algoritmo utilizado para calcular la asignación. El método propuesto en [Sad90] sería un ejemplo de tal influencia: su estrategia usa la función *summed* y se concentra sólo en el término de desbalanceo de carga entre procesadores durante el primer paso del algoritmo; en el segundo paso, ya considera de forma global toda la función intentando reducir el término debido a comunicaciones sin aumentar excesivamente el término de desbalanceo.

CAPÍTULO 3

Estrategia de asignación de tareas basada en agrupaciones

En los capítulos precedentes se ha dado una visión global de las distintas soluciones que se han propuesto para el problema del *mapping* y hemos prestado una atención especial a aquellas soluciones diseñadas para calcular la asignación de programas consistentes en un conjunto persistente de tareas y que, por lo tanto, exhiben un comportamiento que se ha modelado tradicionalmente como grafos de tipo TIG.

Vimos también que, descontando un reducido número de casos, no existe una solución óptima del problema, lo que ha provocado la proliferación de métodos heurísticos que buscan un compromiso entre calidad de la solución encontrada y tiempo necesario para su obtención. Dentro de las aproximaciones heurísticas existen también importantes discrepancias entre los distintos métodos en función del conjunto de factores que se han tenido en cuenta en la resolución del problema. El uso de distintas funciones de coste, las diversas consideraciones relativas a la arquitectura y las características propias del grafo de programa usado hacen difícil una comparación uniforme de las distintas estrategias y su uso como métodos universalmente aceptados.

Sin embargo, de lo expuesto en el capítulo precedente podemos deducir que las soluciones más efectivas para la búsqueda de soluciones de tipo más general para el problema del *mapping* se encuentra en la familia de los **métodos heurísticos mixtos**. En este caso entendemos la efectividad desde el punto de vista del compromiso que existe entre la calidad de la solución obtenida y el tiempo empleado en su obtención.

En este capítulo, presentaremos una estrategia de asignación de tipo mixto que se basa en una heurística de agrupación en la fase *greedy* y en un fase iterativa de la que se han diseñado dos versiones alternativas. La heurística propuesta considera que el sistema paralelo dispone de una topología totalmente interconectada. En el capítulo 5 se presentará la extensión a cualquier arquitectura.

Se estudiará la complejidad computacional de la estrategia completa y se analizarán algunas características importantes relacionadas con el proceso de agrupación y el proceso iterativo de refinamiento.

Por último, se comparará la estrategia con un método óptimo utilizando un conjunto de grafos de ejemplo que cubre un extenso abanico de posibilidades.

3.1 Antecedentes

De la estrategias mixtas analizadas creemos que algunas no son plenamente satisfactorias algunas porque presentan características que las descartan en su uso como estrategias generales. Es el caso de la estrategia de Efe, que puede presentar problemas de convergencia en función de un cierto parámetro para el que no se propone ningún criterio a la hora de fijarlo, o la propuesta de Chaundary que fija un modelo de precedencias en la ejecución de las tareas que condiciona a la política de asignación para evitar que se incurra en asignaciones que provoquen *deadlock.*

De las restantes estrategias mixtas podemos concluir que, en general, todas hacen un uso de estrategias *greedy* en la primera fase que o son muy simples (todas las utilizadas por Ercal, Sadayappan y Ramanujam) o que en determinados momentos toman decisiones que no tienen en cuenta la conectividad del grafo (caso de la de Wu). Este comportamiento entendemos que perjudica el resultado global de las asignaciones; perjuicio que puede traducirse en la obtención de soluciones más alejadas del óptimo o en un mayor tiempo consumido en la fase iterativa.

De hecho, en la literatura existe todo un conjunto de algoritmos que han sido propuestos para resolver el problema del *scheduling* y que bajo el nombre global de estrategias de agrupación (*clustering*) demuestran la importancia de tener en cuenta la conectividad del grafo en todos los pasos del algoritmo [Kim88][Sar89][Yan91][Wu88]. El problema que tratan de resolver las estrategias mencionadas no es el mismo que estamos tratando nosotros porque todas ellas se basan en el modelo de DAGs, pero sí que resulta interesante analizar su comportamiento para observar cuáles son las características que muestran las estrategias que obtienen mejores soluciones y qué razón hay para que ello ocurra. A partir de este análisis intentaremos extrapolar esas características para que sean aplicables a una estrategia diseñada para nuestro ámbito. Todas se basan en la supresión de un arco del grafo en cada paso (lo que ocasiona la agrupación de dos nodos), aunque cada una difiere en el criterio seguido para elegir el arco que en cada momento será candidato a ser suprimido. El objetivo final es conseguir un conjunto de nodos agrupados de tal forma que el tiempo necesario para la ejecución del grafo sea mínimo (suponiendo un número infinito de procesadores).

De la comparación realizada en [Ger92] se concluye que aquellas estrategias que, a medida que van realizando las operaciones de agrupación van actualizando la información referente al grafo, son las que obtienen al final una mejor solución. Por el contrario, las que obtienen peores resultados son las que, por ejemplo, ordenan los arcos al inicio del algoritmo y recorren esa lista empezando por el mayor arco sin tener en cuenta la evolución que experimenta el grafo a lo largo de los pasos de agrupación, ni variar en ningún momento el orden de examen.

Los criterios seguidos en todas estas estrategias para decidir cuándo interesa agrupar dos nodos no son aplicables cuando se usa un modelo basado en TIGs. De todos modos, lo que nos interesa más de este análisis consiste fundamentalmente en la

obtención de unas directrices generales que sean prometedoras para ser aplicadas a una estrategia de *mapping* que use el modelo de TIGs. Las características interesantes que debería aprovechar dicha estrategia son básicamente dos:

- El proceso de reducción del grafo debe estar guiado por la sucesiva supresión de arcos. Esta supresión también suele denominarse conversión a cero de arcos (*zeroing*).
- A medida que se vaya realizando dicho proceso de reducción se debe recalcular la información que se utiliza para determinar cuál es el siguiente arco que debe ser suprimido.

De estas dos características podemos deducir claramente que el algoritmo de *mapping* va a obtener una serie de agrupaciones en las cuales todos los nodos van a formar un subgrafo conexo, con lo que se habrán suprimido todas las comunicaciones que los unen. Esta es una característica deseable en la asignación final pues ya hemos comentado a lo largo de los primeros capítulos que uno de los objetivos de cualquier política de asignación es la reducción del *overhead* debido a comunicaciones.

Por otra parte, la segunda característica va a obligar al algoritmo a utilizar alguna estructura de datos donde las actualizaciones se puedan realizar de forma rápida y eficiente con objeto de conseguir un tiempo de ejecución lo más bajo posible. Por lo dicho en la primera característica, dicha estructura debe contener los arcos del programa que estarán además ordenados, de tal forma que permita la elección en cada momento del mejor candidato para ser suprimido.

Junto con los aspectos señalados se debe fijar también el criterio que en nuestro caso va a determinar la posible agrupación de nodos y que, en definitiva, va a servir para ordenar los arcos del grafo. Para ello debemos fijar previamente cuál será la función de coste adoptada como criterio de evaluación de las soluciones halladas. En nuestro caso optamos por la función ***minimax***, de la que ya se comentó que, en principio, parece la más indicada para evaluar un grafo asignado a una arquitectura donde las comunicaciones se pueden realizar concurrentemente.

3.2 Fundamentos de la estrategia de asignación propuesta

Antes de presentar el diseño completo de la estrategia heurística, analizaremos mediante un pequeño ejemplo la mecánica que se va a seguir para obtener la asignación de un cierto grafo. El funcionamiento básico de la estrategia se resume en ir reduciendo paulatinamente aquellos arcos de mayor comunicación, al tiempo que se intenta mantener equilibradas, desde el punto de vista de la función de coste *minimax,* las agrupaciones resultantes.

Supongamos el ejemplo de la figura 3.1.

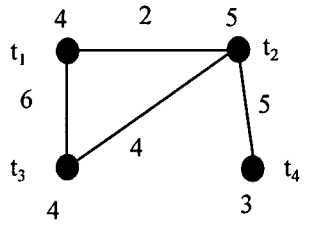


Figura 3.1 Grafo TIG

De forma análoga a la definición del coste de un procesador usada en el cálculo de la función *minimax* podemos definir una función que evalúa el coste de un nodo individual. Dado un nodo t_i, cost(t_i) se define como:

$$\cos t(t_i) = w_i + \sum_{i \neq j} c_{i,j}$$

La función cost(t_i), proporciona la suma del volumen de cómputo de un nodo más el volumen de comunicaciones de todos los arcos de dicho nodo.

Para los nodos de la figura tenemos:

cost (t_1) = 4 + (6 +2) = 12
cost (t_2) = 5 + (2 + 4 + 5) = 16
cost (t_3) = 4 + (6 + 4) = 14
cost (t_4) = 3 + (5) = 8

Con estos valores iniciales el valor de la función de coste *minimax* para este grafo es el que determina el nodo con mayor carga, en este caso el nodo t_2 que tiene coste 16, por lo que éste sería el valor de la función *minimax*.

Supongamos que a continuación hiciéramos la agrupación de los nodos n_2 y n_4. La nueva situación del grafo sería la que muestra la figura 3.2.

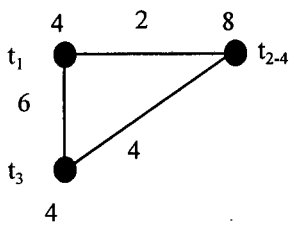


Figura 3.2

Si denominamos $v_{2,4}$ al arco que ha sido eliminado por la unión de los dos nodos, podemos asociarle un coste, que denominaremos coste de agrupación (cost_agrup), y que será:

$$cost\_agrup\ (v_{2,4}) = cost(n_2) + cost(n_4) - 2*c_{2,4}$$

Podemos formalizar este valor para cualquier arco $v_{i,j}$ del grafo como:

$$cost\_agrup\ (v_{i,j}) = cost\ (n_i) + cost(n_j) - 2*v_{i,j}$$

El valor de cost_agrup() para un determinado arco coincide con el valor de cost() del nodo resultante al suprimir dicho arco. En nuestro ejemplo,

$$cost\_agrup\ (v_{2,4}) = 16 + 8 - 2*5 = 14 = 8 + (2 + 4) = cost\ (t_{2\text{-}4})$$

El nuevo nodo resultante de la unión de t_2 y t_4 tiene un coste de 14, con lo que hemos reducido el coste global del grafo a ese mismo valor.

Cada uno de los arcos del grafo va a tener un cost_agrup asociado; para el grafo original de la figura 3.1 los valores son:

$$cost\_agrup\ (v_{1,2}) = 24$$
$$cost\_agrup\ (v_{1,3}) = 14$$
$$cost\_agrup\ (v_{2,3}) = 22$$
$$cost\_agrup\ (v_{2,4}) = 14$$

Como se puede ver, el valor de cost_agrup() para cada uno de los arcos nos está indicando, en definitiva, el interés que puede presentar su supresión. Así, por ejemplo, agrupar los nodos t_1 y t_2 va a suponer la creación de un nuevo nodo de coste 24, superando el valor que tenía la función de coste originalmente. Sin embargo, la agrupación de t_2 y t_4, y que muestra la figura 3.2, da lugar a una situación donde el coste de la función de coste se reduce a 14 unidades, pues el nodo t_2 era el más cargado originalmente y con esta agrupación se reduce su coste (el coste de los demás nodos queda inalterado).

Así pues, podemos esbozar ya cuál será el algoritmo *greedy* utilizado en la primera fase de nuestro estrategia mixta de *mapping*. En primer lugar, se considerará que todos los nodos del grafo constituyen agrupaciones unitarias y se calculará el valor de cost_agrup() para cada uno de los arcos. A partir de ahí, y de forma iterativa, se irán eliminando arcos del grafo y agrupando, por lo tanto, nodos del grafo, en función del valor de cost_agrup(). Para ello se seguirán dos criterios: escoger un arco del nodo con mayor coste, según la función cost(), de forma que la agrupación creada tenga una coste menor que el nodo original, o escoger, en su defecto, aquel arco con menor cost_agrup().

Es fácil comprobar que el número de pasos máximo que deberían ejecutarse del algoritmo suponiendo un grafo con N nodos y un sistema con K procesadores es N-K. Además el criterio de agrupación mencionado anteriormente presenta la ventaja de ser

capaz de producir agrupaciones donde la reducción que se haga del grafo sea incluso menor que el número de procesadores si con ello se consigue reducir el coste final de la asignación. Un ejemplo de esta propiedad la muestra la agrupación hecha en el grafo de la figura 3.1. Si suponemos un sistema con 4 procesadores, podemos pensar que basta con asignar un proceso a cada procesador. Sin embargo, usando la información que proporcionan los respectivos cost_agrup() se puede observar que en este ejemplo concreto resulta beneficioso agrupar los nodos t_2 y t_4 porque con ello se reduce el coste global de la asignación (de 16 a 14), de tal forma que la pérdida de paralelismo ocasionada por el uso de un procesador menos se ve compensada por la supresión de una comunicación relativamente costosa.

3.3 Descripción de la estrategia de asignación propuesta

La estrategia de asignación consiste en dos fases diferenciadas. La primera es la que se denomina fase de agrupación y genera una primera asignación de la tareas en *clusters* o agrupaciones. La segunda fase, denominada de reasignación, intenta iterativamente mejorar la situación generada por la fase de agrupación moviendo las tareas entre los *clusters* formados.

Primera fase: Agrupación

En el apartado anterior hemos esbozado el funcionamiento básico de lo que constituiría la fase *greedy* de nuestra estrategia de asignación. Esta fase es la que denominaremos fase de agrupación (*clustering*) y que identificaremos con las siglas CA (*Clustering Algorithm*).

Esquemáticamente, el algoritmo completo en pseudo-código que sigue la fase de agrupación se muestra en la figura 3.3.

Como se deduce del algoritmo anterior, a lo largo de su ejecución van a ser necesarias muchas operaciones de búsqueda y actualización, tanto en la lista de nodos como en la lista de arcos que en todo momento deben estar ordenadas. Para que estas operaciones se ejecuten con efectividad y mantengan los elementos ordenados, la implementación adoptada se basa en el uso de árboles equilibrados. Recordemos que un árbol equilibrado es aquel árbol binario para el que cada uno de sus nodos cumple que las alturas de sus dos subárboles difieren como mucho en la unidad [Wir80].

En un árbol equilibrado de n elementos se pueden realizar con un complejidad O(log n), incluso en el peor de los casos, las siguientes operaciones:

1. Encontrar un nodo con una clave dada.
2. Insertar un nodo con una clave dada.
3. Borrar un nodo con una clave dada.

El uso de árboles equilibrados va a agilizar la selección en cada paso del algoritmo del arco que debe eliminarse en el grafo. Bastaría precisar que en la

ordenación de los elementos se utiliza el valor de cost() para ordenar los nodos y el valor de cost_agrup() para ordenar los arcos; en caso de igualdad, el orden se fija simplemente por el identificador del propio nodo o de los identificadores de los dos nodos que están en los extremos del arco, respectivamente. Esta decisión implica, en la práctica, que en caso de igualdad de valores, el algoritmo no utiliza ningún criterio de segundo orden significativo para tomar la decisión del elemento a considerar.

```
Para todos los nodos t_i del grafo original hacer
        calcular cost(t_i)
Para todos los arcos v_i,j del grafo original hacer
        calcular cost_agrup(v_i,j)
Ordenar los nodos según cost(t_i)
Ordenar arcos según cost_agrup(v_i,j)
Para I = 1 hasta N-K hacer {
        t_k := Nodo con mayor cost()
        arco_candidato := NULL
        cost_arco_candidato := ∞
        Para todos los arcos v_i,k hacer {
                Si (cost_agrup (v_i,k) < cost (t_k))
                   y (cost_agrup (v_i,k) < cost_arco_candidato))  entonces {
                        arco_candidato := v_i,k
                        cost_arco_candidato := cost_agrup (v_i,k)
                }
        Si (arco_candidato = NULL) entonces
                arco_candidato := Arco con menor cost_agrup()
        Eliminar (arco_candidato)
        Actualizar lista de cost()
        Actualizar lista de cost_agrup()
}
mejora := TRUE
Hacer {
        t_k := Nodo con mayor cost()
        arco_candidato := NULL
        cost_arco_candidato := ∞
        Para todos los arcos v_i,k hacer {
                Si (cost_agrup (v_i,k) < cost (t_k))
                   y (cost_agrup (v_i,k) < cost_arco_candidato))  entonces {
                        arco_candidato := v_i,k
                        cost_arco_candidato := cost_agrup (v_i,k)
                }
        Si (arco_candidato = NULL) entonces
                mejora := FALSE
        sino {
                Eliminar (arco_candidato)
                Actualizar lista de cost ()
                Actualizar lista de cost_agrups ()
        }
mientras mejora
```

Figura 3.3 Algoritmo de agrupación (CA)

Para mostrar la aplicación de este algoritmo a un TIG concreto, consideremos el grafo de la figura 3.4 y veamos cúales son los resultados generados por la fase de agrupación si suponemos que disponemos de 6 procesadores. El conjunto de 15 nodos iniciales ha sido reducido a seis agrupaciones finales, siendo el coste final de esta fase de 200 (coste de la agrupación T5-T6-T8-T9). En la tabla 3.1 se muestran los sucesivos pasos de agrupación realizados, los nodos agrupados, el valor de cost_agrup() para el arco suprimido y el valor de la función de coste al final de ese paso. La figura 3.5 muestra el TIG resultante.

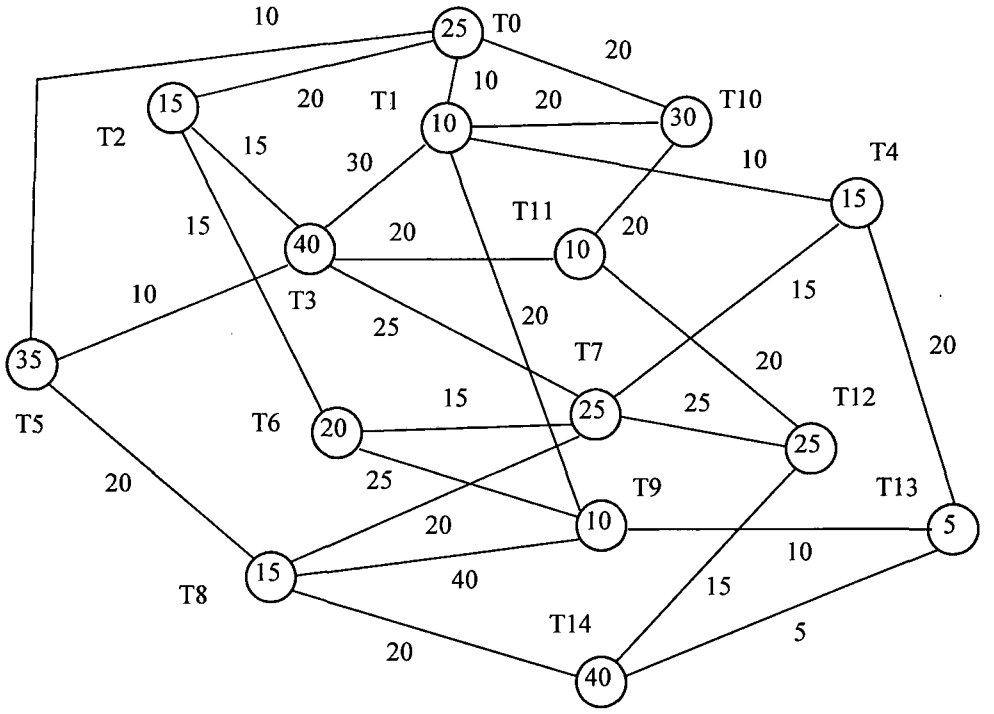


Figura 3.4 Ejemplo TIG

Tabla 3.1: Pasos de agrupamiento del grafo de la figura 3.4			
Paso	Nodos agrupados	valor cost_agrup()	valor *minimax*
1	T_4 y T_{13}	60	125
2	T_{11} y T_{12}	95	125
3	T_0 y T_2	110	125
4	$T_{4\text{-}13}$ y T_{14}	130	130
5	T_6 y T_9	130	130
6	T_{10} y $T_{11\text{-}12}$	145	145
7	T_5 y T_8	150	150
8	T_1 y T_3	160	160
9	$T_{5\text{-}8}$ y $T_{6\text{-}9}$	200	200

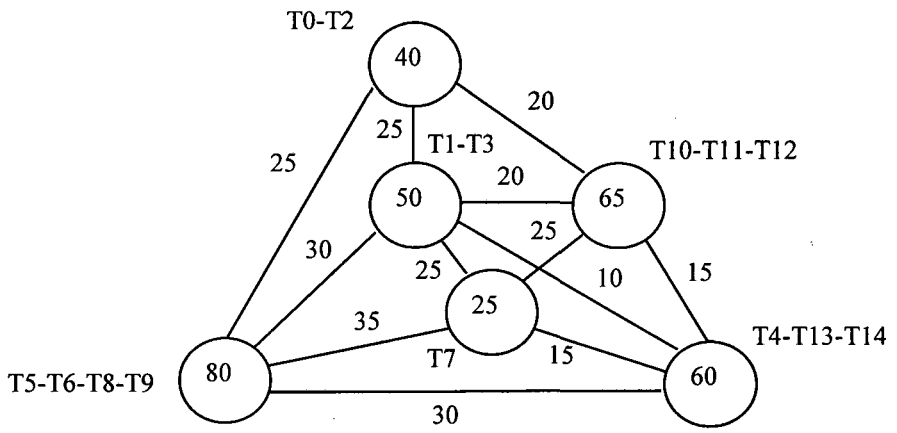


Figura 3.5 TIG después de la fase de agrupación

Segunda fase: Reasignación

Al final de esta primera fase de asignación se obtendrá una asignación subóptima que en la mayoría de casos podrá ser mejorada mediante la realización de movimientos o intercambios de tareas entre las distintas agrupaciones. Esta segunda fase de mejora es la que se corresponde con el esquema de un método iterativo. Hemos desarrollado dos versiones alternativas para esta fase:

- una que sólo efectúa movimientos individuales de tareas. El nombre que se le ha dado a esta alternativa es CRM (*Clustering and Reassignment by Movements*).
- otra que efectúa movimientos individuales, como en el caso anterior, e intercambios. Esta alternativa la denominamos CRME (*Clustering and Reassignment by Movements and Exchanges*).

El algoritmo que describe el movimiento de tareas individuales se muestra en la figura 3.6. Por su parte, el algoritmo específico para realizar intercambios de parejas de nodos se muestra en la figura 3.7.

En ambos algoritmos se sigue una misma estrategia de optimización aunque cada uno de ellos tiene algunos aspectos diferenciales. En primer lugar, se busca la agrupación que tiene mayor coste y se intenta mover o intercambiar, respectivamente, cualquiera de las tareas que contiene esa agrupación siempre y cuando se consiga reducir el coste de la misma. La tarea seleccionada será aquella que produzca una mayor disminución en el valor de la función de coste, valorando también en caso de igualdad, aquel cambio que empeore menos el valor de la función de coste de la otra agrupación que interviene en el cambio (este detalle no se ha reflejado en los mencionados algoritmos para no complicarlos excesivamente).

```
Procedimiento movimientos_individuales (No_movimientos)
Hacer {
        No_movimientos := TRUE
        clust_k := Agrupacion con mayor cost()
        Para todos los nodos t_i ∈ clust_k hacer
        {
                nuevo_coste_k := evaluar_coste (clust_k - t_i)
                nuevo_coste := ∞
                Si (nuevo_coste_k < cost(clust_k)) entonces {
                        Para todas las agrupaciones clust_j ≠ clust_k hacer {
                                nuevo_cost_i := evaluar_coste (clust_j + n_i)
                                Si (nuevo_coste_i < clust_k)
                                y (nuevo_cost_i < nuevo_coste) entonces {
                                        nuevo_coste := min (nuevo_cost_i,
                                                        nuevo_coste_k)
                                        movimiento_encontrado := TRUE
                                        movimiento := (t_i, clust_j)
                                }
        }
        Si (movimiento_encontrado) entonces {
                mover (movimiento)
                Actualizar lista de cost ()
                No_movimientos := FALSE
        }
} hasta No_movimientos
```

Figura 3.6 Procedimiento para movimientos individuales

En el caso de los movimientos, cada tarea de la agrupación más cargada se intenta mover a todas las demás agrupaciones, siempre y cuando su marcha de la agrupación actual sea beneficiosa para ella. De todos los posibles movimientos se escoge el mejor según el criterio enunciado anteriormente. En el caso de los intercambios se intenta intercambiar todas las tareas de la agrupación más cargada con todas las tareas de las otras agrupaciones, aplicando al final el mismo criterio anterior para seleccionar el mejor intercambio.

Después de haber encontrado un movimiento o un intercambio, el proceso se repite a partir de aquella agrupación que sea la más cargada en ese momento. En el caso de no encontrar ninguna mejora para una determinada agrupación cabría la posibilidad de intentar hacer movimientos o intercambios entre agrupaciones que no son la que en este momento corresponde a la más cargada y, por lo tanto, define el valor de la función de coste. Este opción se podría hacer con la esperanza de que un cambio a este nivel secundario permitiese volver a intentar cambios en esa agrupación más cargada. Esta mejora, sin duda podría aportar ligeras reducciones en la función de coste final aunque entendemos que los haría a costa de un incremento en el tiempo de cómputo que probablemente no compensaría la ganancia obtenida, tal y como se demuestra con los resultados experimentales conseguidos con la versión actual.

```
Procedimiento intercambios_parejas (No_intercambios, No_mejora)
Hacer {
        No_intercambios := TRUE
        clust_k := Agrupacion con mayor cost()
        nuevo_coste_hallado := ∞
        Para todos los nodos t_i ∈ clust_k hacer
        {
                Para todas las agrupaciones clust_j ≠ clust_k hacer {
                        Para todos los nodos t_h ∈ clust_i hacer {
                                nuevo_coste_k := evaluar_coste (clust_k - t_i + t_h)
                                nuevo_coste_i := evaluar_coste (clust_j + t_i - t_h)
                                nuevo_coste := max (nuevo_cost_i, nuevo_coste_k)
                                Si (nuevo_coste < cost(clust_k))
                                y (nuevo_coste_i < nuevo_coste_hallado) entonces {
                                        nuevo_coste_hallado := nuevo_coste
                                        intercambio_encontrado := TRUE
                                        intercambio := (t_i, clust_k, t_h, clust_j)
                                }
        }
        Si (intercambio_encontrado) entonces {
                intercambiar (intercambio)
                Actualizar lista cost ()
                No_intercambios := FALSE
                No_mejora := FALSE
        }
} hasta No_intercambios
```

Figura 3.7 Procedimiento para intercambio de nodos

En el caso de la alternativa CRM, el algoritmo completo se consigue mediante la pura concatenación de los algoritmos mostrados en las figuras 3.3 y 3.6. En el caso del método CRME, el algoritmo CA va seguido de una iteración que realiza secuencias de movimientos, según el método de la figura 3.6, seguidos de secuencias de intercambios, según el método de la figura 3.7. Esa iteración termina cuando el algoritmo de intercambios no consigue ninguna mejora. En la figura 3.8 se muestra el algoritmo seguido por CRME en la fase iterativa.

```
Hacer {
        Movimientos_individuales (No_movimientos)
        No_mejora := TRUE
        Intercambios_parejas (No_intercambios, No_mejora)
} hasta No_mejora
```

Figura 3.8 Fase iterativa del método CRME

Aplicando la fase iterativa al ejemplo de la figura 3.5, el resultado final sería el movimiento de la tarea T5 hacia la agrupación formada por T0-T2, consiguiendo reducir el coste de la asignación a 165 (coste de {T0-T2-T5}).

3.4 Análisis de complejidad

Para analizar la complejidad algorítmica del método propuesto, supongamos que disponemos de un grafo de N nodos y M arcos. El algoritmo de agrupación (CA) va a realizar un conjunto de iteraciones acotada por N pues a cada paso reduce en uno el número de nodos del grafo. La reducción de un nodo implica una búsqueda del nodo más cargado, dos operaciones de eliminación de nodos y una operación del nuevo nodo resultado de la agrupación. Cada una de estas operaciones tiene una complejidad O(log N) si mantenemos la lista de nodos ordenada mediante un árbol balanceado; por lo tanto, las operaciones en la lista de nodos tendrán esa misma complejidad. Por lo que respecta a la lista de arcos, si suponemos una situación típica donde todos los nodos tienen un conjunto de arcos acotado en r, podemos deducir que el mayor tiempo se va a consumir en las 2r operaciones de borrado que se necesitan y en las r operaciones de inserción, cada una de ellas de complejidad O(log M), usando también un árbol balanceado para la lista de arcos. Tendremos 3r operaciones de complejidad O(logM) durante N pasos. Este será el factor determinante en la complejidad del algoritmo con lo que la complejidad global de la primera fase será O(NlogM).

Existiría una situación en la que la complejidad del algoritmo sería algo peor si consideramos un grafo totalmente conexo. En este caso el número de arcos M es de orden N^2 lo que implicaría que la complejidad total del algoritmo podría llegar a ser $O(N^2 \log N)$. Sin embargo, en la práctica las aplicaciones paralelas donde todos las tareas se comunican con todas las demás suelen ser minoritarios y, en general, se cumple que una cierta tarea se comunica con un reducido número de tareas vecinas [And91], lo que permitiría aceptar como válida la suposición que hemos hecho para acotar el número de arcos que tiene cada nodo.

Por lo que se refiere a la fase de movimientos individuales, la evaluación de los movimientos de un nodo suponen un coste de O(N) [Fid82]. Suponiendo que se haga un número bajo de movimientos y, en consecuencia, se evalúe un número reducido de movimientos, la complejidad de esta fase será O(N). Para los intercambios, su evaluación tiene un coste $O(N^2)$ [Sel92]. Esta será la complejidad de esta fase si, de nuevo, el número de intercambios realizados es bajo. Estas complejidades se verán multiplicadas por un factor N si el número de movimientos o intercambios es elevado. En la práctica, los resultados experimentales obtenidos confirmaron que en la mayoría de ejemplos probados el número de movimientos e intercambios que se realizaban era suficientemente pequeño y, por lo tanto, la complejidad temporal de la fase iterativa se mantenía en niveles bajos. Los tiempos experimentales se detallarán en el siguiente capítulo.

3.5 Cota aproximada para el número de movimientos

Aunque, en general, resulta imposible predecir teóricamente el número de movimientos que va a realizar el algoritmo, en las fases experimentales realizadas se ha comprobado que el mayor número se producía cuando el tipo de grafo utilizado mostraba una estructura regular. La razón para ello hay que buscarla en el hecho de que la regularidad del grafo, y la relación entre su número de nodos y el número de procesadores van a provocar situaciones donde el algoritmo de agrupamiento ejecuta un número de pasos que provoca un resultado final donde hay un desequilibrio entre nodos con un exceso parecido de carga y nodos con un defecto parecido de carga, como se verá a continuación. El análisis de estas situaciones nos permitirá proponer una cota para aproximar el número de movimientos realizados en ellas.

Supongamos un conjunto de K procesadores y un grafo de N nodos con un coste global α (cómputo + comunicaciones) cada uno de ellos. Supongamos también por simplicidad que al agrupar dos de estos nodos el coste del nodo resultante tiene coste 2α.

Después de N/2 pasos el número de nodos habrá quedado reducido a la mitad y cada uno de los nodos presentes en el grafo reducido tendrá un coste 2α. Este proceso de división por dos del número de nodos y duplicación de las carga de cada uno de ellos se irá repitiendo a lo largo del proceso de agrupación.

El número de pasos de duplicación de carga vendrá determinado por $D = \log_2 N/K$. Si N/K es una potencia exacta de 2, el valor D será un número entero y el resultado de la fase de agrupación será perfecto, en los demás casos se producirá el desequilibrio antes mencionado y D deberá aproximarse por el entero inmediatamente anterior, o sea,

$$D = \lfloor \log_2 N/K \rfloor$$

Tomando este valor de D tenemos el número de pasos hechos en los que se ha duplicado perfectamente la carga de los nodos, que será $2^D\alpha$.

El número de procesos a los que habremos reducido el grafo original será $K' = N/2^D$.

Entonces, si K' = K habremos acabado. Si no, deberemos hacer todavía unos cuantos pasos de agrupamiento para reducir K' a K. El número de pasos será:

$$K' - K = N/2^D - K$$

Para cada uno de estos pasos tendremos un nuevo nodo con carga $2^{D+1}\alpha$ (el doble de la carga que tenían hasta este momento que era de 2^D). Redondeando los valores a números enteros, tendremos, por lo tanto, $\lceil N/2^D - K \rceil$ procesadores con un exceso de carga igual a $2^{D+1}\alpha - N\alpha/K$, siendo $N\alpha/K$ la distribución de carga perfectamente balanceada. En un caso general, habrá que tomar un valor entero para aproximar esta carga balanceada: $\lfloor N\alpha/K \rfloor$.

El equilibrio de carga se conseguirá moviendo este exceso a los otros procesadores que estén por debajo del valor promedio. Por lo tanto, podemos establecer una cota aproximada para este número de movimientos que permiten conseguir el equilibrio de carga en:

$$num\_movs = \left(\left\lceil N/2^D \right\rceil - K \right) * \left(2^{D+1} - \left\lfloor N/K \right\rfloor \right)$$

A continuación, mostraremos sobre un ejemplo concreto la aplicación de la fórmula anterior. Partimos de un grafo original de N = 20 nodos de carga idéntica (α) como el que muestra la figura 3.9(a) que debe asignarse a un sistema de K = 4 procesadores. En este ejemplo obviamos la existencia de comunicaciones entre los nodos para mantener la hipótesis hecha en el razonamiento precedente que suponía que la unión de dos nodos implicaba la aparición de un nodo de carga doble. En nuestro caso, el número de pasos en los que se produce una duplicación perfecta de carga es (fig. 3.9 (b)):

$$D = \lfloor \log_2 N/K \rfloor = 2$$

En este punto el número de agrupaciones es de 5, por lo que será necesario realizar un paso adicional de agrupación que reduzca el grafo a 4 (figura 3.9. (c)). En consecuencia, el número de procesadores con exceso de carga es:

$$\left(\left\lceil N/K \right\rceil - K \right) = 1$$

El exceso de carga de esa agrupación es:

$$\left(2^{D+1} - \left\lfloor N/K \right\rfloor \right) = 3$$

Por lo tanto, el número de movimientos que deberán realizarse para equilibrar la carga entre todos los procesadores es

num_movs = 1 * 3 = **3**

Después de realizar esos 3 movimientos, se corrige el desequilibrio provocado en la fase de agrupación y se alcanza la situación final reflejada en la figura 3.9(d).

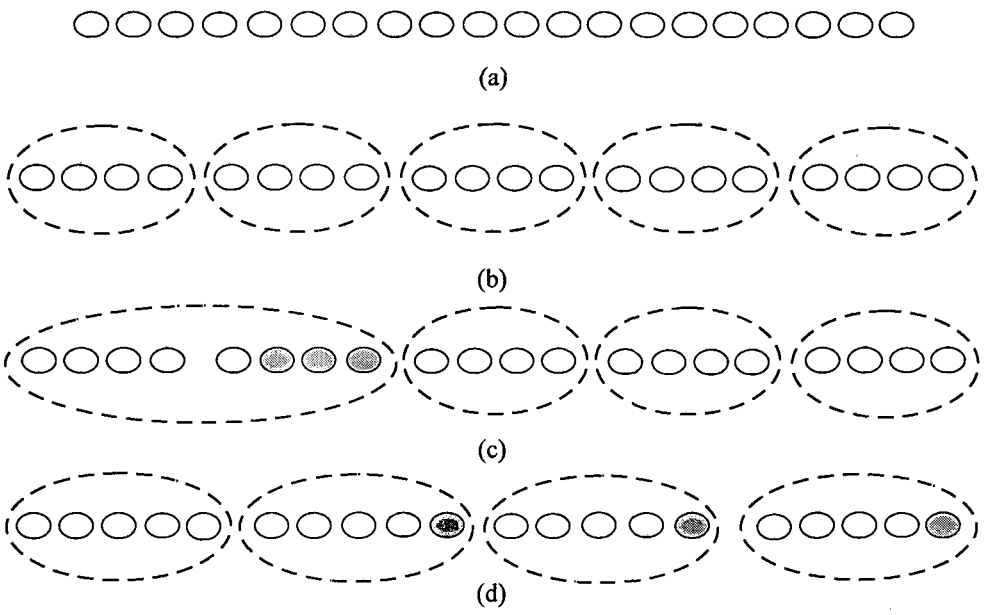


Figura 3.9 Ejemplo de agrupación de grafo regular y su posterior equilibrado

La fórmula que se ha presentado se basa en aquella situación ideal donde haya una perfecta duplicación de carga al ir agrupando nodos. En la práctica, este tipo de situaciones no suele darse y el número de movimientos siempre suele ser inferior debido a factores como la disparidad de volúmenes de cómputo y comunicaciones, y las distintas interconexiones que muestran los nodos del grafo. Sólo en el caso de grafos con una estructura regular, aunque dependiendo también del tipo de grafo concreto, se puede producir un número de movimientos que, como veremos en el capítulo cuatro, se acerca al número de movimientos previsto en la fórmula anterior. Sirva, en cualquier caso, el razonamiento realizado en este apartado para explicar el hecho, un tanto sorprendente a priori, de que se efectúen más movimientos al asignar un grafo regular que uno irregular, cuando un primer juicio intuitivo nos haría afirmar justo lo contrario.

3.6 La bondad de las estrategias CA, CRM y CRME frente al óptimo

Una vez planteada en sus términos algorítmicos la estrategia de *mapping* pasaremos a mostrar en este apartado los resultados obtenidos en una primera experimentación llevada a cabo con objeto de probar la bondad del método en sus tres versiones: CA, CRM y CRME. Como toda heurística pensada para conseguir soluciones subóptimas para un amplio abanico de problemas de entrada a los que no se les impone ninguna restricción especial, no resulta fácil a priori predecir cuán bueno o malo será su comportamiento. En la literatura existen tres aproximaciones seguidas por los distintos autores para aseverar la bondad de una determinada heurística:

1. Comparar sus soluciones con las obtenidas por un método óptimo. Obviamente este tipo de experimentación limita el tamaño de los ejemplos

probados dada la naturaleza exponencial de cualquier algoritmo que encuentre soluciones óptimas.

2. Comparar sus soluciones con las obtenidas por algún otro método óptimo de reconocida fiabilidad (por ejemplo, el ya mencionado Simulated Annealing). Este tipo de experimentación permite trabajar ya con ejemplos de mayor tamaño.
3. Utilizar la estrategia para calcular la asignación de determinadas aplicaciones reales y, a partir de ahí, ejecutar (o simular la ejecución) de dichas aplicaciones y obtener alguna medida de rendimiento (tipo speed-up) de la aplicación asignada de ese modo. Este tipo de experimentación suele realizarse con un conjunto muy reducido de ejemplos y sólo suele evaluar el rendimiento de la estrategia al aumentar el número de procesadores o su rendimiento frente a una colocación de tipo aleatorio.

En nuestro caso, se han llevado a cabo distintas fases de experimentación entre las arriba mencionadas con objeto de tener una mayor certeza para corroborar la validez del método propuesto.

En primer lugar veremos en este capítulo la comparación realizada con una estrategia de tipo óptimo. El algoritmo óptimo pertenece a la categoría de Branch & Bound y, en nuestro caso concreto, se basa en ir recorriendo el árbol de soluciones haciendo una búsqueda en profundidad prioritaria, descartando un determinado subárbol en el momento en que la solución parcial que se está construyendo tiene un valor en su función de coste peor que la mejor solución hallada hasta el momento.

La propia naturaleza de los métodos Branch & Bound les hace especialmente adecuados para su diseño recursivo. Así ocurre en nuestro caso como podemos comprobar en la figura 3.10 que reproduce el procedimiento principal encargado de hacer el recorrido por el espacio de soluciones. En la llamada a dicho procedimiento se le pasan dos parámetros que corresponden a un número de tareas a asignar (n) y el número de procesador (p). A partir de estos parámetros el procedimiento generará iterativamente todas las combinaciones de los n elementos cogidos de k en k (para k = 1..n), calculará el valor de coste de una de esas combinaciones y si está por debajo del mayor coste obtenido hasta el momento y quedan tareas por asignar (n-k > 0) hará una llamada recursiva con valores n-k y p+1. Cuando un procesador haya asignado todas las tareas que recibió y su combinación siga teniendo un coste menor que la mejor solución hasta el momento se habrá encontrado una solución mejor que será almacenada y cuyo coste servirá de valor de referencia para continuar la búsqueda. La primera llamada empieza pasándole las N tareas al procesador 1.

El proceso de búsqueda obedece fundamentalmente a las directrices mencionadas anteriormente aunque se han incluido unas cuantas mejoras para reducir el número de soluciones exploradas, sin que ello signifique, por supuesto, que el algoritmo deje de tener complejidad exponencial:

- se ha hecho uso de un algoritmo muy efectivo para generar las combinaciones de n tareas cogidas de k en k que no hace uso de recursividad [Oli93].

- las combinaciones que prueba cada uno de los procesadores tiene un límite inferior que consiste en el número de tareas que él recibió dividido por el número de procesadores que quedan para que éstas sean asignadas, él incluido. Este valor correspondería a la situación en la que todos los procesadores considerados tengan el mismo número de tareas. De este modo se evita la generación de soluciones simétricas: por ejemplo, para dos procesadores, si el primero genera todas las combinaciones de n elementos tomados de n-3 en n-3 y el segundo evalúa las agrupaciones de 3 tareas, no es necesario que el segundo pruebe las combinaciones de n elementos cogidos de n-3 en n-3 y el primero haga lo propio con las de 3 elementos porque es una situación simétrica ya evaluada.
- una vez encontrada una solución mejor se vuelve al punto de recursión correspondiente al procesador con un valor p menor cuya agrupación tiene el mismo coste que la solución hallada.

```
Procedimiento busqueda (n, p) {
        limite_inferior = (n/numero_procesadores - p + 1)
        Para i = n hasta limite_inferior hacer {
                num_combinaciones = calcular_combinaciones (n, i)
                Para j = 1 hasta num_combinaciones hacer {
                        combinación = generar_combinación (n. i)
                        coste_combinacion = evaluar (combinación)
                        Si (coste < optimo) entonces
                                Si (n - k > 0) entonces
                                        busqueda (n-k, p+1)
                                sino {
                                        actualizar_mejor_solución ()
                                        actualizar_optimo (optimo)
                                }

                }
        }
}
```

Figura 3.10 Algoritmo óptimo de asignación

La experimentación que se llevó a cabo para comparar el método heurístico propuesto frente a un método óptimo se basó en el usó de dos tipos de grafos: grafos irregulares sin ningún patrón topológico simétrico o regular, y grafos regulares que sí exhibían dicha simetría topológica [Sen96].

3.6.1 Resultados experimentales con grafos irregulares

Para la realización de esta fase experimental se generaron un total de 72 grafos con las siguientes características:

- Número de nodos: variable entre 15, y 20.
- Relación entre el número de arcos y el número de nodos variable:
  - grafos dispersos (*sparse*) : la razón entre número de arcos y número de nodos está en el intervalo [1,2). En consecuencia, son grafos que exhiben un nivel de conectividad bajo entre los nodos.
  - grafos medios: la razón está en el rango [2,4).
  - grafos densos: la razón es mayor que 4. El nivel de conectividad entre los nodos en este caso es elevado.
- Granularidad del grafo variable:
  - grafos de grano grueso (*coarse*): el valor de granularidad estaba en el rango 5-15. Son grafos donde su cómputo es substancialmente mayor a sus comunicaciones.
  - grafos de grano medio: el valor de granularidad está en el rango 0.8-1.2. Los costes de cómputo y comunicación son muy parecidos.
  - grafos de grano fino (*fine*): el valor de granularidad está en el rango 0.2-0.05. Los costes de comunicación son substancialmente mayores que los de cómputo.

En nuestro caso, definimos granularidad del grafo como la proporción que existe entre el promedio de los volúmenes de cómputo y el promedio de los volúmenes de comunicación. Este parámetro aproxima de algún modo la relación que existe entre el cómputo que realiza una tarea y las comunicaciones en las que se ve implicada. Esta definición se aproxima más a la que da Gerasoulis en [Ger93] frente a la utilizada por otros autores que dividen el volumen de cómputo por el total de comunicaciones del nodo. Sin embargo, creemos que la definición adoptada en nuestro caso permite por una parte, analizar por separado la influencia de la densidad o grado de conectividad del grafo y, por otra, la relación existente entre el valor del cómputo de un nodo y el valor de sus comunicaciones.

El conjunto total de grafos estaba repartido equitativamente en todas las categorías enunciadas anteriormente, de forma que existían 18 grafos para cada número de nodos, de los que 6 eran dispersos, 6 eran medios y 6 eran densos, y en cada uno de estos subgrupos había 2 grafos de grano grueso, 2 de grano medio y 2 de grano fino.

Los valores correspondientes al volumen de cómputo de los nodos y al volumen de comunicación de sus correspondientes arcos se generaron aleatoriamente en el rango [1,500]. Las conexiones entre los nodos también se generaron aleatoriamente.

Para todos los grafos se calculó su asignación utilizando 2, 4 y 8 procesadores, con objeto de aumentar, por un lado, el número de pruebas y poder comprobar, por otro, si las distintas estrategias presentaban algún comportamiento dependiente del número de procesadores.

Un primer resultado que se obtuvo permitió comprobar que para la práctica totalidad de grafos de grano fino y grafos densos de granularidad media la asignación óptima era aquella que dejaba todos los nodos en una única agrupación. Este resultado era predecible a priori siempre que el valor de R fuese cercano a 0.05 o cercano a 0.2 y el grado de conectividad fuera elevado; sin embargo, la experimentación confirmó

también que incluso los grafos con valor R cercanos a 0.2 de baja conectividad y los de grano medio con alta conectividad experimentaban el mismo fenómeno. Fenómeno que no hace sino corroborar la idea de que la ejecución en paralelo de determinadas aplicaciones que no presentan una granularidad suficientemente elevada puede resultar peor que su ejecución secuencial.

Para el resto de casos los resultados obtenidos por cada una de las estrategias se muestra, respectivamente, en las tablas 3.2, 3.3 y 3.4 (estos resultados se han calculado a partir de los datos contenidos en las tablas del Apéndice B). Recordemos que la fase sólo de agrupación la denominamos estrategia CA, el método que usa sólo movimientos en la fase de mejora se denomina CRM y la estrategia que incluye también intercambios la denominamos CRME. En las tablas se muestra la distribución obtenida del valor M = Tp/To, donde Tp es el valor de la función minimax obtenida por la estrategia heurística y To es el valor obtenido por el método óptimo. Para las asignaciones óptimas se obtiene M = 1,0. La columna *Pro* indica el número de procesadores utilizado y la columna *Granul*, el tipo de grafo en función a su granularidad.

Tabla 3.2: Distribución de M = Tp/To. Estrategia **CA.** Grafos irregulares

			Porcentaje de simulaciones					
Tipo de grafo	Granul	Pro.	= 1.0	≤ 1.1	≤ 1.2	≤ 1.3	≤ 1.4	>1.4
dispersos	media	2	12.5%	25%	100%	100%	100%	100%
		4	0%	62.5%	87.5%	100%	100%	100%
		8	50%	87.5%	100%	100%	100%	100%
	gruesa	2	0%	12.5%	87.5%	100%	100%	100%
		4	0%	0%	37.5%	75%%	100%	100%
		8	0%	16.7%	50%%	100%	100%	100%
	Subtotal		10.8%	34.8%	78.2%	95.6%	100%	100%
medios	media	2	75%	87.5%	100%	100%	100%	100%
		4	0%	37.5%	75%	100%	100%	100%
		8	0%	62.5%	87.5%	100%	100%	100%
	gruesa	2	0%	0%	62.5%	100%	100%	100%
		4	0%	0%	12.5%	50%	100%	100%
		8	0%	0%	25%	62.5%	100%	100%
	Subtotal		12.%	31.3%	60.4%	85.4%	100%	100%
densos	gruesos	2	0%	37.5%	100%	100%	100%	100%
		4	0%	0%	37.5%	87.5%	100%	100%
		8	0%	0%	50%	100%	100%	100%
	Subtotal		0%	12.5%	62.5%	96%	100%	100%
TOTAL			35%	50%	73.5%	88.5%	93.8%	100%

Tabla 3.3: Distribución de M = Tp/To. Estrategia **CRM.** Grafos irregulares							
	Porcentaje de simulaciones						
Tipo de grafo	Granul	Pro.	= 1.0	≤ 1.1	≤ 1.2	≤ 1.3	≤ 1.4
dispersos	media	2	50%	75%	100%	100%	100%
		4	62.5%	87.5%	100%	100%	100%
		8	50%	87.5%	100%	100%	100%
	gruesa	2	12.5%	100%	100%	100%	100%
		4	12.5%	87.5%	100%	100%	100%
		8	12.5%	37.5%	100%	100%	100%
	Subtotal		33.3%	79.2%	100%	100%	100%
medios	media	2	75%	100%	100%	100%	100%
		4	12.5%	100%	100%	100%	100%
		8	25%	87.5%	100%	100%	100%
	gruesa	2	12.5%	100%	100%	100%	100%
		4	0%	62.5%	100%	100%	100%
		8	0%	50%	62.5%	75%	100%
	Subtotal		20.8%	85.4%	93.8%	95.8%	100%
densos	gruesos	2	12.50%	87.5%	100%	100%	100%
		4	12.5%	87.5%	100%	100%	100%
		8	0%	62.5%	100%	100%	100%
	Subtotal		8.3%	79.2%	100%	100%	100%
TOTAL			23.3%	81.7%	97.5%	98.3%	100%

Tabla 3.4: Distribución de M = Tp/To. Estrategia **CRME.** Grafos irregulares					
	Porcentaje de simulaciones				
Tipo de grafo	Granul.	Proc.	= 1.0	≤ 1.1	≤ 1.2
dispersos	media	2	50%	75%	100%
		4	62.5%	75%	100%
		8	62.5%	87.5%	100%
	gruesa	2	12.5%	100%	100%
		4	25%	100%	100%
		8	37.5%	83.3%	100%
	Subtotal		41.7%	87.5	100%
media	media	2	75%	100%	100%
		4	25%	100%	100%
		8	25%	100%	100%
	gruesa	2	25%	100%	100%
		4	0%	100%	100%
		8	0%	75%	100%
	Subtotal		25%	95.8%	100%
densos	gruesa	2	25%	100%	100%
		4	25%	100%	100%
		8	37.5%	100%	100%
	Subtotal		29.2%	100%	100%
TOTAL			32.5%	93.3%	100%

Los resultados que reflejan las tablas nos permiten concluir que tanto las estrategias CRM como CRME alcanzan unos resultados globales muy próximos al óptimo: concretamente, en un 81,7% de los casos para CRM y en un 93.3% para

CRME, la solución hallada no difería en más de un 1.1 respecto al valor óptimo; cifras que se sitúan en el 97.5% y 100%, respectivamente, para una diferencia de 1.2 respecto al valor óptimo. Los resultados de CA, a pesar de ser algo peores, son también significativamente buenos porque consiguen no alejarse en ningún caso más de 1.4 respecto de las asignaciones óptimas.

En general, el número de procesadores a los que se asigna el grafo no tiene, en principio, una influencia reseñable para ninguna estrategia. En cambio, sí que se observan variaciones en función de la granularidad del grafo. En primera instancia, se constata que suelen aparecer más resultados óptimos o más cercanos al óptimo en el caso de grafos de granularidad media (especialmente constatable en la estrategia CA). De todos modos, una vez pasado por la fase de reasignación los resultados para ambos tipos de grafo suelen igualarse, llegando en el caso de la estrategia CRME a ser ligeramente mejor el resultado final para los grafos de granularidad gruesa. Por último, por lo que respecta al grado de conectividad del grafo los resultados de todas las estrategias tienden a conseguir más casos óptimos cuanto menos denso es el tipo de grafo. A medida que nos alejamos de los casos óptimos, las diferencias en función de la densidad del grafo se reducen significativamente y es difícil establecer si existe algún tipo de densidad para el cual alguna estrategia muestre siempre un comportamiento mejor.

3.6.2 Resultados experimentales para grafos regulares

Los resultados obtenidos usando grafos aleatorios confirmaron inicialmente la bondad del método propuesto aunque la propia naturaleza aleatoria de los grafos puede interpretarse como un arma de doble filo: por una parte, se cubre un amplio espectro en la caracterización de los grafos en lo que se refiere a granularidades y conectividades, pero, por otro lado, resulta difícil establecer una correspondencia entre esos grafos y aplicaciones reales. Para utilizar grafos que tengan una mayor correspondencia con las estructuras que aparecen en aplicaciones reales nos hemos basado en grafos regulares que ciertos autores presentan en algunos de sus trabajos [Lo91] [Chau93][And91]. En general, los tipos de grafo regular que hemos usado (véase figura 3.11) corresponden a tres grandes grupos:

- Mallas: suelen ser la estructura topológica de muchas aplicaciones orientadas a la manipulación de matrices, problemas de elementos finitos o tratamiento de imágenes. En nuestros ejemplos utilizamos 2 tipos: la malla abierta y la malla cerrada.
- Árboles: estas estructuras suelen aparecer en muchos algoritmos de ordenación o de búsqueda, o algoritmos de tipo *divide-and-conquer* y *master-slave*. En nuestros ejemplos usamos las estructuras correspondientes al árbol binario, el árbol binomial, la estructura fork-join y el árbol de tipo in-out.
- Anillos: en esta categoría podemos incluir todas aquellas aplicaciones que efectúan un procesamiento de los datos a través de una cadena de procesos siguiendo un modelo semejante al *pipeline*. Hemos usado tres topologías de este tipo: la cadena simple (*pipe*), el anillo y el anillo n-body (característico del problema del mismo nombre).

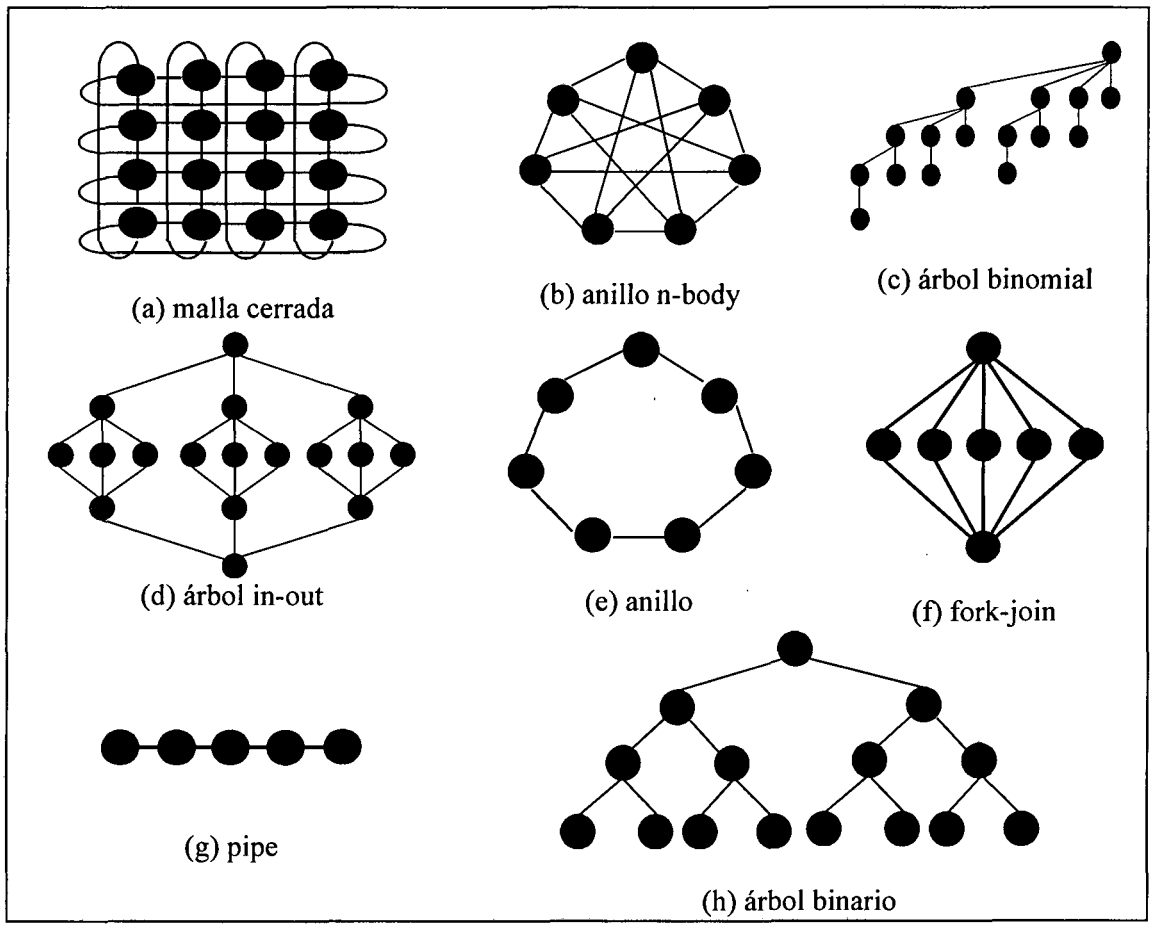


Figura 3.11. Muestra de grafos regulares

Se usaron en total 9 topologías distintas de grafos regulares y para cada una de ellas se generaron 4 grafos distintos (dos de granularidad media y dos de granularidad gruesa, según las definiciones indicadas en el apartado anterior). Para todos ellos se volvió a comparar las soluciones obtenidas por la estrategia heurística frente a las del método óptimo y los resultados se resumen en la tablas 3.5, 3.6 y 3.7.

Tabla 3.5: Distribución de M = Tp/To. Estrategia **CA** Grafos regulares

			Porcentaje de simulaciones					
Tipo de grafo	Granul	Pro.	= 1.0	≤ 1.1	≤ 1.2	≤ 1.3	≤ 1.4	>1.4
árboles	media	2	87.5%	100%	100%	100%	100%	100%
		4	62.5%	75%	87.5%	100%	100%	100%
		8	62.5%	75%	75%	87.5%	100%	100%
	gruesa	2	25%	25%	25%	50%	75%	100%
		4	25%	25%	50%	50%	50%	100%
		8	25%	25%	25%	25%	50%	100%
	Subtotal		47.9%	54.2%	60.4%	68.8%	79.2%	100%
mallas	media	2	100%	100%	100%	100%	100%	100%
		4	50%	50%	50%	75%	100%	100%
		8	50%	50%	100%	100%	100%	100%
	gruesa	2	50%	100%	100%	100%	100%	100%
		4	50%	75%	100%	100%	100%	100%
		8	50%	100%	100%	100%	100%	100%
	Subtotal		58.3%	79.2%	91.2%	95.8%	100%	100%
anillos	media	2	83.3%	83.3%	100%	100%	100%	100%
		4	83.3%	100%	100%	100%	100%	100%
		8	100%	100%	100%	100%	100%	100%
	gruesa	2	100%	100%	100%	100%	100%	100%
		4	100%	100%	100%	100%	100%	100%
		8	100%	100%	100%	100%	100%	100%
	Subtotal		94.4%	97.2%	100%	100%	100%	100%
TOTAL			65.7%	74.1%	80.6%	85.2%	90.7%	100%

Tabla 3.6: Distribución de M = Tp/To. Estrategia **CRM**. Grafos regulares

			Porcentaje de simulaciones		
Tipo de grafo	Granul	Pro.	= 1.0	≤ 1.1	≤ 1.2
árboles	media	2	87.5%	100%	100%
		4	87.5%	87.5%	100%
		8	87.5%	100%	100%
	gruesa	2	50%	100%	100%
		4	75%	100%	100%
		8	100%	100%	100%
	Subtotal		81.2%	97.9%	100%
mallas	media	2	100%	100%	100%
		4	75%	75%	100%
		8	50%	50%	100%
	gruesa	2	50%	100%	100%
		4	50%	75%	100%
		8	50%	50%	100%
	Subtotal		62.5%	83.3%	100%
anillos	media	2	100%	100%	100%
		4	100%	100%	100%
		8	100%	100%	100%
	gruesa	2	100%	100%	100%
		4	100%	100%	100%
		8	100%	100%	100%
	Subtotal		100%	100%	100%
TOTAL			83.3%	95.4%	100%

Tabla 3.7: Distribución de M = Tp/To. Estrategia **CRME**. Grafos regulares					
	Porcentaje de simulaciones				
Tipo de grafo	Granul	Pro.	= 1.0	≤ 1.1	≤ 1.2
árboles	media	2	87.5%	100%	100%
		4	87.5%	87.5%	100%
		8	87.5%	100%	100%
	gruesa	2	75%	100%	100%
		4	100%	100%	100%
		8	100%	100%	100%
	Subtotal		89.6%	97.9%	100%
mallas	media	2	100%	100%	100%
		4	75%	75%	100%
		8	75%	100%	100%
	gruesa	2	50%	100%	100%
		4	50%	100%	100%
		8	100%	100%	100%
	Subtotal		75%	95.8%	100%
anillos	media	2	100%	100%	100%
		4	100%	100%	100%
		8	100%	100%	100%
	gruesa	2	100%	100%	100%
		4	100%	100%	100%
		8	100%	100%	100%
	Subtotal		100%	100%	100%
TOTAL			89.8%	98.1%	100%

En el caso de los grafos con estructuras regulares los resultados mejoran, en general, respecto a los que se obtuvieron con los grafos irregulares. Todas las estrategias aumentan el número de casos donde coinciden con las soluciones del algoritmo óptimo de asignación y se reducen también las diferencias respecto al óptimo cuando no se consigue encontrar la misma solución. Por ejemplo, CA consigue asignaciones óptimas en un 65.7% de los casos, CRM lo consigue en un 83.3% y CRME lo hace en un 89.8%. Además, en el caso de los grafos con topología de anillos las tres estrategias consiguen resultados óptimos (CRM y CRME) o prácticamente óptimos (CA con un 94.4%).

La única salvedad a esa regla la constituye el algoritmo CA cuando intenta asignar grafos de la categoría de árboles. Cuando el tipo de grafo presenta un número elevado de nodos extremos (que sólo contienen una conexión), los resultados finales de la fase de agrupación pueden llegar a desviarse por encima del 1.5 respecto al óptimo. De todos modos este fenómeno no es excesivamente preocupante porque la estrategia completa que se propone en este trabajo es de tipo mixto y, por lo tanto, siempre contiene una fase de refinamiento, ya sea de movimientos o de movimientos e intercambios, y en cualquiera de esos casos se corrige la desviación generada por el algoritmo CA para los grafos con esta característica topológica.

Como conclusión del presente capítulo las figuras 3.12 y 3.13 muestran gráficamente los resultados conjuntos de las tres estrategias para grafos irregulares y regulares , respectivamente (las gráficas se han construido a partir de los valores de los datos reflejados en la fila **TOTAL** de las anteriores tablas. En ambas gráficas se puede apreciar la bondad de las diferentes estrategias reflejada en su rápida aproximación al nivel 100

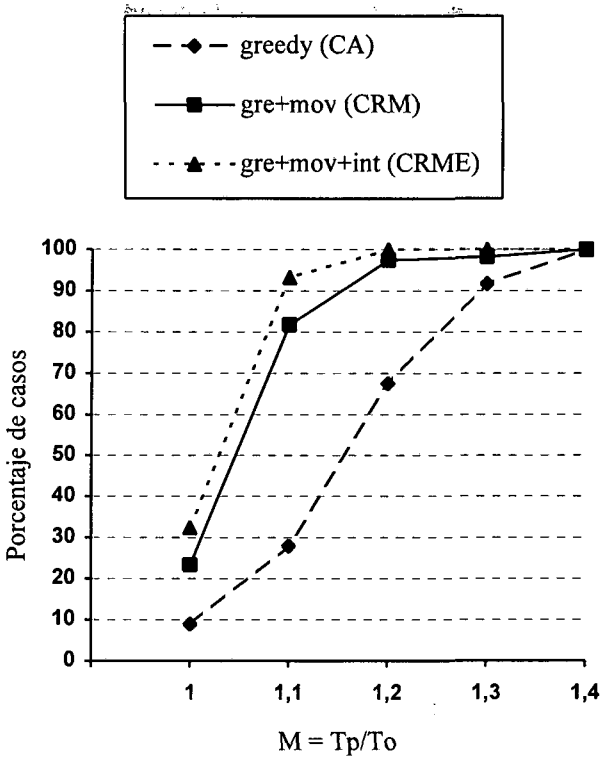


Figura 3.12 Comparación con el óptimo para grafos irregulares

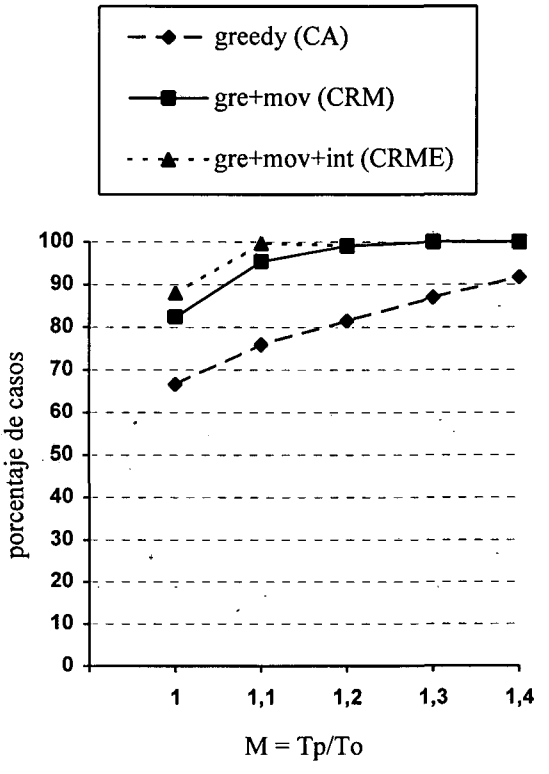


Figura 3.13 Comparación con el óptimo para grafos regulares

CAPÍTULO 4

Estudio comparativo de estrategias de asignación

En capítulo anterior se ha presentado una estrategia mixta para resolver el problema del *mapping* y se ha visto una primera validación experimental de la misma comparándola con los resultados obtenidos por una estrategia óptima de asignación. Dicha validación se realizó mediante un conjunto de grafos que incluía tanto grafos regulares como irregulares, y que presentaban también distintas granularidades. De todas formas, todos los ejemplos utilizados tenían un número reducido de nodos debido a la naturaleza exponencial del método óptimo. Una validación más completa del método propuesto debería, por lo tanto, basarse en el uso de ejemplos de mayor tamaño aunque ello obligue a comparar sus resultados con los obtenidos por otras estrategias heurísticas que también obtengan soluciones subóptimas.

En este capítulo vamos a presentar el conjunto de experimentos realizados a tal fin donde se han utilizado, por una parte estrategias heurísticas de tipo *greedy*, de tipo iterativo y de tipo mixto. De este modo se ha pretendido cubrir un amplio espectro de estrategias que va desde los métodos de muy baja complejidad (heurísticas *greedy*), los de complejidad relativamente alta (heurísticas iterativas) y los de complejidad intermedia (heurísticas mixtas). Lógicamente, la calidad de la soluciones será proporcional a la complejidad requerida por el algoritmo utilizado.

Los grafos utilizados han vuelto a ser de tipo regular y de tipo irregular. En el caso de los grafos regulares se usaron las estructuras enunciadas en el capítulo anterior aumentando el número de nodos y manteniendo la diferenciación referente a granularidad. Por su parte, los grafos irregulares no se generaron aleatoriamente sino que se obtuvieron a partir de DAGs correspondientes a aplicaciones reales que fueron reducidos a grafos TIGs.

La experimentación se centra en la comparación de las asignaciones obtenidas por cada una de las estrategias de la literatura, junto con un estudio de los tiempos de ejecución de cada una de ellas. Por último, se realiza un análisis comparativo más detallado de las fases iterativas de las estrategias mixtas.

4.1 Comparación con estrategias *greedy* e iterativas

Para realizar la comparación de nuestras estrategia con métodos *greedy* e iterativos representativos nos basamos en el uso de varios métodos integrados en el entorno ALTO desarrollado por el equipo del instituto francés IMAG [Bou95]. Se usaron cuatro estrategias para realizar la comparación. Dos de ellas eran de tipo *greedy* y dos de tipo iterativo. En esta experimentación se pretenden cubrir dos grandes objetivos: por una parte, se cotejará la estrategia *greedy* que usa información sobre la conectividad del grafo frente otras estrategias *greedy* que no lo hacen y, por otro lado, se evaluará la estrategia global frente a otras estrategias que son computacionalmente mejores y peores que ella. En este sentido, a las heurísticas de tipo iterativo se les permitió realizar una búsqueda intensiva de soluciones que intentase refinar al máximo su calidad aunque fuese a costa de alargar su tiempo de cómputo. En muchos casos, las mismas heurísticas podían encontrar en menor tiempo soluciones de coste ligeramente peor, sin embargo, la opción escogida nos sirvió para que las asignaciones que hallaban sirviesen como cota subóptima superior a la que debería acercarse nuestra estrategia.

Las dos estrategias *greedy* eran la LPTF (*Largest Processing Time First*) y la LGCF (*Largest Global Cost First*), y las dos iterativas fueron la *Simulated Annealing* y la búsqueda tabú. A continuación, veremos con mayor detalle las características de cada una de ellas.

Largest Processing Time First (LPTF)

Este método ordena inicialmente las tareas en función del volumen de cómputo y, a partir de ahí, las va escogiendo en orden decreciente. La primera tarea se asigna al primer procesador y las restantes se van asignando sobre el procesador menos cargado en cada paso. Este es un algoritmo muy rápido (de complejidad O(NlogN)) que garantiza un buen reparto del cómputo total entre todos los procesadores pero que en presencia de grafos de tipo TIG con comunicaciones no suele proporcionar buenos resultados porque no las tiene en cuenta.

Largest Global Cost First (LGCF)

Este método es similar al anterior en su filosofía aunque intenta obtener mejores resultados teniendo en cuenta la influencia de las comunicaciones. Para cada nodo se calcula se coste global (debido a cómputo + comunicaciones) y se los ordena posteriormente en función de ese coste. Como en el caso anterior, la tarea con mayor coste se asigna al primer procesador y después se va asignando las restantes tareas siguiendo por orden decreciente de coste sobre el procesador menos cargado en cada paso. Este método también tiene una complejidad muy baja (O(NlogN)) y en presencia de grafos con comunicaciones obtiene mejores resultados que el método anterior.

Simulated Annealing (SA)

En el capítulo 2 ya se mencionó que este método es una técnica de optimización de propósito general utilizada por diversos autores y se hizo una descripción general del

mismo. A continuación, detallaremos el algoritmo en el que se basa la versión de SA adaptada a la resolución del problema del *mapping*.

En la tabla 4.1. aparecen las definiciones usuales en el algoritmo de *Simulated Annealing* y la figura 4.1. muestra la estructura de dicho algoritmo

Tabla 4.1. Definiciones del algoritmo SA

Parámetros	Tipo	Significado
T	real	Temperatura del sistema
T_{min}	real	Temperatura final
equilibrio_local	función	Determina si se ha alcanzado una situación de equilibrio local
perturbar (S)	función	Alteración de la configuración actual
S	estructura	Configuración actual (describe una asignación completa)
S_{new}	estructura	Nueva configuración
C_{new}	entero	Valor de la función de coste de la nueva situación generada
ΔC	entero	Variación en la función de coste
gen_unif (0,1)	función	Generación de un número aleatorio entre 0 y 1
actualizar (T)	función	Actualiza el nuevo valor de la variable T

El método empieza con una configuración inicial aleatoria S_0, que tiene un coste asociado C_0. Una nueva configuración, S, se genera a partir de una perturbación en S_0, proporcionando un nuevo coste, C. Se calcula la variación en el coste, $\Delta C = C - C_0$; si $\Delta C < 0$, se acepta la nueva configuración; si $\Delta C \geq 0$ entonces la nueva configuración se acepta con una probabilidad $e^{-\Delta C/T}$, donde T es el parámetro que controla la acción de *hill-climbing* dentro del algoritmo.

El algoritmo se caracteriza por los siguientes elementos:

- una función perturbar (S): permite generar nuevas configuraciones a partir de la situación actual. Las operaciones que tradicionalmente constituyen esta función son el movimiento de tareas individuales entre dos procesadores o el intercambio de tareas entre dos procesadores.
- la función actualizar (T): es la función que actualiza el parámetro T y que típicamente es de la forma

$$T = \alpha(T) * T$$

  donde α es función de la temperatura y cumple $0 < \alpha(T) < 1$. En la práctica se ha observado que valores de α alrededor de 0.95-0.98 proporcionan buenos resultados de forma consistente; en nuestro caso, se usó el valor 0.98.
- la función equilibrio_local: determina la condición de finalización del bucle interno para pasar a actualizar la temperatura actual.
- el valor T_{min}: es el criterio que marca la finalización del algoritmo.

```
T := T0
S := S0
C := C0
mientras ( T > Tmin) hacer {
        mientras (not equilibrio_local) hacer {
                Snew := perturbar (S)
                Cnew := estimar coste de la configuración Snew
                ΔC := Cnew - C
                si ΔC < 0 entonces {
                        S := Snew
                        C := Cnew
                } sino {
                        r := gen_unif (0, 1)
                        si r < e^(-ΔC/T) entonces {
                                S := Snew
                                C := Cnew
                        }
                }
        }
        T := actualizar (T)
}
```

Figura 4.1 Algoritmo de SA

El criterio de equilibrio local usado en el bucle interno suele especificarse de dos formas:

1. Indicando un cierto número de iteraciones que hará dicho bucle, es decir, indicando el número de perturbaciones que van a probarse para una temperatura dada. Este valor suele ser un múltiplo (M) de N (número de tareas del grafo). Se fija como M*N. Se ha observado experimentalmente que a medida que el valor M se incrementa a partir de un valor pequeño como 0.01, la calidad de las soluciones obtenidas van mejorando hasta alcanzar un cierto valor de M, a partir del cual no se producen mejoras significativas en la calidad de las soluciones. Lógicamente, el tiempo de ejecución será claramente proporcional al valor usado de M.
2. Indicando un número de perturbaciones aceptadas a una cierta temperatura; dado que este criterio es difícil de satisfacer a bajas temperaturas, se suele fijar adicionalmente una cota de movimientos intentados. Así, el bucle interno termina cuando se hayan aceptado un cierto número de perturbaciones (Ma*N) o cuando se hayan intentado un número suficientemente grande de ellas (Mt*N).

En nuestro caso, se usaba M = 10.

El criterio de finalización (T_{min}) acostumbra a ser un valor "bajo" de la temperatura, o sea, un valor positivo pequeño (p. ej., 2^{-31}) para el que la probabilidad de aceptar una perturbación de peor coste sea extremadamente pequeña. Se puede observar que si T fuese fijada a un valor infinito, el algoritmo SA se comportaría como un recorrido totalmente aleatorio por el espacio de soluciones; y si T fuese cero, no se efectuaría ningún tipo de acción de *hill-climbing*, lo que daría lugar a un algoritmo de búsqueda local. Este parámetro suele fijarse, por ejemplo, de forma que una perturbación que incremente la función de coste sea aceptada con probabilidad 2^{-31}. En la versión de SA utilizada, el criterio de finalización se podía fijar también indicando un tiempo máximo de cómputo, aunque no se utilizó tal posibilidad y se dejó siempre que el algoritmo realizase su proceso de optimización hasta alcanzar la temperatura mínima.

Por último, la temperatura inicial T_0 se fija habitualmente de forma que exista una probabilidad entre el 80% y el 90% de aceptar para un incremento medio de la función de coste considerando todas las posibles perturbaciones resultantes de un movimiento simple en la configuración inicial. En nuestro caso se situó en el 90%.

El conjunto de parámetros se fijó de tal forma que el algoritmo realizaba un número de iteraciones superior a 40000 para grafos de tamaño superior a 200 nodos, de forma, que según los resultados de [Bou95] el algoritmo suele encontrarse en la zona en la que la búsqueda ha convergido hacia la mejor solución. Adicionalmente, la versión de SA utilizada, a diferencia del algoritmo básico recogido en la figura 4.1, proporcionaba siempre la mejor solución encontrada a lo largo de toda la búsqueda y no el valor final de S, que podía ser peor.

Búsqueda tabú (TS)

Los elementos fundamentales en los que se basa esta técnica de optimización ya fueron presentados en el capítulo 2. Como en el caso de SA detallaremos aquí algunas características particulares de la versión del algoritmo utilizado en nuestros experimentos.

En la tabla 4.2 aparecen las definiciones usuales en el algoritmo de búsqueda tabú y la figura 4.2 muestra la descripción en pseudo-código de dicho algoritmo.

Tabla 4.2. Definiciones del algoritmo de búsqueda tabú

Parámetros	Tipo	Significado
tabu_continua	lógico	Indica si debe continuar la búsqueda
gen_config_ini	función	Generación de una configuración inicial
mejor_movim	(tarea,dest)	Mejor vecino de una configuración dada
vecinaje (conf)	función	Genera de manera sucesiva los vecinos de una configuración
coste (mov)	función	Proporciona el valor de la función de coste al realizar un movimiento
no_es_tabu (mov)	función	Verifica que el vecino no esté en la lista tabú
criterio_aspirac (mov)	función	Verificación del criterio de aspiración
actualizar_tabu (mov)	función	Incluir configuración en la lista tabu
actualizar_cont (tabu_continua)	función	Verificar el interés de continuar la búsqueda

A continuación, se detallan los aspectos fundamentales que caracterizan el algoritmo de búsqueda tabú utilizado [Bou95]:

- las relaciones de vecindad utilizadas incluían tanto movimientos individuales como intercambios de tareas.
- los elementos de la lista tabú contienen la descripción referida al movimiento de una cierta tarea a un procesador. Para disminuir el número de posibles situaciones de vecindad sólo se guarda información correspondiente al volumen de cómputo de la tarea.
- el tamaño de la lista tabú es proporcional al tamaño del grafo de entrada, valor fijado a partir de los resultados experimentales mostrados en [Bou95], que evitaban las posibilidades de caer en búsquedas cíclicas.
- la finalización de la búsqueda se podía producir por tres causas:
    1. cuando todos los vecinos de una cierta configuración se encuentran en la lista tabú y no se puede mejorar la función de coste.
    2. después de un cierto número de iteraciones sin mejorar la mejor solución encontrada hasta el momento.
    3. fijando un tiempo máximo.

En nuestros experimentos se fijó un tiempo máximo de finalización de 90 minutos, aunque, en la práctica, la finalización del algoritmo se producía siempre por la primera de las condiciones enunciadas.

```
generar_config_inicial (conf_actual)
tabu_continua := TRUE
mientras (tabu_continua) hacer {
        mejor_movim := NULL
        para todos los movimientos en vecinaje(conf_actual) hacer {
                si ((coste (movimiento) < coste (mejor_movim))
                   y (no_es_tabu(movimiento))
                   o (criterio_aspirac (movimiento))
                        mejor_movimiento := movimiento
                }
       hacer_movimiento (mejor_movimiento)
       actualizar_tabu (mejor_movimiento)
       actualizar_cont (tabu_continua)
        }
        T := actualizar (T)
}
```

Figura 4.2 Algoritmo de búsqueda tabú

4.2 Comparación con otras estrategias mixtas

Además de realizar la comparación con estrategias *greedy* e iterativas que han servido para probar el rendimiento de la estrategia propuesta frente a métodos computacionalmente mejores y peores respectivamente, también se ha realizado una experimentación destinada a comparar nuestro método con otro método mixto. En este caso ambas estrategias van a tener unas complejidades similares por lo que la elección de una de ellas para su uso práctico no dependería, a priori, de un criterio de velocidad de ejecución. La diferencia entre ambos métodos va a venir de la calidad de las soluciones obtenidas.

En el capítulo 2 se vieron todas las estrategias que se han propuesto dentro de la categoría de métodos mixtos. Sin embargo, no todos los métodos presentados serían candidatos atractivos para una posible comparación. Las razones para ello son diversas:

- Algunos métodos utilizan parámetros de los que se desconocen los criterios para fijar su valor. Estos parámetros en algún caso determinan la convergencia del algoritmo: es el caso del método de Efe [Efe82] y el factor de tolerancia δ que sirve para catalogar a los procesadores de sobre o infracargados. En otros casos, el parámetro va a influir en el resultado final: es el caso del método de Ercal de bipartición recursiva [Erc90], donde utiliza un factor de tolerancia para controlar que las particiones del grafo tengan un volumen de cómputo semejante (la desviación de cada partición respecto a la media debe estar por debajo de esa tolerancia). Aunque en este último caso se menciona un ejemplo de factor de tolerancia del 5%, no se puede saber si ese fue el valor con el que se ejecutaron sus experimentos y qué implicaciones tiene sobre el método el uso de valores mayores o menores.

- Muchos métodos utilizan funciones de coste diferentes a la minimax (en general, suelen usar alguna versión de la función *summed*) y eso repercute de forma más o menos directa en el diseño del algoritmo utilizado, ya que el uso de ese método implicaría un rediseño total de la estrategia para adaptarla al uso de la función minimax. Esto ocurre con los métodos propuestos por Ercal [Erc90] y Sadayappan [Sad90], donde el uso de la función *summed* da lugar a dos estrategias donde la optimización de uno de los términos (equilibrio de carga o minimización de las comunicaciones) se realiza implícitamente por la estrategia algorítmica. En el caso de las estrategias de Agarwal [Lee87][Cha93], la situación se complica debido a la consideración de un modelo que incluye factores adicionales como las fases de comunicación o las prioridades de ejecución de las tareas que convierten sus métodos en casos únicos. Tan sólo uno de sus pasos de asignación es extrapolable con facilidad para ser usado en métodos de asignación que no consideren esos factores.

- En algunos casos las estrategias tienen una dependencia importante de la arquitectura o del tipo de aplicación. Algo de esto ocurre en las estrategias de Sadayappan. La de *nearest-neighbor* [Sad87] está condicionada fuertemente para ser aplicada con grafos que exhiban una fuerte localidad de

comunicaciones y que vayan a asignarse a arquitecturas de tipo malla. La de bipartición recursiva está orientada únicamente a sistemas con un número de procesadores igual a una potencia exacta de 2 [Erc90], y en particular, una de las versiones se orienta exclusivamente a hipercubos [Sad90].

Los argumentos expuestos han motivado que la estrategia finalmente escogida dentro del grupo de métodos mixtos haya sido la propuesta recientemente por Wu [Wu94]. El método que él propone usa originalmente la función *minimax*, no contiene ningún parámetro difícil de determinar y la influencia de la arquitectura sólo se refleja en una fase posterior. En esta última fase, Wu propone una reconfiguración estática de la red de forma que la topología de ésta se parezca tanto como sea posible a la topología que presenta el grafo del programa una vez reducido a tantos nodos como procesadores. Finalmente, se determina cuáles son los arcos del grafo del programa que no pueden coincidir directamente con enlaces de la arquitectura y para los cuales será necesario realizar un encaminamiento de sus mensajes. Si obviamos esta última fase, los primeros pasos realizados por la estrategia de Wu tienen un gran parecido con los que ejecuta nuestro método: por un lado, intenta reducir el grafo original a un grafo que tenga sólo tantas agrupaciones de nodos como procesadores y, por otro lado, define un método *greedy* seguido de uno iterativo para llevar a cabo esta reducción.

La diferencia con este método radica fundamentalmente en la primera fase porque la fase iterativa se reduce a efectuar iterativamente movimientos de tareas individuales entre las distintas agrupaciones generadas inicialmente. Por este motivo, a continuación vamos a analizar sólo el método *greedy*. Para la fase iterativa nos hemos basado en el algoritmo presentado en la sección 3.2. Tan solo se ha incorporado una variación a esta fase iterativa que es la inclusión de una fase de intercambios que su autor no considera en su propuesta original y que simplemente ha permitido comparar ambos métodos cuando hacían uso de esta técnica de mejora que podemos considerar que es la más completa y que proporciona los mejores resultados.

El algoritmo *greedy* de Wu se muestra en la figura 4.3. Suponiendo N tareas y K procesadores, inicialmente toma las K tareas con mayor coste global para que sean el embrión de las respectivas K agrupaciones que finalmente serán asignadas a cada uno de los procesadores. Una vez hecho esto, el algoritmo busca a cada paso cuál es la agrupación menos cargada y le asigna aquella tarea de entre todas las que todavía no han sido asignadas que suponga un incremento menor en el coste global de la agrupación.

Esta estrategia *greedy* la define Wu como *Even Distribution* (ED) y en la fase experimental se analizó separadamente los resultados que obtenía este método individualmente. Lógicamente, también se analizaron los resultados que obtenía la estrategia global, que denominamos con las siglas EDTR (*Even Distribution with Task Reassignment*).

```
Para todos los nodos t_i del grafo original hacer
        calcular cost(t_i)
Ordenar los nodos según cost(t_i)
Para i = 1 hasta K hacer
        P_i := {t_i}
R := {t_i+1, t_i+2, ..., t_N}
Para I = 1 hasta N-K hacer {
        P_j := Buscar procesador con menor cost()
        cost_nodo_candidato := ∞
        Para todos los nodos t_j de R hacer {
                Si (coste_agrupacion (t_j, P_i) < cost_nodo_cadidato) entonces {
                        nodo_candidato := t_j
                        cost_nodo_candidato := coste_agrupacion (t_j, P_i)
                }
        P_i := P_i + {nodo_candidato}
        R := R - {nodo_candidato}
        Actualizar coste (P_i)
}
```

Figura 4.3 Algoritmo greedy de la heurística propuesta por Wu

4.3 Estudio experimental y resultados

Se realizaron un conjunto de experimentos para comparar la nueva estrategia propuesta en este trabajo (en sus tres diferentes versiones) con todas las estrategias que se han mencionado en este capítulo y que incluyen tanto métodos *greedy* puros, iterativos puros y mixtos [Sen97]. La descripción general del conjunto de experimentos llevados a cabo es la siguiente:

- En el grupo de grafos regulares se utilizaron, por una parte, los mismos grafos regulares pequeños comentados en el capítulo anterior y grafos regulares de mayor tamaño con 7 estructuras (árbol binario árbol in-out, malla abierta, malla toroidal, *pipe*, anillo y anillo n-body). Este segundo grupo constituía un total de 24 grafos. La mitad de esos grafos presentaba granularidad media y el resto era de granularidad gruesa. El número de nodos era de 400 por término medio (aunque podía variar dependiendo del tipo de estructura regular).
- Se utilizaron grafos irregulares que provenían de DAGs correspondientes a aplicaciones reales y que fueron convertidos a TIG mediante el método DSC, que se comentará en más detalle en el apartado que describe el tercer experimento.
- El número de procesadores se varió en 2, 4 y 8 cuando se usaron grafos regulares pequeños, y entre 8, 16 y 32 al usar grafos regulares grandes o los grafos irregulares. De esta forma se ampliaba el abanico de resultados obtenidos y se podía constatar también si las heurísticas mostraban algún

comportamiento particular al ir variando el número de procesadores sobre el que se hacía la asignación.

- Se realizó una comparación de las distintas estrategias con respecto a la calidad de la solución dividida en tres experimentos:
    1. Comparación de las estrategias con el método óptimo asignando grafos regulares pequeños. Este experimento permitió establecer también la peor estrategia que se usaría como referencia en los restantes experimentos.
    2. Comparación de las estrategias al asignar grafos regulares grandes.
    3. Comparación de las estrategias al asignar grafos irregulares.
- En el caso de las estrategias mixtas se hizo un análisis más detallado de las respectivas fases iterativas.
- Se hizo, por último, una comparación de las estrategias con respecto al tiempo de cómputo consumido por cada una de ellas para encontrar la solución.

4.3.1 Experimento 1. Comparación de las estrategias con el óptimo para grafos regulares

El primer experimento se basó en la comparación con el método óptimo de todas las estrategias *greedy* e iterativas y las CA, CRM y CRME propuestas utilizando un conjunto de 42 grafos regulares pequeños. En cierta medida el experimento sería ilustrativo de cuál es el comportamiento de cada estrategia frente a un método óptimo usando grafos de tamaño reducido y poder comparar posteriormente el comportamiento relativo entre las distintas estrategias cuando se aumenta el tamaño de los grafos.

La tabla 4.3 muestra los resultados globales de éste experimento (obtenidos a partir de los datos del apéndice B). En la tabla las columnas **MD** y **CR** contienen los resultados promedio para grafos de granularidad media y gruesa respectivamente, la columna **PRO** contiene el promedio para una familia de grafos de una topología dada y la columna **Total** muestra el resultado global de cada estrategia para todos los grafos utilizados. Para cada asignación de un cierto grafo se computó el cociente Tp/To (donde Tp es el coste obtenido por la heurística en cuestión y To es el coste de la asignación óptima). En la tabla se muestra en % el resultado promedio de esos cocientes para todo el conjunto de grafos de una cierta clase.

Dado que las estrategias de *Simulated Annealing* (SA) y *Tabú Search* (TS) parten de configuraciones iniciales aleatorias, cada uno de estos métodos se aplicó cinco veces para cada uno de los ejemplos y el valor promedio de la función de coste obtenida en esas cinco ejecuciones fue el valor que se tomó como resultado del método para ese ejemplo.

Tabla 4.3. Porcentaje de aproximación a una asignación óptima para grafos regulares										
Topología	ARBOLES			MALLAS			ANILLOS			
Granular.	MD.	CR.	PRO.	MD.	CR.	PRO.	MD.	CR.	PRO.	Total
LPTF	65.5	91.1	78.3	67.7	91.2	79.5	64.5	91.9	78.2	78.7
LGCF	81.5	96.4	89.0	81.0	80.6	80.8	82.3	93.1	87.7	85.8
TS	90.0	98.4	94.2	87.4	94.6	91	89.2	98.1	93.7	93.0
SA	97.0	100	98.5	99.6	100	99.8	98.8	100	99.4	99.2
CA	95.6	76.4	86	92.4	97.4	94.9	99.3	100	99.7	93.5
CRM	99.4	99.2	99.3	96.4	97.4	96.9	100	100	100	98.7
CRME	99.4	99.8	99.6	97.3	99.0	98.2	100	100	100	99.3

Análisis de los resultados del experimento 1

De este primer experimento se concluye claramente que LPTF resulta la peor de las estrategias, especialmente para grafos de granularidad media. Por este motivo, en los posteriores experimentos llevados a cabo la comparación que se realizó fue utilizando los resultados de LPTF como valores de referencia.

También constatamos que SA se comporta como una estrategia prácticamente óptima (se acerca a un 99.2% del óptimo) y que TS, a pesar de ser un método iterativo como SA consigue resultados un tanto peores (93%). De hecho, el problema que sufre TS frente a SA es consecuencia del uso de la lista tabú y de la influencia que esto tiene a la hora de determinar la condición de finalización del algoritmo. Mientras que para el método SA es posible fijar los parámetros de forma que se realice una búsqueda de tiempo suficientemente elevado, en el método TS aunque se le fije un límite temporal elevado la búsqueda finaliza porque el método es incapaz de explorar configuraciones que aparecen en la lista tabú que no son necesariamente configuraciones repetidas por las que ya se ha pasado. Por el hecho de guardar en la lista tabú elementos cuya descripción engloba a todo un conjunto de configuraciones distintas, el método TS es incapaz de aceptar movimientos a configuraciones vecinas de la misma familia pero que permitirían mejorar posteriormente la función de coste. Esta incapacidad lleva a situaciones donde no se mejora el resultado actual y todos los vecinos están en la lista tabú, produciéndose, en consecuencia, la finalización de la búsqueda.

A pesar de que las estrategias TS y SA parten de configuraciones iniciales aleatorias, los resultados finales para un mismo grafo de las distintas ejecuciones suelen ser siempre iguales; tan sólo TS en algunos casos proporciona distintas soluciones para un mismo grafo inicial.

Por lo que se refiere a las estrategias CA, CRM y CRME ya se había visto su eficiencia en el capítulo anterior al usar el mismo grupo de grafos, basta constatar que, en particular, tanto CRM como CRME consiguen resultados muy próximos a la estrategia SA para todos los tipos de grafos, acercándose a un 98.7% y 99.3%, respectivamente, del óptimo.

4.3.2 Experimento 2. Comparación entre estrategias para grafos regulares grandes

La tabla 4.4 muestra el resultado global de las soluciones halladas por las distintas estrategias (los datos correspondientes a este experimento se encuentran en el apéndice B). La tabla muestra la mejora promedio que alcanza cada estrategia en relación con el resultado hallado por LPTF. Estos valores se han calculado usando T_p/T_{LPTF}, donde T_p es el coste de la asignación que encontraba una cierta heurística y T_{LPTF} es el coste de la asignación encontrada por LPTF. Un valor de 1 significa que la calidad de las soluciones era la misma y, por lo tanto, valores superiores a la unidad implican que la heurística encontraba asignaciones mejores que las de LPTF, mientras que valores por debajo de uno indican que la heurística encontraba asignaciones peores que las de LPTF. El significado de las columnas MD, CR, PRO y Total coincide con lo que se comentó en el experimento 1.

En este experimento, donde se usaban grafos de mayor tamaño, ya se incluyó en la comparación la heurística mixta de Wu, tanto la fase *greedy* (ED), como la versión completa (EDTR).

Aunque la tabla 4.4 no diferencia los resultados particulares obtenidos para 8, 16 y 32, en la práctica el comportamiento relativo de las distintas heurísticas era similar para un número concreto de procesadores. Así, todos las heurísticas tenían las mayores mejoras frente a LPTF para 8 procesadores y esas mejoras disminuían paulatinamente cuando se usaban 16 y 32 procesadores, respectivamente.

Tabla 4.4 Mejora promedio de la función de coste respecto a LPTF para grafos regulares

Topología	ARBOLES			MALLAS			ANILLOS			
Granular.	MD.	CR.	PRO.	MD.	CR.	PRO.	MD.	CR.	PRO.	Total
LGCF	1.35	1.04	1.19	1.23	1.03	1.13	1.29	1.02	1.15	1.17
TS	2.17	1.05	1.61	2.04	1.05	1.55	1.94	1.03	1.48	1.55
SA	2.33	1.14	1.74	2.52	1.20	1.86	2.61	1.12	1.87	1.82
ED	1.78	1.12	1.45	1.87	1.16	1.51	1.22	1.03	1.12	1.36
EDTR	1.96	1.14	1.55	1.96	1.18	1.57	1.36	1.06	1.21	1.44
CA	1.97	0.89	1.43	2.13	1.01	1.56	2.45	0.97	1.71	1.57
CRM	2.43	1.18	1.81	2.51	1.23	1.87	2.80	1.20	2.00	1.89
CRME	2.43	1.19	1.81	2.52	1.24	1.88	2.80	1.21	2.01	1.90

Análisis de los resultados del experimento 2

De los resultados de la tabla 4.4 se puede concluir que en el conjunto de estrategias *greedy*, tanto ED como CA demuestran ser bastantes efectivas globalmente y substancialmente mejores a LGCF (1.36 y 1.57, respectivamente, frente a 1.17) que, aunque tiene una complejidad menor obtiene peores resultados debido a que en su proceso de agrupación no toma ninguna decisión en base a la conectividad existente entre los nodos. Por contra, tanto ED como CA sí agrupan tareas en base a la supresión de arcos significativos.

Para grafos de granularidad gruesa y topología de árbol o de anillo la estrategia CA obtiene resultados que son ligeramente peores que los de LPTF. La explicación a este hecho hay que buscarla en el razonamiento que se hizo cuando se presentó en el

apartado 3.5 del capítulo anterior la cota para estimar el número de movimientos que se harán en la fase iterativa. Como se vio, el algoritmo CA puede dejar, dependiendo del número de nodos y procesadores, alguna agrupación muy desbalanceada al final. Cuanto mayor sea la granularidad del grafo, más cerca estamos de la hipótesis que se asumía en aquel estudio y, por lo tanto, es más probable que aparezcan esos nodos desbalanceados al final. Sólo en estos casos, de grafos muy regulares donde en la función de coste prevalece el volumen de cómputo de los nodos, es donde LPTF, que busca simplemente el equilibrio de cómputo entre todos los procesadores, se muestra ligeramente superior. De todos modos, si pensamos que el objetivo de CA es obtener una reducción del grafo eficiente que sea mejorada posteriormente en la fase iterativa, estos resultados parciales no son preocupantes porque CRM y CRME corrigen perfectamente ese desbalanceo momentáneo.

En general, las estrategias iterativas, TS y SA, sufren un retroceso relativo en su rendimiento frente a los resultados de la experimentación con grafos pequeños. Sus resultados globales se ven empeorados por el hecho de haber realizado cinco ejecuciones de cada método para cada grafo y haber utilizado el resultado promedio de cada una de las cinco asignaciones. Con los grafos pequeños este fenómeno apenas se observaba y para un mismo grafo siempre solía hallarse la misma asignación. Al aumentar el tamaño de los grafos las diferencias en las asignaciones halladas para un mismo grafo son más significativas. Tanto TS como SA son capaces de obtener buenos resultados pero que en promedio se van a ver compensados por aquellas ocasiones en las que el algoritmo ha obtenido una asignación mucho peor.

La estrategia TS sigue obteniendo peores resultados que SA por los mismos motivos que se comentaron en el anterior experimento.

El rendimiento relativo de SA es inferior al de CRM o CRME, frente a los resultados para grafos pequeños donde SA era mejor que CRM y prácticamente igual a CRME. SA alcanza una mejora de 1.82, frente a la mejora de 1.89 alcanzada por CRM y la de 1.9 alcanzada por CRME. Esta pérdida al aumentar el tamaño de los grafos se explica, en parte, porque el comportamiento del algoritmo SA con un grafo pequeño de unos 16 nodos y tardando 3 ó 4 minutos para alcanzar la solución se acerca al supuesto en el que el método dispone de un tiempo extremadamente grande para buscar en el espacio de soluciones. Al aumentar el número de nodos a 400, aunque el tiempo de ejecución estuviese entre 45 y 60 minutos, no se conseguía el mismo efecto, y la búsqueda y, en consecuencia, el resultado final encontrado quedaban más limitados a un espacio de soluciones menor.

Teniendo en cuenta que tanto la estrategia mixta EDTR como las propuestas CRM y CRME comparten la misma parte iterativa, los resultados de las dos últimas son substancialmente mejores que los de la primera (1.44 de EDTR frente a 1.89 de CRM y 1.9 de CRME). La razón de la calidad de las soluciones halladas por CRM y CRME hay que buscarla, en consecuencia, en el algoritmo CA usado en la primera fase de la estrategia. Igualmente, SA siempre es superior a EDTR mientras que no ocurre lo mismo con CRM y CRME..

Para todos los grafos de granularidad gruesa la ganancia de todas las estrategias frente a LPTF no es muy elevada (al menos en comparación con las ganancias que se obtienen para los grafos de grano medio). Esto se debe a que a medida que aumenta la granularidad de los grafos más se puede considerar que son tareas independientes, y en ese caso se sabe que la estrategia LPTF es capaz de obtener asignaciones que empeoran el resultado de la asignación óptima en menos de 4/3 (o sea, cost_map (LPTF) ≤ 4/3*cost_map (óptimo)) [Gra69]. Lógicamente, las restantes estrategias tendrán la misma cota y, por lo tanto, resultados cercanos a los obtenidos por LPTF.

4.3.3 Experimento 3. Comparación entre estrategias para grafos irregulares

En este experimento no se recurrió al uso de grafos irregulares generados aleatoriamente, sino que se utilizaron TIGs irregulares que provenían de la transformación previa de grafos DAGs. A continuación, presentaremos el método en el que se basa dicha transformación.

Transformación de grafos DAGs a TIGs

La presente heurística, denominada DSC (*Dominant Sequence Clustering*), fue propuesta por Yang y Gerasoulis [Yan94] para asignar las tareas de un DAG a un conjunto no acotado de procesadores y es utilizada para transformar DAGs a TIGS. La elección de este método se debe a dos razones fundamentales: por una parte, es el método que consigue una mejor asignación, y con una complejidad algorítmica muy baja (de O(NlogM), donde N es el número de procesos y M es el número arcos) con respecto a los métodos presentes en la literatura analizados en [Ger92]; por otra parte, el programa para obtener estas asignaciones está disponible en la universidad de los autores y es de libre utilización.

En la figura 4.4 podemos ver un ejemplo del resultado del método DSC. Originalmente, partimos de un DAG compuesto por 7 tareas que al final quedan agrupadas en tres nodos: $\{t_1, t_2, t_7\}$, $\{t_3, t_4, t_6\}$ y $\{t_5\}$. Junto con la agrupación también se obtiene una ordenación parcial de la ejecución de las tareas dentro de cada uno de los nodos finales: así, el arco discontinuo añadido entre t_4 y t_3 está reflejando el hecho de que t_4 debe ejecutarse antes de t_3, creando con ello una dependencia ficticia.

La asignación obtenida tiene un coste, expresado como tiempo de cómputo paralelo de TP = 8 unidades, mientras que el grafo original tenía un valor de TP = 13. Para calcular estos valores se debe proceder de la siguiente forma:

> Los arcos del grafo representan dependencias y pasos de comunicación. Una tarea sólo se puede ejecutar cuando ha recibido todos sus mensajes de entrada. La tarea se ejecuta completamente hasta su finalización y después envía los mensajes de salida a sus tareas sucesoras. El tiempo de ejecución es el volumen de cómputo del nodo. El tiempo de transmisión de los mensajes de salida es igual al volumen de comunicación del arco si las tareas no están asignadas en el mismo procesador y cero, si sí lo están. Inicialmente se pueden empezar a ejecutar todas aquellas tareas que no tienen ninguna dependencia, en el ejemplo de la figura se empezaría la ejecución de las tareas t_1, t_4 y t_5. A partir de este

punto, respetando las dependencias y los tiempos de cómputo y de comunicación de las tareas predecesoras, se van ejecutando el resto de tareas del grafo. El instante de finalización de la última tarea del grafo (t_7, en nuestro ejemplo) determina el valor del tiempo de ejecución paralelo, TP.

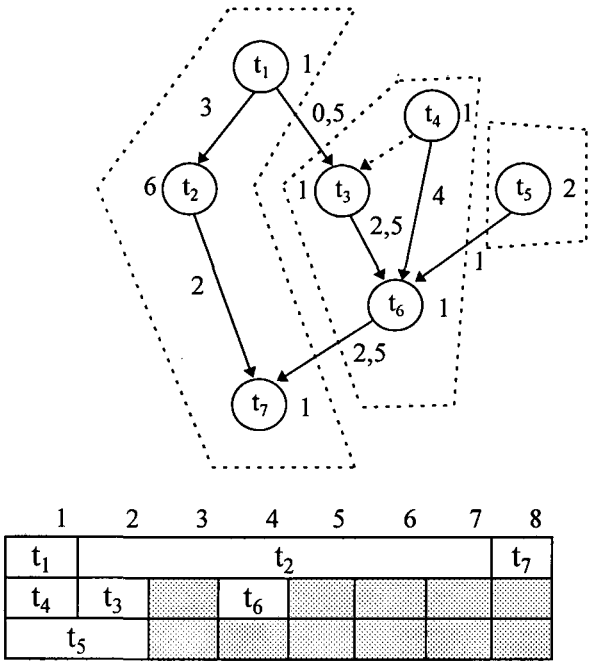


Figura 4.4 Ejemplo de DAG y su ejecución agrupado por DSC

En la parte inferior de la figura 4.4 puede verse una traza de Gantt correspondiente a la ejecución del grafo del ejemplo y puede compararse con la traza obtenida con una asignación diferente y que refleja la figura 4.5. En este segundo caso la asignación se calculó aplicando el método propuesto por Sarkar [Sar89] y el tiempo de ejecución obtenido es de TP = 10 unidades.

El estudio detallado del método DSC está fuera del ámbito de interés de este trabajo por lo que el lector interesado puede dirigirse a la referencia antes citada para conocer detalladamente su funcionamiento; baste indicar aquí cuál es la idea básica en la que se basa. Definimos primero *camino crítico* de un DAG como el camino más largo en el grafo desde un nodo inicial a un nodo terminal. Cuando las tareas han sido agrupadas y se ha fijado un orden de ejecución en cada agrupación el tiempo de ejecución TP dependerá del camino crítico que tiene en cuenta ese orden y que se conoce como *secuencia dominante (Dominant Sequence, DS)*. El algoritmo DSC realiza una serie de pasos de agrupación intentando detectar en cada momento cuál es el conjunto de nodos y arcos que constituyen la DS del grafo y poder eliminar algún arco de ese camino de forma que se reduzca la longitud de ese camino y con ello el tiempo de ejecución paralelo. Lógicamente, a medida que se van haciendo agrupaciones la DS del grafo va cambiando y puede estar formada por otros nodos y arcos que no formaban parte de ella inicialmente. La dificultad del método DSC, y al mismo tiempo su efectividad, estriba en su capacidad para localizar los componentes de la secuencia dominante a medida que van produciéndose los distintos pasos de agrupación. Y todo ello, con una baja complejidad similar o inferior a otros métodos de agrupación para DAGs.

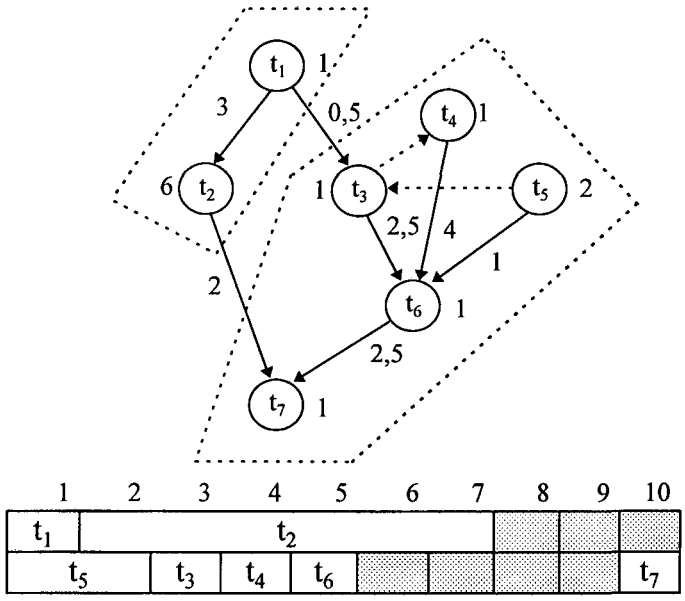


Figura 4.5 Agrupación y ejecución del DAG según método de Sarkar

Después de aplicar el algoritmo DSC a un DAG se obtiene un grafo que podemos considerar que es del tipo TIG y para el cual podemos aplicar los métodos que se han ido proponiendo en este trabajo. Siguiendo con el ejemplo de la figura 4.4, el resultado de la agrupación es un grafo TIG como el que muestra la figura 4.6, donde se muestran los nodos y arcos resultantes y sus respectivos volúmenes. Cada agrupación obtenida por el método DSC es un nodo del grafo TIG y todos los arcos que conectaban una misma pareja de agrupaciones en el DAG pasan a ser un único arco no dirigido que une los dos nodos correspondientes en el TIG. Ahora podemos ver a los tres nodos resultantes en nuestro ejemplo como tres tareas de tipo persistente, que existen durante todo el tiempo de ejecución del programa y que se van intercambiando información a lo largo del tiempo, sin que tengamos presente en qué sentido ni en qué momento se produce la comunicación.

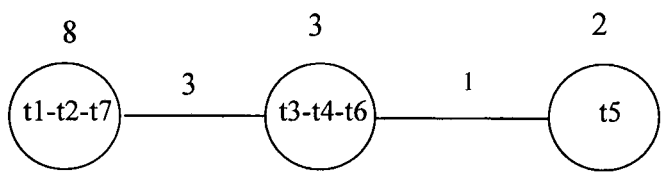


Figura 4.6 TIG resultante de la agrupación realizada por DSC

El ejemplo que hemos presentado muestra la forma de convertir un DAG en un TIG y, de hecho, es la idea utilizada en el entorno PYRROS [Yan92] para realizar la asignación de programas modelados como DAGs a un multiprocesador. En el caso concreto de PYRROS, se utiliza una variante del método que hemos denominado como LGCF para reducir el número de agrupaciones generadas por el algoritmo DSC a un número igual al de procesadores disponibles.

Descripción de grafos irregulares

Los DAGs que se utilizaron para generar los grafos TIGs aleatorios corresponden a un conjunto de aplicaciones reales que fueron modeladas y utilizadas en el entorno ALPES [Kit93] (en el apéndice A se incluyen descripciones gráficas reducidas de estos grafos).

En la tabla 4.3 se detallan todos los grafos utilizados en este conjunto de experimentos, indicando la aplicación a la que corresponde el grafo, el número de nodos que tenía el DAG original y el número de nodos que contiene el TIG generado por el algoritmo DSC.

Tabla 4.3. Descripción del conjunto de grafos TIGs irregulares

Nombre grafo	Aplicación	# nodos DAG	# nodos TIG
g1	Algoritmo de Bellman-Ford que encuentra los caminos más cortos entre cualquier pareja de nodos de un grafo	365	46
g2	Algoritmo sistólico de multiplicación de matrices	1002	100
g3	Algoritmo sistólico para calcular la *transitive closure* de un conjunto de elementos	1001	188
g4	Algoritmo de tipo *divide & conquer*	1534	512
g5	Algoritmo de sustitución de Gauss (*back substitution*)	530	32
g6	Algoritmo de Warshall para calcular la *transitive closure* de un grafo	1026	62
g7	Algoritmo Strassen de multiplicación paralela de matrices	1483	573
g8	Algoritmo de tipo master-slave seguido del algoritmo de Gauss (*back substitution*)	3420	176
g9	Algoritmo *black-red relaxation* para la resolución de la ecuación de Poisson	2818	416
g10	Árbol de búsqueda de Prolog	1002	114

De los grafos presentes en la tabla 4.3 se descartó el TIG obtenido a partir de g10. porque presentaba una granularidad fina. Esto implicaba que la mejor asignación se obtenía agrupando todos los nodos a un sólo procesador. Desgraciadamente, todas las heurísticas utilizadas en la comparación eran incapaces de detectar esta situación y producían asignaciones usando el máximo número de procesadores y, por consiguiente, con peor valor en la función de coste. Tan sólo nuestra estrategia era capaz de superar este inconveniente gracias a la iteración que realiza el algoritmo de agrupación CA una vez reducido el grafo a un número de agrupaciones igual al número de procesadores. La inclusión de esa iteración, sin embargo, no tenía como objetivo que el algoritmo fuese capaz de reducir a un sólo nodo porque este tipo de casos "patológicos" pueden ser detectados por cualquier estrategia añadiendo una simple comprobación que compare el coste de la asignación hallada de forma normal con el coste de la asignación que coloca todos los nodos juntos. Si la segunda situación fuese más favorable, bastaría con seleccionarla, superando así el problema en el que incurre el funcionamiento normal de la estrategia.

Resultados del experimento 3

La tabla 4.5 resume los resultados obtenidos, mostrando para cada grafo la mejora obtenida respecto a LPTF y el comportamiento global de la estrategia se muestra en la columna Total.

Tabla 4.5. Mejora promedio de función de coste respecto a LPTF para grafos irregulares

	g1	g2	g3	g4	g5	g6	g7	g8	g9	Total
LGCF	1.11	1.13	1.11	1.25	1.07	1.07	1.49	1.15	1.24	1.18
TS	1.25	1.41	1.46	1.31	1.08	1.07	1.92	1.33	1.44	1.36
SA	1.25	1.44	1.49	1.31	1.08	1.07	1.85	1.44	2.10	1.45
ED	1.06	1.22	1.35	1.31	0.98	1.07	2.00	1.19	1.65	1.31
EDTR	1.20	1.30	1.45	1.31	1.08	1.07	2.03	1.37	1.83	1.41
CA	1.03	1.28	1.20	1.25	0.88	0.89	1.82	1.11	1.53	1.22
CRM	1.12	1.46	1.46	1.31	0.97	1.07	2.04	1.37	1.93	1.41
CRME	1.19	1.54	1.50	1.31	1.07	1.07	2.05	1.39	2.00	1.46

Análisis de los resultados

De la tabla 4.5 se puede concluir de nuevo que, globalmente, las dos versiones (CRM y CRME) de nuestra estrategia siguen ofreciendo muy buenos resultados en comparación con las otras estrategias (CRME presenta los mejores resultados entre todas las estrategias analizadas con una mejora promedio de 1.46 y CRM está ligeramente por debajo con un 1.41). Sin embargo, es importante notar que para el caso de grafos irregulares los resultados globales de todas las estrategias son muy similares y la máxima diferencia entre las mejores (TS, SA, EDTR, CRM y CRME) es de 0.1 como máximo (TS cierra el grupo por abajo con un promedio de 1.36 y CRME por arriba presenta un promedio de mejora de 1.46). Esta igualdad en los resultados se debe a dos motivos: el primero está relacionado con la granularidad de los grafos y el segundo está relacionado con el grado de conectividad de ciertos nodos en los grafos.

Por una lado todos los grafos irregulares eran, en mayor o menor medida, de granularidad gruesa, por lo que se puede aplicar el mismo razonamiento que se hizo al comentar los resultados de los grafos regulares de granularidad gruesa. Tan sólo en los grafos g7 y g9, que presentaban la menor granularidad con valores de 3.6 y 3.1, respectivamente, es donde se pueden observar diferencias más significativas.

El otro factor influyente para que se consigan resultados similares para todas las estrategias proviene del hecho de que muchos de los grafos irregulares tenían un grado de conectividad elevado y, en particular, en ciertos casos existía algún nodo con un número de conexiones muy por encima de la media. En esos casos, donde un determinado nodo tiene un número muy elevado de conexiones, se puede observar que el resultado final de la asignación viene determinado e influido de forma importante por ese nodo. La agrupación a la que pertenece ese nodo suele ser la que determina el coste de la asignación final y, dado su elevado grado de conectividad y que el grafo exhibe granularidad gruesa, es imposible reducir el coste de dicha agrupación mediante la incorporación de otros nodos que estén conectados con el nodo en cuestión. Un ejemplo de este tipo de situación se puede ver en la figura 4.7

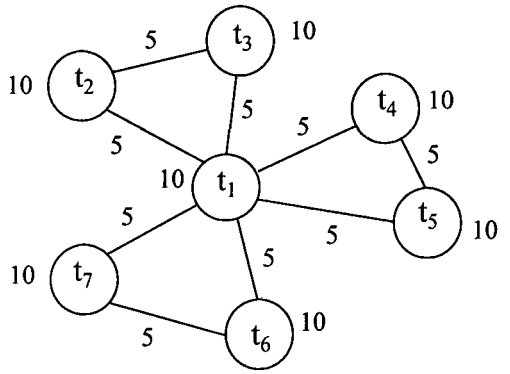


Figura 4.7 Ejemplo TIG

Si suponemos que el grafo mostrado en la figura 4.7 debe asignarse a 4 procesadores, se puede constatar que el coste de tal asignación está determinado desde el inicio por el coste del nodo t_1 (coste = 40). No se puede suprimir ningún arco de ese nodo que permita reducir su coste. En consecuencia, cualquier estrategia de asignación que detectase esa situación se limitará a generar las agrupaciones {t2, t3}, {t4, t5} y {t6, t7}. Notemos, por último, que el tipo de situación comentado no es necesario que aparezca inicialmente en el TIG original, puede aparecer a partir de una cierta reducción del mismo.

Como conclusión del análisis comparativo entre las distintas estrategias, la figura 4.8 muestra gráficamente las mejoras obtenidas por cada una de ellas globalmente para grafos regulares e irregulares (información de la columna Total de las tablas 4.4 y 4.5)

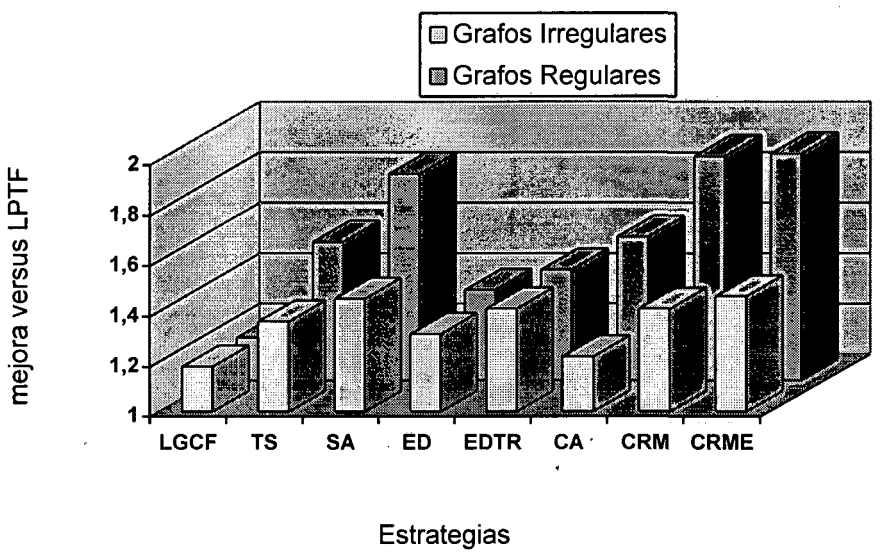


Figura 4.8 Resultados comparativos de las estrategias de mapping

4.3.4 Estudio comparativo de las fases iterativas de las estrategias mixtas

Del análisis efectuado a partir de los resultados obtenidos en los experimentos anteriores se puede deducir que, restringiendo la comparación a las estrategias mixtas (EDTR, CRM y CRME), nuestras dos propuestas (CRM y CRME) son significativamente superiores al método propuesto por Wu (EDTR). Es importante además notar que la estrategia CRM, además consigue mejores resultados que EDTR usando tan sólo movimientos individuales de tareas que, como se comentó, suponen una operación de menor complejidad que los intercambios.

Con objeto de analizar con más detalle la complejidad experimental de los estrategias mixtas en sus respectivas fases iterativas se recopiló el número de movimientos e intercambios que hacían tanto el método CRME como el EDTR en los experimentos 2 y 3. La tabla 4.6 muestra, para cada estrategia, los resultados globales obtenidos, indicando el número promedio de movimientos $\overline{m}$, de intercambios $\overline{e}$, así como su correspondiente desviación estándar σ_N.

Tabla 4.6. Promedio de movimientos e intercambios realizados en la fase iterativa

	GRAFOS REGULARES				GRAFOS IRREGULARES			
	movimientos		intercambios		movimientos		intercambios	
	$\overline{m}$	σ_N	$\overline{e}$	σ_N	$\overline{m}$	σ_N	$\overline{e}$	σ_N
EDTR	14.3	18.5	16.3	18.7	7.2	9.6	15.8	26.0
CRME	40.3	17.9	3	0.9	8.5	8.3	3.1	4.5

Los resultados que refleja la tabla 4.6 corroboran que la fase de mejora de la estrategia CRME se basa fundamentalmente en la realización de movimientos individuales (un promedio de 40.3 en grafos regulares y de 8.5 en los irregulares) mientras que el número de intercambios es significativamente menor (3 y 3.1, respectivamente). En el caso de los grafos regulares el número de movimientos realizados está en consonancia con los valores que podían esperarse aplicando la cota de estimación de movimientos presentada en el apartado 3.5 del capítulo anterior.

Por su parte, la estrategia EDTR mejora los resultados de su fase *greedy* usando un cantidad ligeramente superior de intercambios que de movimientos individuales (los intercambios suelen estar siempre alrededor de 16, mientras que los movimientos varían entre 7 y 14 según el tipo de grafo).

A la vista de los resultados experimentales, un número de movimientos elevados puede elevar la complejidad de esa fase a $O(N^2)$; por su parte, si se eleva el número de intercambios, la complejidad de esta fase pasaría a ser de $O(N^3)$. En consecuencia, podemos concluir que la estrategia EDTR obtiene, en general, asignaciones peores, necesitando además la realización de un mayor número de operaciones de mayor complejidad computacional. Lógicamente, la estrategia EDTR original propuesta por Wu no incluía la realización de intercambios, pero es importante notar que sin esa mejora que nosotros le hemos añadido sus resultados hubiesen sido todavía peores frente a los obtenidos por nuestra estrategia.

4.3.5 Resultados experimentales de tiempos de cómputo

En el apartado anterior se ha realizado la comparación de las estrategias desde el punto de vista de calidad de la solución obtenida. En el presente apartado se presentarán los resultados relativos al tiempo de cómputo real consumido por cada una de las estrategias en la obtención de esas soluciones. En la tabla 4.7 se muestra el tiempo promedio ($\bar{t}$), en segundos, requerido por cada una de las estrategias en cada ejecución, separando el caso de los grafos regulares de los irregulares.. También se proporciona para ambos casos el tiempo mínimo, el tiempo máximo, así como la desviación estándar. Las medidas de tiempo de los grafos regulares se hicieron con los de tamaño grande (entre 255 y 400 nodos). En el caso de los grafos irregulares se utilizaron aquellos que tenían 100 o más nodos, de forma que el tamaño de los grafos de ambos conjuntos, regulares e irregulares, fuese del mismo orden. El promedio se calculó a partir de los tiempos de ejecución de cada estrategia con cada uno de los grafos mencionados asignados sobre 16 procesadores.

Tabla 4.7. Tiempo promedio (en segundos) consumido por las distintas heurísticas

	GRAFOS REGULARES				GRAFOS IRREGULARES			
	$\bar{t}$	σ_N	Max.	min.	$\bar{t}$	σ_N	Max.	min.
LPTF	0.1	0.04	0.2	0.1	0.1	0.02	0.2	0.1
LGCF	0.16	0.04	0.2	0.1	0.3	0.1	0.5	0.1
TS	46.2	78.4	517	4	749.3	861.2	3691	75
SA	2682.4	699.9	4784	1778	2513.9	864	4136	1237
ED	1.3	0.33	1.5	0.6	2.18	1.04	3.9	0.4
EDTR	23.6	39.9	285	0.9	18.5	39.6	188	1.9
CA	0.8	0.39	1.3	0.4	1.4	0.7	2.2	0.3
CRM	1.5	0.5	2.2	0.6	2.1	0.3	2.6	1.2
CRME	4.3	3.4	13.9	0.6	13.6	23.7	72.4	1.7

Los resultados mostrados en la tabla 4.7 confirman, en principio, los supuestos teóricos que indican que las estrategias *greedy* son las de menor complejidad computacional (LPTF, LGCF, ED y CA), las estrategias iterativas son las de mayor complejidad (TS y SA) y las mixtas ofrecen un compromiso entre las anteriores (EDTR, CRM y CRME). Dentro de las estrategias *greedy*, las que clasificábamos como cuantitativas (LPTF y LGCF) suelen mostrar además un comportamiento ligeramente mejor que el de aquellas que son topológicas (ED y CA). .

En el grupo de las heurísticas iterativas, SA se muestra como una estrategia computacionalmente costosa y bastante regular en los tiempos consumidos en todos los casos (tiempos del orden de 2600 segundos), con la lógica dependencia del tamaño del grafo original. Sin embargo, la heurística TS muestra un comportamiento más irregular (tiempos alrededor de 46 segundos en grafos regulares y de 749 en los irregulares). Como ya se comentó brevemente al comparar los resultados del experimento 2, la estrategia TS suele caer en situaciones donde es incapaz de seguir mejorando el valor de la función de coste, lo que lleva a la finalización de la búsqueda, especialmente en el caso de los grafos regulares. Esto conlleva, lógicamente, un menor tiempo de cómputo aunque la calidad de la solución encontrada será también menor. En este sentido, TS evita en mayor medida que SA la exploración de soluciones que momentáneamente son peores que la mejor solución hallada. Por el contrario, SA es capaz de seguir explorando

durante mucho más tiempo porque no rechaza pasar por situaciones ya exploradas a partir de las cuales pueda pasar a soluciones momentáneamente peores, pero que posteriormente permitirán nuevas mejoras en el valor de la función de coste.

Por lo que respecta a las estrategias puramente mixtas, la que ofrece el mejor compromiso en cuanto a calidad de solución y tiempo de cómputo es claramente la CRM. Los tiempos obtenidos (promedios de 1.5 segundos en grafos regulares y 2.1 en grafos irregulares) demuestran también que la operación de intercambio es claramente la técnica de refinamiento más costosa, que puede incrementar substancialmente el tiempo de ejecución total. Este hecho se puede comprobar tanto en la estrategia CRME como, especialmente, en la EDTR que, como habíamos comentado en el apartado anterior, requiere de la realización de muchos más intercambios que la primera para mejorar una determinada asignación.

Hemos de tener en cuenta también que, a partir de una cierta situación donde todas las tareas estén ya agrupadas, los intercambios suelen ser más efectivos que los movimientos individuales en la mejora de la función de coste. Los movimientos son muy útiles para corregir situaciones como las que puede generar el algoritmo CA donde se pueden crear algunas agrupaciones muy desbalanceadas. A partir de una situación de relativo equilibrio, son los intercambios los que ofrecen más mejoras. Vista la complejidad temporal debida a la realización de intercambios, los resultados podrían aconsejar, en cualquier caso, concentrar esfuerzos para intentar optimizar esta fase usando estructuras de datos y algoritmos que permitan reducir el impacto de las operaciones involucradas en la realización de esos intercambios. De todos modos, para nuestro estudio creemos que los resultados obtenidos son plenamente satisfactorios en su estado actual, disponiendo de una estrategia prometedora para resolver el problema del *mapping* de programas paralelos modelados como TIGs.

CAPÍTULO 5

Estrategias de asignación considerando la arquitectura

En los capítulos precedentes se ha propuesto una estrategia para la resolución del problema del *mapping* del tipo N a 1. La estrategia parte de un grafo de N nodos y lo reduce a un número de agrupaciones igual (o menor) al número de procesadores. La única consideración que hace la heurística con respecto a la arquitectura del computador paralelo es que éste es un sistema totalmente interconectado. Esto supone que en la evaluación del coste de las asignaciones encontradas se considera que los procesadores se encuentran a distancia unitaria.

En el presente capítulo se analizará en qué situaciones la suposición hecha es válida y, por lo tanto, la heurística sería directamente aplicable, y en qué casos la suposición no es válida porque la arquitectura no es totalmente interconectada y, en consecuencia, la heurística debería ser ampliada.

La ampliación de la heurística se basará en la realización de una etapa adicional en la que se resolverá un problema de *mapping* del tipo 1 a 1, en el que se tendrá en cuenta un nuevo parámetro del sistema paralelo que será la distancia entre los procesadores. Se propondrán, por lo tanto, un conjunto de estrategias para la realización de esta etapa y se evaluará su rendimiento.

El hecho de considerar arquitecturas no interconectadas totalmente supone un incremento del coste debido a las comunicaciones que repercute también en el valor final de la función de coste asociado a la asignación. Por ese motivo, se presentarán y validarán dos estimadores que, dada una cierta arquitectura y el grafo de agrupaciones de tareas, permitan evaluar a priori el incremento que experimentarán las comunicaciones y la función de coste, respectivamente, al realizar la asignación de dicho grafo en esa arquitectura.

Por último, se presentarán los resultados de un conjunto de experimentos encaminados a evaluar la relación existente entre valores de la función de coste y tiempo de ejecución de las aplicaciones.

5.1 Costes de comunicación y Arquitectura

Desde el punto de vista del problema de la asignación el hecho de considerar la existencia de una cierta red de interconexión va a repercutir fundamentalmente en el tratamiento que se hace de las comunicaciones entre las tareas. Cuando se considera que dos tareas tienen asociado un cierto volumen de comunicación entre ellas dicho valor va a traducirse físicamente en un tiempo de comunicación T_{comm}. Este tiempo puede descomponerse en tres factores:

$$T_{comm} = T_{ini} + T_{term} + T_{rout}$$

Donde T_{ini}, T_{term} y T_{rout} corresponderían, respectivamente, a los tiempos de cómputo del procesador asociados con la inicialización del mensaje, su finalización y su encaminamiento. Estos tres términos serían el *overhead* que implicaría cada envío de un mensaje aunque su incidencia física sea diferente. Así, T_{ini} correspondería al procesamiento ocasionado en el procesador origen, T_{term} al producido en el procesador receptor del mensaje, mientras que T_{rout} repercutirá en los procesadores intermedios atravesados por el mensaje entre el procesador origen y el destino.

Dos de los *overheads* mencionados, T_{ini} y T_{term} son consecuencia de la descomposición del programa en módulos y son independientes de la asignación de los módulos. Incluyen todo el procesamiento involucrado en la generación de paquetes, el coste de las llamadas a las funciones de envío y recepción de paquetes, en la copia de la información entre el espacio de direcciones del sistema y el de los procesos, y el desempaquetado del mensaje. Por lo que se refiere a estos dos valores se puede suponer que el cómputo asociado con en el envío de mensajes es independiente de la localización de la pareja de procesos involucrados. También suele considerarse que cuando los dos procesos están situados en el mismo procesador ambos factores son despreciables porque el sistema podría hacer simplemente una copia entre los espacios de direcciones de los procesos.

Por lo que se refiere a T_{rout}, este *overhead* puede interpretarse de dos formas. Por una parte, en ciertos sistemas sería un valor constante con independencia de los procesadores involucrados en la comunicación. Puede suponerse que estos sistema paralelos están formados por un conjunto de procesadores totalmente interconectados donde se introduce siempre un retardo fijo. Tal suposición es plenamente aceptable, por ejemplo, en los siguientes casos:

- Aquellos sistemas paralelos donde las redes de interconexión se basan en la idea de Redes de Interconexión Multietapa (RIM). Suelen argumentar los defensores de dichos sistemas que ese sistema de interconexión garantiza un elevado ancho de banda y una baja latencia de comunicación entre todos los procesadores, ofreciendo, además un rendimiento uniforme y predecible en todo el sistema incluso bajo condiciones de carga elevada. Un ejemplo de este tipo de arquitectura lo constituye el modelo SP2 de IBM con el esquema de interconexión que muestra la figura 5.1 [Age95] (la figura muestra en negrita posibles caminos entre el procesador P0 y los procesadores P3 y P10). En realidad el camino seguido por los mensajes entre las distintas parejas de

procesadores no siempre atraviesa el mismo número de elementos de interconexión pero, en la práctica, el error cometido al considerar todos los procesadores a una misma distancia los unos de los otros es despreciable [Stu95].

- En las arquitecturas con red de interconexión punto a punto (p. ejemplo, hipercubos o mallas con procesadores tipo Transputer) puede existir una capa de software encargada del encaminamiento (*routing*) de los mensajes que enmascare la distancia física entre los procesadores. En estos casos la mencionada capa puede utilizar alguna estrategia de encaminamiento (como los mecanismos de dos fases en los que el primer paso se realiza aleatoriamente) que en presencia de carga elevada sea capaz de proporcionar latencias promedio similares en todas las comunicaciones sin que influya el lugar físico que ocupan los procesadores. De esta forma la red puede presentar una buena escalabilidad porque a medida que se incremente el tamaño del sistema la latencia promedio crecerá en una proporción cercana a la raíz cuadrada del tamaño de la red [May93].

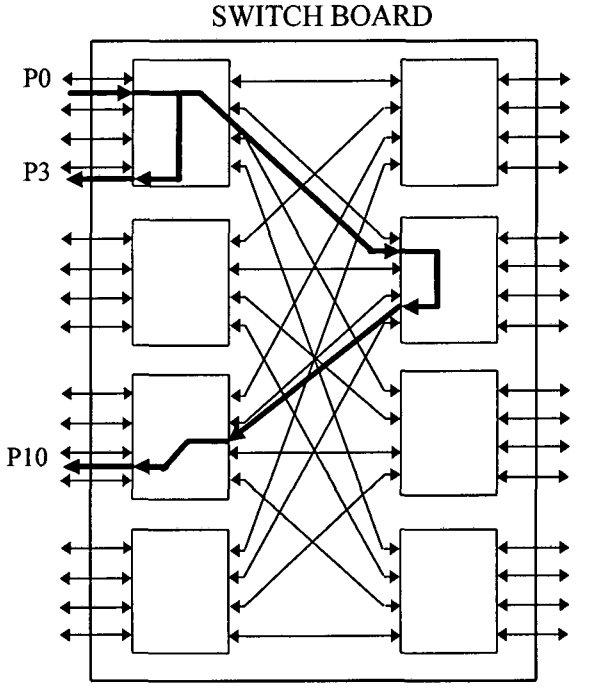


Figura 5.1 Red de interconexión multietapa del sistema SP2

En cualquiera de los dos casos mencionados la estrategia presentada en los capítulos precedentes sería suficiente para abordar el problema del *mapping*.

Si el sistema incluye una red de interconexión punto a punto, como los computadores paralelos basados en Transputers o hipercubos, donde no existiera esa capa de encaminamiento que proporcionase una latencia fija, T_{rout} sería proporcional a la distancia entre los procesadores. En estos casos, la estrategia de asignación que hemos presentado resulta insuficiente para resolver el problema del *mapping* y será necesario

incluir alguna nueva etapa que permita obtener la asignación física final teniendo en cuenta la influencia de la red punto a punto.

5.1.1 Coste de comunicaciones en redes punto a punto

De lo expuesto en el apartado anterior sería necesario introducir cambios en la estrategia de asignación sólo para los sistemas paralelos con una red punto a punto. A continuación analizaremos con mayor detalle las implicaciones que suponen este tipo de redes. Nuestro estudio se centrará en tres aspectos debidos a las características de estos sistemas: las estrategias de control de flujo, las distancias entre los procesadores y la contención en la red.

Estrategias de control de flujo

Desde el punto de vista de los procesos una comunicación tiene lugar mediante el envío de un mensaje que suele descomponerse en paquetes (unidad mínima que contiene información de encaminamiento), los cuales también pueden dividirse posteriormente en *flits* (la menor unidad sobre la que se realizan las operaciones de control de flujo). Mientras todos los paquetes incluyen la dirección del procesador destino, sólo el primer *flit* de un paquete es el que conoce el destino [Don93].

El sistema utiliza estrategias para escoger un camino entre los canales de la red y estrategias de control de flujo para regular el tráfico en la red. Existen dos estrategias de control de flujo habituales: *store and forward* y *wormhole*, siendo la segunda la predominante para conseguir redes de comunicación con mejor rendimiento. Cada una de estas estrategias toman sus decisiones sobre paquetes o sobre *flits* respectivamente y ello supone que tengan una latencia característica suponiendo una red con carga cero (donde no hay contención). En el primer caso la latencia viene dada por:

$$T_{SF} = T_c\,(D * L/W)$$

donde D es el número de saltos (*hops*) necesarios para ir del procesador origen al destino (lo que entendíamos por distancia), Tc es el tiempo de ciclo del canal, L es la longitud del mensaje y W es el ancho de banda del canal.

Para *wormhole* la latencia viene expresada por:

$$T_W = T_c(D + L/W)$$

porque en este caso las transmisiones avanzan en forma de *pipeline*; D representa el coste de inicialización del pipeline y una vez inicializado la información va llegando al destino a una velocidad de L/W.

En el modelo de grafo TIG los arcos que unen las parejas de nodos tiene asociado un peso que hemos venido considerando a lo largo de este trabajo que se puede interpretar como el volumen total de información intercambiada entre cada pareja de procesos. Sin embargo, a raíz de lo expuesto al hablar de las distintas estrategias de

control de flujo deberíamos corregir esta interpretación para que dicho volumen corresponda con una estimación correcta de la latencia que sufrirá el mensaje.

De las dos ecuaciones presentadas se deduce que en una arquitectura que use *store-and-forward* será irrelevante que el volumen de comunicación total se deba al envío de n mensajes de tamaño x o al envío de un mensaje de tamaño nx. En ambos casos la latencia esperada será n veces T_{SF}. Sin embargo, cuando se usa *wormhole* ambas situaciones provocan distintas latencias. En el primer caso, tendremos

$$T_W(n) = nT_c(D + x/W)$$

y el segundo,

$$T_W(1{:}nx) = T_c(D + nx/W)$$

Como se ve, en el caso de *wormhole* la latencia crece linealmente con el número de mensajes enviados debido al término nT_cD.

La consecuencia del análisis anterior repercute básicamente en la forma de calcular el peso asociado a cada uno de los arcos del grafo del programa, que debe hacerse de modo que se tenga en cuenta la política de control de flujo seguida por el sistema y reflejar, por consiguiente, el volumen total de información enviada o el número de mensajes enviados. En definitiva, se debe corregir el valor de dichos pesos y normalizarlos de modo que se ajusten a los parámetros físicos del sistema [Don93]. Para ello, en los sistemas basados en *wormhole* sería necesario diferenciar el volumen de comunicación en dos términos: uno fijo, que sólo dependiese del tamaño de los mensajes enviados, y uno variable, que dependiese del número de mensajes enviados y de la distancia entre los procesadores. En nuestro caso, la política de asignación debería adaptarse ligeramente para manejar separadamente estos dos valores: considerar inicialmente que D = 1 y manejar un único volumen de comunicación por cada arco, y tener en cuenta sólo el término nT_cD, para evaluar el coste de la asignación sobre una arquitectura no interconectada totalmente.

Distancias entre los procesadores

En el apartado anterior se ha visto la influencia que puede tener la consideración de la estrategia de control de flujo usada en el sistema a la hora de calcular el peso asociado a los arcos del grafo del programa. Una vez fijado ese valor, la siguiente consideración que debemos realizar se refiere al modo en que va a ser considerado en la función de coste el coste adicional de las comunicaciones debido a la necesidad de encaminar los mensajes por la red.

En el segundo capítulo se presentó una definición formal del problema del *mapping* en la que el volumen de comunicación asociado a cada arco del grafo TIG era multiplicado por la distancia entre los procesadores en los que se encontraban asignadas las dos tareas unidas por dicho arco. Esta consideración se hacía tanto si se adoptaba la función de coste *minimax* como la *summed* y, según lo mostrado en el apartado anterior, la distancia entre procesadores se correspondería con el término D, que indica el número

de saltos necesarios para que un mensaje vaya desde el procesador origen al procesador destino. Nótese que este factor D influye del mismo modo con independencia del mecanismo de encaminamiento adoptado. La consideración de la distancia entre los procesadores como factor multiplicativo del volumen de comunicación ha sido tradicionalmente el modelo adoptado para aproximar dentro de la función de coste el factor que habíamos denominado T_{rout}. Tal consideración supone, de todos modos, algunas simplificaciones sobre el modelo físico de arquitectura de las que debemos ser conscientes y que, básicamente, son:

- al hablar de distancia entre procesadores, ésta debe ser un valor único e inalterable. Esta suposición es cierta siempre que todos los mensajes entre cualquier pareja de procesadores sigan el mismo camino (o caminos de la misma longitud), con independencia del sentido de los mensajes. En la práctica, esta simplificación supone que se va a cometer una cierta imprecisión en aquellos casos donde el número de saltos efectuados por un mensaje no sea un valor fijo porque, por ejemplo, el encaminamiento se haga de forma dinámica o porque los caminos entre dos procesadores tengan distancias distintas en función del sentido. En el primero de los supuestos, la distancia deberá entenderse como la distancia promedio o número medio de saltos que siguen los mensajes entre una cierta pareja de procesadores. El segundo supuesto tiene una resolución más compleja porque implica la necesidad de ampliar el modelo de grafo TIG de forma que cada arco original entre dos nodos se desdoble en dos arcos direccionales, cada uno de ellos correspondiente a la comunicación en uno de los dos posibles sentidos entre ambos procesos y caracterizados por el volumen de comunicación global en dicho sentido; lógicamente, la suma del volumen de comunicación de ambos arcos deberá ser el valor que tenía el arco original del TIG que unía a la pareja de nodos.

- cuando se usa la función de coste *minimax* el hecho de multiplicar cada volumen de comunicación por la distancia supone alterar ligeramente la interpretación física que poseía esta función de coste. Como se mencionó en el segundo capítulo, la función *minimax* podía interpretarse físicamente si consideramos que está midiendo la carga de trabajo que debe realizar un procesador concreto, tanto de cómputo puro, como de cómputo involucrado en la realización de las comunicaciones. Al multiplicar el volumen de comunicación por la distancia, el coste asociado a cada procesador se ve incrementado por el tiempo de latencia asociado a todos los mensajes. Este incremento puede ser fuente de imprecisiones por dos motivos. En primer lugar, porque se están sumando valores que pueden estar expresados en unidades distintas. La latencia se debería medir en unidades que estarían relacionadas con la red de interconexión y el rendimiento de la misma, mientras que los otros términos de la función de coste se miden en unidades relacionadas con el rendimiento específico del procesador. En segundo lugar, no se tiene en cuenta la posible intervención de los procesadores intermedios que atraviesa un mensaje en su transmisión desde el procesador origen al procesador destino. En este sentido, podría pensarse en una forma alternativa de computar el coste de encaminamiento de los mensajes donde se

consideraría que la transmisión de un mensaje va a repercutir de forma equivalente sobre todos los procesadores que están en el camino entre el procesador origen y el procesador destino. Así, todos ellos van a ver su coste de comunicación incrementado por el volumen de comunicación asociado al arco del TIG que los atraviesa, reflejando así la carga de trabajo que soporta cada procesador para gestionar los mensajes de tránsito.

Si tenemos en cuenta los enfoques adoptados en la literatura se deduce claramente que, en general, han sido una franca minoría los autores que han intentado corregir alguno de los errores mencionados. Citemos, por ejemplo, a [Lee87] y [Cha93] que pueden diferenciar el sentido de los mensajes gracias al uso de un modelo de grafo dirigido, o a [Wu94] que incorpora el coste de una comunicación a todos los procesadores intermedios atravesados por el mensaje.

De todos modos, la corrección de los mencionados errores se justificaría en el supuesto de que, dado un programa paralelo, quisiéramos cotejar los resultados de la asignación y de la función de coste con el tiempo de ejecución de dicho programa sobre un sistema real, con objeto de ver el grado de adecuación o correlación entre los valores obtenidos por la política de asignación y los tiempos reales. Lógicamente, si quisiéramos que los valores de la función de coste fuesen más precisos y más representativos de lo que va a suceder en la ejecución real de una aplicación en el cálculo de la función de coste deberían añadirse todas las correcciones mencionadas. Sin embargo, en nuestro estudio obviaremos la inclusión de dichas correcciones por razones de simplicidad en el diseño de los algoritmos. Nótese, sin embargo, que ninguna de las soluciones propuestas sería descartable por el hecho de suponer la mencionada simplificación. En realidad, todas las correcciones a que hemos hecho referencia podrían incluirse en cualquiera de los algoritmos propuestos, a expensas, eso sí, de una mayor complejidad tanto en las estructuras de datos utilizadas como en los cálculos necesarios para obtener el valor de la función de coste.

Contención de la red

Los factores estudiados en los apartados anteriores se corresponden con características de lo que podríamos denominar la red de interconexión estática. Una problemática diferente aparece cuando se tiene en cuenta la red de interconexión como un elemento dinámico que a lo largo de la ejecución de un cierto programa va a tener un comportamiento variante y que, en la práctica, va a significar la aparición de un nuevo fenómeno: la contención. La aparición de este fenómeno se puede explicar porque durante la ejecución de un programa los mensajes van a competir por el uso de los enlaces físicos de la arquitectura, de tal forma que, si en un mismo instante dos mensajes deben circular por el mismo enlace, el sistema deberá secuencializar su paso por dicho enlace, lo que supondrá la inclusión de un retardo adicional a uno de los dos mensajes.

La contención de la red es un fenómeno dinámico y por ello su consideración pasa por la adopción de modelos que permitan capturar cierta información temporal del comportamiento de las comunicaciones del programa paralelo al lo largo de su ejecución. Los modelos que podrían ser candidatos serían por una parte, el modelo

clásico de grafos dirigidos acíclicos (DAGs), o algunas extensiones al modelo de TIGs que incluyen conceptos como el de *fases de comunicación* [Lo92][Lee87][Cha93]. En la práctica han sido pocos los autores que hayan estudiado, sin embargo, la problemática de la contención en la red.

En el caso de los DAGs los autores [ElR90] la han tenido en cuenta aunque ha sido más como un estudio de su influencia en distintas arquitecturas que como un factor que la estrategia de asignación maneje en el momento de tomar sus decisiones.

En el caso de las extensiones al modelo TIG, algunos autores [Lo92] tienen en cuenta la información sobre las fases de comunicación para determinar el mejor encaminamiento de los mensajes una vez hecha la asignación. El conocimiento de dichas fases de comunicación no influye de ninguna forma en la estrategia de asignación, que maneja el grafo del programa como si fuese un TIG puro. Los otros autores, [Lee87] y [Cha93], proponen el uso de las fases de comunicación como una información adicional a tener en cuenta para el cálculo preciso del coste de comunicaciones, de forma que se tenga en cuenta la posible existencia de contenciones en la red. La definición de fase de comunicación que se maneja tanto en [Lo92] como en [Lee87] y [Cha93] es además bastante simple pues supone que todos los procesos entran en cada una de dichas fases al mismo tiempo y tienen la misma duración para todos. En este sentido, y como sucediera con las correcciones a la función de coste que hemos mencionado en el apartado anterior, las soluciones propuestas en este capítulo podrían contemplar el fenómeno de la contención a la hora de calcular el coste de una cierta asignación de forma similar a como lo hacen [Lee87] y [Cha93].

De nuevo, por motivos de simplicidad se ha obviado dicho cálculo (que hubiese supuesto la consideración de un modelo de grafo substancialmente más complejo). De hecho, los propios [Lee87] y [Cha93] en sus estudios muestran la aplicación de sus estrategias sobre ejemplos que son de un tamaño muy pequeño y donde apenas aparecen fases de comunicación, lo que, desde nuestro punto de vista, convierte a sus propuesta más en una metodología teórica que en una metodología aplicable en la práctica de la que se conocen claramente los costes computacionales debidos a la extensión del modelo frente a la mejor adecuación de los valores de la función de coste y los tiempos de ejecución.

5.2 Estrategias de asignación de tareas teniendo en cuenta la arquitectura

Una vez analizados los factores que influyen en la resolución del problema del *mapping* en arquitecturas no totalmente interconectadas y considerando además las distintas soluciones que aparecen en la literatura para diseñar estrategias de *mapping* que tengan en cuenta una arquitectura concreta, presentaremos en esta sección la solución que hemos adoptado para resolver el citado problema. Nuestra solución pertenece al grupo de estrategias multietapa que calculan la asignación en dos pasos: el primero, reduce el grafo hasta que tenga tantos nodos (procesos) como procesadores

(*mapping* N a 1) y el segundo asigna un procesador físico a cada uno de dichos nodos (*mapping* 1 a 1).

Así, la asignación de las tareas de un programa a los procesadores de un cierto sistema se divide en dos pasos fundamentales: **contracción** y **asignación física** (también conocida por ***layout*** por algunos autores). En el primer paso se pretende reducir el grafo del programa de forma que aquellas comunicaciones más pesadas sean suprimidas y se consiga un equilibrio entre la carga que van a tener todos los procesadores. El segundo paso de asignación física parte de un grafo reducido donde el número de nodos es igual o inferior al número de procesadores y se trata de resolver el problema de calcular una asignación nodo-procesador intentando que las comunicaciones presentes en el grafo reducido (contraído) coincidan al máximo con los enlaces físicos de la arquitectura. De alguna forma, el problema de la asignación física podemos considerar que es equivalente al problema que pretendía resolver Bokhari y que ya presentamos al comentar su solución en el capítulo 2 (apartado 2.3.2). En el caso que nos ocupa, sin embargo, el grafo que pretendemos asignar no es tan simple como el considerado por Bokhari ya que los arcos no tienen todos el mismo volumen y el volumen de cómputo tampoco tiene porqué ser igual.

Este cambio en los parámetros del grafo obligaría a extender y generalizar la solución propuesta por Bokhari. En el entorno PYRROS, por ejemplo, podemos encontrar una aproximación para resolver el problema de la asignación física basada en una adaptación al algoritmo de Bokhari donde se utiliza el volumen total de comunicaciones como parámetro a optimizar y se usa la estrategia original de Bokhari que va realizando intercambios de parejas de nodos. La complejidad del algoritmo es de tipo $O(K^3)$, siendo K el número de procesadores del sistema.

Otra alternativa para la resolución del problema de la asignación física consiste en el uso de un algoritmo que algunos autores denominan de incrustación (*embedding*) [Lo91] y que está presente, por ejemplo, en el entorno OREGAMI. El algoritmo de incrustación no es más que una estrategia heurística de tipo *greedy* que obtiene una asignación a los procesadores físicos intentando minimizar el volumen total de comunicaciones del sistema. En el caso concreto de OREGAMI, el algoritmo construye una lista ordenada de los arcos del grafo agrupado en orden decreciente. En caso de igualdad se considera también el peso de los otros arcos adyacentes a los arcos con pesos iguales. Una vez construida la lista, el algoritmo la recorre linealmente y para cada arco asigna los nodos de sus extremos de la siguiente forma:

- si ambos nodos ya están asignados entonces no se hace nada.
- si sólo hay un nodo asignado, el otro se coloca en el procesador libre más cercano.
- si ninguno de los dos nodos ha sido asignado, se escoge aleatoriamente un procesador libre en el que se asigna uno de los nodos y el otro se coloca su vecino libre más cercano.

Suponiendo que M es el número de arcos del grafo agrupado, la complejidad de esta estrategia es, básicamente, O(MlogM) debido al paso de ordenación de dichos arcos. El recorrido posterior tiene sólo una complejidad lineal.

Otro método de incrustación, aunque sus autores no lo denominen de esa forma, sería el utilizado en [Lee87] y [Cha93]. En este caso, se asigna inicialmente el nodo con mayor volumen de comunicación. A partir de este punto se va seleccionando al nodo con un mayor volumen de comunicación con los nodos ya asignados y se lo coloca en el procesador físico en el que se minimice la función de coste escogida. En la propuesta de [Lee87] este método se aplica a un grafo de programa que originalmente cuenta con un número de nodos no superior al número de procesadores. En el caso de [Cha93], el mismo método se utiliza para asignar un programa con un número de procesos superior al de procesadores, por lo que el algoritmo constituye una estrategia de asignación completa más que un método de incrustación.

Frente a la resolución del problema del *mapping* mediante un paso de agrupación (o contracción) seguido de un paso de asignación física cabe la posibilidad de resolverlo mediante un único paso. Sin embargo, estos métodos presentan un doble inconveniente. En algunos de ellos (como [Cha93]) el hecho de ir colocando físicamente los nodos uno a uno obliga a tener en cuenta en cada caso sólo la información de los nodos ya asignados, por lo que se pueden tomar decisiones que inhiban posteriores asignaciones que resulten más interesantes: se puede agrupar un nodo con uno ya colocado cuando la mejor opción sería agruparlo con otro nodo que todavía no ha sido colocado. En otros casos (como ocurre en [Erc90], [Sad87] y [Sad90]), el diseño de la estrategia va a tener una importante dependencia de la arquitectura o del tipo de grafo de programa que dificultará su generalización para otras arquitecturas u otro tipo de grafos.

Dada nuestra voluntad de diseñar una estrategia de propósito general que pueda aplicarse a cualquier tipo de grafo y arquitectura, creemos más conveniente optar por la metodología multietapa y usar nuestro algoritmo de agrupación y reasignación en el paso de contracción, y otro método específico para la etapa de asignación física.

Lógicamente, el hecho de intentar optimizar el problema del *mapping* mediante el uso de dos pasos subóptimos producirá resultados que, en general, serán peores que los obtenidos con un método que lo resuelva en un único paso. De todas formas, ante la falta de una estrategia de tipo general y de complejidad computacional baja para solucionar el problema en un sólo paso, varios son los autores que han optado también por la metodología multietapa y han obtenido resultados satisfactorios sin requerir estrategias de elevada complejidad [Lo91][Yan92].

5.2.1 Estrategias de asignación física

De lo expuesto en el apartado anterior, hemos reducido el problema del *mapping* considerando una arquitectura punto a punto a un problema de agrupación seguido de un problema de asignación física. El método estudiado en los capítulos precedentes será el encargado de resolver el primero de los problemas; en este apartado nos centraremos en el estudio de posibles estrategias pensadas para la resolución del segundo de los problemas.

Ya hemos comentado previamente dos posibles alternativas para su solución: una variante del algoritmo de Bokhari y el uso de heurísticas *greedy* (lo que denominamos métodos de incrustación). De hecho, ambas opciones no tienen porqué ser excluyentes y en nuestra aproximación hemos optado por conjugar ambos métodos, de forma que usaremos una estrategia heurística de incrustación y posteriormente se intentará mejorar la colocación mediante un algoritmo iterativo adaptado del método de Bokhari. Así, el algoritmo de Bokhari no empezará desde una situación aleatoria como en la propuesta original sino que intentará realizar la optimización a partir de una colocación inicial relativamente buena.

Tanto en la estrategia de incrustación como en la fase iterativa posterior, los algoritmos manejan e intercambian nodos que son las agrupaciones obtenidas de la fase de contracción. Cada una de estas agrupaciones está compuesta, a su vez, por un cierto número de tareas de menor tamaño y que corresponden con los tareas independientes del programa original. Sería posible en este punto intentar un mayor refinamiento de la asignación final mediante el uso de un método iterativo basado en la realización de movimientos e intercambios a nivel de las tareas originales del programa. Ya vimos en su momento que esta fase iterativa, especialmente por lo que a los intercambios se refiere, era la que introducía mayor complejidad global a la estrategia. Incluir otra fase de este tipo significará doblar el tiempo de cómputo debido a la nueva fase iterativa. Una alternativa que no suponga ese incremento en la complejidad computacional sería la ejecución de una única fase de refinamiento teniendo en cuenta sólo la arquitectura. Otra alternativa consistiría en la realización sólo de movimientos en ambas ocasiones.

Así, el conjunto de 5 alternativas que se ha evaluado para resolver el problema del *mapping* considerando la topología es el siguiente:

- Algoritmo de agrupación (CA) + refinamiento iterativo (movimientos e intercambios: CRME) seguido de:
    1. Algoritmo heurístico de incrustación
    2. Algoritmo de incrustación + algoritmo iterativo tipo Bokhari
    3. Algoritmo de incrustación + algoritmo iterativo de tipo Bokhari + algoritmo iterativo de refinamiento (movimientos + intercambios)

- Algoritmo de agrupación (CA) seguido de:
    4. Algoritmo de incrustación + algoritmo iterativo de tipo Bokhari + algoritmo iterativo de refinamiento (movimientos e intercambios)

- Algoritmo de agrupación (CA) + refinamiento iterativo (movimientos: CRM) seguido de:
    5. Algoritmo de incrustación + algoritmo iterativo de tipo Bokhari + algoritmo iterativo de refinamiento (sólo movimientos)

A continuación, se detallará los aspectos más importantes de cada uno de los algoritmos utilizados en estas cinco alternativas.

Algoritmo de incrustación

El algoritmo de incrustación que se ha desarrollado es un híbrido entre los dos métodos comentados anteriormente de esta misma categoría. Como el método de [Lee87] se empieza asignando un procesador físico a aquel nodo que tiene un mayor volumen de comunicación total. A partir de este punto, se van a ir asignando nodos que sean vecinos de los ya asignados de uno en uno. Para ello, cuando un nodo sea asignado todos sus arcos se integrarán a una lista de arcos ordenados en función del volumen de comunicación asociado a ellos. De esta lista se escoge un arco cada vez y se sigue el mismo criterio de asignación explicado en el caso de [Lo91], aunque en nuestro caso todos los arcos tendrán siempre un nodo asignado como mínimo. Por lo tanto, bastará con asignar el nodo del otro extremo en el procesador que garantice un menor volumen de comunicación para ese nodo con respecto a los otros nodos ya asignados físicamente.

De esta forma evitamos los casos que se podían presentar en el algoritmo de Lo en los que había que asignar dos nodos a la vez y uno de ellos era colocado aleatoriamente en cualquier procesador libre. Nuestra versión no requiere nunca tomar decisiones aleatorias de ese tipo e intenta siempre ir colocando los nodos de forma que respeten en la mayor medida posible sus relaciones de vecindad con los demás nodos. En este sentido, el método se asemeja más al criterio propuesto en [Lee87].

La figura 5.2 muestra el pseudo-código del mencionado algoritmo.

```
t_i := encontrar nodo con mayor volumen de comunicación;
asignar (t_i, p_0)
lista_arcos := {}
Para todos los arcos v_ij de t_i hacer
        lista_arcos := lista_arcos + {v_ij}
Mientras lista_arcos ≠ {} hacer {
        v_kh := elemento máximo de lista_arcos
        Si t_k esta asignado entonces
                n_candidato := t_h
        sino
                n_candidato := t_k
        coste_min := ∞
        Para todos los procesadores p_j desocupados hacer {
                coste_actual := evaluar coste asignar n_candidato en p_j
                Si coste_actual < coste_min entonces {
                        p_candidato := p_j
                        coste_min := coste_actual
                }
        }
        asignar n_candidato a p_candidato
        lista_arcos := lista_arcos - {v_kh}
        Para todos los arcos e_ij de n_candidato hacer
                lista_arcos := lista_arcos + {e_ij}
}
```

Figura 5.2 Algoritmo de incrustación

En el algoritmo no se contempla la posibilidad de que más de un nodo sea asignado a un mismo procesador físico porque partimos de la base de que la fase de contracción ha reducido el grafo de tal forma que ya ha eliminado todo el paralelismo inútil.

La complejidad del método viene determinada por las veces que se ejecuta el lazo principal y se intenta asignar un nuevo nodo. En total se asignarán K - 1 nodos (siendo K el número de procesadores) y para cada uno de ellos se evaluará su colocación en todos los procesadores disponibles. Esto hará que la complejidad del algoritmo sea de $O(K^2)$.

Algoritmo iterativo de Bokhari

En el capítulo 2 ya presentamos este método clásico de asignación. Baste recordar que en su versión original estaba diseñado para un grafo de programa donde los volúmenes de cómputo y de comunicación eran unitarios y el resultado de la asignación se media en función de cuantos arcos del grafo del programa coincidían directamente con arcos de la arquitectura. El algoritmo basaba su optimización en la realización de intercambios de nodos a partir de una primera asignación aleatoria y también utilizaba intercambios aleatorios a lo largo de su ejecución para evitar situaciones de mínimo local.

Lógicamente, la versión básica original del algoritmo de Bokhari no es aplicable al problema actual y deberemos adaptarlo teniendo en cuenta los parámetros que presentan nuestros grafos de programa. La variación más importante reside en considerar el volumen de comunicación de los arcos. La bondad de una determinada configuración no se medirá en el número de arcos coincidentes con enlaces de la arquitectura sino que deberemos utilizar alguna función que tenga en cuenta, por ejemplo, el volumen total de comunicaciones multiplicando el peso de cada arco por la distancia que separa a sus nodos extremos.

La otra gran variación introducida reside en la supresión de la asignación inicial aleatoria y los intercambios aleatorios a lo largo del proceso de mejora. Con estas dos medidas pretendemos evitar que la aplicación del algoritmo a un mismo grafo proporcione resultados distintos cuya única justificación reside en el valor que determinada variable aleatoria tomó en cada una de las ejecuciones. En este sentido, somos de la opinión que un posible usuario en la práctica de una política de *mapping* esperará de ella un comportamiento determinista; que, aunque la asignación no sea óptima, siempre sea igual si no cambian ni el programa ni las condiciones físicas del sistema. En definitiva, el comportamiento de la política de asignación debe parecerse al de un compilador, que siempre genera el mismo código ejecutable si compilamos el mismo programa.

Junto con la supresión de la aleatoriedad en los cambios realizados a los largo del proceso de optimización también se ha diseñado una estrategia sistemática para probar los intercambios entre tareas. Se busca en primer lugar cuál es el nodo con un mayor volumen de comunicación asociado y se le intenta intercambiar con todos los

demás. De todos los intercambios posibles se escoge aquel que repercuta en una mayor disminución del coste global debido a las comunicaciones. Si no existe ningún intercambio de este nodo que proporcione ninguna mejora se escoge el siguiente nodo con mayor volumen de comunicación asociado y se repite el proceso de intercambio con todos los demás. Mientras los nodos probados no proporcionen ningún intercambio que mejore la situación, se van seleccionando los nodos candidatos en orden inverso a su volumen asociado de comunicación. En el momento en que uno de esos nodos proporcione una mejora se vuelve a empezar considerando de nuevo como candidato el nodo con mayor volumen de comunicación asociado.

El algoritmo final utilizado es el que muestra la figura 5.3.

```
Ordenar nodos en función de su volumen de cómputo
com_global := Suma del volumen de comunicación de todos los nodos
n_candidato := primer elemento de la lista de nodos ordenados
fin := FALSE
Hacer {
        mejor_coste := com_global
        mejora := FALSE
        Para todos los procesadores p_i hacer {
                coste_intercambio := evaluar el intercambio de n_candidato y p_i
                Si coste_intercambio < mejor_coste entonces {
                        p_candidato := p_i
                        mejor_coste := coste_intercambio
                        mejora := TRUE
                }
        Si (mejora) entonces {
                intercambiar n_candidato y p_candidato
                actualizar la lista de los nodos ordenados
                com_global := mejor_coste
                n_candidato := primer elemento de la lista de nodos ordenados
        }
        sino {
                n_candidato := buscar siguiente nodo en la lista de nodos
                Si (n_candidato = NULL) entonces
                        fin := TRUE
        }
} hasta (fin)
```

Figura 5.3 Algoritmo de Bokhari adaptado

La adaptación realizada al algoritmo de Bokhari tiene en cuenta sólo el valor que en cada instante va tomando la suma global de todas las comunicaciones. Sin embargo, durante la primera fase de contracción del grafo el criterio de optimización tenía en cuenta el volumen de cómputo y de comunicación de cada nodo individual. No se consideraba en ningún momento algún parámetro global sino sólo el máximo de los valores individuales. En este sentido, la versión que muestra la figura 5.3 contradice ligeramente el criterio de optimización que utilizaremos finalmente mediante la función

minimax. Por ello, es posible introducir una pequeña variación al algoritmo mostrado de forma que tenga en cuenta la optimización del valor de la función *minimax*.

La variación introducida se refleja, en primer lugar, en la ordenación de la lista de nodos. El orden seguido se basa en el valor de la función *minimax* para cada uno de ellos. Además, cuando el nodo candidato es el primero de la lista el criterio para aceptar un intercambio es que disminuya su coste de la función *minimax* y, en consecuencia, que disminuya el coste de la función para el conjunto de la asignación. Si no se encuentra ningún intercambio que mejore la situación para este nodo entonces se buscan intercambios entre los restantes nodos de la lista siguiendo su orden dentro de ella, como ya hiciera la versión anterior. Con éstos se aceptará un intercambio siempre que mejoren el volumen global de comunicaciones y no empeoren el valor de la función minimax hasta ese momento. De este modo se intenta provocar un cambio que permita posteriores intercambios por parte del nodo que define el coste *minimax* de la asignación.

Esta segunda variante del método de Bokhari se muestra en la figura 5.4.

La complejidad de las dos adaptaciones propuestas del algoritmo de Bokhari tienen una complejidad que está en función del número de veces que se intente hacer un intercambio de todos los nodos entre sí (de coste K^2) y se vaya mejorando la función de coste. El número de veces que se hará esta repetición dependerá del grafo en cuestión pero basándonos en los resultados de Bokhari, en la práctica estará acotado por K. De esta forma, la complejidad máxima de ambos algoritmos será de $O(K^3)$.

```
Ordenar nodos en función de su valor de la función minimax
com_global := Suma del volumen de comunicación de todos los nodos
n_candidato := primer elemento de la lista de nodos ordenados
fin := FALSE
optimizar_minimax := TRUE
coste_minimax := coste minimax de n_candidato
Hacer {
        Si (optimizar_minimax) entonces
                mejor_coste := coste minimax de n_candidato
        sino
                mejor_coste := com_global
        mejora := FALSE
        Para todos los procesadores p_i hacer {
                Si (optimizar_minimax) entonces
                        coste_intercambio := evaluar intercambio de n_candidato y
                                             p_i siguiendo criterio de reducción
                                             de función minimax
                sino {
                        coste_intercambio := evaluar intercambio de n_candidato y
                                             p_i siguiendo criterio de reducción de
                                             comunicaciones globales
                        nuevo_minimax := evaluar intercambio de n_candidato y
                                             p_i siguiendo criterio de reducción
                                             de función minimax
                }
                Si (coste_intercambio < mejor_coste) AND
                ((optimizar_minimax ) OR (no optimizar_minimax AND
                 nuevo_minimax ≤ coste_minimax)) entonces {
                        p_candidato := p_i
                        mejor_coste := coste_intercambio
                        mejora := TRUE
                }
        Si (mejora) entonces {
                intercambiar n_candidato y p_candidato
                actualizar la lista de los nodos ordenados
                Si (optimizar_minimax) entonces
                        coste_minimax := mejor_coste
                sino
                        coste_minimax := nuevo_minimax
                com_global := Suma del volumen de comunic. de todos los nodos
                optimizar_minimax := TRUE
                n_candidato := primer elemento de la lista de nodos ordenados
        }
        sino {
                n_candidato := buscar siguiente nodo en la lista de nodos
                optimizar_minimax := FALSE
                Si (n_candidato = NULL) entonces
                        fin := TRUE
        }
} hasta (fin)
```

Figura 5.4 Algoritmo de Bokhari adaptado considerando la función minimax

Algoritmos iterativos de refinamiento

Como ya hemos comentado previamente, los algoritmos iterativos de refinamiento intentarán reasignar tareas individuales entre los distintos procesadores siguiendo la misma filosofía que ya se comentó al mencionar la estrategia de contracción. Los algoritmos utilizados intentan movimientos individuales e intercambios respectivamente y son prácticamente iguales a los que se presentaron en el capítulo 3, por lo que no los volveremos a reproducir aquí.

En ambos casos las diferencias respecto a las versiones del capítulo 3 residen en el cálculo de la función de coste porque se tiene en cuenta la distancia entre los procesadores para multiplicarla por el volumen de comunicación de cada arco del grafo de programa. La otra diferencia importante afecta tan sólo a las operaciones necesarias para actualizar el valor de la función de coste. Cuando se hacen movimientos o intercambios de tareas teniendo en cuenta la arquitectura se debe recalcular el valor de la función de coste para todos los procesadores con los cuales estén conectadas las tareas reasignadas. Esto supone un cierto incremento de complejidad frente a la versión que no considera la arquitectura porque en este caso basta recalcular sólo los valores para el procesador origen y el destino, quedando los restantes inalterados.

Del mismo modo que se hizo con la estrategia de contracción, también ahora se ha probado una versión de "baja" complejidad en la que la única técnica de mejora se basa en hacer movimientos individuales en todas las etapas.

5. 3 Modelo de estimación del coste de la asignación física

Una vez planteadas las distintas alternativas que se han propuesto para resolver el problema de la asignación física podemos formular un modelo que permita cuantificar una cota *a priori* del coste que va a suponer el hecho de realizar esta asignación, entendiendo por coste el valor que proporciona la función *minimax*. La estimación se realizará a partir de parámetros del grafo contraído y de la arquitectura, teniendo presente también que se ha intentado usar el menor número de parámetros y que a su vez fuesen de fácil obtención.

Podemos considerar inicialmente que la arquitectura ideal sería aquella que tuviese tantos enlaces físicos (*links*) como arcos presentase un determinado grafo de programa una vez pasado por la fase de contracción pues, en este supuesto, sería posible hallar una asignación física que mantuviese el coste global de la asignación: sería aquella asignación que haría coincidir uno a uno los arcos del grafo del programa con los *links* de la arquitectura. En la medida que el número de *links* de la arquitectura sea inferior al número de arcos del grafo del programa o que existan nodos con un número de arcos mayor al número de *links* de que dispone cada procesador, la fase de asignación física va a suponer indefectiblemente un aumento del valor de la función de coste frente al valor alcanzado en la fase de contracción. Por este motivo, aquellos grafos de programa que, una vez contraídos, presenten un grado de conectividad importante van a experimentar un aumento importante en el valor de la función de coste una vez realizada

la fase de asignación física. El mismo fenómeno se observará cuando se realice la asignación en arquitecturas que tengan un reducido número de *links*, como los anillos.

Lógicamente, el aumento de la función de coste que se experimentará en la fase de asignación física se deberá en exclusiva al aumento del coste de determinadas comunicaciones que van a verse multiplicadas por un factor de distancia superior a uno. Suponiendo el uso de la función de coste *minimax*, ese aumento de comunicaciones repercutirá posteriormente en el incremento del coste global de cada uno de los procesadores, aumentando en consecuencia el valor final de la función de coste. Por ello, vamos a desglosar el modelo de previsión del incremento de la función de coste en dos:

- en primer lugar, presentaremos un modelo que va a estimar el incremento que experimentarán las comunicaciones,
- en segundo lugar, se presentará un modelo que estimará el incremento que experimenterá la función de coste *minimax*.

Estos modelos pueden usarse de dos formas. El primer uso serviría para tener un cierto valor teórico estimado de la nueva función de coste dado un determinado grafo y una cierta arquitectura, valor que nos diría, en definitiva, qué empeoramiento podemos esperar para dicho grafo asignado a esa arquitectura. El segundo uso permitiría catalogar las distintas estrategias de asignación física en función del nivel de acercamiento a dicho valor. Así, se podría decir que una cierta estrategia de asignación física proporciona globalmente unos resultados que se aproximan en un porcentaje superior o inferior al X% a la cota teórica estimada, ya sea para cualquier arquitectura o para una arquitectura concreta. Lógicamente, hacer una previsión del empeoramiento que parece razonable a tenor de las características del grafo del programa y de las características de la arquitectura no significa que siempre vaya a conseguirse estar cerca de esos valores. La eficacia de una estrategia de asignación física influirá en que el coste de la asignación final se aproxime a ese valor de coste "razonable" o no.

Incremento del volumen global de comunicaciones

El estimador para el coste adicional debido a comunicaciones es:

$$comm\_adic = \frac{1}{2}\left(\sum_{\forall i \in Grafo Contraido} max\left(\left(N\_arcs_i - N\_links \right), 0 \right) \right) * \bar{c} * \overline{d\_inc} \qquad \text{(ec. 5.1)}$$

donde, N_arcs_i es el número de arcos del nodo i del grafo resultante de la fase de contracción, N_links es el número de *links* de cada procesador para una arquitectura concreta, $\bar{c}$ es el volumen promedio de comunicación del grafo contraído y $\overline{d\_inc}$ es la *distancia promedio de incremento de comunicaciones* que existe entre los procesadores de la arquitectura.

El valor de comm_adic supone una estimación del incremento que experimentará el conjunto global de comunicaciones. Teniendo en cuenta el número de *links* que tienen los procesadores de una cierta arquitectura (N_links), la fórmula de comm_adic considera cuantos arcos tiene cada nodo del grafo contraído por encima de N_links y

suma el total de esos arcos calculando la diferencia entre el número de arcos (N_arcos_i) del nodo y N_links. El resultado de esa resta es el número de arcos que necesariamente deberán estar a distancia mayor que uno. Si el número de arcos de un nodo es inferior al número de *links* la diferencia será un número negativo y, en consecuencia, en el sumatorio se utilizará el valor 0 para ese nodo. De este modo, el sumatorio final contendrá el número de arcos que necesariamente van a estar a distancia mayor que uno, valor que se divide por 2 para eliminar todos los arcos que se habrán considerado dos veces (una vez al contar los arcos de cada uno de los nodos unidos por dicho arco). El volumen de estos arcos ya ha sido considerado una vez en el coste global de comunicaciones, por lo tanto, el incremento de ese coste global será proporcional al volumen del arco multiplicado por la (distancia - 1) a la que se encuentren los procesos asociados a cada uno de esos arcos. Así, si los procesos se encuentran a distancia 2, el incremento del coste será una vez el volumen del arco; si la distancia es 3, el incremento será dos veces el volumen del arco, etc. El volumen de comunicación adicional se verá multiplicado por un factor proporcional a la distancia entre los procesadores, término que denominamos distancia promedio de incremento de comunicaciones $\overline{d\_inc}$.

De acuerdo con la definición que se dio de distancia entre dos procesadores d_{ij} en el sección 2.1 del segundo capítulo, podemos calcular la distancia promedio de un procesador, d_i, calculando la media de todas sus distancias con los demás procesadores. En el caso de arquitecturas simétricas donde todos sus procesadores tienen el mismo valor d_i, éste es el que definiremos como distancia promedio de la arquitectura, $\overline{D}$. La distancia promedio de incremento de comunicaciones ($\overline{d\_inc}$) se obtiene restando una unidad a todas las distancias de un cierto procesador y, a continuación, calculando el promedio de aquellas que son mayores de cero. La tabla 5.1 muestra para varias arquitecturas, que han sido utilizadas en nuestra experimentación, los valores correspondientes a la distancia promedio de la arquitectura, $\overline{D}$, y la distancia promedio de incremento de comunicaciones, $\overline{d\_inc}$.

Tabla 5.1. Parámetros característicos de arquitecturas punto a punto

Topología	Num. procesad.	Num. *links*	$\overline{D}$	$\overline{d\_inc}$
malla cerrada	9	18	1,5	1
malla cerrada	16	32	2,1	1,6
malla cerrada	32	64	3,1	2,4
hipercubo	8	12	1,7	1,3
hipercubo	16	32	2,1	1,6
hipercubo	32	80	2,6	1,9
anillo	8	8	2,3	1,8
anillo	16	16	4,3	3,8
anillo	32	32	8,1	7,8

En la estimación de comm_adic, por tratarse de una estimación global genérica, no se utiliza el valor individual de los volúmenes de determinados arcos sino que se usa el volumen promedio de las comunicaciones como valor genérico para todos los arcos que van a estar a distancia mayor de uno, multiplicado por el término de distancia promedio de incremento de comunicaciones. Lógicamente, se podrían formular estimaciones más "optimistas" considerando, por ejemplo, sólo el volumen de aquellos

arcos con menor volumen de comunicación, con objeto de formular una estimación orientada a medir la capacidad de la estrategia de asignación física en conseguir que sean los arcos menos pesados los que se encuentren a distancia mayor de uno.

Por otra parte, en aquellos casos donde el grafo contraído tenga un grado de conectividad bajo la división por dos que se realiza para no contar dos veces un mismo arco puede ocasionar que se consideren menos arcos de los que realmente van a estar a distancia mayor que uno. Esta situación se refleja en la figura 5.5. Si suponemos que el grafo de la figura 5.5 va a ser asignado sobre un anillo de 7 procesadores, la fórmula anterior tiene en cuenta sólo un arco cuando deberían ser dos. Los únicos nodos que tiene un número de arcos superior a dos son t_3 y t_5; para cada uno de estos nodos, uno de sus tres arcos necesariamente estará a distancia superior a uno por lo que el número de arcos a distancia mayor que uno será dos en total. En este caso, los dos nodos no comparten ningún arco por lo que la división por dos efectuada en comm_adic para no contar dos veces un mismo arco produce una subestimación en el número de arcos que van a estar a distancia mayor que uno.

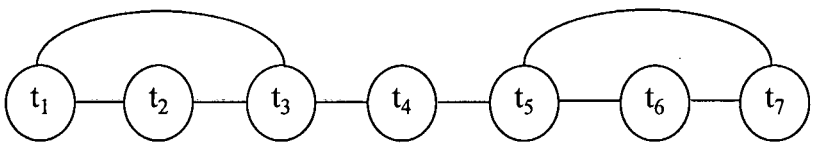


Figura 5.5 Ejemplo de grafo con subestimación de arcos en comm_arc

De todas formas, si consideramos que, en general, los grafos contraídos suelen ser bastante homogéneos y tanto las desviaciones sobre el número de arcos por nodo o el volumen de comunicación de los arcos no suelen ser muy elevadas, el error cometido por la presente formulación no será significativo y permitirá simplificar el cálculo de la estimación.

Incremento de la función de coste

Por lo que se refiere al nuevo valor que tomará la función de coste para un cierto *mapping*, su estimador será:

$$cost\_map\_arq = cost\_map + (Max\_arcs - N\_links) * \bar{c} * \overline{d\_inc} \qquad \text{(ec. 5.2)}$$

donde *cost_map* corresponde al valor de la función de coste del grafo contraído, Max_arcs es el número de arcos máximo que tienen los nodos del grafo contraído, y como en el caso anterior, N_links es el número de *links* de la arquitectura, $\bar{c}$ es el volumen de comunicación promedio del grafo contraído y $\overline{d\_inc}$ es la distancia promedio de incremento de comunicaciones.

Con el estimador anterior se está considerando que el incremento de la función de coste va a estar acotado por el coste de comunicaciones adicional debido a todos los arcos que vayan a estar a distancia mayor que uno de aquel nodo del grafo que tenga mayor número de arcos. Podemos justificar la formulación de este estimador porque, de

nuevo, el grafo contraído suele presentar un conjunto de nodos cuyo coste individual (teniendo en cuenta tanto cómputo como comunicaciones) está equilibrado, por lo tanto, en la fase de asignación física el nuevo valor de la función de coste va a depender fuertemente del incremento que el factor de comunicaciones experimente en aquel nodo que mayor número de arcos tenga.

5.4 Resultados experimentales

Para evaluar las diferentes estrategias de asignación física que se han descrito en el apartado 5.2 se utilizó el mismo conjunto de grafos regulares e irregulares que el utilizado en los experimentos 2 y 3 del capítulo anterior. Para todos esos grafos se calculó su asignación sobre tres tipos distintos de arquitecturas: anillo, hipercubo y malla cerrada (*wrap-around mesh*). El número de procesadores para cada tipo de arquitectura fue, a su vez, de 8, 16 y 32. Hubo una única excepción para la arquitectura de malla donde se utilizó una topología de 9 procesadores en vez de 8 para conseguir un sistema simétrico de dimensión 3*3 (en el caso de la malla de 32 procesadores se optó por definir un sistema asimétrico de 4*8).

Evaluación de las estrategias de asignación física

En la experimentación realizada se ha evaluado, por un lado, cuál era el incremento de la función de coste y, por otro, cuál era el incremento global de las comunicaciones. Las tablas 5.2 y 5.3 muestran los resultados globales referidos al incremento de la función de coste y al incremento de las comunicaciones, respectivamente, obtenidos por cada una de las estrategias evaluadas respecto a los valores obtenidos por la estrategia CRME sin considerar la arquitectura (los datos utilizados en el cálculo de estos resultados están en el Apéndice D). Las siglas utilizadas para designar cada una de las estrategias son las siguientes:

- EMB: Algoritmo de agrupación (CA) + refinamiento iterativo (movimientos e intercambios: CRME) seguido de algoritmo heurístico de incrustación (fig 5.2).
- BOK_1: Algoritmo EMB seguido de algoritmo de Bokhari considerando sólo las comunicaciones (fig. 5.3).
- BOK_2: Algoritmo EMB seguido de algoritmo de Bokhari considerando función de coste *minimax* y comunicaciones (fig 5.4).
- ITER_1: Algoritmo BOK_1 seguido de refinamiento iterativo (movimientos e intercambios).
- ITER_2: Algoritmo BOK_2 seguido de refinamiento iterativo (movimientos e intercambios).
- MOVIM: Algoritmo de agrupación (CA) + refinamiento iterativo (sólo movimientos: CRM) + Algoritmo de incrustación + algoritmo de Bokhari considerando sólo las comunicaciones (fig 5.3) + refinamiento iterativo (sólo movimientos).

- CA_ME: Algoritmo de agrupación (CA) + algoritmo de incrustación+ algoritmo de Bokhari considerando sólo las comunicaciones (fig 5.3) + refinamiento iterativo (movimientos e intercambios).

Las columnas MD y CR, muestran los valores promedios obtenidos para grafos regulares de granularidad media y gruesa, respectivamente; la columna PRO muestra el promedio total para grafos regulares. La columna Irreg muestra el promedio obtenido para grafos irregulares.

Tabla 5.2. Incremento promedio de la función de coste al considerar arquitectura

Arquitec.	HIPERCUBOS				MALLAS				ANILLOS			
Grafo	Regulares			Irreg.	Regulares			Irreg.	Regulares			Irreg.
Granul.	MD.	CR.	PRO.		MD.	CR.	PRO.		MD.	CR.	PRO.	
EMB	1.26	1.06	1.16	1.49	1.29	1.08	1.19	1.57	2.2	1.29	1.75	3.00
BOK_1	1.2	1.06	1.13	1.48	1.19	1.06	1.13	1.45	1.81	1.27	1.54	2.78
BOK_2	1.16	1.05	1.10	1.41	1.18	1.05	1.12	1.45	1.66	1.22	1.44	2.54
ITER_1	1.13	1.04	1.09	1.37	1.14	1.05	1.09	1.42	1.54	1.16	1.35	2.46
ITER_2	1.13	1.04	1.09	1.37	1.19	1.05	1.12	1.41	1.64	1.2	1.42	2.39
MOVIM	1.13	1.05	1.09	1.4	1.14	1.06	1.1	1.44	1.58	1.18	1.38	2.51
CA_ME	1.19	1.23	1.21	1.2	1.22	1.28	1.25	1.42	1.47	1.28	1.38	2.39

Tabla 5.3. Incremento promedio de las comunicaciones al considerar la arquitectura

Arquitec.	HIPERCUBOS				MALLAS				ANILLOS			
Grafo	Regulares			Irreg.	Regulares			Irreg.	Regulares			Irreg.
Granul.	MD.	CR.	PRO.		MD.	CR.	PRO.		MD.	CR.	PRO.	
EMB	1.5	1.64	1.57	1.64	1.5	1.68	1.59	1.69	3.33	3.58	3.45	3.39
BOK_1	1.38	1.52	1.45	1.58	1.36	1.5	1.43	1.53	2.26	3.04	2.65	3.03
BOK_2	1.42	1.55	1.49	1.64	1.42	1.63	1.53	1.68	2.5	2.23	2.86	3.37
ITER_1	1.39	1.52	1.45	1.58	1.4	1.57	1.48	1.6	2.26	3.01	2.64	3.02
ITER_2	1.44	1.55	1.49	1.64	1.42	1.61	1.52	1.68	2.53	3.33	2.93	3.67
MOVIM	1.38	1.57	1.47	1.58	1.39	1.63	1.51	1.59	2.27	3.15	2.71	2.98
CA_ME	1.29	1.28	1.28	1.68	1.27	1.28	1.28	1.66	1.79	2.1	1.95	2.95

Análisis de los resultados

La información reflejada en las tablas anteriores contempla básicamente dos grandes grupos de grafos:

- por una parte, los grafos contraídos que provenían de grafos regulares,
- por otra parte, los grafos contraídos que provenían de grafos irregulares.

Estrictamente hablando, todos los grafos una vez contraídos son, mayoritariamente, grafos irregulares donde difícilmente se distingue alguna característica que los relacione con su estructura original. Tan sólo se puede observar que los grafos originalmente irregulares una vez contraídos exhiben un grado de conectividad substancialmente superior al exhibido por cualquiera de los grafos contraídos que provenía de un grafo regular (sólo en el caso de las mallas, sus grafos contraídos tenían un grado de conectividad más cercano al de los grafos irregulares). Ese mayor grado de conectividad es el factor más importante para que el incremento de

la función de coste sea mayor en el grupo de los irregulares que en el grupo de los regulares. Aunque la tabla no lo refleja, el estudio detallado de los resultados demostró que los incrementos experimentados en el grupo de los regulares eran mayores en el grupo de las mallas, seguidos a una ligera distancia por los árboles y los anillos, hecho explicable también porque eran los primeros los grafos contraídos de mayor grado de conectividad dentro del grupo de los regulares.

La característica más importante dentro de los grafos regulares para la que sí se observan diferencias substanciales es la granularidad del grafo. Aquellos grafos de granularidad media una vez contraídos exhiben una granularidad de tipo grueso que se ve compensada por el incremento del grado de conectividad, lo que origina que la función de coste sufra un incremento sustancia debido al aumento que experimenta el factor de las comunicaciones dentro del coste de cada nodo. Por su parte, los grafos que originalmente eran de grano grueso ven incrementado en un orden de magnitud la relación cómputo/comunicaciones por lo que, a pesar del incremento del grado de conectividad, los resultados finales en la función de coste son poco perceptibles ya que en el valor de dicha función sigue predominando el término correspondiente al cómputo de cada nodo.

Los grafos irregulares, a pesar de ser grafos de granularidad gruesa originalmente, una vez contraídos ven incrementada su granularidad ligeramente (pero no en un orden de magnitud como en el caso de los regulares). Esto, unido al incremento del grado de conectividad, explica el sustancia incremento en su función de coste, pues es el factor de comunicaciones el que marca ese incremento.

Este fenómeno se puede constatar también si vemos que el incremento global de comunicaciones es mayor en los grafos irregulares y los regulares de granularidad gruesa, siendo los que menor incremento experimentan los grafos regulares de granularidad media. Sin embargo, la repercusión de ese incremento en la función de coste no sigue la misma distribución, siendo en este caso los grafos regulares de granularidad gruesa los que menor incremento relativo experimentan. Es decir, las comunicaciones se incrementan por un igual en todos los tipos de grafo y de forma proporcional al grado de conectividad; sin embargo, la función de coste experimenta mayores aumentos si en el coste de cada nodo del grafo contraído la proporción debida al cómputo y a la comunicación era parecida.

Como sucedía en el caso de la comparación de estrategias de *mapping*, el número de procesadores también repercute en el resultado final de la función de coste aunque en la tabla no se ha reflejado porque todas las estrategias muestran el mismo comportamiento, predecible, por otra parte. A medida que el número de procesadores es mayor, los incrementos de la función de coste y de las comunicaciones globales son mayores, fenómeno explicable porque al aumentar el número de procesadores aumentan necesariamente las distancias entre ellos y, por lo tanto, aumenta también el peso final debido a las comunicaciones.

Lógicamente, si los costes se incrementan al aumentar las distancias, los resultados obtenidos son substancialmente peores cuando la arquitectura utilizada era de

tipo anillo pues es la que presenta mayores distancias y menor número de *links*. Para el caso de los hipercubos y las mallas, las diferencias son mínimas.

Por lo que respecta al análisis de las distintas estrategias podemos concluir que, como era de esperar, la estrategia de incrustación (EMB), por ser la más simple, es la que obtiene peores resultados y a partir de ella, las demás estrategias que parten de ella (BOK_1, BOK_2, ITER_1 e ITER_2) van mejorando esos resultados a costa de incrementar el coste computacional.

Al comparar las dos versiones del algoritmo de Bokhari se constata que cada una de ellas obtiene mejores resultados o bien en el coste global de comunicaciones (BOK_1) o en el valor de la función de coste (BOK_2), de acuerdo con el criterio de optimización que sigue cada una de las dos estrategias. Sin embargo, los resultados de las correspondientes fases de refinamiento (ITER_1 e ITER_2) demuestran que, aunque sólo sea de forma muy ligera, el primer método para los grafos regulares es capaz de alcanzar situaciones donde la función de coste sea menor a pesar de partir de una situación peor. En el casos de grafos irregulares, ITER_2 muestra una ligera ventaja. De todas formas, también es importante resaltar que estas dos estrategias consiguen mejorar los valores de la función *minimax* a costa de aumentar el coste global de comunicaciones ya que alteran las agrupaciones originales de los nodos. En el caso de las dos variaciones del método de Bokhari la mejora de la función *minimax* es consecuencia únicamente de una mejor colocación de las agrupaciones que repercute en un menor coste global de comunicaciones.

La estrategia que sólo realiza la fase de refinamiento cuando se considera la arquitectura (CA_ME) tiene un comportamiento similar a los otros métodos con fase de refinamiento en el caso de los grafos irregulares, pero no tanto en el caso de los grafos regulares, en especial para los grafos de granularidad gruesa en mallas e hipercubos. La razón de esta anomalía que presenta esta estrategia hay que buscarla en las consecuencias de la operación de refinamiento aplicada después de la fase de agrupación (CA). En el caso de los grafos regulares el algoritmo CA puede crear un conjunto de agrupaciones donde la agrupación de mayor coste está conectada directamente a agrupaciones de coste un poco menor, que a su vez están conectadas a otras agrupaciones de coste menor y así sucesivamente. De forma gráfica, la situación sería semejante a la estructuración del grafo contraído en forma de círculos concéntricos donde los más internos tienen coste mayor y los más externos tienen coste menor. Ante esa situación el algoritmo de refinamiento consigue reducir el coste de los nodos internos llevándose tareas individuales desde esos nodos a cualquier nodo de la periferia aunque dicha tarea no tenga conexión directa con ninguna tarea de la agrupación externa. Este tipo de acciones permite balancear el coste global de todas las agrupaciones aunque afloren una cantidad de nuevas comunicaciones más o menos importante. Cuando esta misma optimización se quiere realizar cuando las agrupaciones están físicamente asignadas a un procesador de la arquitectura, la estrategia de reasignación es incapaz de obtener ninguna mejora porque el hecho de llevar una tarea hacia una de esas agrupaciones periféricas se ve penalizado fuertemente por la distancia entre ambas agrupaciones lo que inhibe la aceptación de dicha reasignación. En consecuencia, la fase de reasignación apenas no consigue introducir prácticamente ningún cambio frente a las agrupaciones generadas por el método CA.

Por último, por lo que se refiere a la estrategia que sólo efectúa movimientos en todas las fases de refinamiento (MOVIM), sus resultados están muy próximos a los obtenidos por ITER_1 que es la mejor de las estrategias más complejas. Este hecho nos permitiría afirmar que éste es el mejor método desde el punto de vista del compromiso alcanzado entre calidad de la solución y complejidad computacional. Para concluir el análisis de estos resultados, la figura 5.6 muestra gráficamente los resultados globales de la estrategia MOVIM.

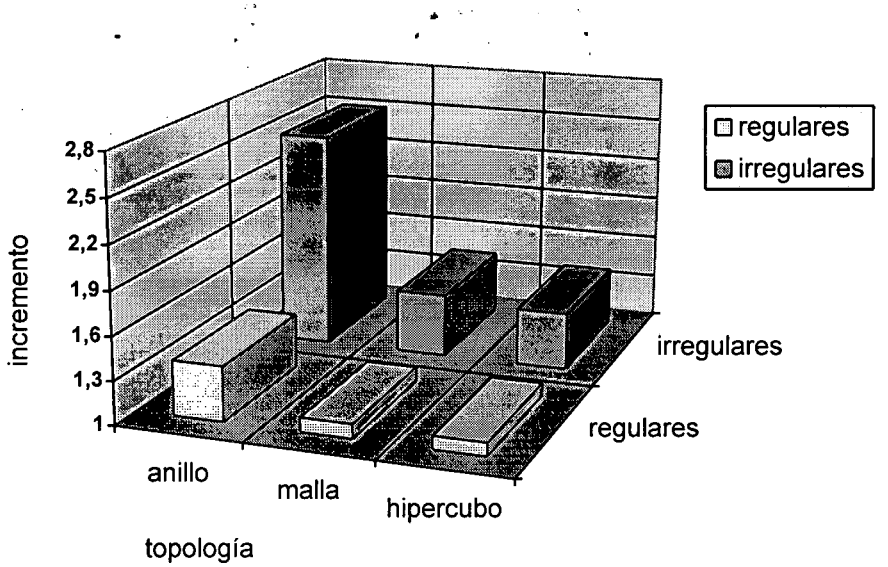


Figura 5.6. Incremento de la función de coste para la estrategia MOVIM

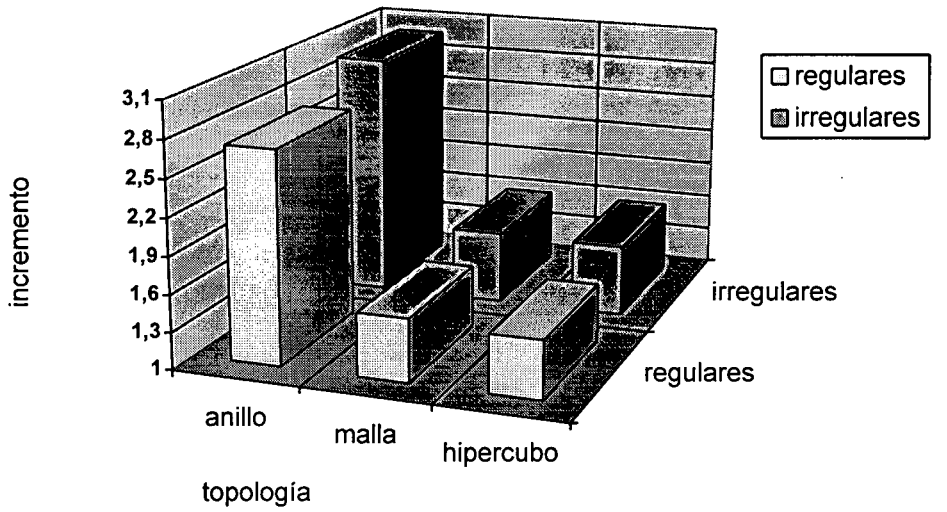


Figura 5.7 Incremento del volumen de comunicaciones para la estrategia MOVIM

Validación de los estimadores del coste de la asignación física

En el apartado 5.3 se presentaron dos estimadores que se habían propuesto con el objeto de evaluar teóricamente el incremento que experimentarían tanto la función de coste *minimax* como el volumen de comunicaciones global dado un cierto grafo contraído y una arquitectura.

Las tablas 5.4 y 5.5 muestran respectivamente el resultado obtenido al contrastar la estimación teórica del incremento de comunicaciones y la estimación teórica del incremento de la función de coste. A diferencia de las tablas anteriores, donde los grafos regulares se habían diferenciado entre aquellos de granularidad gruesa y aquellos de granularidad media, en las tablas 5.4 y 5.5 no se ha hecho tal distinción porque los dos estimadores calculan, dado un grafo y una arquitectura, cuál es el incremento en términos absolutos de la función de coste y del volumen de comunicaciones global. Al proporcionar ambos estimadores un valor absoluto, las características del grafo por lo que se refiere a su granularidad ó grado de conectividad ya se verán reflejadas en ese valor. Tan sólo se ha mantenido la diferenciación entre grafos regulares e irregulares porque en los primeros es donde se suelen presentar mayoritariamente los casos de subestimación del número de arcos que van a estar a distancia mayor que uno que ya se mencionó en la apartado 5.3 al comentar el ejemplo de la figura 5.2.

En la tabla 5.4, se muestra en la columna PRO el promedio de los cocientes $Vol\_comm_{arq}/(Vol\_comm_{CRME} + comm\_adic)$, donde Vol_comm_{arq} es el volumen de comunicación global obtenido después de la fase de asignación física, Vol_comm_{CRME} es el volumen de comunicación global obtenido después de la fase de contracción y comm_adic es el valor teórico estimado de incremento de ese volumen de comunicación. Si el cociente es 1 el coste obtenido y la predicción estimada son iguales. Si el coste final obtenido es inferior a la predicción, el cociente será < 1, si el resultado final supera a la predicción el cociente será > 1. La columna σ_N refleja la desviación estándar obtenida en el conjunto de valores.

De forma análoga, en la tabla 5.5 se refleja el promedio entre los cocientes de Map_{arq} / cost_map_arq, donde Map_{arq} es el coste de la asignación obtenido por cada una de las estrategias y cost_map_arq es el coste de la asignación estimado mediante la fórmula (ec. 5.2). Como en el caso anterior, un promedio de 1 indica coincidencia entre el valor obtenido y estimado, y los casos deseables son aquellos donde el promedio sea < 1, lo que garantizaría la efectividad del estimador como cota superior.

Notemos finalmente que tanto para la estrategia MOVIM como CA_ME no se usó el grafo contraído generado por el algoritmo CRME, sino que se utilizaron, respectivamente, los grafos contraídos obtenidos por los algoritmos CRM y CA, ya que los estimadores son predictores del empeoramiento que experimentará el volumen de comunicación y la función de coste a partir del grafo contraído.

Tabla 5.4. Promedio entre valor obtenido y valor estimado del volumen de comunicación												
Arquitec.	HIPERCUBOS				MALLAS				ANILLOS			
Grafo	Regulares		Irregulares		Regulares		Irregulares		Regulares		Irregulares	
	PRO	σ_N	PRO	σ_N	PRO	σ_N	PRO	σ_N	PRO	σ_N	PRO	σ_N
EMB	1.18	0.21	0.96	0.19	1.14	0.19	0.97	0.2	1.04	0.29	0.86	0.21
BOK_1	1.09	0.15	0.92	0.17	1.04	0.15	0.88	0.19	0.83	0.17	0.79	0.23
BOK_2	1.11	0.16	0.96	0.19	1.09	0.14	0.96	0.2	0.89	0.17	0.87	0.23
ITER_1	1.09	0.17	0.92	0.18	1.06	0.14	0.92	0.19	0.82	0.19	0.79	0.23
ITER_2	1.12	0.17	0.95	0.19	1.09	0.14	0.96	0.2	0.9	0.18	0.87	0.24
MOVIM	1.1	0.17	0.92	0.18	1.08	0.15	0.92	0.2	0.84	0.18	0.78	0.23
CA_ME	1.16	0.18	1.05	0.19	1.15	0.18	1.0	0.13	1.04	0.8	0.8	0.24

Tabla 5.5. Promedio entre valor obtenido y valor estimado de la función de coste												
Arquitec.	HIPERCUBOS				MALLAS				ANILLOS			
Grafo	Regulares		Irregulares		Regulares		Irregulares		Regulares		Irregulares	
	PRO	σ_N	PRO	σ_N	PRO	σ_N	PRO	σ_N	PRO	σ_N	PRO	σ_N
EMB	1.03	0.12	0.96	0.21	1.02	0.11	0.97	0.2	0.97	0.15	1.0	0.38
BOK_1	1.0	0.08	0.95	0.21	0.97	0.09	0.9	0.2	0.89	0.11	0.97	0.41
BOK_2	0.98	0.08	0.92	0.2	0.96	0.09	0.91	0.2	0.85	0.13	0.9	0.39
ITER_1	0.98	0.09	0.89	0.21	0.95	0.08	0.89	0.23	0.81	0.14	0.87	0.42
ITER_2	0.97	0.12	0.89	0.19	0.96	0.09	0.88	0.21	0.83	0.13	0.85	0.38
MOVIM	0.97	0.09	0.91	0.21	0.95	0.08	0.91	0.23	0.82	0.14	0.89	0.42
CA_ME	0.97	0.13	0.71	0.2	0.96	0.09	0.73	0.19	0.9	0.16	0.69	0.31

Del análisis de la tabla 5.4 se puede concluir que el estimador propuesto para el incremento del volumen global de comunicaciones tiene un comportamiento bastante bueno aunque suele subestimar dicho incremento en los grafos contraídos a partir de grafos regulares, donde hay menor grado de conectividad y, por lo tanto, son grafos más proclives a que se les cuenten menos arcos a distancia superior a uno. Por el contrario, en el caso de los grafos irregulares, dado su mayor grado de conectividad, el cómputo de arcos es más preciso y el incremento calculado suele estar por encima del incremento que posteriormente se obtiene. Su único inconveniente está en la mayor desviación estándar observada en este caso.

El estimador actúa también como una cota más precisa para todo tipo de grafos en el caso de la topología de anillo pues los valores de las distancias son substancialmente superiores a las topologías de malla e hipercubo por lo que resulta más fácil obtener valores estimados más pesimistas de lo que en la realidad van a conseguirse.

Por lo que se refiere a las heurísticas, todas suelen estar cerca del promedio 1, lo que depende, lógicamente, de su efectividad en asignar físicamente los nodos del grafo contraído, y que sigue una proporción acorde a los resultados reflejados en la tabla 5.3. La única heurística que parece obtener unos valores por encima de lo que por su complejidad sería previsible es la CA_ME. La razón para este comportamiento hay que buscarla en el hecho de que la mejora de la función de coste en la fase de refinamiento suele conseguirse mediante movimientos de tareas que vienen acompañados de la aparición de nuevos arcos de los que mostraba el grafo contraído por CA.

Por lo que respecta al estimador del incremento de la función de coste (tabla 5.5), los resultados confirman su eficacia en la estimación de una cota para dicho incremento. En general, para todos los tipos de grafos y para todas las arquitecturas se obtienen valores < 1. El único inconveniente observado estaría en la elevada desviación observada en los grafos irregulares y, en especial, para la topología de anillo. Desviación que suele deberse a que en el valor de incremento de la función de coste en ciertos casos es sobreestimado (a lo que lógicamente contribuye las elevadas distancias de la topología de anillo).

De todas formas, creemos que los resultados nos permiten asegurar que ambos estimadores propuestos son buenos (en especial, el de la función de coste), aunque en el caso del estimador del volumen de comunicación haya que tener en cuenta su tendencia a la subestimación para grafos de baja conectividad y arquitecturas de distancias no muy elevadas.

Particularizando el análisis para la estrategia MOVIM, que anteriormente habíamos escogido como la mejor heurística, podemos comprobar que, en promedio, garantiza asignaciones físicas donde la función de coste estará por debajo del valor teórico estimado y cuyo volumen de comunicación estará como máximo un 10% por encima del valor teórico estimado.

5. 5 Función de coste y tiempo de ejecución

Hasta este momento, se ha hecho mención a las funciones de coste como elemento universalmente aceptado para evaluar la bondad de una cierta asignación cuando se maneja el modelo de TIGs. La suposición básica que realizan todos los autores es que minimizar la función de coste minimiza el tiempo de ejecución. Esta situación es distinta cuando se estudia el problema del *scheduling* usando el modelo de DAGs. En este caso, gracias a la existencia de relaciones de precedencia que deben respetarse en la ejecución de una determinada aplicación y conociendo los correspondientes volúmenes de cómputo y de comunicación se puede obtener una estimación del tiempo de ejecución de la aplicación calculando el momento en el que finaliza su ejecución el último nodo del programa. El modelo de DAGs, pues, permite dar una aproximación más directa a lo que será el tiempo de ejecución de un programa paralelo (en el capítulo 4 se vio un ejemplo de tal característica en el apartado donde se mostró la transformación de DAGs a TIGs).

En esta sección pretendemos realizar un estudio que permita establecer qué relación existe entre los valores obtenidos por las funciones de coste en los modelos TIGs y el tiempo de ejecución de un programa. El estudio también intentará determinar qué factores son los que pueden provocar las discrepancias entre tiempo de ejecución y valor de la función de coste, y qué posibles medidas podrían servir para corregirlas. Para realizar este estudio aprovecharemos los ejemplos de grafos de tipo DAG presentados en el capítulo 4 porque trabajando con ellos se podrán obtener valores equivalentes al tiempo de ejecución de las correspondientes aplicaciones. Lógicamente, esos ejemplos