Curso 1994/95
CIENCIAS Y TECNOLOGÍAS

FÉLIX CÉSAR GARCÍA LÓPEZ

Programación en paralelo y técnicas algorítmicas

Director
CASIANO RODRÍGUEZ LEÓN

SERVICIO DE PUBLICACIONES
UNIVERSIDAD DE LA LAGUNA

SOPORTES AUDIOVISUALES E INFORMÁTICOS
Serie Tesis Doctorales

A mis padres por su inagotable paciencia y el apoyo que siempre me han brindado, a mi hermana por serlo de verdad, y a Sergio por estar ahí siempre que lo he necesitado.

Agradecimientos.

Mi agradecimiento más sincero al Dr. D. Casiano Rodríguez León, mi profesor, director y amigo, por su cariño, el compañerismo que siempre muestra como director del grupo de investigación y la dedicación a este trabajo, el cual no hubiera sido posible sin su inestimable ayuda.

También quiero expresar la mayor de las gratitudes a todos los miembros del grupo, Coro, Dani, Paco, Kiko, Jose Luis y Carmelo, por las innumerables ayudas que me han prestado y me prestan. Mi reconocimiento también a todos ellos y a Patricio, Javi, Jose Manuel y Julio, por hacer que nuestro lugar de trabajo tenga un ambiente tan familiar.

Mi agradecimiento asimismo a mi familia y amigos, por su cariño, calidad humana, creer en mí y apoyarme a lo largo de mi vida.

Por último, quiero hacer extensiva mi gratitud a todos los compañeros del departamento que siempre me han brindado su apoyo y me han hecho sentir como una verdadera persona.

PROLOGO

El procesamiento paralelo se ha convertido en un área de vital importancia en los últimos años para la obtención de soluciones y reducción del tiempo de cálculo en problemas de alto coste computacional. Durante este período hemos asistido a la aparición de máquinas masivamente paralelas como la Connection Machine CM5, con un máximo de 256000 procesadores (Thinking Machines, USA), la arquitectura supernodo de Parsys (Inglaterra), la Surface Machine (Meiko, Inglaterra), y la arquitectura supernodo de Parsytec (Alemania), con hasta 1024 procesadores. En este tipo de supercomputadores fuertemente acoplados, la distancia entre procesadores es bastante pequeña. Al mismo tiempo, se han empezado a introducir en el mercado estaciones de trabajo multiprocesador (normalmente una MIMD con un pequeño número de procesadores, entre 4 y 16). Los avances realizados en las redes de conexión tanto locales como remotas, que han llevado a mayores ancho de bandas y velocidades cercanas al Gigabyte por segundo, y el desarrollo de herramientas, como la PVM, creadas para facilitar a los usuarios las comunicaciones entre diferentes máquinas, permiten la utilización de grupos de estaciones como un único computador paralelo, en lo que se ha dado por conocer como sistemas débilmente acoplados.

En cualquier caso, tanto para las compañías que apuestan por el primer tipo de arquitectura como para las del segundo, un problema que aún está por resolver es la creación de entornos de alto nivel que permita una relación amigable para los usuarios con este nuevo tipo de arquitecturas. Estos productos deberían resolver los problemas intrínsecos de la programación paralela (atascos, inanición) de forma automática, así como conseguir portabilidad entre los diferentes sistemas. Sin entrar a decantarse por uno o otro sistema como modelo estándar para la computación paralela, si parece existir un acuerdo generalizado en que los supercomputadores del futuro sólo serán aceptados de forma generalizada por la comunidad científica y la industria, cuando les permitan obtener mayores prestaciones con unos costos de adaptación inexistentes. En este sentido, han comenzado a aparecer en el mercado como ejemplo compiladores de alto nivel como son High Performance Fortran, Parallel C, etc y los grupos de especialistas tanto de América como Europa proponen centrar la investigación y desarrollo, en este marco, de herramientas que se basen en lenguajes secuenciales aceptados y reconocidos por todo el mundo como son el Fortran, C, etc. Estos productos se caracterizan por utilizar una serie de funciones y protocolos, normalmente de librería, que se añaden a las instrucciones y estructuras de datos ya conocidas. Es en este marco donde se puede encuadrar este trabajo.

Desde hace varios años, un gran esfuerzo tanto económico como humano se ha dedicado a la creación de utensilios y herramientas que permitan transformar un código secuencial en su equivalente paralelo, con la introducción automática de primitivas de sincronización. Sin embargo, la mayoría de los resultados obtenidos se han centrado en la

ejecución concurrente de instrucciones que actúan sobre datos independientes entre sí, dando lugar a códigos eficientes en el campo del cálculo numérico y sus relacionados. Las técnicas algorítmicas secuenciales básicas ya consolidadas (divide y vencerás, ramificación y acotación, programación dinámica, etc), han servido a lo largo de los años para intentar diseñar algoritmos eficientes en áreas como la investigación operativa y la inteligencia artificial, y se caracterizan por la ejecución de un mismo algoritmo sobre un número, en principio, indeterminado de casos de un problema. En este terreno, los frutos obtenidos, aunque abundantes y que continúan surgiendo, no han servido para desarrollar esqueletos paralelos generales y eficientes, que permitan transformar los algoritmos secuenciales ya probados sin más que adaptar el código de la aplicación a resolver a la herramienta ya creada. Nosotros creemos que se debe dar un nuevo empuje a esta línea de investigación, y por ello es por lo que se examinan y prueban las diferentes descubrimientos que se han ido produciendo en esta materia. Aplicando estas ideas, se proponen y contrastan otras sobre uno de los tipos de arquitectura paralela existente.

Este trabajo se ha dividido en cuatro capítulos. En el primero de ellos se introducen una serie de conceptos necesarios para el desarrollo posterior y en los tres restantes se presentan y estudian las tres técnicas algorítmicas más extendidas.

El capítulo dedicado a introducción comienza con una clasificación de los computadores actuales, para pasar rápidamente a la introducción de la arquitectura de computador Transputer. Después de una breve descripción de esta familia de procesadores y de los lenguajes de programación más utilizados, se analizan los modelos básicos de redes de conexión. Por último, se trata el tema de la complejidad algorítmica paralela, haciendo referencia a características generales que deben cumplir los problema a tratar de forma concurrente.

Para analizar cada una de las técnicas estudiadas se ha seguido el siguiente esquema: El primer apartado presenta la técnica secuencial de partida, incidiendo en los componentes imprescindibles de cada una de ellas. La siguiente sección muestra las posibles paralelizaciones a las que puede dar lugar el código secuencial, de forma teórica. Por último en un apartado de aplicaciones, se analizan e implementan sobre un problema particular, los códigos existentes basados en ideas propuestas con anterioridad. Se presentan y examinan en cada capítulo, resultados para los problemas de la ordenación, el viajante de comercio y la mochila, respectivamente.

Cada uno de los cuatro capítulos incluye al final, la bibliografía más relevante. En un apartado posterior se incluyen las conclusiones y trabajos futuros a realizar. Para hacer más comprensible la lectura de esta memoria, se suministra en un disco flexible los diferentes códigos experimentados. Al mismo tiempo se incluye un apéndice, donde se da una somera

descripción de cada uno de ellos.

En el capítulo dedicado a la técnica divide y vencerás, se introduce como primer código paralelo un esquema maestro/esclavos donde un procesador especial (maestro) tiene el control sobre los problemas y resultados existentes y comunica a cada uno de los restantes procesadores (esclavos) que trabajo deben realizar. En un segundo apartado, aprovechando la estructura árbol que siguen los algoritmos divide y vencerás, se analizan tres códigos aplicables a esta estructura jerárquica (cada procesador del árbol recibe un problema de su ancestro, divide el problema y combina los resultados recibidos por sus hijos). En este caso, acompañando a los resultados experimentales que siempre se presentan, se realiza un análisis teórico para estimar los resultados obtenibles. Al final del capítulo se propone una nueva metodología que intenta sacar partido de las características ventajosas de los esquemas centralizado y jerárquico anteriores. Se trata de posibilitar el análisis de un número mayor de problemas en cada procesador, manteniendo la aceleración de los esquemas jerárquicos en la generación de los mismos. Como problema de aplicación se ha elegido el algoritmo de la ordenación rápida (*quicksort*), representante muy popular de la estrategia divide y vencerás.

La ramificación y acotación admite a primera vista dos esquemas de paralelización. El primero más sencillo es un esquema centralizado, donde en su forma básica un único procesador controla la estructura de problemas a analizar (esquema maestro/esclavos). El esquema distribuido reparte el control entre los procesadores de la red. Tomando como punto de partida ambos métodos, se implementan y analizan mediante resultados experimentales tres códigos como representantes del amplio abanico de paralelizaciones de la técnica que se han publicado. El primero de ellos se basa en el esquema centralizado, al que se le introducen una serie de mejoras para intentar minimizar el número de comunicaciones y el tamaño de estas. Los otros dos códigos siguen la estrategia distribuida y se diferencian en la forma en que se intenta realizar el balance de la carga entre los procesadores de la red. Un algoritmo secuencial que utiliza una función de búsqueda primero el de mejor cota para el problema del viajante de comercio, sirve como base para los experimentos.

En la programación dinámica, el estudio se ha centrado en la posible paralelización de alguno de los bucles que siempre forma parte de un algoritmo de este tipo. Tomando como base el problema de la mochila, se presentan varios códigos que paralelizan o el bucle de las capacidades o el de los objetos. Estos códigos se contrastan en dos tipos de mochilas, la binaria y la entera. Además en este último caso, se propone un nuevo planteamiento del problema al trasladarlo a un problema del cálculo del camino más largo en un grafo. Para este problema se introduce un código secuencial y su equivalente paralelo, a partir de un análisis matemático que permite reducir el grafo a explorar.

Dos consideraciones se deben tener en cuenta, antes de comenzar con el desarrollo del trabajo: Por un lado se ha intentado utilizar en la mayoría de los códigos presentados un formato lo más cercano a Pascal y C, por ser lenguajes estructurados bastante extendidos (sólo cuando se explicita un código específico no se ha tenido en cuenta esta regla). En los casos donde era necesario la utilización de nuevas instrucciones, no disponibles en un lenguaje de carácter secuencial, se ha optado por utilizar las más normales en la literatura como *parallel* para la ejecución concurrente de una serie de procesos y ? y ! para las comunicaciones entre procesos. Por otro lado, en la expresión de los términos informáticos anglosajones se ha hecho un esfuerzo por utilizar su equivalente castellano, cuando éste no se ha creado de forma artificiosa y se ha mantenido el inglés para aquellos casos donde la comunidad informática lo ha adoptado como habitual.

INDICE

INDICE DE FIGURAS

INDICE DE TABLAS

INDICE DE CODIGOS

CAPITULO I:

Conceptos Básicos

1.1. INTRODUCCION.

La demanda de lo que podemos llamar computadores de altas prestaciones o supercomputadores está creciendo continuamente en áreas de conocimiento como meteorología, medicina, física de altas energías, inteligencia artificial, ingeniería genética entre otras muchas aplicaciones científicas y tecnológicas. Sin el concurso de los supercomputadores, muchos de los logros del ser humano en las últimas décadas hubieran sido imposibles de alcanzar dentro de un período de tiempo razonable.

El alcanzar altas prestaciones en los computadores no depende sólo de utilizar dispositivos hardware más rápidos y fiables sino también de los desarrollos en arquitecturas de computadores y de las técnicas de procesamiento de la información. El procesamiento en paralelo es probablemente la técnica que permite un mayor aumento de prestaciones en los computadores con un coste razonable.

Se puede definir el procesamiento en paralelo [hwa85] como una forma eficiente de procesar información que hace hincapié en la utilización de concurrencia de eventos en el proceso de cómputo. Concurrencia implica paralelismo, simultaneidad y proceso segmentado: eventos paralelos ocurren en diferentes dispositivos durante el mismo intervalo de tiempo; eventos simultáneos ocurren en el mismo instante de tiempo y eventos segmentados pueden ocurrir en lapsos de tiempo superpuestos. El procesamiento en paralelo implica la ejecución concurrente de varios procesos en el computador, lo que está en contraposición con el procesamiento secuencial.

Según Hwang y Briggs [hwa85], los computadores paralelos pueden englobarse dentro de tres grandes clases estructurales:

- Procesadores segmentados (*pipeline processors*).
- Procesadores vectoriales.
- Sistemas multiprocesador.

Un sistema de proceso segmentado realiza computaciones de forma superpuesta en el tiempo en un intento de explotar el paralelismo temporal. Un computador vectorial utiliza múltiples unidades aritmético-lógicas para aprovechar el paralelismo espacial. Un sistema multiprocesador alcanza paralelismo asíncrono gracias a un conjunto de procesadores interactivos con recursos compartidos (memoria, bases de datos, etc.). Estas tres aproximaciones a la computación en paralelo no son mutuamente excluyentes; de hecho, muchos computadores existentes hoy en día poseen varias de estas características. La diferencia fundamental entre un computador vectorial y un sistema multiprocesador radica en que mientras los elementos de proceso del computador vectorial operan de forma síncrona (gobernados por un reloj global), los procesadores de un multiprocesador pueden funcionar de forma asíncrona.

En 1966 Michael J. Flynn [fly66] introdujo la que probablemente es la clasificación de computadores más ampliamente utilizada. La clasificación se basa en la multiplicidad de

flujos de instrucciones y datos. En general, los computadores digitales se pueden clasificar en cuatro categorías atendiendo a esta multiplicidad. Lo esencial de un proceso de computación es la ejecución de una secuencia de instrucciones sobre un conjunto de datos. El término *flujo* se utiliza aquí para denotar una secuencia de elementos (instrucciones o datos) ejecutados u operados por un único procesador. Las instrucciones o datos se definen con respecto a una máquina prefijada. Un flujo de instrucciones es una secuencia de instrucciones ejecutada por la máquina; un flujo de datos es una secuencia de datos incluyendo los de entrada, y resultados parciales o intermedios invocados por el flujo de instrucciones.

Las clasificaciones de ordenadores se caracterizan por la multiplicidad del hardware provisto para adaptarse a los flujos de instrucciones y datos. Los cuatro tipos de máquinas caracterizados por Flynn son:

- SISD (*Single Instruction, Single Data*) Flujo único de instrucciones y datos.
- SIMD (*Single Instruction, Multiple Data*) Flujo único de instrucciones y múltiples flujos de datos.
- MISD (*Multiple Instruction, Single Data*) Flujo múltiple de instrucciones y único flujo de datos.
- MIMD (*Multiple Instruction, Multiple Data*) Flujos múltiples de instrucciones y datos.

Las figuras de la 1.1 a la 1.4 ilustran los cuatro tipos de estructura. La categorización depende de la multiplicidad de eventos simultáneos en los componentes del sistema, y conceptualmente sólo se necesitan tres tipos de componentes en los sistemas de estas figuras: tanto instrucciones como datos son obtenidos de los módulos de memoria MM_i; las instrucciones son decodificadas por las unidades de control (CU_j) que envían el flujo de instrucciones decodificadas (IS) a las unidades de proceso PU_i para su ejecución. El flujo de datos (DS) discurre entre los procesadores y la memoria de forma bidireccional. Se pueden utilizar múltiples módulos de memoria en el subsistema de memoria global, compartida o no. Cada unidad de control independiente genera un flujo de instrucciones. Diferentes flujos de datos se originan en los módulos del sistema de memoria. En estas figuras simplificadas no se muestran los dispositivos de entrada/salida.

La organización SISD representa a la mayoría de los computadores secuenciales disponibles hoy en día. Las instrucciones se ejecutan secuencialmente pero pueden estar solapadas en sus etapas de ejecución (*pipelining*). La mayoría de los monoprocesadores SISD hacen uso de esta técnica de solapamiento de las diferentes etapas de ejecución de sus instrucciones. Todas las unidades funcionales en un computador SISD están bajo la supervisión de una única unidad de control (figura 1.1).

La clase SIMD engloba a los computadores vectoriales. En la figura 1.2 se observa

como hay diferentes unidades de proceso supervisadas por la misma unidad de control. Todos los elementos de proceso reciben de la unidad de control la orden de ejecutar la misma instrucción pero operando sobre diferentes conjuntos de datos procedentes de flujos de datos distintos.

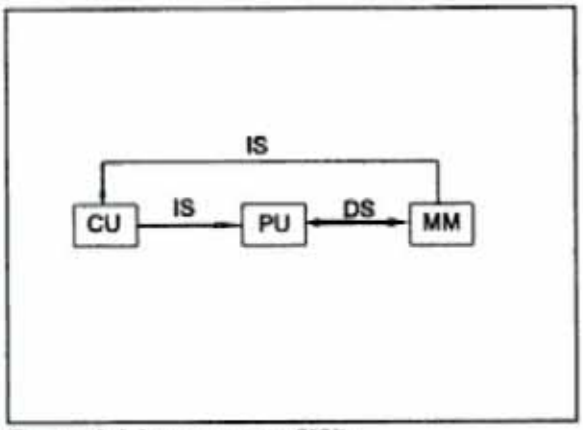

Figura 1.1 La estructura SISD.

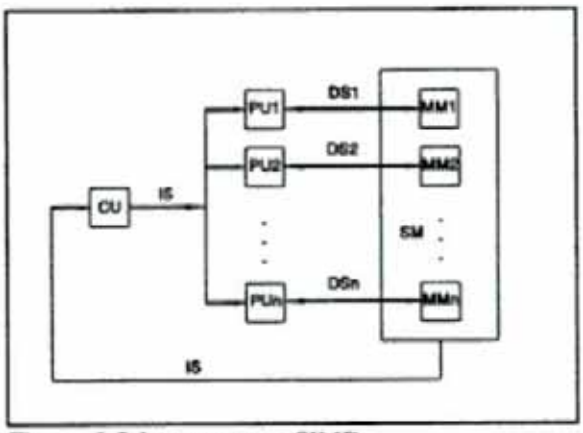

Figura 1.2 La estructura SIMD.

El esquema de la estructura MISD se muestra en la figura 1.3. Hay n unidades de proceso, cada una de ellas recibiendo diferentes instrucciones operando sobre el mismo flujo de datos. Los resultados (salida) de un procesador se convierten en los operandos (entrada) del siguiente. Esta clase de computadores ha recibido mucha menos atención que las otras, y no existe ninguna implementación práctica de este tipo de configuración.

La mayoría de los sistemas multiprocesadores y multicomputadores pueden clasificarse dentro de la categoría MIMD (figura 1.4). Una máquina MIMD se dice fuertemente acoplada si el grado de interacción entre los procesadores es alto. En contraposición se considera débilmente acoplada si esta interacción es escasa. La mayoría de los computadores MIMD comercializados son débilmente acoplados.

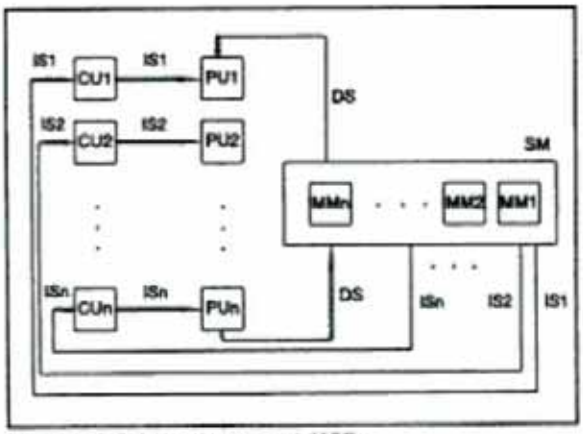

Figura 1.3 La estructura MISD.

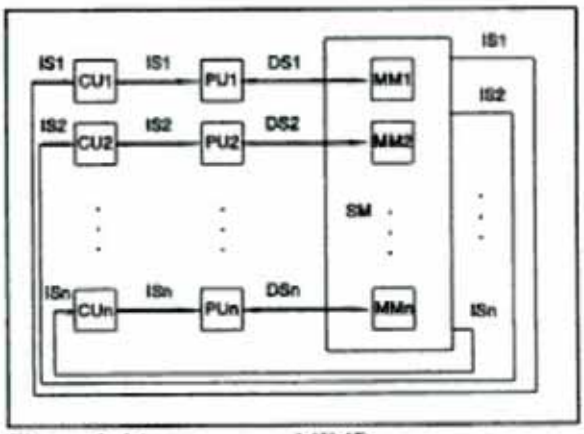

Figura 1.4 La estructura MIMD.

1.2. EL TRANSPUTER. LENGUAJES DE PROGRAMACION.

Un avance en el área del procesamiento paralelo nos lo ofrece una nueva arquitectura de computadores desarrollada por Inmos Ltd en un microprocesador. Este nuevo procesador se llama TRANSPUTER acrónimo de *TRANS(istor)-(com)PUTER* y que además de los componentes de memoria y el procesador, ofrece canales o enlaces (*links*) para comunicarse con otros transputers u otros dispositivos. Estos canales y sus propiedades son una de las características principales de esta nueva arquitectura.

La comunicación por los canales tiene lugar sólo si el emisor y el receptor están preparados, es decir, la comunicación es sincronizada. La sincronización de sucesos fue uno de los mayores inconvenientes en los comienzos del procesamiento paralelo. Cada transputer dispone de cuatro enlaces lo que significa que pueden formar un amplio espectro de configuraciones.

La velocidad básica del procesador transputer está en la región de los 20 MIPS, pero como los transputers no comparten el mismo *bus* de comunicación el poder de comunicación total se incrementa linealmente con el número de transputers añadidos. Un vector de 100 transputers ofrecería una velocidad en la región de los 1000 MIPS. En el caso de los procesadores convencionales, la mejora en la potencia empieza a disminuir con la inclusión de seis o más procesadores.

La arquitectura transputer utiliza los procesos como el bloque fundamental de construcción de software y ofrece una implementación directa de un proceso en hardware. Un proceso es una computación independiente que puede comunicarse con otros procesos que se ejecutan al mismo tiempo.

La comunicación entre procesos que se ejecutan en distintos transputers se consigue utilizando canales explícitamente definidos. Un proceso puede consistir en un número de subprocesos y el transputer es capaz de ejecutar estos subprocesos en tiempo compartido, con instrucciones especiales que soportan las comunicaciones.

El transputer ofrece un número de enlaces que soportan comunicación punto a punto entre transputers, permitiendo de esta forma que los procesos se distribuyan sobre una red de transputers. De esta forma es posible programar sistemas que contienen múltiples transputers interconectados, en los que cada transputer implementa un conjunto de procesos. Debería notarse también que un transputer puede enviar un mensaje sólo a otro transputer con el que esté directamente conectado. Un transputer puede programarse en la mayoría de los lenguajes de alto nivel, sin embargo sus características se han desarrollado de forma consistente con el diseño del lenguaje concurrente Occam (las mayores diferencias que se observan se refieren a la traducción que se lleva a cabo de los protocolos de comunicaciones). Una característica importante, tanto del transputer como de los lenguajes para programarlo, es que un programa configurado para un sistema de varios transputers, puede ejecutarse en un sistema de un único transputer haciendo muy pocas modificaciones,

aunque más lentamente. Un sistema puede diseñarse y programarse completamente en un único lenguaje, desde la configuración del sistema hasta las entradas y salidas a bajo nivel y las interrupciones en tiempo real. Todos ellos permiten que la concurrencia se defina explícitamente en el programa.

1.2.1. Arquitectura transputer.

El transputer es un procesador de alto rendimiento, diseñado básicamente para facilitar las comunicaciones entre procesadores y entre procesos. Tiene características hardware especiales que facilitan el proceso de planificación, está dotado de canales de comunicación y canales seriales de comunicación externa. Estas características están desarrolladas de forma consistente con el diseño del lenguaje Occam, de ahí que su traducción a lenguaje máquina sea la más efectiva.

Occam es un lenguaje de programación con un fuerte soporte para el procesamiento paralelo y sus programas producen un código muy compacto y muy eficiente cuando el procesador es un transputer. Dado que el transputer se ha diseñado para ejecutar código generado por el compilador Occam de forma eficiente, se sigue que se obtendrá un rendimiento razonable con otros lenguajes de alto nivel que posean el mismo tipo de características que éste.

La arquitectura transputer define una familia de componentes VLSI programables con las siguientes características:

- Se trata de un procesador de tipo RISC.
- Memoria interna de alta velocidad.
- Controlador de memoria externa.
- Características hardware *time-slicing*.
- Canales de comunicación seriales de alta velocidad.

La estructura básica de un transputer se corresponde con el diagrama de bloque de la figura 1.5. Aunque hay variantes en la arquitectura, se utiliza el T414 (un miembro de la familia transputer) para describir las características de la misma.

El transputer T414 integra un microprocesador con longitud de palabra de 32 bits, posee 2 Kbytes de memoria interna (RAM) de alta velocidad (50ns) y 4 Gbytes de espacio direccionable con una interfase de memoria de 25 Mbyte/s. Los cuatro canales (Inmos Links) en cada T414 ofrecen comunicaciones punto a punto y, pueden operar a velocidades de 5, 10 ó 20 Mbits/s en ambas direcciones simultáneamente. Una vez que un canal ha iniciado una comunicación, procede autónomamente y de esta forma el procesador puede ejecutar otro proceso (esto se consigue mediante técnicas de acceso directo a memoria (DMA)).

La memoria interna, 2 Kbytes para el T414, ofrece un promedio máximo de velocidad de acceso de 80 Mbytes/s comparado con los 25 Mbytes/s de la memoria externa. Desde el

punto de vista del programador no hay diferencias visibles entre ambos tipos de memoria. Toda la RAM interna es asignada a la parte más baja del espacio direccionable, y si la dirección en un acceso a memoria está en este rango entonces se accede a RAM interna, en otro caso se activa el bus de dirección de datos. La provisión de RAM interna tiene dos ventajas: en primer lugar, puede utilizarse para ubicar secciones críticas de código y datos, obteniéndose un incremento significativo en la velocidad de ejecución del programa. En segundo lugar, permite que los transputers se utilicen sin memoria externa, si el programa y los datos no son demasiado largos.

El T414 puede acceder directamente a un espacio de memoria direccionable linealmente de 4 Gbytes. La interfase de memoria utiliza un *bus* de dirección de datos de 32 bits y suministra datos a un promedio de 25 Mbytes/s.

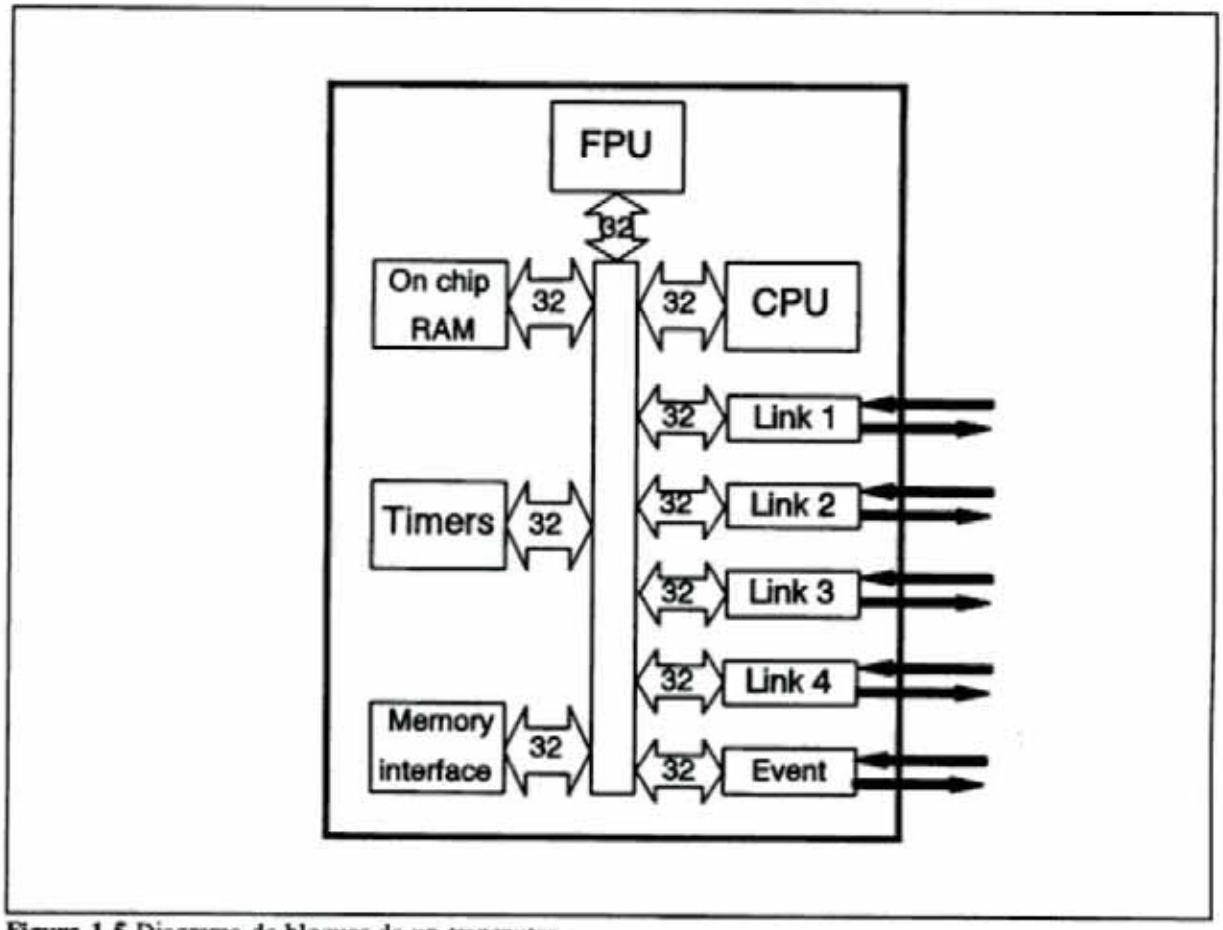

Figura 1.5 Diagrama de bloques de un transputer.

Un controlador de memoria configurable suministra todas las señales de temporización y control. Este controlador posee temporizadores (*timers*) que soportan dos niveles de prioridad. Los procesos de alta prioridad se espera que se ejecuten en intervalos de tiempo cortos. Si uno o más de tales procesos puede proceder, entonces se selecciona uno hasta que tenga que esperar en una comunicación, una entrada en un temporizador o hasta que el

proceso se complete. Si no hay disponibles procesos de alta prioridad, se selecciona uno de baja prioridad.

1.2.2. Canales de comunicación. (Inmos Links).

La arquitectura transputer ofrece comunicaciones mediante el uso de canales llamados Inmos links. En el caso de T414 hay cuatro de tales canales. Cada uno ofrece dos enlaces lógicos (software), uno en cada dirección. La comunicación vía cualquier canal puede ocurrir concurrentemente con comunicaciones en todos los otros canales y con la ejecución de programas. La sincronización de procesos en cada uno de ellos es automática y no requiere programación explícita.

Los canales se conectan uniendo un *LinkOut* de un transputer a un *LinkIn* de otro. Cada enlace consiste en una entrada y una salida serial, ambas se utilizan para llevar datos e información de los mismos.

Los enlaces permiten comunicaciones entre transputers, de aquí la utilización de estos procesadores en redes de interconexión. Es posible una amplia variedad de configuraciones dependiendo en gran medida del número de enlaces utilizados, una limitación obvia viene dada por el número de estos disponibles.

1.2.3. La familia transputer.

El transputer apareció por primera vez en 1985. En aquél momento se anunció como una revolución el lenguaje Occam. El primer procesador de la serie, el T414 se jactaba de una velocidad sin precedentes de 10 MIPS, junto con la capacidad de desarrollar multitarea a través de su propio hardware. Además permitía la comunicación entre procesos que se ejecutarían en distintos transputers. Desde entonces, han aparecido otros miembros de la familia transputer:

- El T212, una versión de 16-bits con 2K de RAM interna y un rango de direccionamiento de 64K, utilizando direccionamiento segmentado y buses de datos.

- El M212, un T212 con 2 de los 4 enlaces sustituidos por circuitería controladora de disco.

- El T800, esencialmente un T414 mejorado con un coprocesador de coma flotante integrado en el chip, instrucciones extra, enlaces mejorados y la RAM interna duplicada a 4K. Este chip altamente extendido ha fijado la reputación de los transputers, porque combina un alto rendimiento del procesamiento en coma flotante (1.5 MFLOPs) con la posibilidad de procesamiento paralelo.

- El T222 un T212 mejorado con 4K de RAM interna.

- El T801, un T800 con direccionamiento segmentado y buses de datos para accesos rápidos a memoria.

- El T425 y T805 versiones superiores al T414 y el T800 con instrucciones extras añadidas para facilitar el seguimiento paso a paso y otras herramientas de depuración.

1.2.4. Desarrollos actuales.

1.2.4.1. Arquitectura *supernode*.

Supernode es un proyecto francobritánico que tiene como objetivo construir una familia de máquinas cuyo grafo de conexiones pueda ser reconfigurado dinámicamente a nivel software, utilizando los transputers T800 diseñados por Inmos.

La arquitectura *supernode* (se emplea deliberadamente las minúsculas para designar y subrayar el carácter genérico y común de este tipo de arquitectura, que puede dar lugar a varias clases de máquinas (T-Nodes fabricados por Telmat en Francia o SN-1000 producidos por Parsys, filial de Thorn-EMI en Gran Bretaña)), es una solución atractiva de coste reducido y una alternativa a las topologías actualmente desarrolladas. Se basa fundamentalmente en un módulo con transputers T800 (cuyos enlaces pueden ser conmutados mediante un componente especial), una estrategia de construcción jerárquica y una metodología de programación elaborada sobre Occam. Este enfoque debería desembocar en la construcción de una familia de *supernodes* que iría desde el puesto de trabajo individual potente hasta el supercomputador, pasando por máquinas dedicadas a una clase especial de aplicaciones. El *supernode* es una arquitectura con la que se puede interconectar un gran número de transputers y, a la vez, optimizar el coste de las comunicaciones. Esto se consigue con un coste de fabricación muy bajo en relación a los superordenadores existentes de potencia equiparable. El elemento básico de la familia está constituido típicamente por un conjunto de dieciocho transputers T800. Estos elementos están interconectados mediante un conmutador programable, funcionalmente parecido a una central telefónica, que realiza la conmutación de los enlaces de los transputers, sin importar cual de los canales de un transputer pueda, en un momento dado, ponerse en relación con uno cualquiera de los canales de otro transputer. La arquitectura de un *supernode* es totalmente modular; los componentes básicos se pueden conectar entre sí según un principio similar a la conexión de los transputers ya descrita. Su configuración puede efectuarse incluso durante la ejecución de una tarea. El lenguaje inicial de programación se ha elaborado en base a Occam pero actualmente existen otros tipos de lenguaje (C, Pascal, Fortran, Prolog, Lisp, Ada).

1.2.4.2. El IMS T9000.

Como resultado de un proceso de revisiones sucesivas a la arquitectura transputer, se

ha obtenido un nuevo componente de la familia, el IMS T9000 (anunciado como el H1). Está destinado a ser un nuevo miembro de la familia transputer que se caracteriza por un alto nivel de compatibilidad con sus predecesores y un rendimiento mucho más alto. Las primeras características del T9000, que ya había sido anunciado en 1990, fueron publicadas oficialmente por INMOS en abril de 1991 y, los primeros modelos están disponibles desde hace pocas fechas [car91], [pou91].

El T9000 aporta el hardware necesario para una nueva concepción del diseño de software paralelo. La idea fundamental es dejar a los diseñadores de software libres de características dependientes del tipo de máquina, tales como la topología de interconexión de los procesadores.

A un nivel de abstracción alto, se puede decir que la arquitectura T9000 CPU/FPU es exactamente la misma que la de la serie transputer T800, con un incremento en la velocidad que permite alcanzar 200 MIPs y 25 MFLOPS.

En el T9000, el multiplexado/demultiplexado de canales se lleva a cabo en un componente hardware llamado *Virtual Channel Processor* (VCP). El VCP es capaz de multiplexar cualquier número de canales virtuales sobre un canal físico, utilizando un protocolo adecuado de comunicaciones. Una importante característica del VCP es que, cuando varios mensajes tienen que ser transmitidos concurrentemente sobre el mismo canal, intercambia paquetes de diferentes mensajes de cara a optimizar el tiempo de entrega.

Los transputers T9000 pueden conectarse directamente como los modelos anteriores. Sin embargo, también pueden utilizarse los VCPs para que los mensajes sean enviados (*routing*) a través de un sistema de comunicaciones que conecta todos (o varios) T9000 en el sistema. En otras palabras, es posible virtualizar las comunicaciones entre procesos asignados a T9000 que no están directamente conectados mediante un canal físico, marcando la ruta del mensaje por software.

En principio, es suficiente conectar un canal de cada transputer a la red canalizadora (*router*), sin embargo, pueden conectarse varios canales T9000 cuando se necesita un ancho de banda más amplio. Otra posibilidad que aparece, es la de conectar los enlaces no utilizados a otras redes canalizadoras (*routers*). El uso de varias redes en paralelo parece bastante interesante porque ofrece la posibilidad de aumentar el ancho de banda y crear planos separados de comunicaciones para diferentes tipos de mensajes de cara a mejorar el rendimiento.

En los últimos meses han empezado a surgir máquinas basadas en T9000, concretamente la arquitectura *supernode* ha sido una de las primeras en adaptar este tipo de procesador a su proyecto. Sin embargo este tipo de ordenadores está todavía en fase de prueba, con lo que es de suponer que haya que esperar un cierto tiempo para poder discernir sobre la eficiencia alcanzable por este tipo de máquinas.

1.2.5. Lenguajes de programación.

El lenguaje Occam y el microprocesador transputer fueron diseñados teniéndose en cuenta mutuamente. Occam es un lenguaje para programación en paralelo sobre redes de procesadores transputer [bur88], [cok91], [ell91], [jon88], [pou88], [wex89].

El modelo de programación Occam consiste en conjuntos de procesos paralelos con capacidad de comunicarse a través de canales. Los canales conectan pares de procesos permitiéndoles el intercambio de datos. A su vez, cada proceso puede estar constituido por una serie de procesos ejecutándose secuencialmente o en paralelo, de forma que un sistema software completo puede describirse como una jerarquía de procesos paralelos intercomunicados.

Un par de procesos se comunican utilizando un canal. Cada canal ofrece una forma de conexión entre dos procesos: uno de ellos coloca un mensaje en el canal y el otro lo recoge. Esta comunicación entre los procesos es síncrona: cuando un proceso envía un mensaje a otro a través de un canal, el proceso emisor de la comunicación no prosigue su ejecución hasta que el proceso receptor no está listo para recibir el mensaje. Las comunicaciones pueden realizarse a través de *buffers* colocándolos explícitamente entre los procesos intercomunicados. Un proceso puede estar preparado para comunicarse mediante cualquier número de canales. La comunicación tiene lugar cuando otro proceso está preparado para comunicarse a través de uno de los canales. Como un proceso puede poseer concurrencia interna, puede tener varios canales de entrada y varios de salida y, desarrollar comunicaciones de forma simultánea.

En Occam el paralelismo puede expresarse directamente. Cada proceso Occam es un proceso ejecutable de forma independiente. Se utiliza un lenguaje de configuración (como una extensión de Occam) para distribuir procesos en redes de transputers, y así el lenguaje permite desarrollar programas en diferentes tipos de sistemas multiprocesador.

El modelo de programación Occam está también capacitado para permitir la construcción de sistemas software basados en diferentes lenguajes. Efectivamente, se pueden integrar en un único sistema componentes escritos en diferentes lenguajes (C, Fortran, Pascal).

Dado que el transputer utiliza el concepto Occam de concurrencia y comunicación, el lenguaje puede utilizarse para programar a un solo transputer o a una red de transputers. En el caso de un único transputer, éste comparte su tiempo entre los procesos concurrentes y las comunicaciones. Para una red de transputers, cada transputer ejecuta los procesos que le han sido asignados. La comunicación entre procesos Occam de distintos transputers se hace a través de los canales físicos de comunicación. En general, la implementación usando una red alcanzará mejores resultados, en lo que se refiere a la velocidad, que el proceso equivalente para un solo transputer. Sin embargo el costo de tal red puede ser significativo, por lo que la configuración elegida para un programa dado depende de un balance rendimiento-coste.

Que Occam y el transputer estén íntimamente ligados no significa que sea el único lenguaje válido para programar transputers. La amplia utilización de los lenguajes de programación secuenciales como C, Fortran y Pascal, ha obligado a varias compañías (Inmos, 3L, etc) a desarrollar compiladores de estos lenguajes para transputers. La portabilidad de los sistemas, unido a las ventajas específicas de los lenguajes de alto nivel no incluidas en Occam (estructuras dinámicas y recursividad), hacen a estos últimos buenos candidatos para servir de soporte en la programación de las máquinas basadas en transputers.

Si bien Occam es el lenguaje que mejor expresa la concurrencia de los procesos secuenciales que se comunican (CSP), y actualmente es el lenguaje que emite el código más eficiente, no es menos cierto que los lenguajes secuenciales ya consagrados como C permiten crear las mismas estructuras de procesos mediante el uso de librerías para las funciones asociadas con la concurrencia y permiten incluso en este caso una portabilidad inexistente en Occam. En el entorno de programación PVM para máquinas heterogéneas [gei93], [rod94] se puede ver un ejemplo similar. En este caso también se dispone de una serie de librerías de rutinas para la manipulación de procesos que se comunican. Como ha ocurrido en los lenguajes de alto nivel secuenciales, donde se ha optado por códigos estándar independientes de las arquitecturas, parece obvio que al menos en los modelos MIMD, el proceso se repita y el futuro se apoye en un lenguaje eminentemente secuencial con la inclusión de librerías para la administración de las comunicaciones.

Los lenguajes utilizados en este trabajo han sido Occam y ANSI C de Inmos [inm88], [inm90]. Este último suministra unas rutinas de librería para manejar la concurrencia. Estas funciones se dividen en tres grupos:

- rutinas para administrar procesos (creación, activación y planificación).
- rutinas para manejar canales de comunicación (creación e inicialización de canales, envío y recepción de mensajes, selección de canal a activar, etc)
- rutinas para controlar semáforos (creación, etc).

Estas rutinas son tratadas del mismo modo que las restantes. El soporte de la concurrencia se completa con tres nuevos tipos de datos.. Para los procesos y los semáforos se usan estructuras de datos específicos. Para la implementación de los canales se utiliza el tipo puntero.

Junto con el compilador se suministra un conjunto de herramientas. Una de ellas consiste en un configurador que, como en el lenguaje Occam, permite describir redes software y hardware por separado, así como las relaciones entre ambas. El lenguaje sigue la sintaxis de C e incluye constructos de alto nivel como instrucciones de repetición y condicionales, lo que lo hacen atractivo para describir diferentes configuraciones (en el apartado siguiente, dedicado a las redes de ordenadores, se presentan algunos ejemplos de configuración mediante este lenguaje).

1.3. MODELOS BASICOS DE REDES DE INTERCONEXION (NETWORKS).

Todos los sistemas multiprocesador que se construyen en la actualidad tienen como una de sus características principales la forma en que cada procesador puede comunicar con otro cuando esto sea necesario. Tanto a nivel físico como lógico, el aumento del número de procesadores de los que dispone el sistema debe ir acompañado de un grafo de interconexión que permita transmisiones de datos entre ellos en un tiempo razonable [qui94]. En estas condiciones, si bien por un lado es necesario crear algoritmos paralelos que se adapten a la topología de la máquina donde se va a realizar la ejecución, no es menos cierto que la tendencia actual lleva hacia sistemas que tratan de realizar una simulación eficiente de cualquier tipo de red (T9000, etc).

Matemáticamente, una red de interconexión se puede ver como un modelo de grafo G = (V, A) dirigido o no: Los N procesadores, en principio idénticos, están localizados en los vértices (nodos) del grafo recursivo, potencialmente infinito, y se comunican a través de los arcos (aristas). El uso práctico de estas estructuras está limitado tanto por restricciones de cableado, como por principios de diseño y fabricación. Entre ellas se pueden destacar:

a.- La dificultad de realizar comunicaciones entre procesadores en tiempo logarítmico, debido a las restricciones sobre la capacidad de las líneas de comunicación.

b.- Las limitaciones físicas de ventilación que requieren que el número de procesadores vecinos sea constante o logarítmico, en términos del número total de estos.

c.- Las restricciones de cableado que no permiten más de dos o tres niveles de este y que ponen obstáculos sobre la densidad y longitud de los cables de conexión en un diseño general, sobre todo en implementaciones VLSI.

Teniendo en cuenta estas limitaciones, se han sugerido una serie de criterios para evaluar estas organizaciones, los cuales ayudan a entender la efectividad de las redes en la implementación de algoritmos paralelos eficientes sobre el hardware real. Estos criterios son los siguientes:

1.- Grado o número de aristas por nodo (G). Es mejor si el número de aristas por vértice es una constante independiente del tamaño de la red, ya que entonces la organización escala más fácilmente a sistemas con un gran número de procesadores.

2.- Diámetro (D). El diámetro de una red es la distancia más larga entre dos nodos cualesquiera. Los diámetros bajos, al menos logarítmicos, son mejores, ya que este es una cota inferior en la complejidad de los algoritmos paralelos que requieren comunicaciones entre pares de vértices arbitrarios.

3.- Ancho (A). Es el mínimo número de aristas que deben ser eliminadas con el fin de dividir la red en dos mitades (ancho de bisección). Son mejores los anchos altos, ya que en algoritmos que requieren largas cantidades de comunicaciones, el tamaño del

conjunto de datos dividido por el ancho proporciona una cota inferior en la complejidad del algoritmo paralelo.

Como apunte adicional y de aprendizaje, en algunas de las organizaciones se muestra el modo de configuración para su utilización sobre máquinas de transputers reconfigurables a nivel software.

1.3.1. Malla rectangular (*rectangular mesh*).

En una malla rectangular de n × n procesadores (supuesta cuadrada por simplicidad), los nodos están organizados en un plano (generalizando hiperplanos de dimensión q). Cada nodo (i, j) comunica con los procesadores (i±1, j) e (i, j±1), supuesto que existan, como se observa en la figura 1.6. La evaluación de este tipo de topología presenta los siguientes resultados: El grado de cada vértice (G) es igual a 4, constante para cualquier número de procesadores. El diámetro (D) es igual a 2(n-1), superior al criterio logarítmico fijado. El ancho (A) es igual a n.

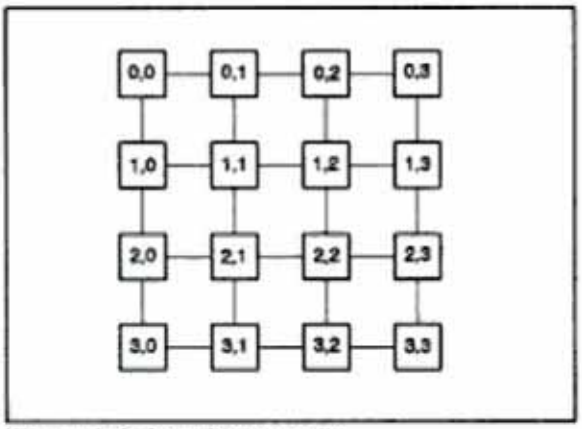

Figura 1.6 Conexión en malla.

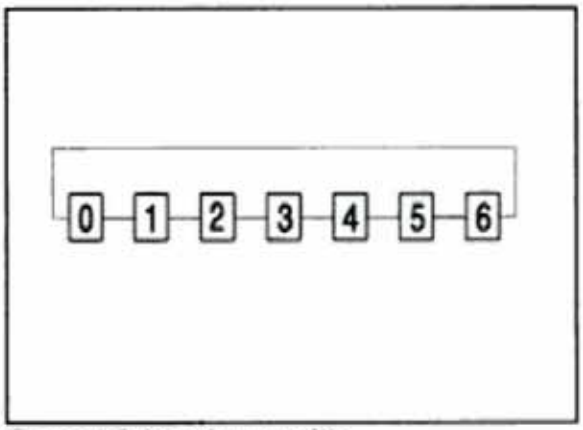

Figura 1.7 Conexión en anillo.

Este tipo de red ha sido utilizado para máquinas de tipo *processor arrays* (MasPar's MP-1, etc) y en el multicomputador Intel Paragon XP/S. A nivel de programación, este tipo de topología es muy útil para aproximaciones de diferencias finitas a problemas de ecuaciones diferenciales y en algunos diseños de tipo sistólico.

1.3.2. Anillo (*ring*).

En un anillo de n procesadores, cada nodo (i) tiene como vecinos a ((i+1) MOD n) e ((i-1) MOD n) (figura 1.7). Los criterios aplicables muestran su poca efectividad en general. Los parámetros toman los valores G = 2, D = n/2, A = 2. Debido a sus malos resultados no ha sido utilizada como organización en ningún tipo de máquina paralela de propósito general. Sin embargo es muy adecuada en los códigos de cadena de montaje

(*pipeline*) con realimentación, así como en los procesos sistólicos.

Debido a que todos los nodos tienen el mismo número de vecinos (independiente del tamaño de la red) y su identificación es muy sencilla (MOD n), la reconfiguración lógica de una máquina como anillo es bastante simple. El código 1.1 muestra en el lenguaje de configuración de Inmos C, la descripción de la topología de anillo.

```
val BootLink 0;          /* Boot Link of Ring */
val n_ring ¿?;           /* Number of nodes of Ring */
val n_trans n_ring+1;    /* Number of Transputers */

process (stacksize = 64K, heapsize = 2048K);

/* The configuration uses (n_trans) T805s, configure from 0 to (n_ring-1)
   as a ring. The root is connected to the host by link 0. */

/* ********** Hardware Description ********** */

T805 (memory = 4M) root;
T805 (memory = 4M) Ring[n_ring];

connect root.link[BootLink], host;        /* Host Link connection */
connect root.link[2], Ring[0].link[2];

rep i = 0 for n_ring-1 {
  if (i%2)
    connect Ring[i].link[2], Ring[i+1].link[2];
  else
    connect Ring[i].link[1], Ring[i+1].link[1];
} /* rep ... */
connect Ring[n_ring-1].link[3], root.link[3];
```

Código 1.1 Configuración como anillo.

1.3.3. Arbol binario (*binary tree*).

En una topología de árbol binario de profundidad k, los 2^{k+1}-1 procesadores se organizan en un árbol binario completo de la misma profundidad (figura 1.8). A cada nodo se le asocia un identificador entre 0 y 2^{k+1}-2 mediante un recorrido del árbol por niveles. El máximo número de aristas de un nodo es tres (G = 3). Cada nodo interior puede comunicar con sus dos hijos y cada vértice distinto de la raíz puede hacerlo con su padre. Su diámetro (D = 2(k-1)) también es admisible. Sin embargo el parámetro ancho es muy pobre (A = 1). De la misma forma que en la conexión mediante un anillo, su pequeña bisección representa el cuello de botella que provoca su inutilidad para ser usada como organización general. Su importancia estriba en que se adapta de manera natural a la paralelización de los procesos recursivos de la técnica de programación divide y vencerás.

A nivel de configuración esta topología no cuenta con un número fijo de vecinos para todos los nodos, como ocurría en la anterior, con lo cual la identificación de las conexiones se vuelve un poco más compleja. El código 1.2 muestra una especificación de una

con guración como árbol binario.

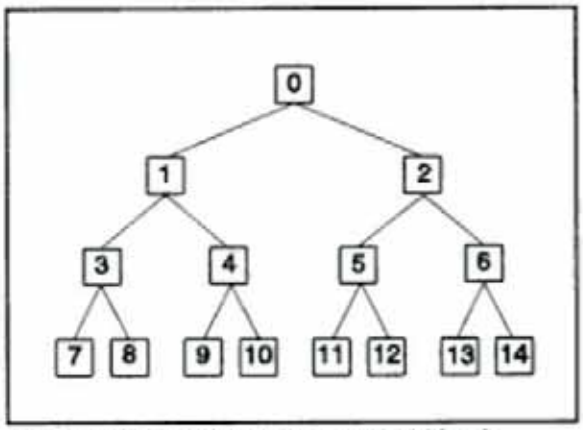

Figura 1.8 Conexión según un árbol binario.

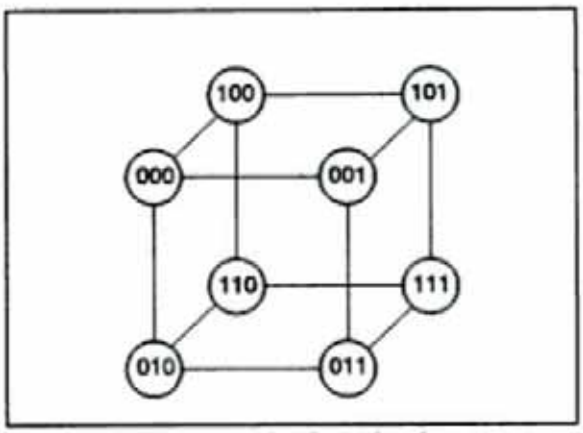

Figura 1.9 Hipercubo de dimensión 3.

```
val BootLink 0;             /* Boot Link of Tree */
val n_level 3;              /* Number of levels of Tree */
val n_leafs 4;              /* Number of leafs of Tree */
val fn {0, 1, 3};           /* Index of first Proc. in Level */
val sl {1, 2, 4};           /* Number of Proc. by Level */
val n_trans 7;              /* Number of Transputers */
val n_nodes n_trans - (n_leafs + 1);

process (stacksize = 128K, heapsize = 2048K);

/* The configuration uses (n_trans) T805s, configured from 0 to (n_trans-1)
   as a tree. */

/* ********** Hardware Description ********** */

T805 (memory = 4M) Tree[n_trans];

/* Physic Connection of Root */
connect Tree[0].link[BootLink], host; /* Host Link connection */
connect Tree[0].link[1], Tree[1].link[1];
connect Tree[0].link[2], Tree[2].link[2];

/* Physic Connection of Left and Right Son of Nodes no Leafs */
rep i = 1 for n_level-2 {
  rep k = 0 for sl[i] {
    if (i%2) {
      connect Tree[fn[i]+k].link[3], Tree[2*(fn[i]+k)+1].link[3];
      connect Tree[fn[i]+k].link[0], Tree[2*(fn[i]+k)+2].link[0];
    } /* if ... */
    else {
      connect Tree[fn[i]+k].link[1], Tree[2*(fn[i]+k)+1].link[1];
      connect Tree[fn[i]+k].link[2], Tree[2*(fn[i]+k)+2].link[2];
    } /* else ... */
  } /* rep ... */
} /* rep ... */
```

Código 1.2 Configuración como árbol binario de tres niveles.

1.3.4. Hipercubo (*Hypercube*).

Este tipo de topología consta de 2^k procesadores formando un hipercubo k dimensional. Si se denotan a los nodos con los índices 0, 1, ..., 2^k-1, dos vértices son adyacentes si sus etiquetas difieren en exactamente un único bit. En la figura 1.9 se muestra un hipercubo tridimensional. El diámetro (D) de un hipercubo k-dimensional (2^k nodos) es k y su ancho (A) es 2^{k-1}. Este tipo de organización tiene un diámetro bajo y una bisección grande. Sus buenos resultados quedan desvirtuados por un grado no constante (G = k). Esta es, sin lugar a dudas, su mayor deficiencia. Debido a las limitaciones físicas existentes, es imposible construir hipercubos más allá de un cierto grado. Esta deficiencia se ha intentado soslayar con la creación de topologías derivables de los hipercubos como son los hipercubos conectados en ciclos, en los que en cada vértice se sitúa un anillo de procesadores.

A nivel de arquitectura, cuando el número de procesadores se encuentra dentro de unos ciertos límites, ha sido utilizada como organización de propósito general. Como ejemplo práctico, en la Connection Machine CM-200 los *clusters* de procesadores elementales se conectan mediante este tipo de red. Existen teoremas que demuestran la optimalidad de esta red como computador paralelo de propósito general.

1.3.5. De Bruijn.

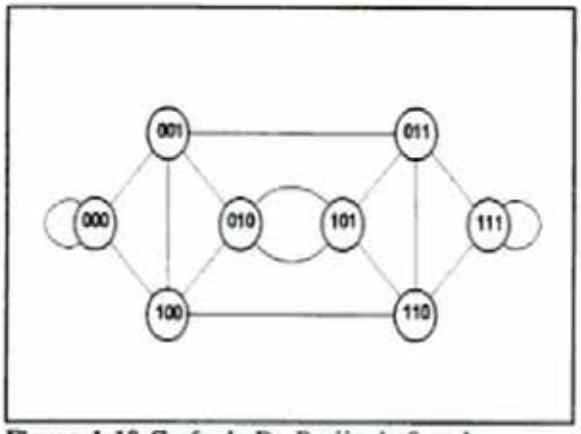

Figura 1.10 Grafo de De Bruijn de 8 nodos.

Una red de Bruijn esta formada por 2^k nodos. Supuesto que se identifica a un nodo por su representación binaria $a_{k-1}a_{k-2}...a_1a_0$, los dos nodos alcanzables directamente mediante arcos (aristas dirigidas) tienen las direcciones siguientes: $a_{k-2}...a_1a_0 0$ y $a_{k-2}...a_1a_0 1$. En la figura 1.10 se muestra una red con ocho procesadores.

Esta es la red que mejor comportamiento muestra: G = 2, por lo tanto, un factor constante de número de enlaces, D = k, un diámetro logarítmico y A = $2^k/k$, una bisección bastante alta.

Los procesadores del Triton/1, un computador paralelo SIMD/MIMD desarrollado en la Universidad de Karlsruhe, están conectados mediante este tipo de organización [her93]. A nivel algorítmico, este tipo de red ha sido utilizado en los experimentos con códigos paralelos basados en la técnica de programación general ramificación y acotación.

1.4. COMPLEJIDAD ALGORITMICA PARALELA.

Como señalan Kinderwater y Lenstra [kin88], además de la resolución en tiempo polinomial y completitud para NP (los cuales son los puntos más importantes en computación secuencial), se han estudiado muchos otros conceptos para computación paralela. Entre los más importantes destacan las nociones de resolución en tiempo polilogarítmico paralelo y la completitud espacial para P. La teoría de la complejidad paralela se desarrolla alrededor del modelo P-RAM. Surgen inmediatamente las siguientes preguntas:

> ¿Es todo problema de la clase NP resoluble en tiempo polinomial por una P-RAM?.
> ¿Cómo influye el número de procesadores activos durante una computación en el tiempo de resolución de un problema ?.

Una fuente importante para el conocimiento de la computación paralela sobre SIMD P-RAM es la referencia de Cook, *Towards a Complexity Theory of Synchronous Parallel Computation* [coo80].

1.4.1. Conceptos y definiciones.

Asociado con cualquier problema de optimización siempre se puede definir un problema de decisión del tipo *Dado un ejemplo del problema I (es decir, un conjunto de datos) ¿ existe una estructura S que satisface una cierta propiedad Q ?*, esto es una cuestión que puede ser contestada *SI* ó *NO*. Un problema se dice *factible* si permite una respuesta afirmativa. El *tamaño de un ejemplo* se define como el número de *bits* necesarios para codificar los datos bajo un esquema razonable de codificación. El *tiempo de ejecución* de un algoritmo se puede ver como el número de operaciones elementales requeridas para su solución. Un algoritmo se dice *polinomial* si su tiempo de ejecución está acotado por una función polinómica del tamaño del problema. Bajo suposiciones bastante realistas, se puede demostrar que el problema de optimización puede ser resuelto eficientemente, siempre que se pueda resolver el problema de decisión. De ahí que la teoría de la complejidad esté basada en los problemas de decisión.

1.4.2. Máquinas paralelas de acceso aleatorio.

Un punto básico en la teoría de la complejidad es hallar el modelo más apropiado de computador. Diferentes modelos conducirán a diferentes computaciones y el número de pasos ejecutados por el mismo algoritmo será diferente, dependiendo del modelo de máquina considerado. El modelo de máquina RAM (figura 1.11) consta, como ya se sabe, de una cinta de sólo lectura, una cinta de sólo escritura, un programa y una memoria.

Fortune y Wyllie [for78] proponen el modelo P-RAM (Parallel Random Access Machine) para computaciones paralelas, el cual es una extensión del modelo RAM. En una

P-RAM (figura 1.12) hay una secuencia potencialmente infinita de registros globales, además de un conjunto potencialmente infinito de procesadores programados idénticamente. Cada procesador tiene un conjunto de registros locales, y en cualquier instante puede acceder a sus registros locales o a los globales. Se permiten lecturas simultáneas, pero no escrituras (esto es, el sistema se detiene en un estado de error en caso de conflicto debido a que dos o más procesadores intentan escribir en el mismo registro global en el mismo instante). Al comienzo de la computación la entrada se carga en los registros globales y únicamente el primer procesador está activo. En cualquier instante cualquier procesador activo puede realizar una de un conjunto de operaciones primitivas, entre ellas la de *activar* a un nuevo procesador (que funcionará en paralelo con los que estén ya en operación desde ese instante hasta que se detenga). El primer procesador está siempre activo y la computación se termina cuando éste se detiene.

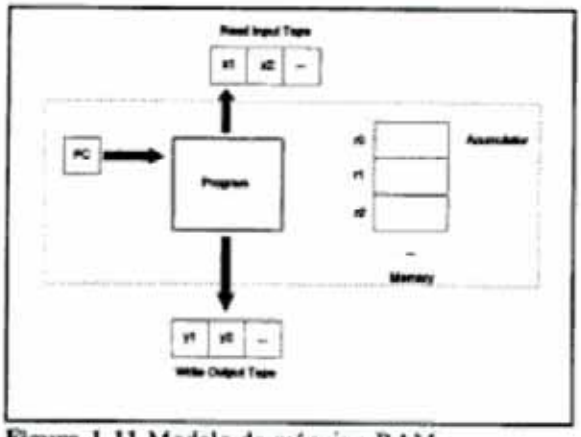

Figura 1.11 Modelo de máquina RAM.

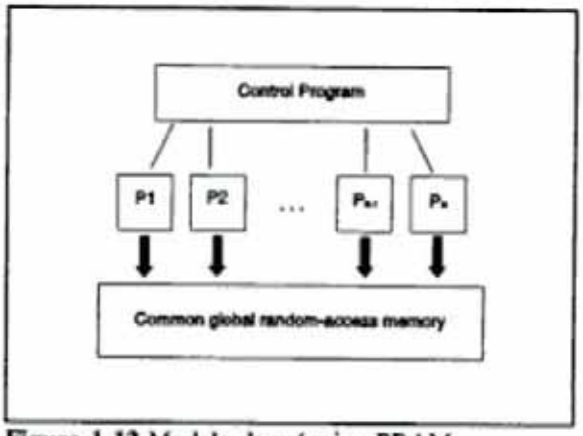

Figura 1.12 Modelo de máquina PRAM.

Puesto que cualquier procesador puede acceder a cualquier registro global, se requiere un tiempo O(log N) para activar N procesadores. Este tiempo de activación es desde luego una cota inferior del tiempo de activación en cualquier modelo razonable, esto es, en cualquier modelo que se pretenda factible, debido a los retrasos introducidos en la práctica por las limitaciones de interconexión y ventilación.

1.4.3. Resultados de la complejidad paralela.

Se ha demostrado que el conjunto de problemas resolubles en tiempo acotado por un polinomio en una función *h* dada por una máquina P-RAM paralela y el conjunto de problemas resolubles con un costo de memoria acotado por un polinomio en esa misma función *h* son iguales. Este enunciado constituye la *TESIS DE LA COMPUTACION PARALELA*. Esta conjetura afirma que la clase de problemas de decisión resolubles en tiempo $T(n)^{O(1)}$ por una máquina con paralelismo no acotado (esto es, polinomial en T(n), donde n es el tamaño del problema) es la misma clase de los problemas resolubles en espacio $T(n)^{O(1)}$ por una máquina secuencial. Dado que los problemas de PSPACE contienen a los problemas de NP, se sigue que los problemas NP-completos son resolubles en tiempo polinomial por

una P-RAM. Sin embargo, Blum [blu83] ha demostrado que la tesis de la computación paralela no es cierta cuando, para el modelo de computador paralelo considerado, se puede activar en un único paso un número arbitrariamente grande aunque finito de procesadores y cualquier procesador puede acceder a cada celda de una memoria arbitrariamente grande aunque finita. Como explica Blum, estas propiedades las verifican varios modelos [shi81], pero no son satisfechas por estos mismos cuando se necesita tiempo O(log N) para activar N procesadores.

La tesis de la computación paralela se cumple, en particular cuando $T(n) = n^{O(1)}$ (es decir, es una función polinomial del tamaño del problema) y el modelo del computador es una P-RAM.

El modelo P-RAM es bastante poderoso: la clase de problemas de decisión que una P-RAM puede resolver en tiempo polinomial es precisamente PSPACE, incluyendo los muy difíciles problemas NP - completos y PSPACE completos. Sin embargo, en el caso más realista de una P-RAM con un número polinomial de procesadores (en lugar de paralelismo no acotado), la clase de problemas resolubles en tiempo polinomial es exactamente P. Bajo esta suposición restrictiva pero realista, todo lo que puede ser obtenido con el paralelismo es acelerar la solución de los problemas en P: los problemas duros (*hard*), para los cuales desafortunadamente es imposible encontrar algoritmos en tiempo polinomial, permanecen tan duros como sin paralelismo, en términos de complejidad teórica.

Se obtiene que muchos problemas en P pueden resolverse en tiempo paralelo de orden $(\log n)^{O(1)}$, es decir, en tiempo que está acotado por un polinomio en el logaritmo del tamaño del problema. Entre estos están los problemas de: Hallar el máximo, sumas parciales, ordenación, planificación con tareas fijas, máximo flujo en grafos planares, programación lineal con un número fijo de variables, etc.

Basada en la tesis de la computación paralela, la clase POLYLOGSPACE de problemas de decisión que pueden ser resueltos con un costo de memoria POLYLOG (esto es, acotado por un polinomio de logaritmos del tamaño de la entrada) por una máquina secuencial, está definida como correspondiente a la clase de problemas resolubles en tiempo POLYLOG por una máquina P-RAM paralela usando un número polinomial de procesadores. Puesto que cualquier modelo realista de computador necesita al menos tiempo O(log N) para activar N procesadores, es bastante improbable que se puedan encontrar algoritmos sublogarítmicos. En este sentido, la clase POLYLOGSPACE puede considerarse como la formada por los problemas en P que pueden ser resueltos eficientemente a través del uso del paralelismo.

Un problema se dice P-completo (en la terminología de Ian Parberry [par87]), si pertenece a P y todos los problemas en P pueden ser reducidos a él por medio de una transformación computable utilizando una cantidad de memoria polilogarítmica. Ejemplos de problemas P-completos son: *Circuit value* (dado un circuito combinatorio, esto es un circuito sin bucles de retroalimentación y una asignación a sus entradas, computar su salida), el

problema de Programación Lineal, el problema del máximo flujo, etc.

La cuestión de si P ⊂ POLYLOGSPACE está todavía abierta y es fundamental para la teoría de la computación paralela. El estado actual de la cuestión es análogo al de la cuestión NP ⊂ P. Al igual que con esta última la conjetura es que P no está contenido en POLYLOGSPACE, como parece hacer pensar la existencia de problemas P-completos, que son los problemas más duros en P en el sentido de que si uno de ellos es miembro de POLYLOGSPACE entonces también lo es cualquier problema en P.

1.4.4. La eficiencia del paralelismo.

Como ya se ha visto por medio de la complejidad algorítmica, no es posible reducir mediante paralelismo real a funciones de diferente tipo que el obtenido por máquinas secuenciales. La única mejora que se puede obtener consiste en disminuir el orden de las funciones. Para comparar los algoritmos paralelos que han surgido se han introducido diversas reglas.

Dos medidas importantes de la calidad de los algoritmos paralelos implementados en máquinas paralelas son la *aceleración* (*speedup*) y la *eficiencia*. Se puede definir la aceleración de un algoritmo paralelo que se ejecuta sobre p procesadores como el cociente entre el tiempo que toma la máquina paralela sobre el algoritmo secuencial más rápido (en un único procesador) y el correspondiente sobre la misma máquina del algoritmo paralelo con p procesadores. La eficiencia de un algoritmo paralelo que se ejecuta sobre p procesadores es la aceleración dividida por p.

Un ejemplo que ilustra esta terminología es el siguiente: Si el mejor algoritmo secuencial tarda 8 segundos sobre uno de los procesadores del ordenador paralelo, mientras un algoritmo paralelo resolviendo el mismo problema necesita 2 segundos cuando se utilizan 5 procesadores, entonces se dirá que el algoritmo paralelo muestra una aceleración de 4 con 5 procesadores. La eficiencia correspondiente es 0.8.

1.4.4.1. Influencia del tamaño del problema.

Muchos factores pueden contribuir a limitar el aumento de velocidad que se obtiene mediante un algoritmo paralelo ejecutado sobre un modelo MIMD. Un obstáculo evidente es el tamaño del problema de entrada. Si no hay bastante trabajo para el número de procesadores disponibles, entonces cualquier algoritmo paralelo podría mostrar un incremento de velocidad bajo. Este fenómeno se conoce con el nombre de *efecto Amdahl* [goo77], y explica porque el aumento de velocidad es casi universalmente una función creciente del tamaño del problema de entrada.

1.4.4.2. Influencia del tamaño del grano.

Se debe minimizar el número de activaciones y sincronizaciones de los procesos que componen el algoritmo paralelo. Puesto que la activación de un proceso es más costosa que su sincronización, la táctica normal es crear el número deseado de procesos cuando comienza la ejecución del algoritmo y sincronizarlos cuando sea necesario. Si la frecuencia de sincronizaciones es alta, la sobrecarga puede ser muy significativa. Una de las metas del diseñador del algoritmo debería ser maximizar la cantidad de trabajo a realizar entre sincronizaciones (grano), mientras mantiene ocupados a todos los procesadores.

1.4.4.3. Ley de Amdahl.

La parte de código intrínsecamente secuencial limita la aceleración de un algoritmo paralelo (lo que se conoce como ley de Amdahl). Sea f la fracción de operaciones de un algoritmo paralelo que deben ser realizadas de forma secuencial ($0 \leq f \leq 1$). La ley de Amdahl establece que la máxima aceleración S que se puede conseguir mediante un modelo paralelo de p procesadores ejecutando ese algoritmo es:

$$S \leq \frac{1}{f + \frac{1-f}{p}}$$

Una segunda limitación provocada por el código secuencial, es el retraso que producen en la activación de los restantes procesadores. Este fenómeno se conoce con el nombre de *efecto árbol* (este efecto es muy evidente en los algoritmos de la técnica divide y vencerás, como se verá posteriormente).

1.4.4.4. Asignación de tareas.

La carga de trabajo se debe repartir entre los procesadores disponibles. Si se conoce a priori las tareas a ejecutar y sus relaciones de precedencia (como en algunos algoritmos de tipo divide y vencerás), la asignación se denomina estática. Si por el contrario el trabajo se genera durante la ejecución (algoritmos tipo ramificación y acotación) se conoce con el nombre de dinámica. A priori, una descomposición estática reduce la cantidad de comunicaciones entre procesadores, sin embargo en algoritmos donde la cantidad total de carga es desconocida al comenzar, la descomposición dinámica hace más fácil la obtención de un equilibrio computacional entre todos los procesadores de la red.

1.4.5. Medidas de rendimiento.

La métrica que más se utiliza para comparar los algoritmos paralelos en máquinas reales es el tiempo de ejecución. Sin embargo a lo largo de los años han aparecido otra serie

de medidas que son actualmente aceptadas.

Tiempo de ejecución total (*Execution time*).

Es el tiempo necesario para computar la solución del problema global. Incluye el tiempo necesario para distribuir el problema inicial a los nodos, detectar que se ha encontrado una solución, recolectar esta y detener a todos los procesadores.

Aceleración (*Speedup*).

Se define como el cociente entre el tiempo de ejecución del algoritmo secuencial y el del paralelo. El número de procesadores utilizados en la versión paralela es una cota superior para esta medida.

Eficiencia (*Efficiency*).

Es una medida de lo bien que han sido utilizados los procesadores. Se calcula como el tanto por ciento de la aceleración dividida entre el número de procesadores utilizados.

Tiempo ocioso (*Idle time*).

Suma de todos los tiempos que un procesador permanece ocioso, esperando por nuevos problemas. Se excluye el tiempo invertido en la recepción del primer paquete de subproblemas.

Número de operaciones (*Number of operations*)

Es una medida teórica del número de operaciones básicas realizadas por un algoritmo. En el caso paralelo se computa una única operación para todos los procesadores que trabajen en cada unidad de tiempo.

Trabajo (*Work*)

Se trata de calcular el número de operaciones totales realizada por un algoritmo paralelo. Al contrario que la medida anterior, en el caso de varios procesadores actuando a la vez se computa todas y cada una de las operaciones realizadas.

1.5. BIBLIOGRAFIA.

[blu83]. N. Blum. A note on the "parallel computation thesis". *Information Processing Letters* 17, pp. 203-205.

[bur88]. A. Burns. *Programming in Occam 2*. Addison-Wesley Publishing Company. England.

[car91]. U. de Carlini and U. Villano. *Transputers and parallel architectures. Message-passing distributed systems*. Ellis Horwood Series in Computers and their applications. England.

[cok91]. R.S. Cok. *Parallel programs for the transputer*. Prentice Hall. Englewood Clifs, NJ.

[coo80]. S.A. Cook. Towards a complexity theory of synchronuos parallel computation. *L'Enseignement Mathematique* 30.

[ell91]. D. Ellison. *Understanding occam and the transputer*. Sigma Press. England.

[for78]. S. Fortune and J. Wyllie. Parallelism in random access machines. *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pp. 114-118.

[fly66]. M.J. Flynn. Very high-speed computing systems. *Proceedings of IEEE* 54 (12), pp. 1901-1909.

[gei93]. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck and V. Sunderam. *PVM 3.1 User's guide and reference manual*. Oak Ridge laboratory.

[goo77]. S.E. Goodman and S.T. Hedetniemi. *Introduction to the design and analysis of algorithms*. McGraw-Hill, NY.

[her93]. C.G. Herter, T.M. Warschko, W.C. Tichy and M. Philippsen. Triton/1: A massively-parallel mixed-mode computer designed to support high level languages. *Proceedings of the International Parallel Processing Symposium*.

[hwa85]. K. Hwang and F.A. Briggs. *Computer architecture and parallel processing*. McGraw-Hill.

[inm88]. INMOS Limited. *Occam2 reference manual*. Prentice Hall. Series in Computer Science. C.A.R. Hoare Series Editor.

[inm90]. INMOS Limited. *AnsiC toolset reference manual*. Inmos Limited.

[jon88]. G. Jones and M. Goldsmith. *Programming in Occam 2*. Prentice Hall. Series in Computer Science. C.A.R. Hoare Series Editor.

[kin88]. G.A.P. Kindervater and J.K. Lenstra. Parallel computing in combinatorial optimization. *Annals of Operations Research* 14, pp. 245-289.

[par87]. I. Parberry. *Parallel complexity theory*. John Wiley & Sons. NY.

[pou88]. D. Pountain and D. May. *A Tutorial Introduction to Occam Programming*. Inmos. Oxford, BSD Professional Books. England.

[pou91]. D. Pountain. The transputer strikes back. *BYTE*, pp. 265-275.

[qui94]. M.J. Quinn. *Parallel computing: Theory and practice*. McGraw-Hill. OR.

[rod94]. J.L. Roda. Computación distribuída sobre redes heterogéneas: El entorno PVM. *Memoria de Licenciatura*. Dept. Estadística, Inv. Operativa y Computación, Univ. de La Laguna.

[shi81]. Y. Shiloach and U. Vishkin. Finding the maximun, sorting and merging in a parallel computation model. *Journal of Algorithms* 2 (1), pp. 88-102.

[wex89]. J. Wexler. *Concurrent Programming in Occam 2*. Ellis Horwood Series in Computers and their applications. England.

CAPITULO II:

Divide y Vencerás

2.1. INTRODUCCION.

Una de las técnicas más importante y que más se ha aplicado en el diseño de algoritmos eficientes, es la estrategia *divide y vencerás* (*divide and conquer*). Consiste en dividir el problema en subproblemas más pequeños del mismo tipo, resolver estos subproblemas de forma separada, y combinar los resultados parciales para obtener la solución total. Este método es utilizado, recursivamente, para dividir el problema en problemas más y más pequeños hasta alcanzar un punto donde cada uno de ellos es fácil de resolver [aho83].

Este concepto ha llevado al diseño de algoritmos secuenciales eficientes en los campos de la ordenación y búsqueda, transformada de Fourier, multiplicación de matrices, etc. [aho74].

2.2. EL METODO DIVIDE Y VENCERAS.

El método general se puede expresar mediante el siguiente pseudocódigo recursivo:

```
Entrada: Problema P0
Salida:  Resultado R0
Método:  Procedure DivideAndConquer(Input P; Output R);
         begin
           if small(P) then R := solve(P)
           else begin
             divide (P, P1, ..., Pk);
             for j := 1 to k do DivideAndConquer(Pj, Rj);
             R := combine(R1, ..., Rk);
             end; { else }
           end; { DivideAndConquer }
```

Código 2.1 Esquema de la técnica divide y vencerás.

En los siguientes apartados, se desarrollan cada una de las operaciones que componen este método. Para mejorar la exposición de las características de la estrategia, se introducen dos ejemplos que resuelven el problema de ordenación: El método de ordenación *rápido* (*QuickSort*) y el método de ordenación por *fusión* ó *mezcla* (*MergeSort*).

El problema de la ordenación (o clasificación) consiste en reorganizar un conjunto dado de objetos en una secuencia específica. El objetivo de este proceso es facilitar una posible búsqueda posterior entre los elementos del conjunto ordenado.

Matemáticamente, el problema de la clasificación se puede formular de la siguiente manera:

Entrada: Una secuencia de n elementos $a_1, a_2, \ldots, a_n$ y una relación de orden total $\leq$.
Salida: Una permutación π de los n elementos $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$ tal que $a_{\pi(i)} \leq a_{\pi(i+1)}$, $1 \leq i \leq n$.

La importancia teórica de este tipo de problema se debe a que suele aparecer como parte esencial de muchos algoritmos. En el código 2.2 se muestran dos de los mejores algoritmos para resolverlo, la clasificación rápida y la ordenación por mezcla, ambos basados en la estrategia divide y vencerás.

```
                    var a : array [1..n] of item;

Procedure QuickSort(l, r : index);
var i, j : index;
begin
  if l < r then begin
    { Divide problem }
    i := l;
    j := r;
    Partition(l, r, i, j);
    if (l < j) then QuickSort(l, j);
    if (i < r) then QuickSort(i, r);
  end; { if ... }
end; { QuickSort }

Procedure Partition(l, r : index,
  var i : index, var j : index);
  var piv : index;
  var w : item;
begin
  piv := a[(l + r) div 2];
  { or piv :=
       mean of some elements
       median, ... }
  repeat
    while a[i] < piv do
      i := i+1;
    while piv < a[j] do
      j := j-1;
    if i ≤ j then
    begin
      w := a[i];
      a[i] := a[j];
      a[j] := w;
      i := i+1;
      j := j-1;
    end; { if ... }
  until i > j;
end; { Partition }
```

```
Procedure MergeSort(l, r : index);
var med : index;
begin
  if l < r then begin
    { Divide problem }
    med := (l + r) div 2;
    MergeSort(l, med);
    MergeSort(med+1, r);
    Merge(iz, med, r); { Combine }
  end; { if ... }
end; { MergeSort }

Procedure Merge(l, med, r : index);
var b : array [1..n] of item;
var h, i, j, k : index;
begin
  h := l; i := l; j := me+1;
  while (h ≤ med) and (j ≤ r) do
  begin
    if a[h] ≤ a[j] then begin
      b[i] := a[h]; h := h+1;
    end { if ... }
    else { a[h] > a[j] } begin
      b[i] := a[j]; j := j+1;
    end; { else }
    i := i+1;
  end; { while ... }
  if h > med then
    for k := j to r do begin
      b[i] := a[k]; i := i+1;
    end { for ... }
  else { j > r }
    for k := h to med do begin
      b[i] := a[k]; i := i+1;
    end; { for ... }
  for k := l to r do a[k] := b[k];
end; { Merge }
```

Código 2.2 Ejemplos de aplicación de la técnica divide y vencerás. Ordenación rápida y por mezcla.

2.2.1. La operación de división (*divide*).

Los algoritmos que se estudian se basan en la división del problema de entrada en unos cuantos subproblemas de menor tamaño, independientes entre sí y del mismo tipo que el de partida. La posterior combinación de los resultados llevará a la resolución del problema

general. En los dos ejemplos que se plantean, la división es la siguiente:

El problema P_0 con n elementos se divide en dos problemas P_1 y P_2, de tamaño n/2 en la clasificación por mezcla (en la ordenación rápida, no se asegura la igualdad de tamaños entre los problemas)[1]. Los problemas parciales quedan identificados por su tamaño y los índices de los elementos dentro de la estructura de datos utilizada.

A continuación, se muestran los problemas generados para los dos algoritmos.

a) Ordenación rápida

P_1: $a_{k(1)}, \ldots, a_{k(index)}$,
$a_{k(i)} \leq piv,\ 1 \leq i \leq index$

P_2: $a_{k(index+1)}, \ldots, a_{k(n)}$,
$a_{k(i)} \geq piv,\ index+1 \leq i \leq n$

k(i) es la nueva posición del elemento i-ésimo después de la operación de división

b) Ordenación por mezcla

P_1: $a_1, \ldots, a_{n/2}$

P_2: $a_{n/2+1}, \ldots, a_n$

La resolución de los problemas P_1 y P_2, conlleva la obtención inmediata de la solución R_0, para el caso a). En el caso b), se hace necesaria además una operación de combinación.

Una de las razones que hace que este tipo de algoritmos sea tan eficiente en algunos casos, es su rapidez en la ramificación (división) del problema. Lo único que se ha de determinar es el pivote que divide el espacio de exploración. En la mayoría de los casos este índice es el punto medio de la estructura, caso a), con lo que se consigue una distribución equilibrada del trabajo entre los subproblemas. En el caso b), la forma en la que trabaja el algoritmo provoca la reorganización de los conjuntos generados y no asegura una división equilibrada.

2.2.2. La operación de combinación (*combine*).

Debido al esquema de trabajo de esta técnica, una vez resueltos los subproblemas generados se hace necesario la utilización de una nueva función que recibiendo como entrada los subproblemas resueltos devuelva el problema original resuelto: las soluciones R_i de los problemas parciales $P_1, \ldots, P_k$ se deben combinar en el cómputo de la solución R_0. La operación de combinación *combine* que se ha de realizar debe aprovechar la optimalidad de las soluciones R_i de los subproblemas P_i. En general, existe una relación inversamente proporcional entre las dificultades asociadas a las operaciones de división y combinación.

[1] A pesar de que en los dos ejemplos indicados, la división produce siempre dos problemas, es posible realizar una descomposición en un número superior de problemas de tamaño menor. Es más, la aplicación del paralelismo a este tipo de estrategia recomienda la creación de más trabajo independiente como se verá.

En los algoritmos de ordenación, las operaciones asociadas son:

a) Ordenación por mezcla.

$R_0 = (x, \text{combine}(R_1', R_2'))$

$R_1 = (a_1, \ldots, a_k)$
$R_2 = (b_1, \ldots, b_m)$

si $a_1 \leq b_1 \Rightarrow x = a_1, R_1' = (a_2, \ldots, a_k)$
si $a_1 > b_1 \Rightarrow x = b_1, R_2' = (b_2, \ldots, b_m)$

b) Ordenación rápida.

$R_0 = \text{combine}(R_1, R_2) = (R_1, R_2)$

2.2.3. Funciones de parada (*small*).

Debido al esquema eminentemente recursivo, debe existir una función que asegure la finalización de la operación de división. Esta función *small*, deberá discernir sobre la conveniencia o no de la división del problema. En caso negativo, debe solucionar el problema por otro tipo de método *solve* que no necesite descomposición (en el caso más sencillo, el problema puede estar compuesto por un único elemento).

2.2.4. Procedimiento Divide y Vencerás Generalizado.

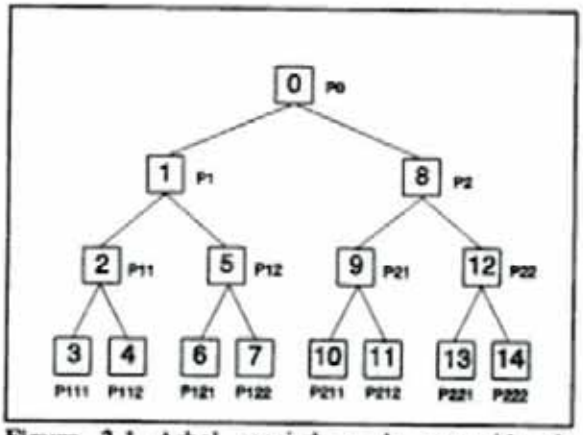

Figura 2.1 Arbol asociado a la operación de división.

Los elementos que hasta el momento se han definido o utilizado, asociados con el esquema divide y vencerás son:

- Una operación de división (*divide*).
- Una operación de combinación (*combine*).
- Una función de parada (*small*).

Si se representa gráficamente la forma de trabajo de esta técnica, la aplicación de la operación de división a todos los problemas posibles genera una estructura de árbol de división. Para los ejemplos se obtiene un árbol como el de la figura 2.1. Asociado a cada nodo del árbol se tiene un problema P_i. En la raíz se encuentra el problema inicial P_0. Las aristas indican implícitamente la operación de división realizada en la fase descendente (*top-down*), a la que se incorpora en el recorrido ascendente (*bottom-up*), la operación de combinación.

Es conveniente reseñar que no es necesario finalizar todas las operaciones de división

antes de comenzar las de combinación. La principal ventaja del recorrido en anchura en la operación de división se presenta en el diseño de los algoritmos paralelos. Sin embargo, la obligación de explorar todo el árbol de búsqueda, conlleva una ganancia nula en la utilización de cualquier estrategia de elección del problema a ramificar para el caso secuencial. Es bueno recordar que el árbol correspondiente a la técnica divide y vencerás es un árbol del tipo AND, esto es donde la solución a cada problema se obtiene de las soluciones de todos los hijos.

```
tipo:    tree = record
           { union-struct for the problem or the solution }
           solve : boolean;
           next : pointer to tree;
           father : pointer to tree;
         end;
         heap = struct of pointer to tree;

Datos:   var heap_div, heap_comb : heap; Tree : tree;

Método:  Procedure GeneralDivideAndConquer;
         var j : 1..k;
             decision : tag;
         begin
           heap_div := heap_comb := ϕ;
           create_node(P0, nil); insert_div(P0);
           while (not_empty(heap_div) or not_empty(heap_comb)) do begin
             decision := select_operation();
             if (decision = combine) then begin
               P := select_comb(heap_comb);
               combine(P^.next); P^.solve := TRUE;
               if solved_son(P^.father) then insert_comb(P^.father);
             end { if ... }
             else begin { decision = divide }
               P := select_div(heap_div);
               if small(P) then begin
                 solve(P); P^.solve := TRUE;
                 if solved_son(P^.father) then insert_comb(P^.father);
               end { if ... }
               else begin
                 divide(P, P1, P2, ..., Pk);
                 for j := 1 to k do begin
                   create_node(Pj, P); insert_div(Pj);
                 end; { for ... }
               end; { else }
             end; { else }
           end; { while ... }
         end; { GeneralDivideAndConquer }
```

Código 2.3 Divide y vencerás genérico.

En el caso del recorrido en profundidad, un esquema recursivo es adecuado para su exposición, o su equivalente iterativo que utiliza una estructura de datos tipo Pila (*LIFO*). Por el contrario, para la exploración en anchura se hace necesaria una estructura tipo Cola (*FIFO*). En el código 2.3 se muestra un esquema general, donde se engloban los dos casos anteriores. Para mantener unificada la notación se introducen tres nuevas operaciones, *select_operation*, *select_div* y *select_comb*, que se utilizan para decidir el tipo de operación y las estrategias de elección del problema. Estas dos últimas funciones actúan sobre las

estructuras respectivas (*heap_div* y *heap_comb*). Esta formulación relaja la estructura de orden que se mostró en la versión recursiva inicial de la técnica divide y vencerás. El pseudocódigo expresa el funcionamiento de esta técnica. En este tipo de estrategia es de vital importancia la estructura de datos utilizada para el almacenamiento de los diferentes problemas que se van generando a lo largo de la ejecución. Debido a la similitud con los árboles que tiene el método divide y vencerás, se emplea una estructura de este tipo (*Tree*). Para mantener enlazados los subproblemas creados en la fase de división de un problema, se dispone de dos enlaces, *next* y *father*, para conectarlos entre sí y conectarse con su padre. Estos punteros son asignados cuando se realiza la división del problema. Estos vínculos permiten, de forma sencilla, analizar cuando un problema determinado puede pasar a engrosar la estructura reservada para los problemas que se pueden combinar, aquellos cuyos hijos ya están resueltos y en la zona asignada han guardado su solución.

La función *solved_son* se encarga de controlar cuando un problema esta listo para ser tratado en la fase de combinación. Mediante un análisis del árbol que tiene al problema como raíz, en esta función se pueden ir liberando las zonas asignadas a los nietos de este problema, si existen. Las tres funciones auxiliares *insert_div*, *insert_comb* y *create_node* se utilizan para introducir trabajos en las estructuras respectivas.

2.3. PARALELIZACION DE LA TECNICA DIVIDE Y VENCERAS.

Partiendo del procedimiento divide y vencerás del apartado anterior, son posibles dos tipos de paralelización:

La primera de ellas no afecta al tipo de operación de división ó combinación a realizar, y se asocia con el paralelismo de *grano grueso* [chi90], [kim92], [kin88]. De la estructura de división que contiene los problemas que no han sido ramificados, se pueden seleccionar tantos problemas parciales P_i como procesadores estén disponibles, por ejemplo, p. Del mismo modo, este sistema se puede utilizar para la elección concurrente de problemas de la estructura de combinación (obsérvese que los problemas incluidos en esta segunda estructura disponen de todos sus datos de entrada).

Sin embargo en algunos problemas, tanto las operaciones de división como de combinación aceptan un paralelismo intrínseco (*grano fino*), la realización de la operación correspondiente se puede llevar a cabo de forma concurrente.

2.3.1. Procedimiento de partida.

Para implementar el primer esquema, los datos requeridos por los procesadores se deben almacenar en una estructura compartida global. El algoritmo paralelo resultante selecciona concurrentemente p problemas para su descomposición y/o combinación. En este esquema, cada procesador m, $1 \leq m \leq p$, ejecuta el código 2.4.

```
Método: Procedure ConcurrentCentralizedD&C(m);
        var j : 1..k;
            not_end : boolean;
            tag : signal;
        begin
          not_end := TRUE;
          while (not_end) do begin
            send_request(m); { solicite work }
            tag := receive_signal;
            if tag = finish then not_end := FALSE
            else begin
              P := receive_prob_and_son;
              if tag = prob_comb then begin
                combine(P^.next); P^.solve := TRUE;
                send_for_comb(P, P^.father);
              end { if ... }
              else { tag = prob_div } begin
                if small(P) then begin
                  solve(P); P^.solve := TRUE;
                  send_for_comb(P, P^.father);
                end { if ... }
                else begin
                  divide(P, P1, P2, ..., Pk);
                  for j := 1 to k do begin
                    create_node(Pj); send_for_div(Pj, P);
                  end; { for ... }
                end; { else }
              end; { else }
            end; { else }
          end; { while ... }
        end; { ConcurrentCentralizedD&C(m) }
```

Código 2.4 Código de un obrero en un divide y vencerás centralizado.

Para prevenir conflictos entre los procesadores es necesario mantener las estructuras compartidas, los *heaps* y *Tree* dentro de *secciones críticas* (en cada instante de tiempo, a lo sumo un procesador tiene acceso al *heap* correspondiente para la selección de un problema). Dentro de las máquinas paralelas con memoria compartida, algunas disponen de dispositivos internos para manejar este acceso simultáneo de varios procesadores a una misma estructura de datos. Tanto en algunas máquinas de memoria compartida como en las de memoria distribuida es necesario que el usuario se haga cargo de la manipulación exclusiva de estas estructuras. El pseudocódigo que se presenta se adapta a este último caso (código 2.5).

Cada procesador mediante el envío y recepción de mensajes, se comunica con el proceso administrador de los problemas. Un protocolo específico de señales (*signals*) indica a cada uno de los procesadores obreros el tratamiento a realizar con el trabajo asignado. Manteniendo en mente la estructura definida para el caso secuencial, cada obrero divide un problema en subproblemas y envía cada uno de ellos al administrador ó combina los subproblemas recibidos del administrador y responde con la solución del problema padre. Las nuevas funciones *send_for_comb* y *send_for_div* se encargan de la comunicación de estos resultados y de la liberación final de las zonas reservadas para su cómputo. Una hipótesis muy realista a tener en cuenta, es la imposibilidad de que un mensaje enviado después que otro, sea recibido antes por el administrador (código 2.5).

```
Datos:  var heap_div, heap_comb : heap; Tree : tree;
            idle_proc : struct of processor;
Método: Procedure CentralizedAdmin;
        var proc : 1..p;
            tag : signal;
            not_end : boolean;
            P, Pfather : pointer;
        begin
          heap_div := heap_comb := ϕ; not_end := TRUE;
          create_node(P0); P0->father = nil;
          while (not_end) do begin
            tag = receive_signal(proc);
            if tag = request then begin
              receive_and_insert(idle_proc, proc);
              if idle_all_proc(idle_proc) && empty(heap_div)
                 && empty(heap_comb) then begin
                not_end := FALSE;
                for proc := 1 to p do send_signal(finish, proc);
              end { if ... }
              else if not_empty(heap_comb) OR not_empty(heap_div) then begin
                proc := select_proc(idle_proc);
                if not_empty(heap_comb) then begin
                  P := select_comb(heap_comb);
                  send_signal(prob_comb, proc);
                end { if ... }
                else { not_empty(heap_div) } begin
                  P := select_div(heap_div);
                  send_signal(prob_div, proc);
                end; { else }
                send_prob_and_son(P, proc);
              end; { else }
            end { if ... }
            else if tag = prob_comb then begin
              receive_for_comb(P, Pfather);
              organizate_tree_comb(P, Pfather);
              if solved_son(Pfather) then begin
                insert_comb(Pfather);
                if not_empty(idle_proc) then begin
                  proc := select_proc(idle_proc);
                  send_signal(prob_comb, proc);
                  P := select_comb(heap_comb);
                  send_prob_and_son(P, proc);
                end; { if ... }
              end; { if ... }
            end { else if ... }
            else { tag = prob_div } begin { insert on heap_div }
              receive_for_div(P, Pfather);
              organizate_tree_div(P, Pfather);
              insert_div(P);
              if not_empty(idle_proc) then begin
                proc := select_proc(idle_proc);
                send_signal(prob_div, proc);
                P := select_div(heap_div);
                send_prob_and_son(P, proc);
              end; { if ... }
            end { else }
          end; { while ... }
        end; { CentralizedAdmin }
```

Código 2.5 Proceso administrador de un divide y vencerás centralizado.

El proceso administrador se mantiene en un bucle a la espera de la llegada de resultados y solicitudes de parte de los procesadores, trabajo que realizan las diferentes funciones *receive*. En reciprocidad las funciones *send* envían los mensajes correspondientes. Las funciones *organize_tree* se encargan de administrar la estructura árbol utilizada (*Tree*), manejando los problemas que se reciben de los obreros.

El modo de trabajo empleado, permite que este proceso administrador lleve el control, en todo momento, del trabajo y los procesadores que permanecen ociosos (estos últimos son almacenados en una estructura específica, *idle_proc*). Por lo tanto, el administrador es también el responsable de la fase de finalización, cuando en la estructura de procesadores ociosos se encuentran todos los obreros y los *heaps* están vacíos.

2.3.2. Ejecución concurrente eficiente.

Si $t_p(P_0)$ denota el tiempo de cálculo necesario para resolver un determinado problema P_0 con p procesadores, es de esperar en el mejor caso que:

$$t_p(P_0) \approx t_1(P_0)/p$$

donde t_1 es el tiempo requerido para el procedimiento secuencial en un procesador. Sin embargo, en las situaciones reales se produce un empeoramiento y se verifica la relación $t_p(P_0) > t_1(P_0)/p$. Este comportamiento se atribuye a dos razones; la primera, es general a cualquier tipo de arquitectura paralela utilizada y a cualquier tipo de implementación realizada. La segunda depende del tipo de máquina que se use.

1.- No es fácil mantener, en todo momento, el tamaño de las estructuras de datos correspondientes lo suficientemente grande para que todos los procesadores puedan encontrar inmediatamente el siguiente problema parcial a seleccionar. Este problema aumenta si el número de procesadores disponibles se incrementa. En particular en la técnica divide y vencerás, hay que tener en cuenta que la activación inicial de todos los procesadores se ve retrasada por el modo de trabajo de la misma, etapa donde el número de procesadores es superior al número de problemas a investigar (al comenzar la ejecución sólo se dispone de un problema asignado a un procesador y los restantes deben esperar, mientras éste divide el problema). Esta fase del algoritmo se mantiene durante $\log_k p$ etapas, supuesto que cada problema se divide en k subproblemas. Lo mismo ocurre, esta vez en sentido contrario, en la fase de combinación. Este hecho conlleva unas aceleraciones limitadas por estas etapas donde el número de problemas a resolver no cubre el número de procesadores disponibles [qui94].

2.- Se necesita intercambiar información entre procesadores y entre memoria global y local. Se introduce entonces, la *sobrecarga de las comunicaciones*. Es de gran importancia en este caso, desarrollar arquitecturas que tiendan a minimizar el tiempo que se gasta en comunicaciones, y diseñar algoritmos adaptados a tales arquitecturas.

2.3.3. Implementación de divide y vencerás centralizado.

Tal y como se ha comentado brevemente en un apartado anterior y que ha servido de base para el diseño de un código paralelo general para la estrategia divide y vencerás centralizada, es necesario distinguir el tipo de máquina subyacente donde se implementará esta técnica.

Si se dispone de una máquina paralela con memoria compartida (multiprocesador), todos los procesadores tienen acceso a la memoria global. Para regular el acceso a las estructuras globales [iba87], se debe desarrollar una estructura o código que asegure su uso como sección crítica. Una posibilidad es la construcción de una cola (*FIFO*), donde se guarden todos los procesadores que están preparados para la ejecución de este paso. Así, sólo el primer procesador puede iniciar la ejecución del paso. Cuando se completa, el procesador abandona la cola.

En el caso de un multicomputador (máquina con memoria distribuida y paso de mensajes), la centralización de los datos en uno de los procesadores, de ahí el nombre de *divide y vencerás centralizado*, provocará el uso de un proceso especial (administrador o maestro), que controle el acceso a los *heaps*. Los procesadores deben intercomunicar con él, para la distribución y recolección de problemas y resultados. El uso exclusivo por parte del maestro de los datos, asegura su consistencia. Los restantes procesos (obreros), se limitan a la resolución de los problemas asignados. Este fuerte trasiego de problemas y resultados, puede producir embotellamientos en las comunicaciones. Sólo si el tiempo de cómputo de los problemas parciales asignados es superior al tiempo de transferencia de estos, se pueden evitar estos atascos. A este tipo de estrategia, se le conoce con el nombre de *proceso granja* (*farming process*), y a ella se hará referencia posteriormente, en los experimentos realizados tanto en esta técnica como en la de ramificación y acotación.

2.3.4. Divide y vencerás jerarquizado.

Debido a las dependencias recurrentes que existen entre ciertos problemas (la solución de un problema parcial P_i, implica el cómputo de todo el árbol que lo tiene como nodo raíz), y la independencia respecto de otros (el espacio del problema P_i, es independiente de los espacios de sus problemas hermanos), es admisible distribuir el trabajo entre los procesadores, manteniendo las relaciones respecto a la operación de combinación. El esquema que surge se puede denominar *divide y vencerás jerarquizado*, debido a la jerarquía existente en el modo de funcionamiento.

2.3.4.1. Implementación del divide y vencerás jerarquizado.

En principio, la independencia tolerada obliga a disponer de tantos grupos de estructuras (*heaps* y *Tree*) como procesadores disponga la máquina paralela y almacenarlos

en cada uno de los procesadores. La fase de inicialización, suponiendo que se poseen p procesadores, podría ser la siguiente:

$$\begin{array}{ll} heap\_div_1 = P_0, & heap\_comb_1 = \phi \\ heap\_div_2 = ¿?, & heap\_comb_2 = \phi \\ \\ heap\_div_p = ¿?, & heap\_comb_p = \phi \end{array}$$

Se supone que el proceso jerárquico se inicia en el procesador uno, que almacena el problema original. Nada se asegura sobre la carga de los restantes procesadores, aunque normalmente comienzan vacíos y en espera de algún problema. El procedimiento que ejecuta cada procesador m, $1 \leq m \leq p$, es el que se muestra en el código 2.6.

Obsérvese que a la estructura de datos utilizada con anterioridad, se le ha añadido un nuevo campo *assign_proc*, que se utiliza para reconocer en la fase de combinación el procesador propietario del problema. De este modo se puede comunicar con él, siguiendo el camino de vuelta correcto en el árbol inducido por la estrategia divide y vencerás.

A pesar de que los *heaps* y la estructura *Tree* son locales a cada procesador, se ha optado por indicar en cada uno de ellos el procesador propietario para facilitar la legibilidad del código.

En el esquema que se muestra, cada procesador repite un bucle donde una vez recibido un problema para dividir, anota quien es el propietario del problema padre y se convierte en el dueño de este. A partir de aquí es el encargado de realizar las subdivisiones sobre el problema, si estas tuvieran que llevarse a cabo, y mediante una función de selección *select_heap* asigna los nuevos subproblemas a diferentes procesadores, los cuales repiten de nuevo el proceso en esta primera fase. En la etapa de combinación, un procesador que resuelve un problema consulta si sigue siendo el propietario del problema padre, mediante una simple comparación con el campo *assign_proc*. Si resultara falsa esta hipótesis, envía su solución al procesador propietario, *assign_proc*, y libera la zona reservada al asegurar que ha finalizado su cometido en este árbol. En caso contrario, al igual que en el algoritmo secuencial, procede a estudiar si el problema ancestro esta en condiciones de pasar a su fase de combinación.

No se han tenido en cuenta los procesos necesarios para el tratamiento de los interbloqueos que esta estrategia genera, para no aumentar y complicar aún más el algoritmo introducido. No obstante, la búsqueda de la mayor asincronía posible provoca la utilización de estos procesos extra.

La finalización del algoritmo, sólo puede ser detectada por el procesador uno (elegido como propietario del problema inicial), Cuando éste advierte que se ha resuelto el problema inicial, debe comunicar a los demás procesadores que terminen su ejecución. El método de trabajo de la estrategia divide y vencerás asegura que todos los restantes procesadores están en estado de espera.

```
tipo    tree = record
          { union-struct for the problem or the solution }
          solve : boolean;
          next, father : pointer to tree;
          assign_proc : 1..p;
        end;

Método: Procedure ConcurrentHierarchicalD&C(m);
        var heap_div_m, heap_comb_m : heap; Tree_m : tree;
            P, Pfather : pointer to tree; father : 1..p;
            j : 1..k; not_end : boolean; tag : signal;
        begin
          if (m = 1) then begin
            heap_div_m := P; Pfather := nil; father := m;
          end { if ... }
          else heap_div_m := receive_div(P, Pfather, father);
          heap_comb_m := Tree_m := φ; not_end := TRUE;
          create_node(P, Pfather, father); insert_div(P);
          while not_end do begin
            if exist_message then begin
              tag := receive_signal;
              if tag = finish then begin
                not_end := FALSE; send_finish_to_sons;
              end { if ... }
              else if tag = prob_comb then begin
                receive_result(P, Pfather);
                organizate_tree_comb(P, Pfather);
                if solved_son(Pfather) then
                  insert_comb(Pfather, heap_comb_m);
                else if (m = 1 and solve_total_prob) then begin
                  not_end := FALSE; send_finish_to_sons;
                end; { else if ... }
              end { else if ... }
              else { tag = prob_div } begin
                receive_div(P, Pfather, father);
                create_node(P, Pfather, father); insert_div(P);
              end { else }
            end { if ... }
            else { not_exist_message } begin
              if not_empty(heap_comb_m) then begin
                P := select_comb(heap_comb_m);
                combine(P^.next); P^.solve := TRUE;
                if P^.father^.assign_proc = m then
                  if solved_son(P^.father) then
                    insert_comb(P^.father, heap_comb_m);
                  else { P^.father^.assign_proc <> m }
                    send_result(P, P^.father, heap_comb_assign_proc);
              end { if ... }
              else { not_empty(heap_div) } begin
                P := select_div(heap_div_m);
                if small(P) then begin
                  solve(P); P^.solve := TRUE;
                  if P^.father^.assign_proc = m then
                    if solved_son(P^.father) then
                      insert_comb(P^.father, heap_comb_m);
                    else { P^.father^.assign_proc <> m }
                      send_result(P, P^.father, heap_comb_assign_proc);
                end { if ... }
                else begin
                  divide(P, P_1, P_2, ..., P_k);
                  for j := 1 to k do begin
                    create_node(P_j, P, m);
                    send_div(P_j, P, m, heap_div_select_heap);
                    { send_div = insert_div if select_heap = m }
                  end; { for ... }
                end; { else }
              end; { else }
            end; { else }
          end; { while ... }
        end; { ConcurrentHierarchicalD&C(m) }
```

Código 2.6 Esquema divide y vencerás jerarquizado.

2.3.4.2. Estrategia eficiente de balance de carga. Minimización de los tiempos de espera (*idle times*).

En ocasiones los problemas resolubles mediante esta técnica permiten el cálculo a priori de la dificultad del problema (tamaño de los problemas parciales generados). No parece entonces muy difícil, elegir una ramificación adecuada que garantice la uniformidad de, al menos, los tamaños de los subproblemas así creados.

Brinch Hansen [bri94] realiza un estudio detallado de los tiempos de ejecución de dos versiones de un algoritmo paralelo basado en el método de ordenación rápida, debido a Hoare [hoa62]. En la primera versión se realiza un balance entre los tamaños de los problemas generados. Los resultados indican una ganancia promedio del 30% respecto a la versión no balanceada, a pesar de la introducción de un factor temporal adicional para la obtención de la uniformidad.

2.3.5. Paralelización de las operaciones de división y combinación.

En la mayoría de los casos, las dos operaciones básicas que componen esta estrategia se pueden computar de forma concurrente. En los códigos 2.7 y 2.8, aprovechando los dos algoritmos de ordenación presentados, se muestran dos códigos paralelos en lenguaje II [leo91] basados en sus versiones secuenciales (por simplicidad, en ambos se supone que los elementos a ordenar son todos distintos). En el primero, la fase de división se realiza de forma paralela, mientras que en el segundo, es la de combinación la que aprovecha la disponibilidad de otros procesadores en la máquina. Este tipo de algoritmos puede ser de utilidad en máquinas síncronas de memoria compartida, donde existe un alto grado de conexión entre procesadores y memoria.

Lee y otros [lee88] utilizan esta técnica en la resolución de un Problema de la Mochila 0-1 sobre un hipercubo de transputers. El cálculo de la operación de combinación en cada vértice activo de éste, se distribuye entre los vecinos. Este algoritmo se ha implementado y se muestra en el capítulo dedicado a la programación dinámica por su relación con ella.

2.3.6. Técnicas intermedias.

Jacquemin y Griffiths [jac90] presentan un esquema distribuido para la ordenación rápida. Apoyándose en una topología de *anillo bidireccional*, cada procesador ocioso recoge del *heap*, que viaja a través del anillo, un problema para su ramificación. De los dos problemas generados, uno pasa a engrosar el *heap*, mientras que el otro se resuelve del mismo modo (uno de los hijos se vuelve a enviar al *heap*). Los autores aseguran haber obtenido eficiencias cercanas al 90%. Nosotros hemos realizado la implementación del esquema propuesto, obteniendo rendimientos muy inferiores a estos. El *heap* distribuido, que

se mantiene siempre viajando, provoca el retraso de la resolución del problema capturado por cada procesador. La sincronización necesaria en la operación de combinación, se elimina debido al tipo de problema tratado, pues consiste simplemente en la union de resultados independientes.

Brinch Hansen [bri91] propone una metodología para el divide y vencerás paralelo, más que desarrollar un algoritmo paralelo concreto para un determinado problema. Utilizando una topología de *árbol binario,* cada uno de los procesadores del mismo nivel del árbol recoge el trabajo asignado de su padre, le aplica la operación de ramificación y envía los subproblemas creados a sus hijos. La sincronización necesaria en la operación de combinación provoca que los procesadores intermedios y el raíz permanezcan ociosos, en espera de los resultados. Como ya se ha comentado, el artículo muestra la importancia de la generación de problemas equilibrados mediante una serie de resultados experimentales. Otros autores [cha93], [li93], [qui88], [rot85], [tod78], [whe92] aprovechan esta misma metodología para analizar diferentes algoritmos de ordenación paralelos basados en los códigos de ordenación rápida y por fusión, y que serán tema de estudio en los resultados.

```
SHARED PROCEDURE Merge(left, right, middle : SHARED index);
BEGIN
  PARALLEL left..right DO BEGIN
    VAR low, high, i : index; x : item;

    RELAX
      IF NAME <= middle THEN BEGIN
        low := middle+1; high := right;
      END { IF ... }
      ELSE BEGIN
        low := left; high := middle;
      END; { ELSE }

      x := a[NAME];
      REPEAT
        index := (low+high) DIV 2;
        IF x < a[i] THEN high := i-1
        ELSE { x > a[i] } low := i+1;
      UNTIL low > high;

    a[high + (NAME-middle)] := x;
  END; { PARALLEL ... }
END; { Merge }

SHARED PROCEDURE ParMergeSort(left, right : index);
VAR i : index;
BEGIN
  IF (right-left <= small) THEN
    Sort(left, right) { May be the QuickSort or the MergeSort ... }
  ELSE BEGIN
    i := (left+right) div 2;
    PARALLEL DO
      ParMergeSort(left, i) || ParMergeSort(i+1, right);
    Merge(left, right, i);
  END; { ELSE }
END; { ParMergeSort }
```

Código 2.7 Implementación en 11 del algoritmo de ordenación por mezcla.

```
sum, mark : ARRAY [..] OF index;

SHARED PROCEDURE Partition(left, right : index ; VAR PivPos : index);
VAR numMarked, oldPivotPos : index;

SHARED PROCEDURE PrefixSum(i, j : index);
VAR k : index;
BEGIN
  IF i < j THEN
  BEGIN
    PARALLEL DO
      PrefixSum(i, (i+j) DIV 2) || PrefixSum(((i+j) DIV 2)+1, j);
    k := (i+j) DIV 2;
    PARALLEL k+1..j DO
      sum[NAME] := sum[NAME] + sum[k];
  END
  ELSE { i = j } sum[i] := mark[i]
END; { PrefixSum }

BEGIN
  PARALLEL left..right DO
    IF a[NAME] < a[PivPos] THEN mark[NAME] := 1
    ELSE mark[NAME] := 0;

  PrefixSum(left, right);

  oldpivpos := pivpos;
  nummarked := sum[right];

  PARALLEL left..right DO
  VAR pos : index;
  BEGIN
    IF NAME = pivpos THEN
    BEGIN
      pivpos := nummarked + left; { New position of pivot }
      posn := pivpos;
    END
    ELSE IF mark[NAME] = 1 THEN
      pos := sum[NAME] + left-1
    ELSE
    BEGIN
      pos := NAME + nummarked - sum[NAME];
      IF NAME < oldpivpos THEN pos := pos+1
    END;
    a[pos] := a[NAME]; { All changes }
  END  { PARALLEL }
END; { Partition }

SHARED PROCEDURE ParQuickSort(left, right : index);
VAR pivpos : index;
BEGIN
  IF left < right THEN
  BEGIN
    pivpos := (left+right ) DIV 2;
    Partition(left, right, pivpos);
    PARALLEL DO
      ParQuickSort(left, pivpos-1) || ParQuickSort(pivpos+1, right);
  END; { IF ... }
END; { ParQuickSort }
```

Código 2.8 Implementación en 11 del algoritmo de ordenación rápida.

En los mismos términos se pronuncian P.G. Clayton y otros [cla94]. En su artículo proponen un esqueleto para la estrategia divide y vencerás paralela basada, de nuevo, en un árbol binario como caso base al que se puede trasladar cualquier otro tipo de árbol. La técnica utilizada asegura que todos los procesadores físicos resuelven uno de los subproblemas creados mediante un código secuencial y algunos de ellos realizan antes y después las fases de división y combinación.

2.3.7. Medidas admisibles.

Como no es posible la reducción del espacio a explorar en este tipo de estrategia, las medidas deben actuar sobre el tiempo de ejecución y en ningún caso tiene sentido sobre otro parámetro. Las más utilizadas son, el tiempo de ejecución total, la aceleración y la eficiencia.

2.4. APLICACIONES.

Como ya se ha mencionado uno de los mejores algoritmos de ordenación que emplea la técnica divide y vencerás es el de *ordenación rápida* (*quicksort*). Su filosofía es bien conocida y es un punto de partida adecuado para contrastar los diferentes esquemas paralelos generales que se pueden utilizar [akl85], [bri91], [che84], [eva85], etc.

A lo largo de esta sección se presentan algoritmos basados en los dos esquemas paralelos anteriormente expuestos (centralizado y jerarquizado). Para mantener siempre el mismo orden, primero se profundizará en el caso centralizado y a continuación se abordará el caso jerarquizado.

La topologías de *árbol binario* se ha elegido como soporte para las implementaciones de los diferentes códigos. Sus características se han comentado en el capítulo de introducción y se ha elegido en este caso, por su similitud con el árbol de búsqueda que se crea al utilizar la estrategia divide y vencerás.

En los experimentos se han elegido redes de cuatro tamaños diferentes, árboles de altura 2 (3 procesadores), 3 (7), 4 (15) y 5 (31). (Ver figura 1.8).

Los problemas elegidos se han generado aleatoriamente con valores en el rango [0,99]. Se han elegido cuatro tamaños [32768, 65536, 131072, 262144] y para cada uno de ellos se han utilizado cinco problemas. Las medidas que se toman son siempre la media entre estos cinco problemas. Se incluyen para todos los algoritmos dos medidas, el tiempo en segundos y la aceleración.

2.4.1. Una implementación mediante una granja de procesadores.

El esquema inicial de trabajo en máquinas paralelas con memoria compartida

propugna la ejecución de códigos idénticos en cada procesador sobre problemas diferentes [dem82], [qui94]. Cada uno de ellos, recoge un subproblema y lo divide, combina o resuelve. En el caso de no disponer de una memoria global, ésta se puede simular de una manera trivial en uno de los procesadores. Los restantes le solicitan tareas y le envían otras (o resultados). Como ya se ha comentado, este tipo de paralelización es conocido con el nombre de *granja de procesadores*.

Un procesador especial, denominado *maestro*, se encarga de almacenar los datos y los resultados. Los restantes procesadores, *obreros*, realizan la división, resolución y combinación de los problemas generados. En general, en cada obrero debe ejecutarse un proceso *aplicación*, encargado de dividir, combinar ó solucionar el problema recibido. Los restantes procesos (*ruteros*) que se ejecutan concurrentemente se usan para el tratamiento de los mensajes y aseguran la existencia de un camino entre cualquier obrero y el maestro (grafo conexo).

El número de procesos para comunicaciones dependerá de la topología escogida. Para intentar minimizar la fase de generación de tantos problemas como procesadores se disponga y la de recolección de resultados, estos procesos se ejecutan con una prioridad mayor que el específico de cada tipo de problema (aplicación). Siguiendo una notación similar al lenguaje Occam2, el código sería el siguiente (código 2.9):

```
priority parallel
  parallel
    router_out1;
    router_in1;
    ...
    router_outN;
    router_inN;
    admin;
  application;
```

Código 2.9 Estructura de procesos a ejecutar en cada procesador.

Se deben asociar a cada enlace *(link)* físico de un *transputer* dos ruteros: uno para la entrada de mensajes y el otro para la salida. La ausencia de elementos de sincronización y la falta de conocimiento, a priori, del número de mensajes en la red, impone la creación de un proceso *administrador*, que debe evitar la producción de condiciones de interbloqueo entre los procesadores de la red (hay que asegurar que los mensajes de entrada a un procesador siempre son atendidos y que los de salida son enviados cuando el receptor esté a la espera del mismo), y que se ejecuta concurrentemente con los procesos ruteros. En la topología de árbol binario, se tendrán seis ruteros para los tres enlaces de conexión (la mitad de entrada y la otra mitad de salida) (código 2.10).

```
procedure rout_in(in, out :
                  channel);
begin
  while not(finish) do
  begin
    in ? case
      { receive by in }
      { send by out }
  end; { while ... }
end; { rout_in }
```

```
procedure rout_out(out, in_ad,
                   out_ad : channel);
begin
  while not(finish) do begin
    out_ad ! ready
    in_ad ? case
      { receive by in_ad }
      { send by out }
  end; { while ... }
end; { rout_out }
```

Código 2.10 Ruteros de entrada y salida.

Cada rutero dispone obviamente de un enlace con el proceso administrador (de salida hacia el administrador, para los de entrada y viceversa). Adicionalmente, los encargados del envío de los mensajes hacia otros transputers (ruteros de salida), usan otro enlace en el que comunican al administrador que están dispuestos a recibir trabajo para su posterior traslado (su ausencia puede provocar interbloqueos entre los procesadores). En el código 2.2 se presentan los algoritmos asociados a cada tipo de rutero.

El proceso administrador debe estar dispuesto a atender a todos los ruteros (se les sirve trabajo a los ruteros de salida cuando este trabajo exista y ellos estén dispuesto a recibirlo, los de entrada siempre que tengan mensajes deben ser atendidos). El administrador debe seguir el esquema del código 2.11.

El proceso aplicación dependerá del problema tratado. En este caso estará compuesto por un código similar al proceso de partición del método de ordenación rápido iterativo dado en [hor78] ó [wir76]. El procedimiento divide la lista en dos partes y envía una de ellas al maestro. La otra sublista se continúa particionando hasta alcanzar un tamaño prefijado para su resolución. Una vez alcanzado este tamaño se resuelve siguiendo el mismo método de ordenación (*quicksort*).

```
procedure admin(  in_rout_in1, ...,   in_rout_inN,   -- channels rout_in
                 in_rout_out1, ...,  in_rout_outN,   -- channels rout_out
                out_rout_out1, ..., out_rout_outN,   -- channels rout_out
                     in_appl, out_appl);             -- channels appl
begin
  while not(finish) do begin
    alternative
      in_appl ? case
        { receive by in_appl }
        { modify if_for_out? }

      if_for_out1 & in_rout_out_1 ? case ready
        { send by out_rout_out1 }

      ...

      if_for_outN & in_rout_out_N ? case ready
        { send by out_rout_outN }

      in_rout_in1 ? case
        { receive by in_rout_in1 }
        { modify if_for_out1 if neccesary }
        { otherwise send by out_appl }
      ...

        in_rout_inN ? case
          { receive by in_rout_inN }
          { modify if_for_outN if neccesary }
          { otherwise send by out_appl }
  end; { while ... }
end; { admin }
```

Código 2.11 Proceso administrador.

Resultados.

La mayoría de los algoritmos del tipo divide y vencerás resuelven el problema más rápidamente si los subproblemas generados tienen un tamaño similar, esto es, se realiza una división equilibrada del problema. Sin embargo el procedimiento encargado de esta división equitativa, en el caso de la ordenación rápida puede ser excesivamente lento [bri91]. Como en el esquema de granja de procesadores se trata de minimizar esta etapa de división generando el máximo número de subproblemas lo antes posible y en la ejecución paralela también hay que tener en cuenta el tiempo de comunicaciones, se opta por ensayar esta última estrategia: particionar el problema (no necesariamente por la mitad), enviar el más pequeño al maestro, y continuar la división del más grande hasta un tamaño prefijado (en el cual se pasa a resolver).

Aprovechando las ideas utilizadas en los algoritmos de ramificación y acotación paralela para el caso centralizado (capítulo tres), se opta por minimizar los recorridos de los problemas que se deben comunicar. El procesador maestro lleva el control de los problemas que aún quedan por explorar pero no contiene el problema en sí, sino un identificador del mismo (es decir, el obrero en cuya estructura está almacenado). La solicitud de trabajo por parte de un procesador provoca la búsqueda dentro de la estructura global que contiene el maestro (sólo almacena descriptores de problemas): si el procesador que ha pedido tiene trabajo en su organización, se le comunica que puede comenzar a resolverlo y se anota que es un problema procesado, en otro caso, mediante una función se busca el procesador más cercano que disponga de trabajo y se le comunica que realice el envío del mismo al procesador que lo solicitó y de nuevo se anota este problema como procesado. Esta última optimización no ha sido llevada a cabo, ya que el diámetro de la red utilizada no es lo suficientemente grande. Los resultados obtenidos se muestran en las tablas 2.1 y 2.2. En la primera de ellas (tabla 2.1) se indica el promedio de tiempo en segundos para los cinco problemas de cada tamaño elegido y la aceleración obtenida. Para analizar el número de problemas (tabla 2.2) se utilizan como medidas la media del número total de problemas resueltos sin ulterior división por los obreros para cada uno de los tamaños elegidos, así como el promedio que resuelve cada obrero dependiendo del número de procesadores utilizados.

Tabla 2.1 Resultados temporales para el esquema en granja de procesadores (FARM).

FARM	1 PROC.	3 PROC.		7 PROC.		15 PROC.		31 PROC.	
TAM.	TIEMPO	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.
32768	3.94	3.01	1.31	1.87	2.10	1.62	2.43	1.62	2.44
65536	8.27	6.20	1.33	3.73	2.22	3.19	2.59	3.21	2.58
131072	17.68	13.36	1.32	8.14	2.17	7.06	2.51	7.05	2.51
262144	36.48	26.90	1.36	15.99	2.28	13.43	2.72	12.96	2.81

Tabla 2.2 Número de problemas resueltos secuencialmente.

FARM	3 PROC.		7 PROC.		15 PROC.		31 PROC.	
TAM.	N. TOT	PROM	N. TOT	PROM	N. TOT	PROM	N. TOT	PROM
32768	12	6.00	27	4.50	54	3.86	107	3.57
65536	10	5.00	27	4.50	55	3.93	106	3.53
131072	12	6.00	27	4.50	55	3.93	107	3.57
262144	11	5.50	27	4.50	53	3.79	107	3.57
MEDIA	11	5.50	27	4.50	54	3.86	107	3.57

Se ha elegido como *tope de división* de un problema, el siguiente valor *TamañoInicial/(2 * NúmeroProcesadores)* y como cota mínima para comunicar un problema al maestro *TopeDivisión/4*. Si se observan las columnas de la aceleración, es evidente que no se detecta un aumento continuo si se incrementa el tamaño del problema, del mismo modo parece obvio que una altura de nivel 3 en el árbol parece ser un tope, en el cual comienza a suponer más tiempo la comunicación de los problemas al maestro que su resolución. Sin embargo, el número de problemas resueltos por cada obrero se mantiene constante para un mismo árbol aunque se multiplique por ocho el tamaño de la lista a ordenar, desde 32768 a 262144.

2.4.2. Una distribución del maestro a lo largo de la red.

En el esquema centralizado anterior nada impide distribuir los problemas entre los diferentes procesadores, haciendo que cada uno de ellos se encargue del almacenamiento y resolución de una cierta cantidad de aquellos. sin embargo es obvio que este sistema dependerá mucho del procedimiento utilizado en la división del trabajo, tanto por su tiempo de cómputo como por su eficacia en la generación de problemas de tamaño equilibrado. Un esquema más ambiguo y que es utilizado por Jacquemin y Griffiths [jac90] consiste en utilizar la propia topología como soporte para el almacenamiento de los problemas aún sin resolver, estos viajan a lo largo de la red (que por supuesto debe ser elegida de manera que forme un grafo conexo). Los procesadores desocupados capturan trabajo cuando éste pasa por ellos y en su resolución crean nuevos problemas que añaden al que ya se encuentra viajando. Los autores eligen como ejemplo de utilización de su método el algoritmo del *quicksort* debido a su adaptabilidad para evitar emplear complicados procesos que se encarguen de detectar la finalización del proceso (fase de combinación), basta con llevar la cuenta en un procesador especial de los elementos ya ordenados que han llegado hasta él. Para aprovechar los enlaces físicos que disponen los transputers la topología elegida es un *anillo doble* donde los problemas viajan siempre en una dirección y los resultados en la contraria. Cada procesador de la red, salvo el especial, usa dos procesos ruteros de entrada y otros dos de salida. El procesador especial no necesita rutero de salida de resultados. Si bien los autores aseguran una eficiencia cercana al 90% para un anillo de 4 procesadores,

las pruebas realizadas siguiendo estas pautas indican serios problemas en su implementación debido al hecho de que los ruteros que se ejecutan a alta prioridad retrasan la realización de las ordenaciones de forma variable, esto es, dependiente de la ejecución.

```
procedure node(to_up, from_up                    -- channels with father.
               to_left, from_left                -- channels with left son.
               to_right, from_right);            -- channels with right son.
begin
  from_up ? P;                                   -- read_prob() for root.
  split(P in P1 and P2);
  parallel
    to_left ! P1;
    to_right ! P2;
  parallel
    from_left ? R1;
    from_right ? R2;
  combine(R1 and R2 in R);
  to_up ! R;                                     -- nothing for root.
end; { node }
```

Código 2.12 Código asociado a los nodos intermedios y la raíz de un árbol.

```
procedure leaf(to_up, from_up);                  -- channels with father.
begin
  from_up ? P;
  R := solve(P);
  to_up ! R;
end; { leaf }
```

Código 2.13 Código asociado a las hojas de un árbol.

2.4.3. Implementación de un esquema jerarquizado.

La traslación directa del esquema de trabajo de este tipo de algoritmos a una máquina paralela del tipo MIMD, induce la utilización de una topología de árbol (en nuestro caso binario) [pet81]. La fase de división de un problema en un determinado procesador, introduce la activación de los procesadores hijos en la red asociada. En este sentido Brinch Hansen [bri91] propone un algoritmo de tipo divide y vencerás genérico para una máquina árbol. Cada procesador, excepto las hojas, recibe el problema a resolver de su padre y envía a sus hijos los subproblemas resultantes. En la etapa de combinación el proceso se intercambia, se reciben los subproblemas resueltos de sus descendientes, se combinan y se devuelve el problema finalizado a su ancestro. Este tipo de metodología introduce una jerarquía obvia en la ejecución de los algoritmos: en la fase de división y suponiendo un costo equivalente para la partición de los problemas asignados a los procesadores de cada nivel, estos son activados por niveles y en ningún momento se encuentran trabajando procesadores de distinto nivel. Cuando el trabajo llega a las hojas estas resuelven los subproblemas asociados y comienza la fase de combinación. En cualquier fase si se explora un camino desde la raíz a una cualquiera de las hojas sólo uno de los nodos intermedios se encuentra trabajando mientras que los demás permanecen ociosos. En los códigos 2.12 y 2.13 se muestran los algoritmos asociados a los tres tipos de nodos del árbol, raíz (*root*), nodo intermedio (*node*)

y hoja (*leaf*).

Debido al esquema totalmente simétrico de la topología utilizada, el balance de la carga entre los hijos se convierte en un factor extremadamente importante tanto en la utilización equitativa de todos y cada uno de los procesadores como en la disminución del tiempo de ejecución. La mayoría de los algoritmos paralelos creados a partir del método de ordenación rápida utilizan el esquema anterior. La única diferencia estriba en el método elegido para particionar los problemas (*split*) ó combinar los resultados (*combine*). Para la evaluación de los diferentes métodos se utilizará tanto los resultados experimentales (tiempo experimentado) como un análisis teórico del tiempo promedio de ejecución de cada uno de ellos (tiempo estimado). Para cada algoritmo se contrastarán los cálculos teóricos con los resultados obtenidos experimentalmente. Como introducción se especifican las notaciones necesarias para estos cálculos teóricos.

n	=	tamaño de la entrada.
p	=	número de procesadores totales.
q	=	número de procesadores hojas del árbol = $(p+1)/2$.
$T_{sort}(1, n)$	=	tiempo de ejecución del algoritmo secuencial.
T_{comu}	=	tiempo de comunicaciones del algoritmo paralelo.
T_{comp}	=	tiempo de cómputo (partición y/o combinación) del algoritmo paralelo.
$T_{sort}(p, n)$	=	tiempo de ejecución del algoritmo paralelo con p procesadores.

El tiempo de un algoritmo paralelo se define como la suma de tres operandos.

$$T_{sort}(p,n) = T_{sort}(1, n/q) + T_{comu} + T_{comp}$$

Los dos primeros sumandos se mantienen fijos para todos los algoritmos, el tercero es el único variable. El tiempo del algoritmo secuencial se ha calculado como el producto de una constante dependiente del procesador por la complejidad teórica promedio del algoritmo secuencial $O(n\log_2 n)$.

$$T_{sort}(1, n/q) = a_{sort} n/q \log_2(n/q)$$

Supongase que *b* denota a la constante asociada a las comunicaciones, entonces el tiempo de éstas resulta el siguiente:

$$T_{comu} = 2b(n/2 + n/4 + \ldots + n/q) = 2bn(q-1)/q$$

Brinch Hansen propone un procedimiento de partición que asegura la división del problema en dos subproblemas de igual tamaño, se trata de repetir el proceso de intercambio (*find*) hasta que los índices de recorrido de la lista superen por encima y por debajo, respectivamente, el elemento medio (código 2.14). La fase de combinación (*combine*) en los procesadores intermedios (*nodes*) se reduce a una simple concatenación de dos listas, mientras que la etapa de partición (*split*) es la que conlleva el mayor esfuerzo computacional.

Si se denota por a_{BH} (*(B)rinch (H)ansen*) la constante relacionada, el tiempo de cálculo esperado debe ser:

$$T_{comp}(BH) = a_{BH}[n + n/2 + \ldots + n/(q/2)] = a_{BH}2n(q-1)/q$$

```
procedure find(first, last, middle);
var left, right, i, j;
begin
  left := first; right := last;
  while left < right do begin
    partition(i, j, left, right);
    if middle <= j then right := j
    else if i <= middle then left := i
    else left := right;
  end; { while ... }
end; { find }
```

Código 2.14 Partición equilibrada (*split*) de una lista.

En el esquema más simple de ordenación la lista se divide en sendas sublistas de igual tamaño (la mitad de tamaño del de partida) que se envían a los procesadores hijos. En la etapa de combinación se mezclan las sublistas por pares (cada procesador no hoja recibe sus dos sublistas ordenadas, las mezcla y envía a su padre). El proceso finaliza cuando se obtiene la total en el procesador raíz. Este método que se basa en el algoritmo secuencial de la ordenación por mezcla (*mergesort*) fue propuesto por Wagar en una topología de hipercubo [wag86]. Si bien la fase de división, obviamente se reduce a la partición de la lista de entrada, la etapa de combinación conlleva el mayor gasto computacional al estar formada por la mezcla (*merge*) de dos listas ordenadas (código 2.15). Si se identifica como a_{SM} (*(S)imple (M)erge*) a la constante correspondiente, el tiempo de cómputo es el siguiente:

$$T_{comp}(SM) = a_{SM}[n + n/2 + \ldots + n/(q/2)] = a_{SM}2n(q-1)/q$$

```
procedure merge(Al, Ar, At, DIM)
var i, j, k;
begin
  i := j := k := 0;
  while k < DIM do begin
    if Al[i] <= Ar[j] then begin
      At[k] := Al[i]; i := i+1;
    end { if ... }
    else { Al[i] > Ar[j] } begin
      At[k] := Ar[j]; j := j+1;
    end; { else }
    k := k+1;
  end; { while ... }
end; { merge }
```

Código 2.15 Mezcla de dos listas ordenadas.

Un último esquema con el mismo orden de complejidad que los anteriores pero que no asegura la división balanceada de los problemas, es el basado en la aplicación del procedimiento secuencial de partición de una lista. Este método lo utiliza Won [won89] en

una red hipercúbica. Para intentar acercarse a una división equitativa del problema, se seleccionan aleatoriamente una serie de elementos de la lista y se escoge la mediana de todos ellos como elemento pivote. El número de elementos a elegir es de vital importancia en este algoritmo y se debe determinar de forma experimental (muy pocos elementos pueden provocar la elección de un pivote no adecuado, mientras que el exceso de ellos aumenta el tiempo de cálculo innecesariamente). Suponiendo que a_{KS} (*(K)ey (S)elect*) es la constante prefijada, n_{KS} es el tanto por ciento de elementos elegidos, y que se realiza una distribución equivalente de los elementos de la lista entre las dos de salida, el tiempo de cómputo coincide con el algoritmo anterior añadiendo el número de elementos utilizados para este cálculo por la constante respectiva a_{nec}:

$$\begin{aligned} T_{comp}(KS) &= a_{KS}(n + n/2 + \ldots + n/(q/2)) + \\ &+ a_{nec}(n \cdot n_{KS} + n \cdot (2n_{KS}) + \ldots n \cdot ((q/2)n_{KS})) = \\ &= 2n(q-1)/q[a_{KS}+a_{nec} \cdot n_{KS}] \end{aligned}$$

Sin embargo, es muy difícil obtener esta división idéntica, de ahí que haya que introducir un factor adicional δ_i que almacene la desviación producida respecto al punto medio ideal.

$$T_{comp}(KS) = 2n(q-1)/q[a_{KS}+a_{nec} \cdot n_{KS}] + a_{KS} \sum_{i=1}^{\log q - 1} \delta_i$$

Las constantes definidas b, a_{BH}, a_{SM}, a_{KS} y a_{nec} dependen de la arquitectura utilizada y mediante estudios experimentales le serán asignados valores reales asociados al tipo de máquina paralela que se use. Sin embargo, se puede aseverar un orden entre las constantes intrínsecas a cada método (a_{BH}, a_{SM} y a_{KS}): Debido a la naturaleza del problema en el último caso el valor comprende el incremento de un contador y en algunos casos el intercambio entre dos valores, mientras que en el algoritmo basado en mezclas este costo computacional se traduce en la variación en dos variables contador y una asignación, lo cual conlleva a priori un mayor gasto. La única ventaja de este algoritmo es que asegura una división equilibrada, lo cual no se puede afirmar en el anterior, sin un esfuerzo excesivo. Para el caso de la partición idéntica, el número de veces que se debe recorrer la lista puede alcanzar en el peor caso un factor dos veces mayor, de ahí que sea la mayor constante de todas.

Resultados.

Para poder contrastar los resultados obtenidos experimentalmente frente a los cómputos teóricos estimados es necesario dar valores numéricos a las diferentes constantes asociadas a la máquina y a los algoritmos. Todos los valores se han computado experimentalmente, obteniéndose los siguientes datos numéricos:

$$\begin{aligned} a_{com} &= 7.73\ \mu s,\ b = 7.00\ \mu s,\ a_{SM} = 6.23\ \mu s, \\ a_{BH} &= 13.09\ \mu s,\ a_{KS} = 4.40\ \mu s,\ a_{nec} = 23.44\ \mu s. \end{aligned}$$

Las tablas contemplan los resultados teóricos esperados (EST) y los resultados obtenidos experimentalmente (EXP). Hay que especificar que para el algoritmo de selección (*Key Select*) se ensayan diferentes porcentajes de n_{KS}, dos concretamente, ambos dependientes de los tamaños de los problemas a resolver, y se ha supuesto en los resultados teóricos que el factor de desequilibrio es despreciable ($\delta_i=0$). Sin embargo, en este caso se da como medida el factor de desequilibrio producido en media (FAC). Este se ha obtenido a partir de la diferencia en valor absoluto entre el pivote y la media del array de entrada para cada uno de los procesadores. Estos valores se han tipificado para equiparar las unidades de medida. Se han tomado como valores n_{KS}, un 0.5% (N/200) y un 0.2% (N/500) de los elementos de entrada a cada partición.

En el algoritmo de la división equitativa (tabla 2.3), los resultados experimentales parecen mantenerse por encima de los estimados. Sin embargo, el incremento en el número de procesadores parece que aproxima ambos valores. El contraste de las aceleraciones indica también un acercamiento cuando se incrementa el número de procesadores, aunque las diferencias para los menores tamaños siempre favorecen, contrariamente, a la experimentación.

Tabla 2.3 Resultados para el esquema de división equilibrada.

BI	1 PROC.		3 PROC.				7 PROC.				15 PROC.				31 PROC.			
TAM.	TEXP	TEST	TEXP	TEST	AEXP	AEST	TEXP	TEST	AEXP	AEST	TEXP	TEST	AEXP	AEST	TEXP	TEST	AEXP	AEST
32768	3.94	3.80	2.41	2.43	1.63	1.56	1.73	1.81	2.23	2.10	1.46	1.53	2.71	2.48	1.31	1.41	3.00	2.70
65536	8.23	8.11	4.99	5.12	1.60	1.58	3.68	3.75	2.24	2.16	3.02	3.13	2.74	2.59	2.73	2.85	3.03	2.85
131072	17.68	17.22	10.74	10.74	1.65	1.60	7.86	7.75	2.25	2.22	6.43	6.38	2.75	2.70	5.77	5.76	3.07	2.99
262144	36.48	36.47	22.71	22.49	1.61	1.62	16.34	16.01	2.23	2.28	13.31	13.02	2.74	2.80	11.87	11.65	3.07	3.13

Para el caso del algoritmo de mezcla (tabla 2.4), los valores experimentales tienden a equipararse a las estimaciones desde casi el primer momento. En las aceleraciones, los valores reales casi siempre mejoran a los conjeturados, aunque esta ganancia es mínima e incluso inexistente.

Tabla 2.4 Resultados para el esquema de mezcla.

SM	3 PROC				7 PROC				15 PROC				31 PROC			
TAM.	TEXP	TEST	AEXP	AEST	TEXP	TEST	AEXP	AEST	TEXP	TEST	AEXP	AEST	TEXP	TEST	AEXP	AEST
32768	2.28	2.21	1.73	1.72	1.52	1.47	2.59	2.58	1.17	1.14	3.38	3.34	1.00	0.99	3.96	3.85
65536	4.76	4.67	1.74	1.74	3.13	3.07	2.64	2.64	2.37	2.34	3.49	3.46	2.01	2.01	4.11	4.04
131072	9.86	9.84	1.79	1.75	6.47	6.40	2.73	2.69	4.84	4.81	3.65	3.58	4.07	4.07	4.35	4.23
262144	20.80	20.69	1.75	1.76	13.34	13.31	2.73	2.74	9.90	9.87	3.69	3.70	8.26	8.28	4.42	4.41

El análisis (tablas 2.5 y 2.6) indica ventajas mínimas para los valores computados frente a los teóricos, sobre todo al incrementar el número de procesadores. Las aceleraciones reales son siempre mejores que las estimadas. Los porcentajes promedio de desviación respecto al valor medio se mantienen en la mayoría de los casos por debajo del 5%, lo cual corrobora la hipótesis de desechar de la estimación el desequilibrio de la carga.

Tabla 2.5 Resultados para el esquema de selección con n_{ES} de N/200.

ES-200	3 PROC.					7 PROC.					15 PROC.					31 PROC.				
TAM.	TEXP	TEST	AEXP	AEST	FAC	TEXP	TEST	AEXP	AEST	FAC	TEXP	TEST	AEXP	AEST	FAC	TEXP	TEST	AEXP	AEST	FAC
32768	2.20	2.15	1.79	1.77	1.22	1.42	1.39	2.77	2.75	1.55	1.05	1.04	3.75	3.65	2.29	0.87	0.85	4.52	4.31	4.21
65536	4.64	4.55	1.78	1.78	1.10	2.91	2.91	2.84	2.79	1.08	2.14	2.14	3.86	3.78	2.71	1.78	1.80	4.64	4.52	4.74
131072	9.68	9.62	1.83	1.79	1.15	6.06	6.06	2.92	2.84	0.97	4.41	4.41	4.01	3.90	2.54	3.65	3.65	4.85	4.71	3.85
262144	19.98	20.24	1.83	1.80	1.05	12.45	12.63	2.93	2.89	1.02	9.06	9.08	4.05	4.02	2.60	7.39	7.43	4.94	4.91	3.32

Tabla 2.6 Resultados para el esquema de selección con n_{ES} de N/500.

ES-500	3 PROC.					7 PROC.					15 PROC.					31 PROC.				
TAM.	TEXP	TEST	AEXP	AEST	FAC	TEXP	TEST	AEXP	AEST	FAC	TEXP	TEST	AEXP	AEST	FAC	TEXP	TEST	AEXP	AEST	FAC
32768	2.21	2.15	1.78	1.77	1.59	1.43	1.39	2.75	2.75	2.48	1.07	1.04	3.68	3.65	4.41	0.91	0.88	4.34	4.31	10.54
65536	4.67	4.55	1.77	1.78	1.45	2.95	2.91	2.81	2.79	1.93	2.17	2.14	3.81	3.78	2.72	1.81	1.80	4.57	4.52	5.81
131072	9.68	9.62	1.83	1.79	1.15	6.05	6.06	2.92	2.84	0.97	4.40	4.41	4.02	3.90	2.75	3.62	3.65	4.88	4.71	4.97
262144	19.97	20.24	1.83	1.80	1.06	12.44	12.63	2.93	2.89	1.02	9.04	9.08	4.04	4.02	2.63	7.36	7.43	4.95	4.91	3.32

La comparación de los tres algoritmos, tanto mediante los experimentos como las estimaciones, indica que el mejor comportamiento corresponde al algoritmo basado en la selección, seguido del basado en las mezclas, quedando como peor algoritmo, la división equitativa. Este es un resultado predecible mediante las constantes asociadas a cada método. El mayor desajuste se produce en los valores de la parte superior derecha de las tablas (tamaños pequeños con gran número de procesadores). En este caso, los experimentos muestran resultados más satisfactorios que las conjeturas. Este hecho es imputable a que el factor de estimación de la ordenación secuencial no se comporta de manera tan correcta para tamaños pequeños.

2.4.4. Maximizando el número de procesadores que resuelven problemas.

La mayoría de los algoritmos creados utilizan como topología subyacente una red hipercúbica [var92], [wag86]. La ventaja del hipercubo respecto a la máquina árbol radica en que se hace uso de todos los procesadores disponibles en la etapa de resolución de los problemas parciales. En un cierto instante todos y cada uno de los procesadores se convierten en hojas del árbol de computación. El tiempo asociado a comunicaciones es como mucho

similar al realizado en el caso anterior (obsérvese que tanto en la etapa de división como en la de combinación uno de los mensajes tiene como destino el mismo procesador), con lo cual parece recomendable hacer uso de este tipo de red. Sin embargo un inconveniente imputable a este tipo de topología, es que este tipo de estructura exige un aumento lineal del número de conexiones distintas existentes para cada procesador cuando se incrementa el grado del hipercubo (relación uno por nodo).

Para el caso del problema que nos ocupa, la ordenación, este impedimento no se puede obviar. Recuérdese que en el caso de los transputers sólo se dispone de cuatro enlaces físicos de conexión reconfigurables y cualquier simulación de grado superior introduce un efecto negativo en el tiempo de comunicaciones así como una dificultad adicional en la complejidad del algoritmo utilizado (es cierto que existen máquinas construidas con una topología hipercúbica de grado superior a cuatro, sin embargo no están disponibles de forma generalizada debido a su alto coste).

¿Cómo se puede entonces intentar maximizar el número de procesadores utilizados, manteniendo el grado de conexión entre ellos dentro de un rango razonable y el código utilizado no excesivamente complejo?. La respuesta obvia es que debe ser una topología que simule fácilmente el comportamiento de los algoritmos sobre la máquina árbol. Todos aquellos que jugaban el papel de raíz o nodo intermedio en la máquina anterior deben cargar con la ejecución de al menos un proceso como hoja y uno o más como nodo intermedio.

Si bien en los algoritmos donde la fase de división realiza cambios sobre la lista de elementos a ordenar, las modificaciones pueden provocar retraso en la expansión de los problemas, no ocurre lo mismo para el código basado en la mezcla. Un parámetro experimental debe discernir cual es el tamaño ideal del problema que se asigna al procesador que realiza la división, es decir, en la fase de división el problema de entrada se divide en tres subproblemas, dos de ellos de igual tamaño (que serán enviados a los dos hijos) y un tercero que resolverá este procesador. En este punto, es necesario recalcular los tiempos estimados de forma teórica. Con esta restricción el tiempo de comunicación en un árbol, se reduce a:

$$T_{comm} = 2bn/p[1 + 3 + 7 + \ldots + (p-1)/2] = 2bn/p\left(\sum_{i=1}^{\log_2((p+1)/2)} (2^i-1)\right)$$
$$= 2bn/p[p - \log_2(p+1)] = 2bn[1 - \log_2(p+1)/p]$$

Del mismo modo se obtiene una reducción en el tiempo de ordenación de cada procesador, que pasa a ser $T_{son}(1, n/p) = a_{son}n/p\log_2(n/p)$.

El tiempo de computación también se ve afectado debido al hecho de que pasan a intervenir tres listas y, por lo tanto, varía el tamaño a controlar en la combinación.

$$T_{comp} = a_{SM}'n/p[3 + 7 + \ldots + p] = a_{SM}'n/p\left(\sum_{i=2}^{\log_2((p+1)/2)+1} (2^i-1)\right) =$$
$$= a_{SM}'n/p[2(p-(1/2)) - \log_2(p+1)]$$

Resultados para el algoritmo de mezcla usando todos los procesadores.

A pesar de que en el caso teórico se ha supuesto que los tamaños de los problemas a resolver son iguales, en la práctica se ha ensayado también variar el tamaño a resolver en cada procesador atendiendo al nivel del árbol donde se encuentre (a mayor nivel, menor tamaño). Experimentalmente se ha obtenido la siguiente regla: Si el número de procesadores supera un límite determinado (en las pruebas mayor o igual a 15), el primer procesador resuelve un problema de tamaño doble al de los demás. El motivo de sólo aplicarlo a los árboles mayores, es que la ganancia en los pequeños queda anulada por tener que resolver secuencialmente este problema tan grande en la raíz. Esta implementación se introduce para mostrar que la ganancia que se puede obtener mediante este esquema se convierte rápidamente en mala debido al factor de mezcla. Las tablas 2.7 y 2.8 muestran los resultados obtenidos en ambos casos. En la estimación, el parámetro a_{SM}' es fuertemente dependiente de los diferentes tamaños de entrada posible. Se ha elegido que tome el valor 9.50 μs como promedio entre todos. También se ha introducido un factor adicional de sobrecarga en el parámetro de las comunicaciones, ya que en la experimentación se ejecutan en paralelo los dos procesos de envío y recepción a los hijos junto al proceso que resuelve el problema local (b = 7.50 μs).

Tabla 2.7 Resultados para el esquema de mezcla generalizada con división equilibrada.

SM'	3 PROC				7 PROC				15 PROC				31 PROC			
TAM.	TEXP	TEST	AEXP	AEST	TEXP	TEST	AEXP	AEST	TEXP	TEST	AEXP	AEST	TEXP	TEST	AEXP	AEST
32768	1.71	1.61	2.30	2.36	1.24	1.17	3.18	3.26	1.11	1.07	3.54	3.56	1.09	1.06	3.61	3.60
65536	3.59	3.38	2.30	2.39	2.53	2.41	3.26	3.37	2.24	2.17	3.70	3.74	2.18	2.13	3.80	3.81
131072	7.41	7.11	2.39	2.42	5.16	4.96	3.43	3.48	4.50	4.40	3.93	3.91	4.35	4.29	4.07	4.01
262144	15.34	14.89	2.38	2.45	10.51	10.20	3.47	3.58	9.12	8.94	4.00	4.08	8.74	8.65	4.18	4.22

Tabla 2.8 Resultados para el esquema de mezcla generalizada con división modificada.

SM'	15 PROC				31 PROC			
TAM.	TEXP	TEST	AEXP	AEST	TEXP	TEST	AEXP	AEST
32768	1.07	1.07	3.69	3.56	1.07	1.06	3.69	3.60
65536	2.15	2.17	3.85	3.74	2.13	2.13	3.89	3.81
131072	4.32	4.40	4.09	3.91	4.26	4.29	4.15	4.01
262144	8.73	8.94	4.18	4.08	8.56	8.65	4.26	4.22

La característica principal de este esquema es que sólo tiene un buen comportamiento para redes pequeñas, si se comparan los resultados con los algoritmos jerárquicos se observa que aquellos obtienen mejores resultados a la larga. En general, si se dividiera el problema en un número mayor de subproblemas, la mejora se anularía con la fase de combinación.

```
procedure TreeFarming(fr_up, to_up, fr_left, to_left,
                      fr_right, to_right, fr_applic, to_applic,
                      fr_gest_div, to_gest_div,
                      fr_gest_comb, to_gest_comb : channel);
var QueueProb, QueueSmallProb : queueproblems; QueueResult : queueresults;
    Prob : problem; Result, ... : result;
begin
  initialize(QueueProb);
  initialize(QueueSmallProb);
  initialize(QueueResult);
  while not_end do begin
    fr_gest_div ? Tag;
      TagPet    : if not_empty(QueueProb) then begin
                    Prob := select_prob(QueueProb);
                    to_gest_div ! TagProb; Prob;
                  end { if ... }
                  else update(idle_gest_div);
      TagProb   : fr_gest_div ? Prob;
                  if for_himself(Prob) then begin
                    saveTProb, QueueSmallProb);
                    if idle_applic then begin
                      Prob := select_prob(QueueSmallProb);
                      to_applic ! TagSmallProb; Prob; update(idle_applic);
                    end; { if ... }
                  end { if ... }
                  else begin
                    to_left; TagProb; Prob; ó
                    to_right; TagProb; Prob;
                  end; { else }
    fr_gest_comb ? Tag;
      TagPet    : if not_empty(QueueResult) then begin
                    select_results;
                    to_gest_comb ! TagResult; Results;
                  end { if ... }
                  else update(idle_gest_comb);
      TagResult : fr_gest_comb ? Result; to_up ! TagResult; Result;
    fr_applic ? Tag;
      TagPet    : if not_empty(QueueSmallProb) then begin
                    Prob := select_prob(QueueSmallProb);
                    to_applic ! TagSmallProb; Prob;
                  end { if ... }
                  else update(idle_applic);
      TagResult : fr_applic ? Result; save(Result, QueueResult);
                  if idle_gest_comb then begin
                    select_results;
                    to_gest_comb ! TagResult; Results;
                    update(idle_gest_comb);
                  end; { if ... }
    fr_up ? Tag;
      TagFinish : update(not_end); to_applic ! TagFinish;
                  to_gest_div ! TagFinish; to_gest_comb ! TagFinish;
      TagProb   : fr_up ? Prob; save(Prob, QueueProb);
                  if idle_gest_div then begin
                    Prob := select_prob(QueueProb);
                    to_gest_div ! TagProb; Prob;
                    update(idle_gest_div);
                  end; { if ... }
    fr_left ? Tag;
      TagResult : fr_left ? Result; save(Result, QueueResult);
                  if idle_gest_comb then begin
                    select_results;
                    to_gest_comb ! TagResult; Results;
                    update(idle_gest_comb);
                  end; { if ... }
    fr_right ? Tag;
      TagResult : fr_right ? Result; save(Result, QueueResult);
                  if idle_gest_comb then begin
                    select_results;
                    to_gest_comb ! TagResult; Results;
                    update(idle_gest_comb);
                  end; { if ... }
  end; { while ... }
  parallel to_left ! TagFinish; to_right ! TagFinish;
end; { TreeFarming }
```

Código 2.16 Proceso control en cada nodo de la granja árbol.

2.4.5. Un compromiso entre jerarquía, utilización de procesadores y aceleración inicial.

La búsqueda de códigos que sean fáciles de implementar en topologías con un factor de escalabilidad alto, esto es, cuyo crecimiento sea realizable con la tecnología actual (ejemplo árboles) y que aprovechen al máximo todos y cada uno de los procesadores disponibles, induce aplicar una serie de mejoras al esquema centralizado de partida. El esquema centralizado tiene como uno de sus inconvenientes, el retardo que produce la generación de un número de problemas mayor que el número de procesadores de la red. En los algoritmos jerarquizados este problema se intentaba minimizar haciendo que la generación de los mismos fuera lo más simple posible, bien usando una partición de lista o bien una división simple. En el primer caso, no se podía asegurar la distribución equitativa de trabajo. En el caso de la división simple, el equilibrio estaba garantizado. En ambos casos, todos los procesadores superados en el árbol quedaban desaprovechados. El esquema que se propone pretende ser un compromiso entre la jerarquía inherente a los algoritmos, un aprovechamiento mayor de los procesadores y una aceleración inicial del mismo orden.

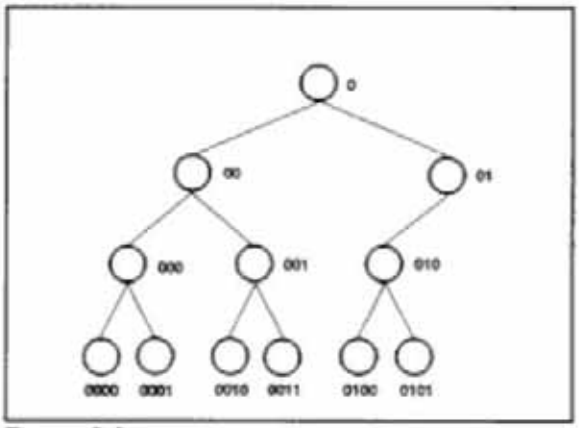

Figura 2.2 Esquema y numeración de problemas.

Este esquema se puede denominar granja árbol, por su procedencia inicial. De nuevo se implementa un divide y vencerás en un árbol de procesadores. Un pseudocódigo posible para el procedimiento que controla los procesos asignados a cada procesador intermedio se muestra en el código 2.16. En cada procesador se ejecutan, además de los tres procesos ruteros, un proceso *gest_div* que gestiona la división de los problemas, un proceso *gest_comb* que se encarga de la combinación y un proceso *applic* para la resolución de los problemas asignados al procesador.

La topología considerada es la de árbol k-ario. Por similitud con los esquemas anteriores se supone que el problema se divide según un árbol binario, aunque el método es válido para cualquier otro tipo de árbol. Una posible identificación de cada problema sería la utilización de su codificación binaria (k-aria), en la que se tiene en cuenta su profundidad en el árbol y la codificación de su padre, así como el número de hermanos que tiene (figura 2.2). Esta nueva forma de identificar los problemas es una alternativa a la utilizada en el divide y vencerás jerárquico introducido anteriormente, que se aprovecha de la red subyacente para evitar tener que especificar de modo explícito toda la estructura árbol que induce la técnica.

Para concretar el método es necesario definir los procedimientos *gestor_divide* y *gestor_combine* que resuelven los problemas y los resultados almacenados en los *buffers* *QueueProb* y *QueueResult*, respectivamente.

```
procedure gestor_divide(fr_buffer, to_buffer : channel);
var QueueLocProb : queueproblems; Prob, ProbOut... : problem;
    ChargeLeft, ChargeRight : word;
begin
  initialize(QueueLocProb); ChargeLeft := ChargeRight := 0;
  while not_end do begin
    to_buffer ! TagPet;
    fr_buffer ? Tag;
      TagFinish : update(not_end);
      TagProb   : fr_buffer ? Prob;
                  { QueueLocProc is empty }
                  save(Prob, QueueLocProb);
                  while not_empty(QueueLocProb) do begin
                    Prob := select_prob(QueueLocProb);
                    if small(Prob) then
                      to_buffer; TagProb; Prob { for applic }
                    else begin
                      divide(Prob in ProbOut...);
                      compare_charge_son_and_send_if_necessary;
                      update_queue; update_charge;
                    end; { else }
                  end; { while ... }
  end; { while ... }
end; { gestor_divide }
```

Código 2.17 Esquema de división (*gestor_divide*).

```
fr_buffer ? Prob
if small(Prob) then
  to_buffer; TagProb; Prob { for applic }
else begin
  fin := FALSE;
  while not(fin) do begin
    partition(Prob in Prob1 and Prob2);
    { Code for:
        size(Prob1) ≤ size(Prob2) and ChargeLeft ≤ ChargeRight }
    if (ChargeLeft + size(Prob2)) <= (ChargeRight + size(Prob1)) then
    begin
      to_buffer ! TagProb; Prob1; { for Right }
      to_buffer ! TagProb; Prob2; { for Left }
      ChargeRight := ChargeRight + size(Prob1);
      ChargeLeft := ChargeLeft + size(Prob2);
      fin := TRUE;
    end { if ... }
    else begin
      to_buffer ! TagProb; Prob1 { for Left }
      Chargeleft := ChargeLeft + size(Prob1);
      Prob := Prob2;
    end; { else }
    ...
  end; { while ... }
end; { ... }
```

Código 2.18 Esquema de división *gestor_divide* para un algoritmo de ordenación.

Cada procesador no hoja recibe trabajo de su ancestro y lo guarda en su *buffer* de problemas (*QueueProb*). El proceso *gestor_divide* recibe problemas de este *buffer*. Una vez que se ha recibido este problema *Prob*, se procede a computar su tamaño (se supone definida una función *size(Prob)* que mide con exactitud el costo computacional del problema *Prob*).

Si el problema es *pequeño*, *small(Prob)* lo resuelve (en este caso, pasa a engrosar la estructura *QueueSmallProb* de problemas listos para resolver). Con el fin de equilibrar la carga enviada a los subárboles, se almacena en todo momento el trabajo enviado a cada uno de ellos (no resuelto) y se procede en función de la búsqueda de un equilibrio entre las cargas de los dos hijos. La rutina utilizada se muestra en el código 2.17.

En el caso particular de un algoritmo de ordenación como el *quicksort*, el esquema general se convierte en el mostrado en el código 2.18. No es necesario la utilización de la cola *QueueLocProb*, puesto que en cada momento se mantiene a lo sumo un problema sin explorar.

```
procedure gestor_combine(fr_buffer, to_buffer : channel);
var QueueLocResult : queueresults; Result, ResultIn... : result;
begin
  initialize(QueueLocResult);
  while not_end do begin
    to_buffer ! TagPet;
    fr_buffer ? Tag;
      TagFinish : update(not_end);
      TagResult : fr_buffer ? Result;
                  save(Result, QueueLocResult);
                  while (seguir) do begin
                    if not_local(father(Result) then begin
                      to_buffer ! TagResult; Result
                      update(seguir);
                    end { if ... }
                    else if solved_sons(father(Result)) then
                    begin
                      combine(sons(father(Result));
                      Result := father(Result);
                    end; { else if ... }
                  end; { while ... }
  end; { while ... }
end; { gestor_combine }
```

Código 2.19 Esquema de combinación (*gestor_combine*).

El proceso encargado de gestionar las combinaciones, *gestor_combine*, recibe resultados con identificadores de problema. A partir de los identificadores determina el padre del problema y el número de hijos. Si están resueltos todos los subproblemas hijos y este es el último por el que se esperaba, se realiza la operación de combinación. Como consecuencia de la resolución de este problema, otro que espera en la cola *QueueResult* puede resultar resuelto. Como el otro gestor, continua trabajando hasta que se vacía su cola de subproblemas. El esquema se muestra en el código 2.19.

Si bien es obvio, que con esta política no es posible alcanzar los tiempos obtenidos con una división simple no deja de ser cierto que se mejora el factor obtenido con los esquemas basados en particiones, manteniendo la puerta abierta (mayor posibilidad cuanto más se profundice en el árbol) a que los procesadores intermedios puedan resolver más de un problema, límite obtenido con los algoritmos jerárquicos.

En el esquema presentado son evidentes diversas mejoras, sin más que desacoplar del

proceso administrador todas aquellas comunicaciones que se pueden hacer directamente entre otros procesos. En la figura 2.3, se muestra un gráfico con las posibles optimizaciones a nivel de enlaces entre procesos. Sin embargo, se ha optado por mantener un único proceso central que controla todos los mensajes por respetar el esquema usado en las restantes estrategias y una mayor legibilidad.

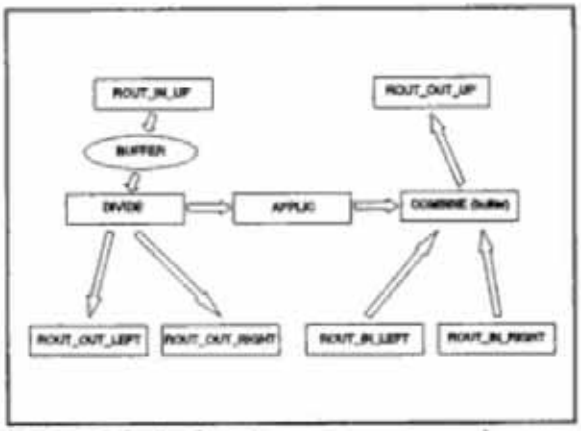

Figura 2.3 Gráfico de procesos y conexiones en cada procesador.

Resultados.

La variable a controlar en este caso, es el tamaño máximo de problema resoluble en cada procesador mediante un algoritmo de ordenación rápida secuencial. Este parámetro varía con la profundidad del nodo y el número de procesadores disponibles en la red. Este mismo valor es utilizado como diferencia máxima admisible entre las cargas asignadas a cada procesador. Se han llevado a cabo dos tipos de pruebas:

Con el fin de poder comparar los tamaños de las topologías entre sí, en la primera experiencia se ha hecho coincidir el valor del parámetro en las hojas de las diferentes redes, tomando como punto de partida el valor N / (número de hojas de la red mayor). Por lo tanto en el nodo raíz, se analiza respecto a los valores N / (16 * 8) ó N / (16 * 4) ó N / (16 * 2) ó N / (16), dependiendo de la altura del árbol (el primero para altura 1, el segundo para altura 2 y así sucesivamente). En cada nivel este valor se divide por la mitad hasta llegar a las hojas, donde coinciden todos, independientemente de la altura del árbol que se estudie.

En el segundo experimento se fija el tamaño mínimo a resolver (dividir) por el procesador raíz (16384), igual para todas las redes. Al igual que antes este parámetro se divide a la mitad, cada vez que se avanza un nivel en el árbol.

En las tablas 2.9 y 2.10 se muestran las medidas temporales comunes, así como el número total de problemas resueltos (SOL) y divididos (DIV). Se ha optado por mantener las hojas como únicos procesadores que resuelven problemas, debido a que la introducción en los restantes procesadores producía mayores retrasos que las ganancias obtenidas por disminuir el camino a recorrer hasta las hojas. Posiblemente, en redes de mayor profundidad si ofrecerían mejoras.

Si se observa esta primera tabla (tabla 2.9), se puede comprobar la falta de simetría que demuestran los resultados experimentales. En ninguna de las cuatro redes utilizadas, se observa un crecimiento de la velocidad a medida que aumenta el tamaño a ordenar. Si se atiende al número de problemas solucionados y particionados existe un punto de inflexión en el tamaño 65536, a partir del cual el número de problemas generados va casi siempre en aumento. Esta reducción inicial parece el hecho más chocante, sin embargo es debida a que

el valor que se toma de entrada es demasiado pequeño para el tamaño 32768. Si se utiliza un parámetro mayor para este caso, el número de problemas analizados se mantiene creciente. A pesar de todo ello, si se comparan las cuatro topologías, fijado a priori un tamaño, la tendencia es siempre creciente, lo cual es esperanzador en el sentido de que una mejora en la heurística puede producir un comportamiento más de acuerdo con la teoría.

Tabla 2.9 Resultados para parámetro igual en hojas.

TREE FARM	TAM.	3 PROC.		7 PROC.		15 PROC.		31 PROC.	
TIEMPO / ACEL.	32768	2.34	1.68	1.60	2.46	1.43	2.75	1.36	2.90
SOL / DIV		11.00	11.00	37.40	20.00	72.40	43.80	102.80	79.20
TIEMPO / ACEL.	65536	5.11	1.62	3.46	2.39	2.85	2.90	2.60	3.19
SOL / DIV		7.20	7.20	28.00	10.00	58.60	28.00	92.20	57.80
TIEMPO / ACEL.	131072	10.37	1.70	7.36	2.40	6.48	2.73	6.28	2.81
SOL/ DIV		11.80	11.80	32.80	15.20	69.80	32.60	106.00	77.00
TIEMPO / ACEL.	262144	21.21	1.72	15.20	2.40	12.77	2.86	12.08	3.02
SOL/ DIV		11.00	11.00	39.80	16.40	73.60	40.60	112.00	84.20

Los resultados para este segundo caso (tabla 2.10), muestran de nuevo la importancia de mantener la diferencia entre problemas resueltos y problemas divididos de forma creciente, a medida que aumenta el tamaño del problema. En aquellos casos donde esta condición no se cumple, la aceleración cae. Sin embargo al igual que antes, el incrementar el número de procesadores produce mejoras en el tiempo de ejecución, sin importar el tamaño de problema.

Tabla 2.10 Resultados para parámetro igual en raíz.

TREE FARM	TAM.	3 PROC.		7 PROC.		15 PROC.		31 PROC.	
TIEMPO / ACEL.	32768	2.95	1.34	2.17	1.81	1.73	2.28	1.54	2.56
SOL / DIV		2.40	2.40	6.80	4.80	15.20	11.60	27.80	26.80
TIEMPO / ACEL.	65536	5.38	1.54	3.86	2.14	3.08	2.69	2.73	3.03
SOL / DIV		2.40	2.40	8.80	4.80	22.00	13.60	45.60	35.60
TIEMPO / ACEL.	131072	11.02	1.60	7.80	2.27	6.64	2.66	6.27	2.82
SOL/ DIV		4.80	4.80	17.00	9.60	42.80	26.60	86.40	69.40
TIEMPO / ACEL.	262144	22.00	1.66	15.09	2.42	12.84	2.84	12.24	2.98
SOL/ DIV		6.60	6.60	27.00	13.20	67.80	40.20	152.80	108.00

2.4.6. Comparaciones gráficas entre los esquemas presentados.

Para tener una medida más global de los resultados obtenidos, se presentan una serie de figuras que contrastan las aceleraciones de los códigos. En primer lugar, se muestran tres gráficos que representan las aceleraciones obtenidas para los tamaños 65536, 131072 y 262144. Las abreviaturas utilizadas para indicar cada algoritmo son las siguientes: (Central) para el esquema de granja inicial, (Equil) para el algoritmo jerárquico que asegura la división equilibrada de la entrada, (Mezcla) para la jerarquía que se basa en la mezcla de las entradas ya resueltas, (Selec) para el tercer esquema con jerarquía que muestrea la entrada para la elección del pivote, y (Compr) para el compromiso entre los esquemas centralizado y jerarquizado.

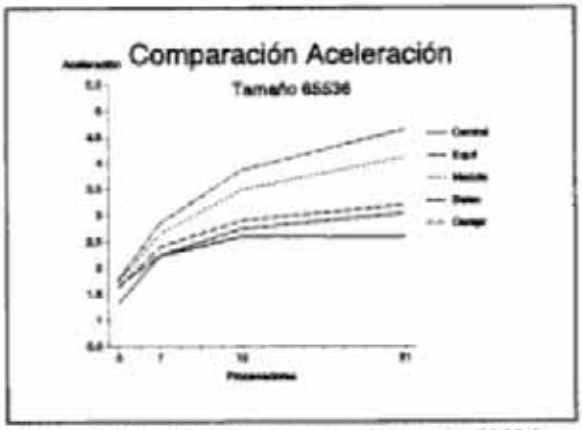

Figura 2.4 Aceleraciones para el tamaño 65536.

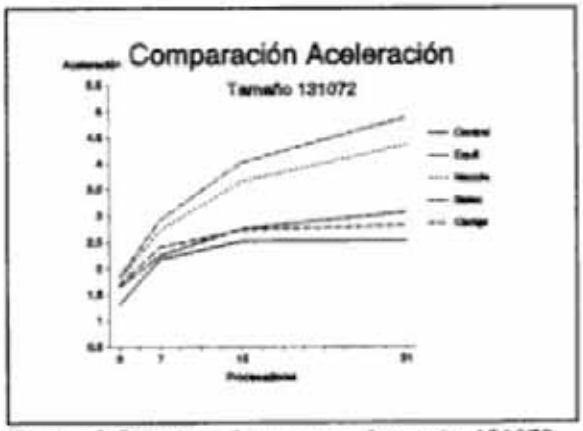

Figura 2.5 Aceleraciones para el tamaño 131072.

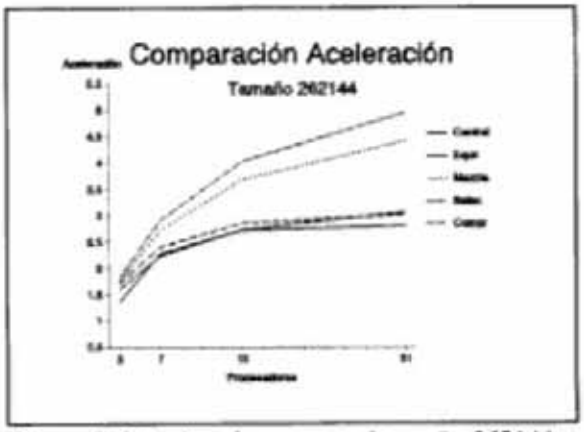

Figura 2.6 Aceleraciones para el tamaño 262144.

En las tres figuras (2.4, 2.5 y 2.6) se muestra que el orden se mantiene invariable. Los algoritmos del esquema jerárquico presentan el mejor comportamiento, aunque desaprovechan los procesadores. La estrategia compromiso (Compr) parece equipararse con el esquema equilibrado (Equil), lo cual da cierta esperanza a este esquema que puede utilizar de forma más razonada los procesadores disponibles. Tanto esta estrategia como la centralizada, pueden mejorar sus resultados si se mejora la función de elección de los problemas pequeños.

Las restantes figuras están dedicadas a los esquemas jerárquicos. En primer lugar, se confrontan los tamaños examinados para cada uno de los tres algoritmos (figuras 2.7, 2.8 y 2.9). Por último se presenta para el tamaño 262144, las figuras 2.10 y 2.11 con los resultados experimentados y los estimados, respectivamente. En ambos gráficos, se muestra el mismo comportamiento.

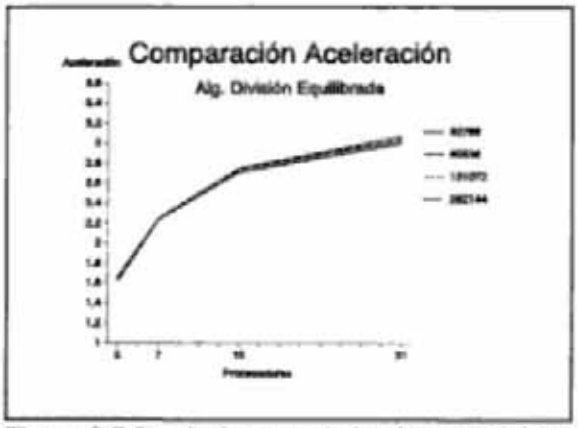

Figura 2.7 Resultados para el algoritmo de división equilibrada.

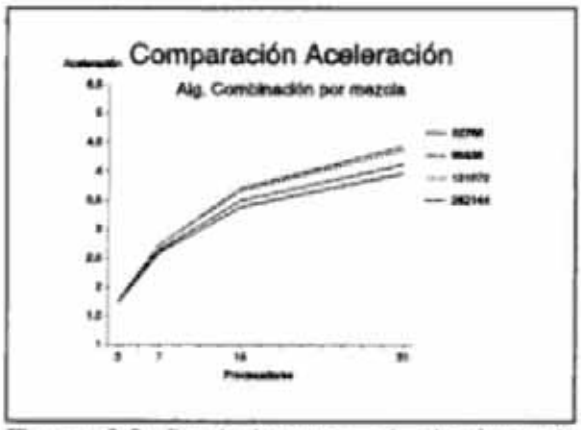

Figura 2.8 Resultados para el algoritmo de combinación por mezcla.

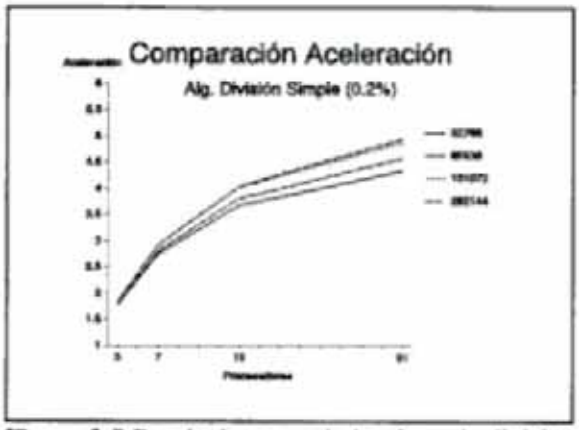

Figura 2.9 Resultados para el algoritmo de división simple (0.2%).

Observando las tres figuras presentadas, se puede concluir que en todos los casos los códigos presentados indican que el aumento del tamaño del problema no provocará el aumento de la aceleración y que el límite de eficiencia de los mismos es rápidamente alcanzable. Es obvio que el factor de comunicaciones y el de división o combinación anula la ganancia que se obtiene en el último nivel del árbol. Esta estabilización, evidentemente es más rápida, para aquellos códigos cuya fase de división ó combinación es más costosa computacionalmente.

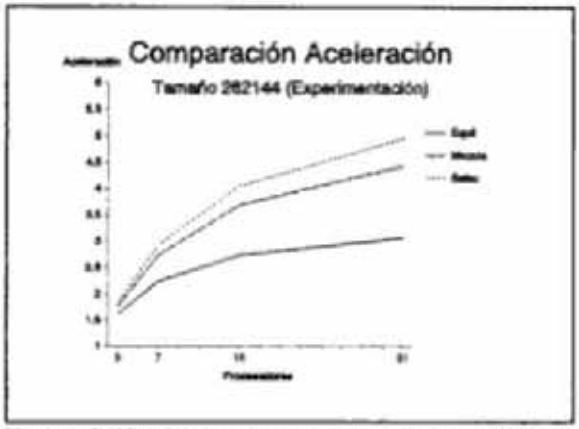

Figura 2.10 Resultados experimentales para 262144 elementos.

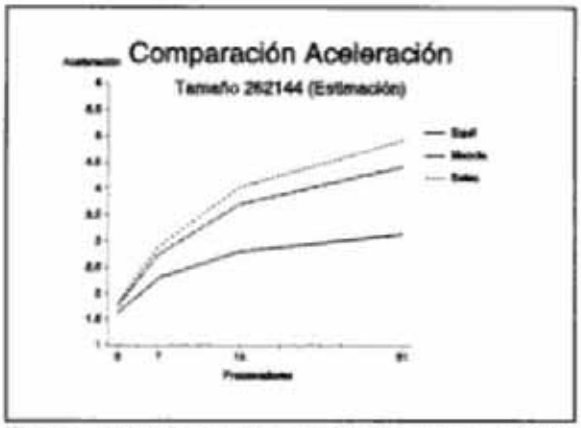

Figura 2.11 Resultados estimados para 262144 elementos.

2.5. BIBLIOGRAFIA.

[aho74]. A.V. Aho, J.E. Hopcroft and J.D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley. Reading, MA.

[aho83]. A.V. Aho, J.E. Hopcroft and J.D. Ullman. *Data structures and algorithms*. Addison-Wesley. Reading, MA.

[akl85]. S.G. Akl. *Parallel sorting algorithms*. Academic Press. Orlando, FL.

[bri91]. P. Brinch Hansen. Parallel divide and conquer. *Technical Report*, School of Computer and Information Science, Syracuse University, Syracuse, NY.

[bri94]. P. Brinch Hansen. Do hypercubes sort faster than tree machines?. *Concurrency: Practice and Experience* 6 (2), pp 143-151.

[cha93]. S. Chandra, M. Jain, A. Basu and P.S. Kumar. Sorting algorithms on transputer arrays. *Parallel Computing* 19, pp. 595-607.

[che84]. J. Chen, E.L. Dagless and Y. Guo. Performance measurements of scheduling strategies and parallel algorithms for a multiprocessor quick sort. *IEE Proceedings, Part E. Computers and Digital Techniques* 131, pp. 45-54.

[chi90]. S. Chiba, H. Honda, H. Maezawa, T. Tsukioka, M. Uematsu, Y. Yoshida and K. Maeda. Divide and conquer in parallel procesing. *Proceedings of the 3rd Transputer/Occam International Conference*, pp. 279-293. Tokyo, Japan. IOS Press.

[cla94]. P.G. Clayton, R.C. Watkins and E.P. Wentworth. A pilot implementation of some algorithmic skeletons on transputers. *Proceedings of the 7th Conference of the North American Transputer Users Group*, pp. 295-302. Atlanta, GA. IOS Press.

[dem82]. J. Deminet. Experiences with multiprocessor algorithms. *IEEE Transactions on Computers* C-31 (4), pp. 278-288.

[eva85]. D.J. Evans and N.Y. Yousif. Analysis of the performance of the parallel quicksort method. *BIT* 25, pp. 106-112.

[hoa62] . C.A.R. Hoare. Quicksort. *The Computer Journal* 5, pp. 10-15.

[hor78]. E. Horowitz and S. Sahni. *Fundamentals of computer algorithms*. Computer Science Press. Potomac, MD.

[iba87]. T. Ibaraki. *Enumerative approaches to combinatorial optimization. Part I-II*. J.C. BALTZER AG. Basel, Switzerland.

[jac90]. J.L. Jacquemin and M. Griffiths. Implementing recursion on a double ring topology. *Proceedings of the 3rd Transputer/Occam International Conference*, pp. 57-62. Tokyo, Japan. IOS Press.

[kim92]. D. Kim and Y. Hah. A parallel TSP algorithm based on divide and conquer strategy. *Parallel Computing and Transputer Applications*, pp 119-127. Barcelona. CIMNE, IOS Press.

[kin88]. G.A.P. Kindervater and H.W.J.M. Trienekens. Experiments with parallel algorithms for combinatorial problems. *European Journal of Operational Research* 33, pp. 65-81.

[lee88]. J. Lee, E. Shragowitz and S. Sahni. A hypercube algorithm for the 0/1 knapsack problem. *Journal of Parallel and Distributed Computing* 5, pp. 438-456.

[leo91]. C. León. Un compilador pascal paralelo para el modelo P-RAM. *Memoria de Licenciatura*. Dept. Estadística, Inv. Operativa y Computación, Univ. de La Laguna.

[li93]. X. Li, P. Lu, J. Schaeffer, J. Shillington, P.S. Wong and H. Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing* 19, pp. 1079-1103.

[pet81]. F.J. Peters. Tree machines and divide-and-conquer algorithms. *Proceedings of the CONPAR 81*, pp. 25-36. Springer Verlag Berlin.

[qui88]. M.J. Quinn. Parallel sorting algorithms for tightly coupled multiprocessors. *Parallel Computing* 6, pp. 349-357.

[qui94]. M.J. Quinn. *Parallel computing: Theory and practice*. McGraw-Hill. OR.

[rot85]. D. Rotem, N. Santoro and J. Sidney. Distributed sorting. *IEEE Transactions on Computers* C-34 (4), pp. 372-376.

[tod78]. S. Todd. Algorithms and hardware for a merge sort using multiple processors. *IBM Journal of Research and Development* 22 (5), pp. 509-517.

[var92]. P.T. Varman and K. Doshi. Sorting with linear speedup on a pipelined hypercube. *IEEE Transactions on Computers* 41 (1), pp. 97-103.

[wag86]. B. Wagar. Hyperquicksort: A fast sorting algorithm for hypercubes. *Proceedings of 2nd Conference on Hypercube Multiprocessors*, pp. 292-299.

[whe92]. M. Wheat and D.J. Evans. An efficient parallel sorting algorithm for shared memory multiprocessors. *Parallel Computing* 18, pp. 91-102.

[wir76]. N. Wirth. *Algorithms + data structures = programs*. Prentice-Hall. Englewood Cliffs, NJ.

[won89]. H.Won and S. Sahni. Hypercube-to-host sorting. *J. Supercomputing*, 3 (1). pp. 41-61.

CAPITULO III:

Ramificación y Acotación

3.1. INTRODUCCION.

En el método de *ramificación y acotación* (*branch and bound*), como en la técnica divide y vencerás, se parte de la creación de un algoritmo de tipo enumerativo, esto es, el problema original se descompone en una serie de problemas parciales de menor tamaño y del mismo tipo que el de partida. Estos problemas deberían permitir resolver el original una vez calculadas sus soluciones [aho83], [hor78]. Esta estrategia mejora el esquema divide y vencerás, al introducir una serie de operaciones adicionales (funciones de acotación, relaciones de dominancia, etc) que permiten descartar la búsqueda en ciertas partes del árbol. La operación de combinación desaparece al almacenarse en la etapa de división toda la información correspondiente al problema de partida.

La ramificación y acotación viene siendo utilizada con resultados provechosos en Optimización Combinatoria e Inteligencia Artificial, al evitar en muchos casos analizar todo el espacio de búsqueda [aho74]. En particular, se ha revelado como una de las técnicas más adecuadas para la resolución de problemas *NP-completos* (aquellos para los que no se ha encontrado un algoritmo en tiempo polinomial que los resuelva). Los problemas NP-completos constituyen una de las clases de problemas que mayor atención ha recibido y recibe por su complejidad computacional.

3.2. EL METODO DE RAMIFICACION Y ACOTACION (BRANCH AND BOUND).

En los problemas de optimización se trata de maximizar o minimizar el valor de una función sobre una región de búsqueda arbitraria. El código 3.1 explica la estrategia de ramificación y acotación para un problema de mínimo[1].

Un procedimiento de ramificación y acotación consiste en la repetición sucesiva de generación y comprobación de problemas parciales con el fin de encontrar una solución al problema de partida examinando la menor parte posible del árbol de búsqueda. Son varios los factores que intervienen en la consecución eficiente de este fin.

La primera disyuntiva es la elección del siguiente problema a explorar (*select*). Se pueden aplicar diferentes estrategias, dependiendo del tipo del problema, puede ser más idónea una que otra. Como se observa en la estructura *heap* se almacenan los problemas sin analizar. Esta estructura de datos debe ser tal que optimice los tiempos de acceso de la función *select*.

Se debe comprobar si el problema que está siendo examinado, puede ser acotado o resuelto. En la posible eliminación del problema actúan los test de cota inferior (*lower*) y las relaciones de dominancia (*dominated*). La cota superior (*upper*) da mayor fuerza al otro tipo

[1] Por simplicidad, se tratará a partir de ahora el problema del mínimo. La maximización se resuelve de la misma forma sin más que cambiar el signo de la función, max f(x) ⇔ min -f(x).

al otro tipo de acotación (sobre todo en las estrategias de selección que no profundizan en el árbol rápidamente). La cota inferior P^.L ha sido calculada durante la etapa de generación del mismo y es contrastada con la mejor solución obtenida hasta ese instante, z, con el fin de discernir si el problema puede alcanzar una mejor solución. Una vez comprobada la factibilidad del problema sobre el que se trabaja, se debe realizar su ramificación (branch).

```
Entrada: Problema P0
Salida:  Resultado R0 y valor z
Datos:   heap:
      { The structure contains the active problems. The associated
      information stores, at least, the identifier of the problem and the
      lower bound. }

Método: Procedure BranchAndBound;
        var P, P1, ... : problem;
        begin
          insert(P0); z := -∞; R0 := φ; P0^.L := lower(P0);
          while not_empty(heap) do begin
            P := select(heap); { search strategy }
            if (P^.L < z) and not dominated(P) { bounding } then begin
              upper(P, z', R');
              if z' < z then begin { new solution }
                z := z'; R0 := R';
              end; { if ... }
              if P^.L < z then begin
                branch(P, P1, ..., Pk);
                for i := 1 to k do begin
                  Pi^.L := lower(Pi);
                  if Pi^.L < z then insert(Pi);
                end; { for ... }
              end; { if ... }
            end; { if ... }
          end; { while ... }
        end; { BranchAndBound }
```

Código 3.1 Esquema de la técnica ramificación y acotación.

Existen algunas variantes del código presentado, aunque no afectan al esquema general. Por ejemplo, es posible realizar un análisis de los problemas activos cada vez que se encuentra una solución mejor. En este examen pueden ser retirados de la estructura *heap*, todos aquellos problemas que no puedan llevar a una mejor solución. Sin embargo, a efectos de eficiencia se suelen producir pérdidas de tiempo debido a que la eliminación de vértices del *heap* es bastante complicada.

A continuación se desarrollan cada una de las operaciones asociadas con el método. Para ilustrar el procedimiento se consideran como ejemplos dos problemas de optimización NP-completos: *El problema del viajante de comercio* (*(T)ravelling (S)alesman (P)roblem o TSP*) y *el problema de la mochila* (*(KNA)psack (P)roblem o KNAP*).

El enunciado del problema del viajante de comercio es muy simple: Un viajante debe visitar cada ciudad de la zona que le han asignado, exactamente una vez y retornar entonces al punto de partida. Supuesto que se conoce el costo (distancia) de los viajes entre cada par de ciudades, el problema se reduce a planificar un itinerario de forma que visite cada ciudad

una única vez y el costo (distancia) del viaje total sea mínimo. En términos de teoría de grafos, el problema es encontrar un circuito de longitud n (número de vértices del grafo) de costo mínimo en un grafo completo de n vértices con costos asociados a cada arco (i, j) (c_{ij}). Este problema es un clásico en optimización combinatoria y existen diversas variantes del problema, si se toma la anterior definición [hel70], [law85], [lit63]. Una posibilidad consiste en asignar costos diferentes a una pareja de vértices cuando el vértice origen es uno ó el otro (matriz no simétrica). En la literatura este problema es conocido con el nombre de *viajante de comercio asimétrico* (ATSP), denotando al otro como problema *simétrico* (STSP).

El problema de la mochila se puede enunciar de la siguiente manera: Un excursionista dispone de n tipos de objetos de diferentes tamaño y valor, pero sólo dispone de una pequeña mochila de capacidad b que usará para llevar los mejores. El problema consiste en encontrar la combinación de objetos que debería elegir para meter en su mochila, de manera que maximice el valor suma de todos los objetos que elige. Sobre este esquema general, se han ido introduciendo diferentes restricciones que han dado lugar a los diferentes tipos de problema de la mochila que se conocen. Como ejemplos más comunes, se pueden citar los siguientes:

- Si los objetos no pueden ser divididos en partes, se hablará del *problema de la mochila entera*.

- Si además de lo anterior, sólo se permite elegir un objeto de cada tipo, se denota como el *problema de la mochila 0-1*.

Matemáticamente hablando, el problema Knap(n, b) ó (n, b)[2] se representa por: Dados n objetos y una mochila de capacidad b. Para cada objeto i se tienen asociados un tamaño a_i y un valor c_i. Si el objeto i se introduce en la mochila n_i veces, se añade un valor de $c_i \cdot n_i$ al valor total. El objetivo es maximizar este último valor, manteniendo la suma de los tamaños de los objetos introducidos dentro de la capacidad de la mochila.

Ambos problemas se pueden representar mediante la siguiente formulación:

a) Mochila 0-1

$$P_0: \quad \max \sum_{j=1}^{n} c_j x_j$$

$$\text{s.a.} \quad \sum_{j=1}^{n} a_j x_j \leq b$$

$$x_j = 0,1 \quad 1 \leq j \leq n$$

b) Viajante de Comercio Asimétrico

$$P_0: \quad \min \sum_{j=1}^{n} \sum_{i=1}^{n} c_{ij} x_{ij}$$

$$\text{s.a.} \quad \sum_{i=1}^{n} x_{ij} = 1,\ 1 \leq j \leq n, \quad \sum_{j=1}^{n} x_{ij} = 1,\ 1 \leq i \leq n$$

$$\sum_{j \in S} \sum_{i \in S} x_{ij} \leq |S| - 1, \ \forall S$$

$$x_{ij} = 0,1 \quad 1 \leq i, j \leq n$$

[2] En esta memoria se utilizarán indistintamente las dos formulaciones.

3.2.1. La operación de ramificación (*branch*).

Si un problema no es lo suficientemente pequeño para ser resuelto y puede alcanzar una solución, se debe realizar una operación que lo divida en subproblemas del mismo tipo, pero de menor tamaño. Para los dos problemas ejemplo una posible división sería la siguiente:

El problema P_0 con n (n^2) variables se puede descomponer en 2 (n-1) problemas, con n-1 ($(n-1)^2$) variables cada uno. Cada problema parcial P_i queda determinado por el conjunto de variables que se fijan a 0 y 1. Un problema con todas las variables fijadas tiene una solución trivial y se representa mediante un vértice hoja en el árbol de búsqueda. Debido al hecho de que en esta operación descendente se almacena el camino recorrido, la estrategia de ramificación y acotación no necesita la exploración ascendente (operación de combinación).

a) Mochila 0-1

Se elige ramificar el objeto k-ésimo.

$$P_1: \quad \max \sum_{j \neq k} c_j x_j$$

$$\text{s.a.} \quad \sum_{j \neq k} a_j x_j \leq b$$

$$x_j = 0,1 \; j \neq k, \; x_k = 0$$

$$P_2: \quad \max \sum_{j \neq k} c_j x_j + c_k$$

$$\text{s.a.} \quad \sum_{j \neq k} a_j x_j \leq b\text{-}a_k$$

$$x_j = 0,1 \; j \neq k, \; x_k = 1$$

b) Viajante de Comercio Asimétrico

Se elige como variable de ramificación el arco km.

$$P_m: \quad \min \sum_{j \neq m} \sum_{i \neq k} c_{ij} x_{ij} + c_{km}$$

$$\text{s.a.} \quad \sum_{i=1}^{n} x_{ij} = 1, \; j \neq m, \; \sum_{j=1}^{n} x_{ij} = 1, \; i \neq k$$

$$\sum_{j \in S} \sum_{i \in S} x_{ij} \leq |S| - 1, \; \forall S$$

$$x_{ij} = 0,1 \; i \neq k, \; j \neq m, \; x_{km} = 1$$

$$x_{im} = 0, \; i \neq k, \; x_{kj} = 0, \; j \neq m$$

Si se resuelven P_1 y P_2 (P_m $2 \leq m \leq n$), aquel que tenga mayor (menor) valor proporciona la solución óptima al problema original P_0. El mismo razonamiento es válido para cualquier problema parcial P_i que se obtenga del de partida.

Se denotará por X_k al espacio de búsqueda de un problema P_k. La variable (ó variables) elegida para la ramificación depende del problema tratado y de la función de búsqueda utilizada. En el caso de los problemas elegidos, los nuevos conjuntos de búsqueda se pueden caracterizar por las ecuaciones siguientes. Aunque estas son dos posibles estructuras de ramificación, se han considerado muchas otras [iba87].

a) Mochila 0-1

X_k se puede descomponer en dos conjuntos X_{k1} y X_{k2}, fijando una variable de ramificación x_k a 0 y 1.

$X_{k1} = \{ x \in X_k \mid x_k = 0 \}$
$X_{k2} = \{ x \in X_k \mid x_k = 1 \}$

b) Viajante de Comercio Asimétrico

X_k se puede descomponer en tantos conjuntos como vértices queden por utilizar, sin más que fijar uno de los vértices.

$X_{km} = \{ x \in X_k \mid x_{km} = 1,\ x_{kj} = 0\ j \neq m,\ x_{jm} = 0\ j \neq k \}$

Para acotar el espacio de búsqueda, es necesario utilizar una serie de funciones y relaciones de acotación. Es deseable que sean fácilmente computables y que proporcionen una buena medida de la optimalidad y conveniencia de exploración del problema tratado.

3.2.2. Funciones de acotación inferior (*lower*).

Una de las funciones utilizadas para reducir el espacio de búsqueda es la de *acotación inferior* (*lower*). Para cada problema P_k generado, se computa este valor y se desecha el problema si se comprueba que no puede mejorar la solución óptima actual (lower(P_k) > z, para un problema de minimización). El objetivo es que la función *lower* ofrezca buenas cotas inferiores (lo más cercanas posibles al óptimo global). Debido a que este test debe ser efectuado sobre cada problema, el tiempo necesario para su cómputo se convierte en un factor crucial en la determinación de la eficiencia del algoritmo.

a) Mochila 0-1

La relajación convierte el problema en:

$$\overline{P_k}: \quad \max \sum_{j=k}^{n} c_j x_j$$

$$\text{s.a.} \quad \sum_{j=k}^{n} a_j x_j \leq b$$
$$x_j = 0,1 \quad k \leq j \leq n$$

Debido a la simplicidad de $\overline{P_k}$, se puede construir una solución óptima directamente, sin más que reordenar los objetos de forma decreciente según la razón beneficio-peso de cada objeto.

$$\text{lower}(P_k) = \sum_{j=1}^{q} c_j + [b - \sum_{j=1}^{q-1} a_j] c_q / a_q$$

b) Viajante de Comercio Asimétrico

En este caso, se obtiene un problema sin la condición del ciclo:

$$\overline{P_k}: \quad \min \sum_{j \in S_k} \sum_{i \in S_k} c_{ij} x_{ij}$$

$$\text{s.a.} \quad \sum_{i=1}^{n} x_{ij} = 1,\ j \in S_k, \quad \sum_{j=1}^{n} x_{ij} = 1,\ i \in S_k$$
$$x_{ij} = 0,1 \quad \forall\ i, j$$

La solución óptima surge del cálculo del árbol generador mínimo del conjunto S_k unido a un arco que conecte los vértices primero y último.

$$\mathit{MinSpanTree}(S_k) + c_{\text{lastfirst}}$$

La técnica utilizada normalmente en la construcción de esta función consiste en relajar el problema en exploración P_k, investigando ahora en otro problema $\overline{P}_k$, cuyo espacio de búsqueda contiene al de partida P_k. La finalidad de esta simplificación es aproximarse a un tipo de problema para el que sea fácilmente computable una solución óptima.

¿Cómo son las funciones de acotación inferior para los ejemplos tratados?. En ambos casos existen varias posibilidades dependientes de la forma de planteamiento del problema. Para la mochila 0-1 se ha utilizado la propuesta en [mar90] y para el viajante de comercio asimétrico la propuesta en [law85].

3.2.3. Relaciones de dominancia (*dominated*).

Otra importante fuente de operaciones de acotación, tan poderosa en algunos casos como la función de acotación inferior (*lower*), es la de los *test de dominancia* (*dominated*). Están basados en relaciones binarias, conocidas como relaciones de dominancia, sobre el conjunto de los problemas generados. Dados dos problemas P_i y P_j, se trata de analizar si uno de los dos domina al otro. En caso afirmativo, el problema dominado se puede excluir de la ramificación.

En la práctica, las relaciones de dominancia suelen investigar sobre las variables no fijadas o variables libres asociadas a cada problema. Si se encuentran dos problemas con el mismo conjunto de variables libres se puede descartar aquel con peor valor computado.

Para los problemas anteriores puede ser test de dominancia el siguiente (sólo se explica el caso de la Mochila 0-1, para el Viajante de Comercio el test es similar).

> Sean P_k y P_h dos problemas parciales que se obtienen de P_0, como resultado de fijar las variables x_j con $j \in S$, a x_j^k y x_j^h respectivamente. $S \subset \{1, 2, \ldots, n\}$ es un conjunto de índices y x_j^k, $x_j^h = 0, 1$. Los subproblemas P_k y P_h comparten el mismo conjunto de variables libres x_j, $j \in \{1, 2, \ldots, n\}$-S. Si se verifica:
>
> $$\sum_{j \in S} c_j x_j^k \geq \sum_{j \in S} c_j x_j^h$$
> $$\sum_{j \in S} a_j x_j^k \leq \sum_{j \in S} a_j x_j^h$$
>
> cualquier solución factible en P_h es también factible en P_k, y su valor objetivo no es más pequeño que el valor objetivo para P_h, luego P_k domina a P_h (se invierte la primera relación al hablar de problema de mínimo).

3.2.4. Funciones de acotación superior (*upper*).

Dado un problema P aunque el cálculo de su valor óptimo sea muchas veces complejo, se puede obtener fácilmente una solución factible: *upper(P, z', R')*. Los llamados *algoritmos aproximados* o *heurísticas* se usan con este propósito. En concreto, en un problema de minimización P, tal solución factible R', con valor de función objetivo z', suministra una cota superior del valor óptimo.

Las cotas superiores se utilizan para modificar el valor óptimo actual z. Normalmente, si se obtienen buenas cotas superiores en las primeras etapas del algoritmo, el test de la cota inferior se puede convertir en una herramienta muy potente. Sin embargo, es de destacar que las funciones de acotación superior no se toman como imprescindibles en la construcción del esquema de ramificación y acotación, aunque si pueden ser muy útiles.

a) Mochila 0-1

Aplicando la ordenación anterior, se puede obtener una solución inmediata mediante un algoritmo de tipo *greedy*, cuyo valor será

$$u(P_i) = \sum_{j=1}^{q-1} c_j$$

Obsérvese que en este caso la obtención de una cota superior no implica cálculo adicional, puesto que este valor se computa en la fase de ramificación.

b) Viajante de Comercio Asimétrico

Los algoritmos aproximados utilizados en este caso, se basan en la construcción de un *ciclo hamiltoniano*. En cada paso se elige, mediante una función determinada, un vértice a añadir al camino. Los criterios de selección del vértice constituyen la única diferencia entre los algoritmos, aunque la mayoría están basados en las ideas de Prim y Dijkstra para el cálculo de un árbol generador mínimo [sys83].

3.2.5. Estrategias de selección (*select*).

Debido al esquema de trabajo de este tipo de algoritmos puede ser muy importante la eleción del siguiente problema a explorar. La función de selección *select* no afecta a la convergencia del algoritmo, pero sí, a su eficiencia y al espacio de memoria requerido. Existen varios tipos de funciones de selección, la inmensa mayoría se basan en los recorridos de un árbol. A continuación se introducen las más importantes. Se denota por A al conjunto de problemas no computados o problemas activos. Este conjunto se almacena en la estructura *heap*.

a. Selección heurística.

En este tipo de selección, aunque existen muchas variantes denotadas por $h(P_i)$ [iba87], se puede utilizar para cada vértice activo P_i una combinación de los valores *upper*(P_i) y *lower*(P_i). Este valor sólo sirve como estimación del valor óptimo, pero requiere mucho

menor tiempo de cómputo. En este caso, la *función de selección heurística* $select_h$ debe verificar lo siguiente (se elige como problema a tratar, aquel con menor valor h):

$$select_h(A) = \{ P_i \mid h(P_i) = \min \{h(P_j)\ \forall P_j \in A\} \}$$

Suponiendo que se dispone de una buena función h, la importancia de la selección heurística radica en que se pueden obtener algoritmos relativamente eficientes sin un excesivo consumo de espacio. Puede comprobarse que las mejores funciones h son las conocidas con el nombre de *nonmisleading* [iba87], aunque también son las más difíciles de obtener pues siguen las mismas reglas que la función objetivo de partida.

Muchos autores entienden que la selección heurística es muy general, en el sentido de que muchos de los métodos de selección existentes se pueden identificar como casos particulares.

b. Selección primero en profundidad (*depth first*).

Es el método de selección más económico desde el punto de vista del espacio de memoria utilizado, también conocido como *selección lineal* o *selección LIFO (Last-In-First-Out)*. Si se denota por D al conjunto de vértices activos de mayor profundidad (función *depth*), esto es:

$$D(A) = \{ P_i \in A \mid depth(P_i) = \max \{depth(P_j)\ \forall P_j \in A\} \}$$

La función de selección $select_D$ se define como:

$$select_D(A) = \{ P_i \in D(A) \mid h(P_i) = \min \{h(P_j)\ \forall P_j \in D(A)\} \}$$

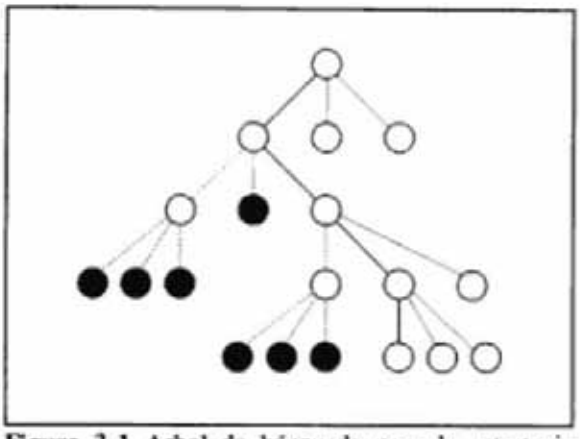

Figura 3.1 Arbol de búsqueda para la estrategia primero en profundidad.

Entre los vértices de mayor profundidad, se elegirá aquel con menor valor de la función heurística h. Como se puede observar en la figura 3.1[3], la estructura asociada a este tipo de selección siempre estará representada por un árbol en el que uno de los nodos hijo se expande hasta obtener una solución o un problema no factible. Si se atiende a los posibles recorridos de árboles, esta estrategia coincide con el recorrido en postorden de los mismos. Este esquema de trabajo conlleva la utilización de una estructura tipo Pila (*LIFO*), para el *heap*. Si se considera A como una lista ordenada de

[3] Los vértices negros indican los problemas parciales terminados, mientras que los blancos denotan subproblemas activos ó descompuestos.

vértices activos almacenados en una pila, en el paso de selección se elige el último elemento P_i de la estructura asociada. En la etapa de ramificación se descompone P_i y se introducen al final del *heap* P_{i1}, ... P_{ik}, en orden decreciente respecto a h, además de eliminar P_i.

Una desventaja de este tipo de selección, se encuentra en su forma de trabajo. El número de vértices inspeccionados es normalmente mayor que los realizados en otro tipo de selecciones (heurística o primero el de mejor cota). Además, si se entra en una rama que no lleva a la solución óptima, se necesita mucho tiempo para salir de esta ramificación. Sin embargo, la exploración en profundidad conduce más rápidamente a la obtención de una cota superior que otras estrategias, lo que representa una ventaja desde el punto de vista práctico.

c. Selección primero el de mejor cota (*best bound first*).

La selección heurística que usa la función de cota inferior *lower* en lugar de h se llama *selección mejor cota* (o *primero mejor*, *mínimo valor*). En otras palabras, la *función mejor cota* $select_{lower}$ satisface:

$$select_{lower}(A) = \{ P_i \mid lower(P_i) = \min \{lower(P_j)\ \forall P_j \in A\} \}$$

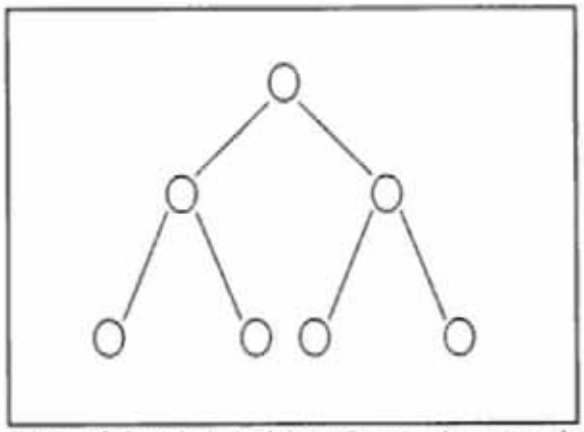

Figura 3.2 Arbol de búsqueda para la estrategia primero mejor cota.

Se elige como vértice activo aquel con la menor cota inferior. En el caso de que hubieran varios, se elige el más profundo. Esta estrategia se caracteriza por minimizar el número de problemas parciales descompuestos. Esto es una ventaja, al mantenerse el tiempo de cómputo proporcional al número de vértices inspeccionados. Sin embargo se plantean como inconvenientes, el retraso para obtener una solución z y el crecimiento exponencial en el espacio de memoria necesario cuando se incrementa la profundidad del árbol. Ambos defectos se ven atenuados si se combina la selección con una función de acotación superior. Este tipo de selección suele expandir primero los nodos de menor profundidad, construyendo un árbol que se asemeja bastante a un paraguas (figura 3.2).

d. Selección primero en amplitud (*breadth first*).

Si se le concede a los vértices de menor profundidad la mayor prioridad, se tiene en contraste con la selección *primero en profundidad*, la *primero en amplitud* (función *depth*). Sea:

$$D(A) = \{ P_i \in A \mid depth(P_i) = \min \{depth(P_j)\ \forall P_j \in A\} \}$$

Entonces el vértice $select_D$ elegido mediante esta estrategia verifica:

$$select_D(A) = \{ P_i \in D(A) \mid h(P_i) = \min \{h(P_j) \ \forall P_j \in D(A)\} \}$$

Es decir, el vértice con menor valor h de entre aquellos que tienen menor profundidad. La ejecución mediante esta selección procede como el recorrido en preorden de un árbol. Esto da pie a una implementación del *heap* por medio de una lista *FIFO (First-In-First-Out)* o *cola*. El esquema de manipulación del *heap* es similar a la pila para la selección primero en profundidad, excepto que el problema elegido es el primero de la cola.

Este tipo de estrategia no es recomendable ni desde el punto de vista del tiempo de computación ni desde el de ahorro de memoria. Sin embargo, si la estructura del problema considerado admite la aplicación de un test de dominancia entre los vértices de la misma profundidad, esta selección puede resultar ventajosa. Su mayor utilidad se presenta en la técnica de programación dinámica como se comentará en el siguiente capítulo.

3.3. PARALELIZACION DE LA TECNICA DE RAMIFICACION Y ACOTACION.

Si se considera el procedimiento de ramificación y acotación general del apartado anterior, son admisibles dos tipos de paralelización:

(a) El proceso de cálculo de las operaciones asociadas a cada problema parcial P_i seleccionado puede realizarse concurrentemente. Sin embargo, este paralelismo es específico del tipo de problema considerado, y no admite reglas generales de implementación.

(b) Si un sistema dispone de p procesadores se pueden seleccionar del *heap* para su comprobación p problemas parciales P_i. Entonces, cada procesador ejecuta independientemente el cómputo asociado con el problema P_{ip} seleccionado [bur82], [kin88a], [qui94].

3.3.1. Procedimiento general de partida.

La implementación de la segunda posibilidad, requiere que los datos globales se guarden en un *heap* compartido por todos los procesadores. Tales datos deberían incluir, en este caso, los problemas activos, la solución actual, etc.

Inicialmente en la estructura *heap* se encuentra el problema de partida P_0. Las variables z y R_0 toman los valores $-\infty$ y ϕ respectivamente. Cada procesador m del intervalo $1 \leq m \leq p$, debería ejecutar el procedimiento del código 3.2.

Cada procesador acaba cuando recibe del proceso administrador la señal de finalización. Los datos del *heap* compartido deben ser manipulados en regiones críticas. Se

definen las diferentes funciones *send* y *receive* para destacar estos problemas adicionales que surgen. Todo lo comentado para la implementación del esquema paralelo centralizado en la técnica divide y vencerás, es válido para la estrategia de ramificación y acotación. La única diferencia estriba en la existencia de un único *heap*. El proceso administrador asignado a estas tareas es equivalente al mostrado en la técnica anterior, por lo que se ha optado por no repetirlo en este caso. Cada vez que el administrador recibe una solución que mejora a la actual, criba los problemas del *heap* y la transmite a los restantes procesos obreros. En los tests efectuados por cada uno de los obreros en la generación de los nuevos problemas sólo se ha incluido el test de la cota inferior, al ser la dominancia un concepto que necesita de informaciones globales no guardadas en cada uno de los procesadores.

```
Método: Procedure ConcurrentCentralizedB&B(m);
        var i : 1..k; not_end : boolean; tag : signal;
        begin
          not_end := TRUE;
          while not_end do begin
            send_request(m); { solicite work }
            tag := receive_signal;
            if tag = finish then not_end := FALSE
            else begin
              if tag = z_val then z_m := receive_z { update z }
              else begin
                P := receive_prob(heap);
                upper(P, z', R');
                if z' < z_m then begin { new solution }
                  send_z(z');
                  R_m := R';
                end; { if ... }
                if P^.L < z_m then begin
                  branch(P, P_1, ..., P_k);
                  for i := 1 to k do begin
                    P_i^.L := lower(P_i);
                    if P_i^.L < z_m then
                      send_prob(P_i); { insert on heap }
                  end; { for ... }
                end; { if ... }
              end; { else }
            end; { else }
          end; { while ... }
        end; { ConcurrentCentralizedB&B }
```

Código 3.2 Esquema paralelo centralizado de la técnica ramificación y acotación.

3.3.2. Ejecución concurrente eficiente.

La ejecución concurrente eficiente viene dada por la siguiente igualdad:

$$t_p(P_0) \approx t_1(P_0)/p$$

Aunque en algunas situaciones anómalas se puede obtener $t_p(P_0) < t_1(P_0)/p$ como se vera posteriormente, sigue siendo más usual la desigualdad contraria. A las dos causas ya

expuestas, cuando se trató la ejecución concurrente del procedimiento divide y vencerás, se une una tercera, específica de la ramificación y acotación:

Puesto que se procesan en cada uno de los procesadores problemas diferentes al mismo tiempo, es posible que estos trabajen en problemas que no habrían sido explorados por el secuencial.

3.3.3. Anomalías en la ramificación y acotación concurrente.

En algunos experimentos computacionales se pueden observar comportamientos anómalos respecto al tiempo de ejecución [lai84], [lai85]. Asumiendo que el cociente $t_1(P_0)/p$ (linealidad), puede utilizarse como una cota que permite medir la eficiencia de un algoritmo paralelo, se encuentran dos tipos de casos anormales:

$$t_p(P_0) > t_q(P_0) \quad p > q \tag{1}$$

$$t_p(P_0) < t_1(P_0)/p \tag{2}$$

En el primer caso se produce un detrimento con respecto al algoritmo paralelo con q procesadores (anomalía de tipo I o detrimental) y en el segundo una aceleración superlineal (anomalía de tipo II o aceleratoria).

El primer fenómeno se debe, básicamente a lo siguiente: Varios procesadores pueden explorar secciones del árbol que retrasan el análisis de una parte prometedora del espacio de selección y que no hubieran sido procesadas en caso de haber encontrado una buena cota en iteraciones anteriores. En el segundo caso se da la situación contraria: El algoritmo paralelo puede encontrar una buena cota mucho antes que la versión secuencial, lo cual provoca la eliminación de la estructura *heap* de nodos del árbol que deberían ser expandidos en el caso secuencial.

Como situación ideal, se trata de minimizar el número de anomalías detrimentales y maximizar el de aceleratorias. En [iba87] se discuten condiciones suficientes para evitar anomalías de tipo I y condiciones necesarias para producir anomalías de tipo II. Lai y Sahni [lai84] estudian dicho comportamiento ofreciendo algunos resultados experimentales para problemas simulados. Todo el estudio se basa en la definición de *nodo crítico*:

Un nodo P_k se dice crítico si y sólo si $lower(P_k) < z$ (supuesto un problema de mínimo)

La principal característica de los nodos críticos es que no dependen de la regla de selección utilizada, sino que únicamente dependen del problema a resolver y de la función de acotación inferior usada.

3.3.4. Ramificación y acotación distribuida.

Si se observa detenidamente el algoritmo anterior, se advierte que no es necesario que cada procesador acceda a todos los elementos del espacio de búsqueda, basta con controlar la mejor solución actual y poder investigar en una sección del árbol. Se puede entonces distribuir el *heap* que almacena los subproblemas generados, para reducir las comunicaciones en la red (*multicomputer*) o eliminar el acceso simultáneo a la memoria común (*multiprocessor*). Sin embargo este hecho puede empeorar el comportamiento del algoritmo, en vez de mejorarlo, si esta es la única modificación que se realiza. Evidentemente, la distribución inicial del trabajo reduce las comunicaciones de forma drástica, pero hay que contar con el hecho de que se desconoce a priori cual es el árbol de búsqueda a explorar, con lo que esta distribución no tiene por que ser equitativa. Para arreglar este y otros inconvenientes se introducen una serie de técnicas que se desarrollan a continuación y que en conjunto se denomina *ramificación y acotación distribuida*.

A continuación, y siguiendo con el esquema de representación de los diferentes algoritmos que se basan en la técnica de ramificación y acotación, se presenta una posible implementación del algoritmo paralelo correspondiente. Se parte de un algoritmo básico donde cada procesador analiza el árbol de búsqueda que se le suministra al comenzar sin realizar comunicaciones posteriores para evitar complicar el código de partida.

3.3.4.1. Implementación distribuida.

Ahora se dispone de tantas estructuras *heaps* como procesadores forman el computador paralelo. Cada $heap_m$ se almacena en el procesador m.

De partida, las estructuras de cada procesador pueden contener:

$heap_1 := P_0$, $z_1 := -\infty$ y $R_1 := \phi$
$heap_2 := \phi$, $z_2 := -\infty$ y $R_2 := \phi$

$heap_p := \phi$, $z_p := -\infty$ y $R_p := \phi$

Uno de los procesadores, el raíz (por simplicidad denotado como uno), es un procesador especial que se encarga de arrancar la red y controlar el resultado final. Nada se asegura con respecto a los datos de los restantes procesadores. En esta situación, una metodología de trabajo de cada procesador en el rango $1 \leq m \leq p$ podría ser la mostrada en el código 3.3.

El algoritmo finaliza cuando todos los *heaps* están vacíos. En un apartado posterior se discute como resolver el problema de la finalización distribuida.

A pesar de que se siguen manteniendo las relaciones de dominancia en el algoritmo, en la implementación distribuida su uso queda desvirtuado. Por ello normalmente, son

eliminados de este tipo de esquema. Hay incluso algunos autores [ort93] que no incluyen estas relaciones de dominancia (*dominated*) como factores intrínsecos a un procedimiento de ramificación y acotación.

```
Método: Procedure ConcurrentDistributedB&B(m);
        var heap_m;
        begin
          while not_empty(heap_m) do begin
            P := select(heap_m); { Search strategy }
            if (P^.L < z) and not dominated(P) { bounding } then begin
              upper(P, z', R');
              if z' < z_m then begin { new solution }
                z_m := z';
                R_m := R';
              end; { if ... }
              if P^.L < z_m then begin
                branch(P, P_1, ..., P_k);
                for i := 1 to k do begin
                  P_i^.L := lower(P_i);
                  if P_i^.L < z_m then
                    insert(P_i); { insert on heap_m }
                end; { for ... }
              end; { if ... }
            end; { if ... }
          end; { while }
        end; { ConcurrentDistributedB&B }
```

Código 3.3 Esquema paralelo distribuido de la técnica ramificación y acotación.

Si se observa detenidamente el código anterior, surgen las siguientes situaciones problemáticas, para las cuales se deben desarrollar estrategias que las prevengan (algunas ya se introdujeron con el esquema jerárquico del divide y vencerás):

- Algún procesador puede permanecer ocioso, mientras otros están aún trabajando y disponen de trabajo esperando en su *heap*. Este fenómeno se conoce con el nombre de tiempo ocioso (*idle time*).

- Un procesador puede explorar subproblemas que podrían ser descartados si se tuviera conocimiento de la mejor solución que ha encontrado otro procesador (sobrecarga en la búsqueda, *search overhead*).

- Detectar la condición de parada es una tarea más compleja que en el caso centralizado.

- Se debe inicializar cada procesador con un conjunto apropiado de subproblemas.

La parte más complicada en la implementación de un algoritmo de ramificación y acotación paralelo es la de encontrar una estrategia para:

(a) asegurar que la mejor solución encontrada por un procesador es conocida por todos

los restantes (*search overhead*),

(b) alimentar a los procesadores con *buenos* subproblemas con el fin de reducir los tiempos ociosos (*idle time*),

(c) impedir que se produzca un exceso de tráfico en la red, para evitar los embotellamientos (*traffic overhead*).

Desgraciadamente estas tres metas están íntimamente relacionadas. Las soluciones que se proponen para la resolución de alguna de ellas pueden no favorecer a las restantes. En conjunto se trata de conseguir un equilibrio que beneficie al algoritmo globalmente.

3.3.4.2. Criterios de inicialización.

La inicialización en este tipo de algoritmos se puede llevar a cabo encargando a un procesador que genere una cantidad suficiente de subproblemas y los distribuya a los restantes procesadores. En este caso, siguiendo la metodología expuesta, el paso de inicialización para los procesadores $2 \leq m \leq p$, debería ser el siguiente:

P := receive_initial_prob;

Un método con menos comunicaciones, pero que consume más tiempo en cómputos locales, consiste en dejar a todos los nodos de la máquina comenzar a trabajar en el mismo problema y contar los subproblemas generados [vor86]. Tan pronto como el número de subproblemas del *heap* de un procesador supere el índice que lo identifica, se eliminan todos los subproblemas de su estructura local, excepto el último generado, que será el subproblema a trabajar. Una posible desventaja de este método es que el último procesador no comienza hasta haber generado p-1 problemas, lo que ocurre en tiempo de orden p. Además, si el árbol de exploración es equilibrado, este esquema conduce a un desequilibrio en el que los primeros procesadores asumen mayor carga de trabajo. Además se pierde rápidamente la ventaja de la ramificación primero el de mejor cota de la versión secuencial.

Una alternativa que podemos considerar consiste en que todos los procesadores esperen a que el *heap* contenga al menos p problemas mejores. Aunque se retrasa la entrada en acción de los primeros procesadores, el tiempo de comienzo para ellos sigue siendo de orden p. El procedimiento asociado a todos los procesadores podría ser el del código 3.4.

```
heap_m := P_0;
while (|heap_m| < p) do begin
  .... Classic Branch and Bound algorithm
end; { while }
heap_m := select_and_free(m, heap_m);
```

Código 3.4 Una posible fase de inicialización de esquema distribuido.

Un tercer método consiste en dividir los procesadores según una jerarquía siguiendo

el método usual de distribución de carga en hipercubos: El primer problema es ramificado por todos los procesadores. La división del primer problema en k subproblemas da lugar a la formación de k grupos de procesadores. Cada uno de los grupos de procesadores procede a la ramificación de su correspondiente problema. En un determinado nivel de la búsqueda, se habrán formado tantos grupos de procesadores como problemas. Después de log(p) etapas cada procesador dispone de un problema para trabajar. Aunque con este último método propuesto se reduce el tiempo de inicialización de nuevo se pierde la ventaja de la búsqueda primero-mejor.

3.3.4.3. Minimización de la sobrecarga en la búsqueda (*search overhead*).

Debido a que cada procesador realiza la exploración concurrente de los problemas locales a su *heap*, algunos de estos problemas podrían ser descartados si las cotas que han encontrado los otros procesadores fueran emitidas al resto de la red. De forma trivial, se puede observar que la ausencia de estas comunicaciones podría producir la exploración de subárboles que nunca serían tratados por el algoritmo secuencial, con la consiguiente sobrecarga. La actualización de los valores óptimos z_m se expresaría de la siguiente forma en el algoritmo asociado (código 3.5).

```
if z' < z_m then begin { upper bounding }
  z_m := z';
  broadcast(z_m);
  R_m := R';
end; { if ... }
```

Código 3.5 Actualización de valores óptimos.

La recepción de este nuevo valor óptimo por parte de un procesador, hace necesario algún mecanismo que asegure la exclusión mutua sobre estas variables. Con el esquema utilizado para representar los algoritmos, se necesita un nuevo proceso en cada procesador que se encargue de la recepción y posterior comunicación de estas nuevas cotas y que se ejecute concurrentemente con la aplicación.

3.3.4.4. Redistribución equitativa de la carga. Tratamiento de los tiempos ociosos (*idle times*).

Los problemas que resuelve cada procesador de la red deben tener un tamaño o grano lo suficientemente grande como para superar el tiempo de comunicaciones. Para aprovechar toda su potencia de cálculo, el número de estos problemas debe ser amplio [vor88]. Una heurística debe decidir si el problema en exploración es lo suficientemente pequeño como para no ser expandido. Esta forma de trabajar provoca que sólo los niveles más superficiales del árbol de búsqueda sean explorados y expandidos. Si este proceso únicamente se realiza en el procesador que contiene al proceso administrador, se está hablando de la estrategia de ramificación y acotación centralizada.

En el caso de que este esquema se repita en todos y cada uno de los procesadores, se produce un algoritmo distribuido. Esta distribución del *heap* global no conlleva obligatoriamente una distribución equitativa del trabajo asignado a cada procesador. En la mayoría de los problemas es necesario una redistribución de la carga de trabajo [chi90], con el fin de minimizar los tiempos ociosos.

La distribución de la carga puede ser dinámica ó estática, según se tengan en cuenta ó no los cambios en el factor de carga que se producen durante la ejecución del algoritmo. Los algoritmos de redistribución de carga dinámica se caracterizan por:

(a) los conocimientos que se utilizan en la decisión de la distribución de las unidades de carga (*espacio de decisión*),

(b) el entorno de la red al que puede emigrar una determinada unidad de carga (*espacio de migración*).

En ambos casos se puede distinguir entre espacios globales y locales. En la toma de decisiones, con el fin de discernir si debe realizarse una redistribución de la carga, un procesador puede necesitar información relativa al sistema en su conjunto (global) o de sólo algunos (local). En este último caso, se analiza la situación del procesador propietario de la carga y a lo sumo, la de sus vecinos. De la misma manera se puede razonar entre espacio de migración local o global.

En este sentido Lüling y Monien [lül91] utilizan una estrategia distribuida con espacios de decisión y migración locales. Para describirla definen la carga de un procesador como el resultado de una función peso w en los elementos del *heap* local del procesador, que depende del problema concreto y la topología utilizada. Cada procesador procura mantener equilibrado su peso y el de sus vecinos. Una posible variación en este equilibrio puede producir la emigración de subproblemas en el entorno. Nada se asegura sobre qué subproblema, de entre los *mejores* del *heap*, debe de ser enviado. Esta estrategia de migraciones locales da lugar a un alto grado de simetría en el sistema.

Otra política parecida es la utilizada por McKeown y otros [mck91] en su estrategia SHL sin optimización (*Select Highest Locally*). En este caso, los procesadores solicitan trabajo cuando agotan su *heap*. Aquellos procesadores que reciben la señal de solicitud de trabajo de uno de sus vecinos, enviarán una cantidad de trabajo directamente proporcional al número de problemas que contenga su *heap* (su función peso w cuenta el número de subproblemas que contiene). Se utiliza la serie de Fibonacci para indicar qué subproblemas enviar. Estas modificaciones afectan al código en la parte dedicada a la selección del problema y a la condición de parada (código 3.6).

Evidentemente, un nuevo proceso se debe encargar de administrar las recepciones de solicitud de trabajo y los envíos de éste. Como en el caso de la reducción de la sobrecarga en la búsqueda, este proceso actúa al mismo tiempo que el código expresado.

```
function not_end : boolean;
begin
  if not_empty(heap_m) then not_end := FALSE
  else { heap weight has changed } begin
    send_request;
    if receive_finish then not_end := TRUE
    else if receive_prob then not_end := FALSE;
  end; { else }
end; { not_end }

while not_end do begin
  P := select(heap_m); { update_weight }
  ...
end; { while ... }
```

Código 3.6 Selección de nuevo problema y condición de parada.

Sin embargo esta distribución de trabajo, como respuesta a cambios en el entorno comunicados explícitamente, puede provocar que se expandan más nodos que en las estrategias centralizadas. En este sentido, tanto Lüling como Mckeown proponen enviar problemas prometedores entre los procesadores a intervalos regulares. Este intervalo se convierte en un parámetro adicional que caracteriza a la estrategia. Para Mckeown se crea una subdivisión de la estrategia SHL entre infinito y uno, según sea el número de expansiones realizables antes de una comunicación.

Troya y Ortega proponen una estrategia diferente [tro89]. No será obligatorio almacenar los subproblemas generados en el *heap* local intentando evitar una redistribución posterior. Utilizan unas funciones de distribución que indican él o los *heaps* destino para los subproblemas generados. Experimentan con tres tipos de funciones, intentando abarcar el conjunto de las posibles distribuciones de carga no centralizadas: estática y dinámica (con espacio de decisión local en este último caso). Una diferencia fundamental es la ausencia de control sobre el estado del *heap* local. El código del algoritmo se ve modificado en la fase de ramificación (código 3.7).

```
branch(P, P_1, ..., P_k);
for i := 1 to k do begin
  P_i^.L := lower(P_i);
  if P_i^.L < z_m then begin
    n := distributed(i);
    send_prob(P_i, n); { insert on heap_n }
  end; { if ... }
end; { for ... }
```

Código 3.7 Ramificación e inserción de problemas generados.

Se debe asegurar el acceso en exclusión mutua a los conjuntos $heap_m$. La recepción de los problemas se maneja en otro proceso paralelo al cual se debería consultar para la obtención de uno de ellos.

3.3.4.5. Reglas de parada.

Si se admite un esquema de trabajo como el expresado en el algoritmo distribuido de partida (código 3.3), sin tener en cuenta la redistribución de la carga entre los procesadores, la regla de finalización será una simple generalización del caso centralizado. Un procesador maestro (por comodidad el raíz) recopila señales del resto de los nodos, estos envían la señal una vez agotado su *heap* de problemas. Con un contador el maestro puede detectar la finalización del algoritmo.

En el caso más general la finalización distribuida se puede realizar definiendo un *circuito hamiltoniano* en la red y un procesador maestro [lül89]. Si el procesador maestro está ocioso envía una *señal ámbar* a su vecino en el circuito hamiltoniano. Los nodos ociosos pasan la señal a su vecino hasta que se alcance de nuevo al maestro. Si el maestro está libre y no ha recibido ningún nuevo subproblema desde que la señal empezó a circular envía otra *señal roja* a su vecino. Todos los nodos pasan esta *señal roja* si están desocupados y no han recibido ningún subproblema desde la llegada de la *señal ámbar*. Si la *señal roja* regresa al maestro todos los procesadores están libres y el maestro puede comenzar la recolección de soluciones.

3.3.5. Otras estrategias intermedias.

Son muchos y variados los esquemas que han aparecido en la literatura en los que se dan versiones intermedias de los dos tipos de ramificación y acotación paralela expuestos (centralizado y distribuido) [akl82], [alm92], [bof91], [el80], [kin88b], [lee94], [loo92], [mck91], [moh83], [mon87], [rod92], [wah85].

Loots y Smith [loo92] contemplan un esquema completamente centralizado para el problema de la mochila 0-1, donde las comunicaciones se restringen a las cotas inferiores (soluciones parciales). No existe redistribución de la carga de trabajo, pues el esquema de división del problema asegura que en la primera fase la carga asignada a cada procesador es equivalente.

En [mck91] se presentan otras dos estrategias alternativas para la posible paralelización de un esquema de ramificación y acotación: SHO (*(S)elect (H)ighest (O)verall*) y SHA (*(S)elect (H)ighest (A)vailable*). Se tratan de esquemas centralizados con estrategia de distribución dinámica y espacios de decisión y migración globales. Cada uno de los p procesadores almacena en su *heap* local los subproblemas generados. Sin embargo, la elección de los p problemas que se van a tratar en cada etapa se establece en un procesador maestro o administrador. No se asegura que los problemas elegidos por cada procesador estén almacenados en sus *heaps* locales. La distribución de la carga viene decidida desde el maestro, quien guarda la información mínima para conocer todos y cada uno de los problemas que se contemplan en la red.

3.3.6. Medidas para describir el rendimiento (*performance*) de un algoritmo de ramificación y acotación.

En la técnica de ramificación y acotación, el árbol de búsqueda se puede ver reducido dependiendo de la estrategia seleccionada para elegir el problema, así como de la efectividad de las acotaciones realizadas. En el capítulo introductorio se han introducido las medidas generales más utilizadas [tau91]. En este caso se dan algunas más específicas de esta técnica.

Iteraciones (*iter*).

Número de iteraciones realizadas por cada procesador. Es un indicador del balance de la carga de trabajo en la red y de cuantas iteraciones más (menos) realiza el algoritmo paralelo que el secuencial.

Número de nodos expandidos (*ne*).

Como su nombre indica es el número de nodos totales expandidos. Da una medida de la sobrecarga de búsqueda que realiza el algoritmo.

Número de nodos críticos expandidos por iteración (*nce*).

Asumiendo la definición de nodo crítico dada en [lai84], el número de nodos críticos del árbol constituye una cota inferior al número de iteraciones realizadas por el algoritmo secuencial. Por lo tanto es una medida del trabajo efectivo llevado a cabo por el algoritmo paralelo en cada iteración.

3.4. APLICACIONES.

Se desarrolla en este párrafo una aplicación de las diferentes estrategias expuestas al problema del viajante de comercio. Este problema fue introducido en el apartado 3.1.

Little y otros [lit63] desarrollaron un algoritmo del tipo de ramificación y acotación para resolver el problema del ATSP. Cuando se elige un problema no resoluble directamente, su división genera dos subproblemas que representan los caminos que deben incluir o excluir un arco (i, j) determinado. Una heurística (*bestedge*) utilizada para escoger este arco pretende maximizar la cota inferior en el costo del viaje actual excluyendo a este arco. En otras palabras cuando se rompe un problema en subproblemas, el algoritmo examina los arcos que producen el incremento mínimo en la longitud del camino cuando son incluidos y elige aquel que causa el mayor incremento en el costo cuando se excluye.

Un procedimiento *reduce* se usa para encontrar cotas inferiores en el costo del trayecto, teniendo en cuenta las restricciones realizadas. El algoritmo trabaja de la forma siguiente: Para cada vértice i del grafo, se elige la longitud c_i del arco más cercano que llega a i. Si $c_i > 0$, se puede incrementar la cota inferior de esta ruta con esta cantidad, si la misma se substrae del costo de cada arco que lleva a i. Después de realizada esta etapa, se pueden reducir las filas de forma similar. Para cada vértice j en el grafo, se escoge el costo

c_j del arco más corto que sale de j. Si dicho costo es mayor que cero, la cota inferior puede ser aumentada con este valor, si el mismo se resta de las longitudes de los arcos que salen del vértice j. Mediante este esquema el test de la cota inferior actúa sobre los valores computados. Si la cota inferior a este trayecto supera la mejor solución encontrada, el subárbol correspondiente no necesita ser analizado.

En el código 3.8 se da una descripción del algoritmo que utiliza una función de búsqueda primero mejor cota. Obsérvese que la partición de un problema genera subproblemas, concretamente dos, que contienen restricciones adicionales y que por lo tanto son de menor tamaño. Por un lado, incluir un arco reduce el número de estos que deben ser añadidos para completar el trayecto; excluirlo reduce el número de arcos candidatos.

```
Procedure B&BTSP
var heap, P; { heap's element }
begin
  reduce(); { Initially only the root is on the heap }
  while (true) do begin
    P = select_node(); { node with smallest lower bound }
    if is_a_tour(P) then begin
      update_solution();
      exit_while();
    end { if ... }
    else begin
      bestedge(); { edge whose exclusion increases lower bound the most }
      for (cases including and excluding this edge) do begin
        create_node(); { containing additional constraints }
        reduce(); { compute lower bound for the child node }
        insert_on_heap();
      end; { for ... }
    end; { else }
  end; { while ... }
end; { B&BTSP }
```

Código 3.8 Un esquema de ramificación y acotación para el TSP.

Apoyándose en este código como método de resolución de los diferentes problemas a explorar en cada procesador, se especifican en los siguientes apartados los posibles esquemas paralelos implementables y los resultados conseguidos. La paralelización ideal debería examinar sólo aquellos problemas ramificables por el equivalente secuencial e incluso intentar si es posible explorar sólo los nodos críticos [lai84], lo cual puede conllevar anomalías de tipo II (aceleratorias).

Los problemas elegidos abarcan un rango de número de ciudades entre 25 y 40, en intervalos de 5. Los grafos generados son siempre completos, lo que asegura la existencia de una solución y los costos asociados a los arcos se han escogido aleatoriamente en el intervalo [1, 99]. Para cada número de ciudades, se toman como muestra 5 problemas creados bajo el esquema anterior y las medidas tomadas son el promedio entre las mismas para cada tamaño. Estas medidas se han dividido en dos grupos:

Por un lado el tiempo de ejecución, la aceleración (medida como el cociente entre el tiempo de ejecución del algoritmo secuencial y el paralelo sobre el número de procesadores

correspondiente), y la variabilidad de esta aceleración, tomada como el porcentaje de desviación máxima de la aceleración media (cociente entre aceleración más alejada de la media y esta última).

Por el otro, se considera la medida del número de problemas expandidos por el sistema, y en el caso centralizado el porcentaje de estos mismos problemas que se comunican desde el procesador maestro.

En los análisis que se realizan a continuación se han dividido los problemas atendiendo por un lado al tamaño de estos y por otro al número de problemas necesarios para su resolución en el caso secuencial, debido a que para los casos donde el número de problemas no es lo suficientemente grande los algoritmos paralelos no son alimentados con suficientes problemas, lo que repercute en su comportamiento.

Debido a la relación evidente entre las técnicas divide y vencerás y ramificación y acotación, parece en principio bueno utilizar la topología de árbol binario para, al menos, el caso en el que se dispone de un control centralizado. Para el caso distribuido, la red en árbol binario posee dos características adecuadas (G y D) que aseguran un número alto de vecinos para cada nodo y una distancia no muy grande entre los nodos extremos. Como en la técnica anterior se han elegido cuatro tamaños de árbol para los experimentos (altura dos (3 procesadores), altura tres (7), altura cuatro (15) y altura cinco (31).

3.4.1. Un esquema maestro/esclavos (granja de procesadores).

Tomando como punto de partida el pseudocódigo anterior, se puede adoptar un esquema *maestro/esclavos* donde únicamente el maestro tiene la potestad de asignar trabajo a los esclavos, es decir, lleva el control de todos los nodos activos del árbol de búsqueda y es responsable de identificar aquellos de mayor prioridad (en este caso, menor cota inferior). Como en la técnica divide y vencerás para la granja de procesadores, es necesario el uso de procesos ruteros y administradores para las diferentes comunicaciones entre maestro y esclavos. Todo lo comentado para la estrategia anterior es válido en este caso. Se restringe entonces este estudio a los conceptos específicos al tipo de problema y técnica tratados.

Este esquema centralizado es la manera obvia de identificar y distribuir los n problemas de mayor prioridad, manteniendo todos los problemas activos en la memoria del procesador maestro. Sin embargo, los problemas tratados mediante la técnica de ramificación y acotación (por lo menos el TSP) conllevan en la práctica grandes sobrecargas de comunicaciones, ya que el tiempo de comunicación es proporcional al tamaño del mensaje y la cantidad de datos asociados que deben ser enviados entre procesadores suele ser bastante grande. El tamaño del nodo depende de la definición del tipo de problema, pero normalmente requiere al menos O(n) bytes y posiblemente polinomios de orden superior para un problema de tamaño n. En este caso (código 3.9), el tamaño es bastante significativo $O(n^2)$.

```
problem = record
            n : integer;
            w : pointer to array [1..n][1..n] of integer;
            lowerbound : integer;
            row, col : pointer to array [1..n] of integer;
            fwdptr, backptr : pointer to array [1..N] of integer;
            { N is the size of the initial problem }
          end; { problem }
```

Código 3.9 Estructura asociada a un nodo del árbol de búsqueda.

Una posible solución [mck91] que ha sido implementada, es mantener los nodos generados por cada procesador en su propia memoria local y enviar un pequeño descriptor que identifica el problema al procesador maestro, de forma que este último pueda aún identificar los nodos de mas alta prioridad (menor cota inferior). Estos descriptores contienen los valores de prioridad de los problemas generados (cota inferior y número de ciudades por asignar) junto al identificador del procesador que actualmente lo guarda. De esta forma se consigue una reducción temporal al enviar mensajes de tamaño inferior, así como la ampliación del tamaño del problema a explorar al repartir el almacenamiento entre todos los procesadores de la red. Si la fase de selección de problemas fuera completamente síncrona, esto es, se determinaran los n problemas asignados a los n esclavos al mismo tiempo, se asegura la exploración de los problemas más prometedores. Sin embargo, el funcionamiento asíncrono utilizado requiere que el maestro asigne un nodo activo (si existe) a un procesador desocupado tan pronto como éste lo solicite. Un problema adicional que surge de la asincronía es la posibilidad de que el maestro contenga información no actualizada cuando elige el problema a enviar (el procesador que guarda dicho problema puede haber generado otros posteriores cuyos descriptores no han alcanzado aún el maestro). Para evitar esta variación la comunicación se acompaña del descriptor del problema con el fin de identificar el problema en cuestión.

La implementación rigurosa y que se ha utilizado, requiere enviar el nodo de mayor prioridad de cada vez, pero una posible relajación que indican Mckeown et al. es contrastar un determinado porcentaje de mejores trabajos con la finalidad de encontrar alguno de ellos en el procesador solicitante. Incluso en caso de acordar que el procesador que solicita no dispone en su memoria local de un problema prometedor, se podría elegir para su estudio aquel que estuviera almacenado a menor distancia, para ello se puede utilizar un algoritmo que, en caso necesario, explora los problemas admisibles y escoge para su traslado uno que se encuentre a la mínima distancia del procesador solicitante. Sin embargo, en los experimentos realizados esta política más flexible da lugar en la mayoría de los casos a una sobrecarga de cómputo y una alta desviación del árbol de nodos críticos explorados por el algoritmo secuencial.

Cuando alguno de los subproblemas hijos que se generan en el análisis de uno dado no incrementa el valor de la función cota inferior, este subproblema no es comunicado al maestro y el procesador continua trabajando en ese nuevo problema. La razón principal de esta optimización, que fue utilizada por Lüling [lül89], es suponer que al no haber modificación de la cota, el problema sigue estando dentro del rango de los prometedores.

Una técnica más agresiva que hemos introducido, continua con el análisis del problema si este sigue teniendo la mejor cota inferior de entre los problemas almacenados en la memoria local del procesador (si el procesador no dispone de más problemas se comunica de nuevo con el maestro). De nuevo, la probabilidad de que el problema esté entre los prometedores se sigue manteniendo alta. Esta técnica reduce al máximo el número de comunicaciones a realizar, como demuestran los experimentos realizados.

La condición de parada del algoritmo se contrasta mediante una estructura (en concreto, una cola) que almacena los procesadores desocupados que solicitan trabajo. Cuando esta contiene a todos los esclavos y el maestro no dispone de trabajo se finaliza. En caso de que todavía existan nodos a explorar, estos serán asignados a los procesadores libres en el orden en el que ellos lo solicitaron.

Obviamente la consecución de una solución debe ser comunicada de forma inmediata al procesador maestro, sin embargo el esquema de almacenamiento utilizado aconseja también su envío a todos los restantes obreros para la actualización de la carga asignada (incluso en el caso de que el maestro guardara toda la información, esta replicación de la nueva solución podría acelerar la detección de la acotación de parte del árbol de búsqueda).

Resultados.

En los resultados que se exponen, se contrasta dos tipos de medidas: por un lado las medidas temporales (tiempo en segundos, aceleración, etc) y por el otro las medidas relacionadas con el número de problemas (número de problemas generados, porcentaje de número de problemas enviados desde el control). Debido a la relajación introducida que evita comunicar problemas que mantengan cotas inferiores más pequeñas que las almacenadas en la memoria local del procesador en cuestión, el porcentaje de problemas distribuidos desde el maestro es bastante bajo.

	1 PROC.	3 PROC.			7 PROC.			15 PROC.			31 PROC.		
TAM.	TIEMPO	TIEMPO	ACEL.	VARIA	TIEMPO	ACEL.	VARIA	TIEMPO	ACEL.	VARIA	TIEMPO	ACEL.	VARIA
25	5.57	2.98	1.87	0.21	1.17	4.75	0.32	0.88	6.36	0.38	0.85	6.54	0.48
30	24.24	10.61	2.29	0.40	3.69	6.58	0.51	2.56	9.47	0.57	1.71	14.15	0.75
35	62.22	29.62	2.10	0.18	8.32	7.48	0.48	4.02	15.47	0.68	2.76	22.53	0.77
40	191.85	55.85	3.43	0.43	37.13	5.17	0.75	14.62	13.12	0.78	6.05	31.72	0.58

Tabla 3.1 Resultados temporales para los problemas agrupados por tamaño.

Las tablas 3.1 y 3.2 muestran los resultados de tiempo. Se han realizado dos clasificaciones sobre los problemas tratados. En la primera clasificación se agrupan los problemas según el tamaño de los mismos (25, 30, 35 y 40). En la segunda clasificación, cada entrada de la tabla se corresponde con un conjunto de problemas que en su solución

secuencial generan un número de problemas dentro de cierto intervalo (clase)[4]. En ambas tablas las columnas etiquetadas con TIEMPO, ACEL y VARIA contienen el tiempo de ejecución, la aceleración y la variabilidad de esta última para el número de procesadores (PROC.) indicados. En la primera tabla, la primera columna indica el tamaño del problema (número de vértices). En la segunda tabla, la primera columna contiene las entradas para el conjunto de problemas en los que el secuencial genera un número de problemas similar.

	1 PROC.	3 PROC.			7 PROC.			15 PROC.			31 PROC.		
CLA.	TIEMPO	TIEMPO	ACEL	VARIA	TIEMPO	ACEL	VARIA	TIEMPO	ACEL	VARIA	TIEMPO	ACEL	VARIA
1	7.41	3.66	2.02	0.23	1.66	4.46	0.28	1.16	6.41	0.39	1.13	6.57	0.50
2	35.79	16.36	2.19	0.20	5.58	6.41	0.19	3.38	10.60	0.27	2.39	14.97	0.45
3	221.92	70.29	3.16	0.16	39.83	5.57	0.73	15.50	14.32	0.63	6.22	35.69	0.23

Tabla 3.2 Resultados temporales para los problemas agrupados por número de nodos explorados.

A pesar de que los resultados de esta primera división son satisfactorios, existe una alta variabilidad entre las medidas recogidas (como demuestra la columna de variabilidad). En la tabla 3.2 los grupos tienen en cuenta el número de problemas generados por el secuencial y demuestran un mejor comportamiento general, tanto para las aceleraciones (en uno de los grupos se obtienen resultados superlineales) como para las variaciones resultantes.

En las tablas 3.3 y 3.4 se comparan el número de problemas explorados. Al igual que antes, los grupos están definidos o por el número de vértices a recorrer o por la vecindad en el número de problemas generados por el algoritmo secuencial. En las tablas se presentan: número promedio de problemas generados del secuencial (TS), número promedio de problemas generados del paralelo (TP), porcentaje que este representa respecto al secuencial (PORC), y porcentaje del número de problemas promedio asignados desde el control frente al total (CONT), para cada una de los árboles binarios utilizados.

	1 PROC.	3 PROC.			7 PROC.			15 PROC.			31 PROC.		
TAM.	TS	TP	PORC	CONT	TP	PORC	CONT	TP	PORC	CONT	TP	PORC	CONT
25	228	261	1.14	0.25	268	1.18	0.21	345	1.51	0.19	458	2.01	0.22
30	598	617	1.03	0.38	658	1.10	0.30	740	1.24	0.25	895	1.50	0.21
35	895	932	1.04	0.33	994	1.11	0.29	1041	1.16	0.25	1219	1.36	0.20
40	1880	1962	1.04	0.30	1983	1.06	0.32	2066	1.10	0.27	2266	1.21	0.22

Tabla 3.3 Número de problemas generados para la clasificación por número de vértices del problema.

[4] Los problemas se han agrupado en tres intervalos con los rangos siguientes: 1 entre 100 y 400, 2 entre 400 y 1100 y 3 entre 1100 y 3500.

	1 PROC.	3 PROC.			7 PROC.			15 PROC.			31 PROC.		
CLA.	TS	TP	PORC	CONT	TP	PORC	CONT	TP	PORC	CONT	TP	PORC	CONT
1	224	250	1.12	0.24	291	1.30	0.18	347	1.55	0.18	510	2.28	0.19
2	690	720	1.04	0.34	759	1.10	0.28	860	1.25	0.22	1063	1.54	0.18
3	2277	2363	1.04	0.32	2403	1.06	0.30	2587	1.14	0.25	2581	1.13	0.23

Tabla 3.4 Número de problemas generados para la clasificación por número de nodos explorados por el secuencial.

A pesar de que los porcentajes (columna PORC) de los problemas expandidos son equivalentes en ambas tablas, se observa en la segunda agrupación un comportamiento más acorde con la relación número de problemas del secuencial (TS) respecto al número de procesadores utilizados (cuanto más pequeño es este cociente, mayor es el porcentaje).

3.4.2. Un esquema completamente distribuido.

Si bien el esquema centralizado debe asegurar una mayor fiabilidad a la hora de elegir los problemas correctos, es necesario reconocer que cuando el diámetro de la topología crece (redes de procesadores suficientemente grandes) las comunicaciones con el maestro se van a convertir en una losa difícil de sobrellevar. En estas condiciones parece factible ensayar un esquema donde todos los procesadores realicen ramificaciones sobre los problemas que se le asignan en propiedad, esto es, siguiendo la estrategia de selección elegida a priori tomen de su estructura el problema más prometedor y lleven a cabo sobre él los cálculos correspondientes. Como se expresó en el apartado 3.3, la distribución introducida conlleva una serie de decisiones adicionales que se deben resolver: minimización de la sobrecarga de búsqueda, balance de la carga, criterios de inicialización y detección de la condición de parada.

Al igual que ocurría en el caso centralizado la detección de una mejor solución en un determinado procesador debe ser comunicada de forma inmediata al resto de la red, con el fin de realizar las podas correspondientes lo antes posible (si así fuera el caso).

El compromiso necesario entre cantidad de carga asignada a un procesador y calidad de la misma afecta de manera fundamental a la posible aceleración obtenible con el código paralelo distribuido. Como la selección de los problemas de las estructuras locales a cada procesador no asegura estar examinando las ramas del árbol que contienen nodos críticos, se deben utilizar estrategias que intenten equilibrar la carga asignada a cada uno de ellos y con ello intentar suministrarles aquellos problemas que, realmente deban ser investigados (para ser más preciso, los explorados por el algoritmo secuencial). Las investigaciones que han sido llevadas a cabo con anterioridad, siguen dos esquemas bien definidos:

- por un lado algunos autores [lül89], [lül91], [mck91] tratan de redistribuir la carga de aquellos procesadores a los que se les ha dotado con abundante cantidad de

trabajo, entre los vecinos de estos que han finalizado sus tareas (la cantidad de problemas enviados es siempre una función de la carga asociada). Sin embargo esta única redistribución no puede asegurar el balanceo de los problemas críticos, por ello se introduce también un intercambio de problemas entre los procesadores adyacentes en la red (la asiduidad de estas conversaciones se suele obtener o de las diferencias de peso entre los procesadores, o de algún parámetro controlado por el usuario).

- por otro lado existen investigadores [cap92], [qui90], [tro89] que tratan de realizar la buena distribución de los problemas clave mediante la asignación de algunos de los generados por cada procesador, no a la estructura local asociada sino que experimentan con envíos a otros procesadores (vecinos o no) atendiendo a diferentes reglas, tanto para la elección del(los) problema(s) a enviar como el(los) procesador(es) destino.

Como es obvio es necesario mantener un equilibrio entre el número de comunicaciones realizadas y la carga de los procesadores con trabajo adecuado. Dos han sido las estrategias implementadas, una para cada uno de los esquemas anteriores. En ambas los espacios de decisión y migración se han elegido locales. Las reglas básicas aplicadas en cada caso son las siguientes:

En el primero, cada procesador almacena en su memoria local el peso asociado a cada uno de sus vecinos y el suyo propio. Esta cantidad ha sido definida como la cota inferior del mejor problema. Una variación de alguna de estas cantidades provoca el intento de una comunicación a través de la siguientes reglas:

- cada vez que se modifica el peso local, se comunica a los vecinos. Si el peso ha disminuido se procede, antes de la comunicación, a contrastar los pesos de los vecinos con la carga local (en este caso con el segundo mejor problema almacenado). Se comprueba si es necesario enviarles uno de los mejores problemas que se guardan.

- cada vez que se recibe un peso de uno de los vecinos se actualiza y se contrasta con la carga local (en concreto cota inferior del segundo mejor problema). Si esta última es menor se envía el problema al vecino.

Al menos una de estas reglas se cumple cada vez que se realiza una comunicación o se genera un problema. De esta forma el número de mensajes que viajan a través de la red se convierte en una carga muy difícil de llevar. Para minimizar estos conflictos, se controla la asiduidad de las informaciones por cada canal (tanto para solicitar como para enviar trabajo) llevando la cuenta de los momentos en que se realizaron las últimas actualizaciones (solicitudes ó problemas por separado) y no permitiendo la concreción de ninguna otra del mismo tipo, hasta pasado un intervalo prefijado.

En la segunda estrategia el concepto de reequilibrio de carga se apoya por completo en la distribución de problemas en cuanto son generados. Debido a este hecho el concepto de solicitud de trabajo se anula y los paquetes que se envían son siempre problemas (sólo los

intentos de finalizar la ejecución son tratados, en caso que fallen como solicitud de trabajo). Para no provocar embotellamientos producidos por la repartición de problemas que realiza cada uno de los procesadores que disponen de trabajo, cada vez que se genera un problema se comunica uno de los mejores (en concreto el segundo) a uno sólo de los vecinos (el vecino elegido se obtiene de ciclar entre todos los posibles) y no a todos ellos.

Una disyuntiva se crea entre la dotación, lo más rápida posible, a la red de suficiente trabajo para aprovechar el número de procesadores de los que se dispone y la necesaria certeza de que los problemas asociados pertenecen al rango de aquellos que deben ser realmente explorados. Debido a que el rango que se ha experimentado (máximo 31 procesadores) y el diámetro de la red (máximo 8) no es excesivamente grande, se opta por el esquema más simple en el que inicialmente sólo uno de los procesadores está trabajando y en cada etapa cada uno de ellos activa a otro procesador.

La falta de control en estos esquemas distribuidos de la carga asociada a la red en su conjunto, obliga a la utilización de un código para detectar la finalización del algoritmo de ramificación y acotación paralela. Como la topología utilizada es un árbol y la mayoría de los algoritmos de detección se basan en la definición de árboles generadores [top84], implícitamente se tiene definido el camino a explorar. Para su ejecución se envían dos tipos de *tokens*, rojo y amarillo, el primero indica que el status del nodo es ocioso mientras que el segundo comunica que el procesador permanece aún ocupado. El algoritmo actúa según las siguientes reglas:

- Todos los nodos, excepto las hojas, deben contrastar su status dos veces. El primero, cuando reciben la señal de su padre (menos el raíz) y el segundo, cuando han recogido las señales de sus hijos.

- Cualquier señal que salga de un nodo convertido en principal siempre debe recibir respuesta, tanto afirmativa como negativa. Esta prohibido iniciar una nueva ronda de consultas, hasta que no se haya recibido el reconocimiento de la anterior. Con este sistema se evita que en la red viaje más de una señal de este tipo.

- El procesador raíz del árbol es el encargado de inicializar las comprobaciones pertinentes con el envío en los casos oportunos del *token* prefijado de antemano (en concreto, cuando ha agotado los problemas que se le han asignado y está seguro de no haber enviado otro del cual todavía no tiene conocimiento).

- Cualquier nodo que reciba de su padre una señal roja y tiene trabajo, responde inmediatamente con una señal amarilla. Los nodos no terminales recolectan las señales de sus hijos, antes de responder a su padre. Esta señal debe viajar de vuelta, hasta la raíz del árbol.

Las redes en árbol se adaptan de manera natural a estos algoritmos minimizando el recorrido de las señales de finalización. Además existe una optimización, ya utilizada por Mckeown, que consiste en tratar una de estas señales como solicitud de trabajo, cuando su

comprobación produce una respuesta negativa.

Resultados.

Los experimentos tratan de analizar las dos estrategias más utilizadas en los esquemas distribuidos. Se ha elegido enviar siempre el segundo problema con mejor (menor) cota inferior de los disponibles, debido a que en la mayoría de los trabajos ya realizados este es el esquema usado. En ambos casos, se comunica un único problema de cada vez, para evitar colapsos en la red. Los problemas se han agrupado atendiendo al número de problemas generados por el secuencial, ya que como se ha observado para el caso centralizado, esta clasificación da un mejor punto de vista sobre la eficiencia del esquema aplicado.

	1 PROC.	3 PROC.			7 PROC.			15 PROC.			31 PROC.		
CLA.	TIEMPO	TIEMPO	ACEL	VARIA	TIEMPO	ACEL	VARIA	TIEMPO	ACEL	VARIA	TIEMPO	ACEL	VARIA
1	7.41	2.73	2.72	0.11	1.59	4.67	0.37	1.40	5.31	0.44	1.40	5.30	0.53
2	35.79	11.57	3.09	0.08	5.50	6.51	0.22	3.44	10.39	0.46	2.99	11.98	0.90
3	221.92	60.41	3.67	0.12	24.77	8.96	0.18	11.60	19.13	0.23	7.22	30.75	0.35

Tabla 3.5 Resultados temporales para la estrategia basada en la redistribución atendiendo al peso.

La primera política analizada será la de la redistribución de la carga atendiendo al peso asignado a cada nodo. En la tabla 3.5 se contrastan las medidas temporales observadas (tiempo en segundos, aceleración y variabilidad) y en la tabla 3.6, se analizan los problemas expandidos. Los intentos de comunicación entre cualesquiera dos nodos vecinos, para enviar un paquete de un tipo determinado, son realizados como mínimo en intervalos de una centésima de segundo (0.01 seg.) desde el último envío efectuado de esta misma clase al mismo procesador. Aquellos intentos producidos antes de superar este intervalo son abortados. De entre los dos problemas generados en el análisis de un problema dado, el menor de ambos que no incremente la cota inferior del original, no es objeto de posible comunicación. Se mantiene en la aplicación para su expansión inmediata.

	1 PROC.	3 PROC.		7 PROC.		15 PROC.		31 PROC.	
CLA.	TS	TP	PORC	TP	PORC	TP	PORC	TP	PORC
1	224	240	1.07	268	1.20	410	1.83	726	3.25
2	690	737	1.07	765	1.11	852	1.23	1128	1.63
3	2277	2443	1.07	2623	1.15	2642	1.16	2760	1.21

Tabla 3.6 Número de problemas analizados para la estrategia basada en la redistribución atendiendo al peso.

Existe una gran relación entre el porcentaje de problemas expandidos y la reducción

de tiempos obtenida. Este tipo de algoritmo asegura resultados eficientes cuando el número de problemas explorados se mantiene cercano al del secuencial.

	1 PROC.	3 PROC.			7 PROC.			15 PROC.			31 PROC.		
CLA.	TIEMPO	TIEMPO	ACEL	VARIA	TIEMPO	ACEL	VARIA	TIEMPO	ACEL	VARIA	TIEMPO	ACEL	VARIA
1 CH	7.41	3.17	2.34	0.13	1.73	4.28	0.27	1.44	5.15	0.39	1.44	5.14	0.40
1 SI		3.04	2.44	0.18	1.74	4.25	0.30	1.46	5.09	0.41	1.38	5.38	0.40
2 CH	35.79	15.75	2.27	0.15	6.73	5.32	0.21	4.27	8.39	0.46	3.83	9.35	0.65
2 SI		13.09	2.73	0.15	6.11	5.86	0.28	4.06	8.80	0.48	3.57	10.02	0.53
3 CH	221.92	70.49	3.15	0.12	28.06	7.91	0.27	14.22	15.60	0.13	11.37	19.51	0.26
3 SI		56.84	3.90	0.26	24.50	9.06	0.16	13.08	16.97	0.17	9.17	24.20	0.31

Tabla 3.7 Resultados temporales para la estrategia basada en la redistribución tan pronto se genera el trabajo.

Para la segunda estrategia, la que basa el equilibrio en la distribución de la carga tan pronto como la misma es generada, se han realizado dos ensayos. Las diferencias entre ambas radican, únicamente en los problemas posibles candidatos a ser distribuidos entre los vecinos. En el primer caso, sólo los problemas generados que no incrementan el valor de la cota inferior son excluidos. Estas aparecen en las tablas correspondiendo a las filas etiquetadas CH (Charlatán). En el otro caso, este grupo se amplia con todos aquellos que no superan la cota inferior del siguiente mejor problema que guarda el nodo en cuestión. En las tablas aparecen etiquetados con SI (Silencioso). Se trata del mismo esquema que fue utilizado para el caso centralizado. El motivo de tal reducción, es limitar la sobrecarga de comunicaciones que se produce en el caso más simple, utilizando una regla similar al intervalo temporal que se usa en la estrategia anterior. Si bien la reducción que se observa no es excesiva, si es cierto que su comportamiento presenta la característica de mejorar cuanto mayor es la red. Las tablas 3.7 y 3.8 muestran los resultados para las dos experiencias realizadas (tiempos obtenidos y número de problemas generados).

	1 PROC.	3 PROC.		7 PROC.		15 PROC.		31 PROC.	
CLA.	TS	TP	PORC	TP	PORC	TP	PORC	TP	PORC
1 CH	224	260	1.16	285	1.27	362	1.62	529	2.37
1 SI		272	1.22	300	1.34	385	1.72	521	2.33
2 CH	690	792	1.15	878	1.27	978	1.42	1274	1.85
2 SI		767	1.11	803	1.16	928	1.34	1198	1.74
3 CH	2277	2402	1.05	2614	1.15	2552	1.12	3000	1.32
3 SI		2464	1.08	2444	1.07	2544	1.12	2909	1.28

Tabla 3.8 Número de problemas analizados para la estrategia basada en la redistribución tan pronto se genera el trabajo.

De nuevo como en el esquema distribuido anterior, las ganancias temporales son directamente proporcionales al número de problemas expandidos. Cuanto más se incremente este último número, peores resultados obtenidos. En lo referente a la comparación entre los dos códigos de esta misma estrategia, es obvia la mejoría que introduce la reducción del número de candidatos a enviar, que es más clara cuanto mayor es el problema a resolver.

3.4.3. Comparaciones gráficas entre los tres esquemas presentados.

Se presentan dos tipos de gráficas para cada uno de los tres intervalos en los que se han dividido los experimentos realizados. El primer tipo contrasta las aceleraciones obtenidas frente al número de procesadores utilizados, mientras que en el segundo se muestran el número de problemas expandidos para los tres códigos escogidos. Los tres algoritmos comparados son: el esquema centralizado (Central) y dos esquemas distribuidos. Estos dos últimos son representantes de las dos estrategias introducidas para el rebalance de la carga. El primero (Distr1) asocia pesos a la carga de cada procesador e intercambia trabajo a intervalos temporales. El segundo (Distr2) se basa en el envío de los problemas tan pronto como son generados. De entre las dos opciones posibles (Charlatán y Silencioso) se ha elegido el esquema que reduce las comunicaciones (Silencioso).

La razón fundamental de presentar este tipo de confrontaciones, es la relación que existe en estos algoritmos entre tiempo de cómputo y número de problemas analizados. En las figuras presentadas es obvia la relación existente entre ambos parámetros. El algoritmo centralizado es el que mejores resultados presenta, aunque también es quien más se ve afectado por resultados anómalos. En la figura 3.7 se observa que la aceleración se mantiene por debajo de la linealidad en los primeros tres casos, mientras que pasa a superar este valor para 31 procesadores. Este hecho concuerda con el número de problemas generados para este intervalo (figura 3.8), ya que entre 15 y 31 procesadores se produce un decremento muy leve en el número de problemas.

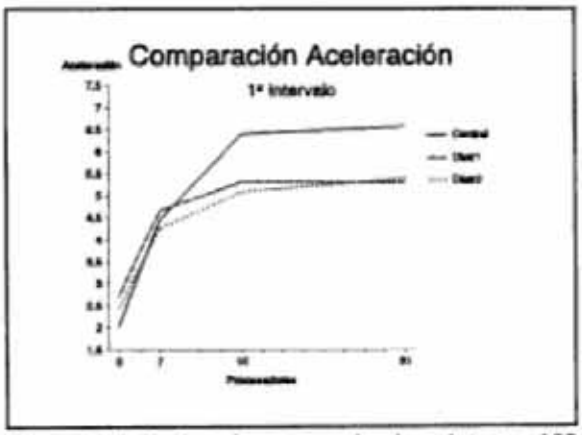

Figura 3.3 Aceleraciones para la clase 1 (entre 100 y 400 problemas).

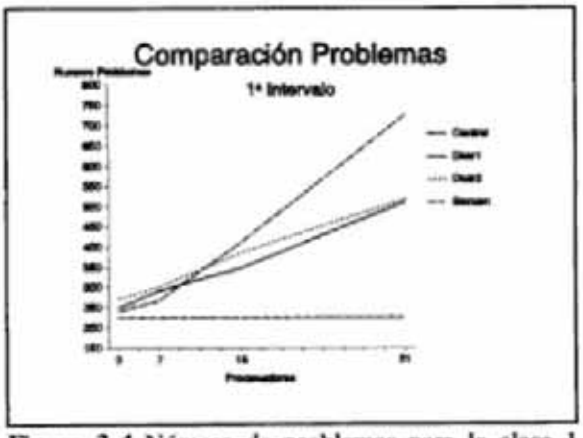

Figura 3.4 Número de problemas para la clase 1 (entre 100 y 400 problemas).

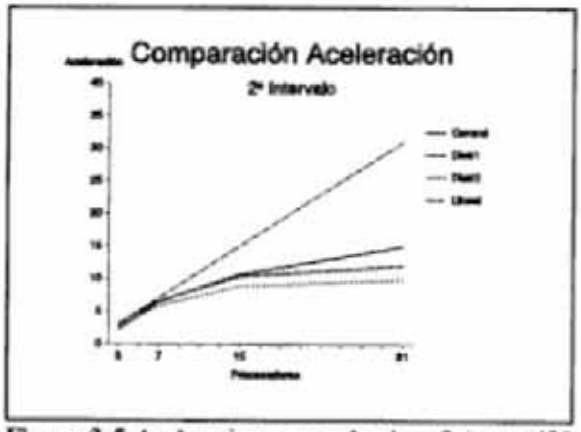

Figura 3.5 Aceleraciones para la clase 2 (entre 400 y 1100 problemas).

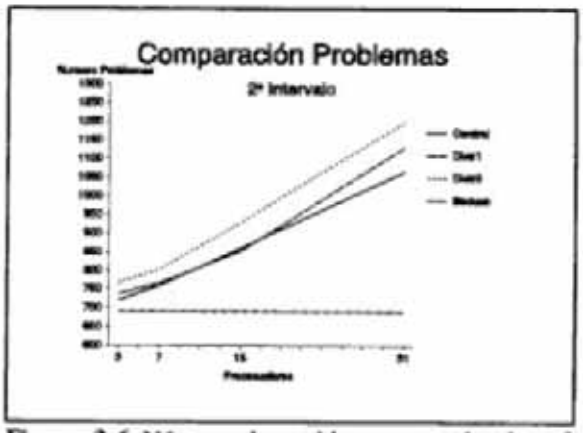

Figura 3.6 Número de problemas para la clase 2 (entre 400 y 1100 problemas).

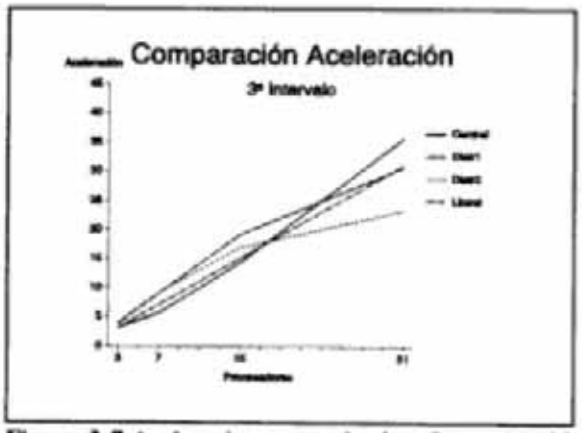

Figura 3.7 Aceleraciones para la clase 3 (entre 1100 y 3500 problemas).

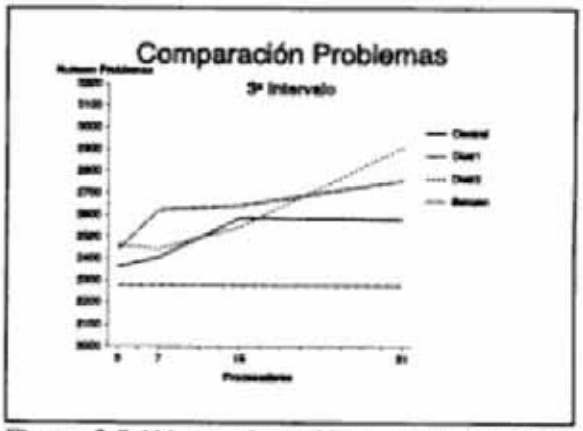

Figura 3.8 Número de problemas para la clase 3 (entre 1100 y 3500 problemas).

3.5. BIBLIOGRAFIA.

[aho74]. A.V. Aho, J.E. Hopcroft and J.D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley. Reading, MA.

[aho83]. A.V. Aho, J.E. Hopcroft and J.D. Ullman. *Data structures and algorithms*. Addison-Wesley. Reading, MA.

[akl82]. S.G. Akl, D.T. Barnard and R.J. Doran. Design, analysis, and implementation of a parallel tree search algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI 4 (2), pp 192-203.

[alm92]. F. Almeida. Paralelismo: Utilidades y aplicaciones en programación combinatoria. *Memoria de Licenciatura*. Dept. Estadística, Inv. Operativa y Computación, Univ. de La Laguna.

[bof91]. T.B. Boffey and P. Saeidi. Parallel branch-and-bound using shared memory. *Technical Report*, SCM Dept. University of Liverpool.

[bur82]. F.W. Burton. G.P. Mckeown, V.J. Rayward-Smith and M.R. Sleep. Parallel processing and combinatorial optimisation. *Proceedings of the Combinatorial Optimisation III Conference*, pp. 19-36. Stirling.

[cap92]. M. Capel and A. Palma. A programming tool for distributed implementation of branch-and-bound algorithms. *Parallel Computing and Transputer Applications*, pp. 138-147. Barcelona. CIMNE, IOS Press.

[chi90]. S. Chiba, H. Honda, H. Maezawa, T. Tsukioka, M. Uematsu, Y. Yoshida and K. Maeda.Divide and conquer in parallel procesing. *Proceedings of the 3rd Transputer/Occam International Conference*, pp. 279-293. Tokyo, Japan. IOS Press.

[eld80]. O.I. El-Dessouki and W.H. Huen. Distributed enumeration on network computers. *IEEE Transactions on Computers* C-29 (9) pp. 818-825.

[hel70]. M. Held and R.M. Karp. The traveling salesman problem and minimum spanning trees. *Operations Research* 18, pp. 1138-1162.

[horZ78]. E. Horowitz and S. Sahni. *Fundamentals of computer algorithms*. Computer Science Press. Potomac, MD.

[iba87]. T. Ibaraki. *Enumerative approaches to combinatorial optimization. Part I-II*. J.C. BALTZER AG. Basel, Switzerland.

[kin88a]. G.A.P. Kindervater and J.K. Lenstra. Parallel computing in combinatorial optimization. *Annals of Operations Research* 14, pp. 245-289.

[kin88b]. G.A.P. Kindervater and H.W.J.M. Trienekens. Experiments with parallel algorithms for combinatorial problems. *European Journal of Operational Research* 33, pp. 65-81.

[lai84]. T. Lai and S. Sahni. Anomalies in parallel branch-and-bound algorithms. *Communications of the ACM* 27 (6), pp. 594-602.

[lai85]. T. Lai and A. Sprague. Performance of parallel branch-and-bound algorithms. *IEEE Transactions on Computers* C-34 (10), pp. 962-964.

[law85]. E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys. *The traveling salesman problem. A guided tour of combinatorial optimization*. John Wiley & Sons. England.

[lee94]. Y.N. Lee, G.P. Mckeown and V.J. Rayward-Smith. Solving the convoy movement problem using branch-and-bound on a network of transputers. *Transputers Applications and Systems '94*, pp 786-796. Italy. IOS Press, Ohmsha.

[lit63]. J.D.C. Little, K.G. Murty, D.W. Sweeney and C. Karel. An algorithm for the travelling salesman problem. *Operations Research* 11 (6), pp. 972-989.

[loo92]. W. Loots and T.H.C. Smith. A parallel algorithm for the 0-1 knapsack problem. *International Journal of Parallel Programming* 21 (5), pp. 349-362.

[lül89]. R. Lüling and B. Monien. Two strategies for solving the vertex cover problem on a transputer network. *3rd International Workshop on Distributed Algorithms*, LNCS392, pp. 160-171.

[lül91]. R. Lüling, B. Monien and F. Ramme. Load balancing in large networks: A comparative study. *Proceedings of the 3th IEEE Symposium on Parallel and Distributed Processing*.

[mar90]. S. Martello and P. Toth. *Knapsack problems: Algorithms and computer implementations*. John Wiley & Sons. England.

[mck91]. G.P. Mckeown, V.J. Rayward-Smith, A. Rush and H.J. Turpin. Using a transputer network to solve branch-and-bound problems. *Proceedings of the TRANSPUTING '91 Conference*, pp. 781-800. IOS Press.

[moh83]. J. Mohan. Experience with two parallel programs solving the travelling salesman problem. *Proceedings of the 1983 International Conference on Parallel Processing*, pp. 191-193. IEEE, NY.

[mon87]. B. Monien and O. Vornberger. Parallel procesing of combinatorial search trees. *Proceedings of International Workshop on Parallel Algorithms and Architectures*, pp. 60-69. Springer Verlag Berlin.

[ort93]. M. Ortega. Curso de formación de profesorado sobre programación paralela. *Dept. de Estadística, I.O. y Computación*.

[qui90]. M.J. Quinn. Analysis and implementation of branch -and-bound algorithms on a hypercube multicomputer. *IEEE Transactions on Computers* C-39 (3), pp. 384-387.

[qui94]. M.J. Quinn. *Parallel Computing: Theory and practice*. McGraw-Hill. OR.

[rod92]. C. Rodríguez, F. Garcia, C. León y F. Almeida. A parallelization of a branch and bound algorithm for the set covering problem. *Proceedings of the VI Meeting of the EURO Working Group on Location Analysis*, pp. 183-193.

[sys83]. M.M. Syslo, N. Deo and J.S. Kowalik. *Discrete optimization algorithms with pascal programs*. Prentice-Hall. Englewood Cliffs, NJ.

[tau91]. A. Taudes and T. Netousek. Implementing branch-and-bound algorithms on a cluster of workstations. A survey, some new results and open problems. *Proceedings of the workshop on parallel algorithms and Tranputers for optimization*, pp. 79-102. Siegen. Springer-Verlag.

[top84]. R.W. Topor. Termination detection for distributed computations. *Information Processing Letters* 18, pp. 33-36.

[tro89]. J.M. Troya and M. Ortega. A study of parallel branch-and-bound algorithms with best-bound-first search. *Parallel Computing* 11, pp 121-126.

[vor86]. O. Vornberger. Implementing branch-and-bound in a ring of processors. *Proceedings of CONPAR 86*, LNCS 237, pp. 157-164. Springer Verlag.

[vor88]. O. Vornberger. Load balancing in a network of transputers. *Distributed Algorithms*, LNCS 312, pp. 116-126. Springer Verlag.

[wah85]. B.W. Wah, G. Li and C.F. Yu. Multiprocessing of combinatorial search problems. *Computer* 18 (6), pp. 93-108.

CAPITULO IV:

Programación Dinámica

4.1. INTRODUCCION.

Con frecuencia, en la descomposición de un problema en un número arbitrario de subproblemas, estos últimos no tienen porqué ser independientes entre sí. Es más, a veces los problemas generados se repiten. Si se conserva la solución de cada problema resuelto, el espacio de búsqueda se puede ver ampliamente reducido. La *programación dinámica* (*dynamic programming*) se aprovecha de este hecho y de lo que se conoce con el nombre de *principio de optimalidad* [bel57], que permite crear relaciones de recurrencia entre los problemas y sus descendientes [sed83].

4.2. LA TECNICA DE LA PROGRAMACION DINAMICA.

Como en los métodos precedentes, el esquema de partida es siempre el mismo, un algoritmo enumerativo. Sin embargo, en la programación dinámica no se tiene en cuenta si se necesita realmente un subproblema particular en la solución total (se resuelven todos y se almacenan sus soluciones para ser usadas en la resolución de los problemas ancestros). Si en el caso de la ramificación y acotación era la etapa de combinación la que se obviaba, en este caso, la etapa de ramificación se supone creada a priori. Se parte directamente desde las hojas del árbol, y se va subiendo en el árbol hasta la raíz.

La forma de un algoritmo de programación dinámica puede variar, pero hay un esquema común que siempre se repite: una tabla a rellenar y un orden específico en el cual se hacen las entradas. A la formación de la tabla es a lo que se conoce con el nombre de programación dinámica, nombre procedente de la teoría de control [smi91].

A pesar de que no existe un algoritmo universal que se pueda aplicar a todos los problemas, como en los capítulos anteriores se intentará crear un pseudocódigo que exprese su modo de funcionamiento (código 4.1).

```
Entrada: Problema P0
Salida:  Resultado f[|x1|]..[|xn|]
Método:  Procedure DynamicProgramming;
         var i1, i1', i2, i2', ..., in, in';
         begin
           initialize(f);
             for i1 in set(x1) do
               for i2 in set(x2) do
                 ......
                   for in in set(xn) do
                     f[i1]..[in] := optimize g(f[i1']..[in']);
                                    0≤i1'≤i1, ..., 0≤in'≤in y existe k / ik'<ik
         end; { DynamicProgramming }
```

Código 4.1 Esquema de la técnica programación dinámica.

Se supone introducida una relación de orden entre los elementos de los conjuntos, el mismo orden que se utiliza para rellenar la tabla asociada.

Se disminuye la dificultad del problema a resolver en cada nodo (en cada uno de los bucles anidados se fija una variable, reduciendo al final el problema a una única variable). Normalmente, es más fácil resolver varios problemas en una variable que un problema con varias variables.

Para entender mejor el método se introducen como ejemplos, sendos algoritmos para resolver los problemas de la mochila 0-1 y los caminos mínimos entre cualesquiera par de vértices de un grafo dirigido (código 4.2).

```
Mochila 0-1

for y := 0 to b do f[0][y] := 0;
for i := 1 to n do
  for y := 1 to b do
    if (y >= a[i]) then
      f[i][y] := max (f[i-1][y],
                      f[i-1][y-a[i]]+c[i])
    else
      f[i][y] := f[i-1][y];
```

```
Caminos mínimos

{ Initially, A is adyacency matrix }
{ for a path with highest vertex k }
for k := 1 to n do
  { for all pairs of vertices }
  for i := 1 to n do
    for j := 1 to n do
      A[i][j] := min (A[i][j],
                      A[i][k]+A[k][j]);
```

Código 4.2 Algoritmos para los problemas de la mochila 0-1 y caminos mínimos.

4.2.1. El principio de optimalidad.

Los algoritmos basados en la técnica de programación dinámica calculan la solución óptima a un problema mediante secuencias de decisiones óptimas hasta ese momento. En los algoritmos ávidos (*greedy*), la decisión que se toma sólo depende de información local y nunca provoca la generación de resultados erróneos, por lo tanto basta generar una decisión en cada estado. Sin embargo, para otro tipo de problemas es necesario ensayar varias posibilidades en cada caso (mediante la fuerza bruta, se enumerarían todos). La eliminación de algunas secuencias que posiblemente pueden no ser óptimas, reduce el espacio a explorar. La generación de las soluciones óptimas se basa en lo que se conoce como *principio de optimalidad*, que puede ser enunciado de las siguientes formas: *Toda subsecuencia de una secuencia óptima es óptima* o *Una política óptima debe estar compuesta de subpolíticas óptimas*.

Para los dos problemas elegidos como ejemplo, el principio de optimalidad asegura lo siguiente:

a) Mochila 0-1

Sea $x_1', \ldots x_n'$, una solución óptima para la mochila de capacidad b y n objetos. Para cada j, $1 \leq j \leq n$, $x_1', \ldots, x_j'$ y $x_{j+1}', \ldots x_n'$ deben ser soluciones óptimas a los problemas $\sum_{i=1}^{j} a_i x_i'$ y $b - \sum_{i=j+1}^{n} a_i x_i'$.

b) Caminos mínimos

Sea k cualquier vértice intermedio de un camino de longitud mínima (i, ..., k, ..., j) de i a j. Para cada k, $i \leq k \leq j$, los caminos (i, .., k) y (k, .., j) deben ser caminos mínimos de i a k y de k a j.

Este principio permite una definición recursiva de la solución a cualquier problema, en función de los datos iniciales suministrados (los problemas son del mismo tipo, lo que varía son los datos). Además, se asegura que la decisión tomada en cada momento como mejor, se mantiene indefinidamente óptima para los datos de entrada.

En los dos modelos expuestos, se pueden obtener relaciones de recurrencia entre los problemas generados.

Para el primer caso (Mochila 0-1), el valor óptimo f[i][y] se obtiene de otros dos valores óptimos f[i-1][y] y f[i-1][y-a[i]], sin más que fijar la variable x_n a los dos posibles valores (0 o 1)[1].

En el caso de los caminos mínimos, la distancia mínima de un vértice i a otro vértice j A[i][j] se alcanza comparando ese valor con este otro A[i][k] + A[k][j], donde k varía desde n hasta 1 y se supone que A[i][j], A[i][k] y A[k][j] contienen los caminos más cortos desde el vértice i, (i), (k) al vértice j, (k), (j), no pasando por vértices con índices superiores a k.

4.2.2. Construcción de tablas.

Si bien la aplicación del principio de optimalidad es básica para la reducción del cálculo de las posibles secuencias no óptimas, su utilidad quedaría bastante restringida si no se guardaran los resultados ya computados en una tabla. Obsérvese que con las reglas de recurrencia se crean problemas que pueden ser iguales. Por lo tanto sólo es necesario computarlos una vez y que todos los otros problemas implicados recojan la solución. Este empleo de tablas de almacenamiento sugiere reconvertir las ecuaciones recursivas en un programa iterativo[2].

4.2.3. Relaciones de dominancia.

Algunas veces es posible reducir aún más la exploración a realizar, haciendo uso de las relaciones de dominancia explicadas en el capítulo de ramificación y acotación. Si se representa geométricamente el problema tomando como ejes las variables del problema, al hacer la proyección sobre alguno de ellos, se obtienen funciones del tipo *continuas con saltos*. Por tanto basta con computar los valores de la tabla donde se producen esos saltos, ya que los demás están dominados por éstos (se supone que se ha introducido el orden

[1] Si la variable x_i toma el valor 1, hay que añadir al valor óptimo de la mochila con i-1 objetos el beneficio del objeto i (c[i]).

[2] Si se mantuvieran tanto el esquema recursivo como la tabla, estaríamos ante un esquema divide y vencerás optimizado por la salvaguarda de los valores computados.

creciente entre los valores de las variables).

Un ejemplo de esta optimización lo tenemos en el problema de la mochila 0-1. L función f[1] toma el salto en un único punto, concretamente a[1] con un valor de c[1] (s puede representar completamente mediante el par (c[1], a[1])). Por motivos de inicialización es necesario también el primer valor (0, 0). Por lo tanto, la función se puede definir de siguiente modo.

$$f[1][y] = \begin{cases} 0, & y < a[1] \\ c[1], & a[1] \le y \le b \end{cases}$$

La función f[2] toma los dos valores (0, 0) y (c[1], a[1]), y se le aplica el mism proceso. Si se observa la regla de recurrencia (f[2][c] = max {f[1][c], f[1][c-a[2]]+p[2]}) los valores óptimos para $x_2 = 0$ son los valores recibidos y para $x_2 = 1$ se pueden obtene sin más que contrastar con el par asociado (a[2], c[2]), obteniendo los valores (0+c[2] 0+a[2]) y ([c1]+c[2], a[1]+a[2]). Los saltos de la función f[2] serán (0, 0), (a[1], c[1]) (a[2], c[2]), (a[1]+a[2], c[1]+c[2]), supuesto que a[1] ≤ a[2]. Sin embargo, puede ocurri que c[1] ≥ c[2] con lo cual no existe salto en a[2], o lo que es lo mismo el par (a[1], c[1] *domina* al par (a[2], c[2]). En este caso, sólo es necesario utilizar para el cómputo de lo posibles saltos de la función f[3], los valores *no* dominados ((0, 0), (a[1], c[1]), (a[1]+a[2] c[1]+c[2])). En general, si se denota por S_{i-1} el conjunto de los pares donde se produce saltos en la función f[i-1], el conjunto S_i estará formado por $S_{i-1} \bigcup S_i^{i-1}$, donde el conjunt $S_i^{i-1} = \{ (a,c) \mid (a-a[i], c-c[i]) \in S_{i-1} \}$, y se han eliminado los pares dominados.

El código aplicable para la resolución del problema, no necesita ahora la tabl completa f[n][b]. En su lugar, se utiliza una representación de conjuntos. En la versió óptima sólo son imprescindibles dos conjuntos, sin embargo en aras de una mayor legibilida se muestra el código 4.3 con todos los conjuntos.

```
S0 := {(0, 0)};
for i := 1 to n-1 do begin
  Si+1 := {(a, c) / (a - a[i], c - c[i]) ∈ Si y a ≤ b};
  Si := merge_purge(Si, Si+1);
end; { for ... }
(cap0, prof0) := last_tuple(Sn-1);
{ last_validated_tuple = tuple with the largest cap in Sn-1 such that
                         cap+a[n] ≤ b }
(cap1, prof1) := last_validated_tuple(Sn-1, a[n]) + (a[n], c[n]);
if (prof1 > prof0) then
  (cap, prof) := (cap1, prof1)
else
  (cap, prof) := (cap0, prof0);
```

Código 4.3 Algoritmo para la resolución del problema de la mochila 0-1 cor eliminación de valores dominados.

4.2.4. Funciones de búsqueda.

Si bien en las dos técnicas anteriormente citadas era factible utilizar distintas funcione

de búsqueda (cada una ofrecía alguna ventaja respecto a las otras), no ocurre lo mismo en esta aproximación. Como no se puede asegurar que la secuencia de decisiones verifica el principio de optimalidad hasta computar todas sus subsoluciones, solamente es admisible una estrategia de búsqueda en anchura (primero en amplitud) [iba87]. Sin embargo si es factible encontrar aproximaciones a la resolución del mismo problema, que modifican el orden de ejecución de los diferentes bucles. Si bien para el caso secuencial su utilidad es normalmente nula (en algunos casos, empeoran la legibilidad de la solución), no ocurre lo mismo en paralelo, donde los códigos que se han intentado se aprovechan de esta posibilidad y se ensaya la paralelización de bucles alternativos.

4.3. PARALELIZACION DE LA PROGRAMACION DINAMICA.

El código que se ha introducido para representar el método general de esta técnica induce un único tipo de paralelización posible. La dependencia existente entre los valores a computar, hace inclinarse (en los casos en que la paralelización sea posible) por un sistema de división del trabajo estático y equivalente entre los procesadores (se asignan las diferentes cargas a priori), y mediante una serie de sincronizaciones se activa cada procesador en el instante adecuado (aquel que asegura el cálculo de valores de entrada en etapas anteriores). Todos los intentos que se han llevado a cabo para crear algoritmos concurrentes que siguieran esta técnica, están basados en este hecho. Las únicas variaciones que se intentan radican en los bucles a paralelizar.

4.3.1. Implementación sobre modelo SIMD con memoria compartida.

Si la máquina de la que se dispone es del tipo SIMD con memoria compartida y p procesadores, la sincronización viene impuesta por la forma de trabajo de este tipo de máquina [kin88], [li82], [ulm92]. Por lo tanto, los datos de entrada necesarios para resolver el problema de la etapa actual, se han obtenido en etapas anteriores. Un posible pseudocódigo realizable por cada procesador m en el rango $1 \leq m \leq p$, puede ser el del código 4.4.

```
Método: Procedure ConcurrentSIMDDP(m);
        var m', i2, i2', ..., in, in';
        begin
          for i2 in set(x2) do
            ......
              for in in set(xn) do
                f[m][i2]..[in]  := optimize g(f[m']..[in']);
                                   0≤i2'≤i2, ... 0≤in'≤in y existe k / ik'<ik
        end; { ConcurrentSIMDDP }
```

Código 4.4 Esquema de paralelización sobre modelo SIMD.

La única condición que se debe asegurar, es que el cómputo de los valores de la función f[m][i_2]..[i_n] dependa de valores menores o iguales en las n-1 variables no paralelizadas (al menos una de ellas debe ser estrictamente menor). Por simplificación se ha

supuesto que el número de valores asociados con la variable paralelizada coincide con el número de procesadores disponibles, esto es p.

Los posibles códigos paralelos de los ejemplos utilizados para este modelo podrían ser los siguientes. De nuevo como en la técnica divide y vencerás, se usa el lenguaje ll [leo91] para expresar los algoritmos (código 4.5).

```
Mochila 0-1

FOR i := 1 TO n DO
  PARALLEL 1..b DO
    IF (NA > a[i]) THEN
      f[i][NA] := max (f[i-1][NA],
                  f[i-1][NA-a[i]]+c[i])
    ELSE
      f[i][NA] := f[i-1][NA];
```

```
Caminos mínimos [kuc82]

{ Initially, A is adyacency matrix
  Finally is shortest path matrix. }
times := 0;
WHILE (times < logn) DO BEGIN
  PARALLEL 1..n² DO
  var i, j : INTEGER;
  BEGIN
    i = NA DIV n; j = NA MOD n;
    PARALLEL 1..n DO
      M[i][j][NA] := A[i][j]+A[j][NA];
    { A[i][j] := min(A[i][j],
          M[i][1][j], .., M[i][n][j] }
     minimum(M[i][j],    n,    logn,
A[i][j]);
  END; { PARALLEL ... }
  times := times+1;
END; { WHILE ... }
```

Código 4.5 Algoritmos ll para los dos problemas ejemplo.

4.3.2. Implementación sobre modelo MIMD con paso de mensajes.

En el supuesto de utilizar el modelo MIMD, cada procesador dispone de su contador de programa. La obligatoriedad de esperar por los datos de entrada hace necesaria la introducción de puntos de sincronización para emular el comportamiento de los algoritmos del modelo SIMD. En la mayoría de los casos los valores utilizados para la solución de un problema particular no han sido computados en el procesador asignado para su resolución, de ahí que deban ser enviados a éste para que pueda realizar la operación asociada. Dependiendo del bucle (o bucles) paralelizado (y por lo tanto de la carga asignada a cada procesador), los diferentes procesadores podrán comenzar a efectuar sus operaciones en instantes de tiempo distintos.

Sin embargo, normalmente el número de procesadores de los que dispone la máquina es inferior al número de valores a explorar en paralelo. Se debe entonces proceder a dividir el conjunto de los valores entre los procesadores, con este fin se debe realizar una repartición adecuada de los valores a computar[3]. Es asimismo importante minimizar el tiempo de espera inicial de los procesadores.

[3] A pesar de que esta dificultad adicional ha sido incluida en el modelo MIMD, también está presente en los programas correspondientes al modelo SIMD.

Un posible código para este modelo puede ser el mostrado en el código 4.6. El primer bucle debe contener los valores a explorar por cada procesador (set(x_1, m)). La diferencia fundamental con respecto al modelo completamente síncrono, radica en la necesidad de realizar las comunicaciones y sincronizaciones pertinentes.

```
Método: Procedure ConcurrentMIMDDP(m);
        var i1, i1', i2, i2', ..., in, in';
        begin
          for i1 in set(x1, m) do
            for i2 in set(x2) do
              ......
                for in in set(xn) do begin
                  { receive necessary for current }
                  f[i1][i2]..[in] := optimize g(f[i1']..[in']);
                                    0≤i1'≤i1, ... 0≤in'≤in y existe k / ik'<ik
                  { send necessary for next }
                end; { for ... }
        end; { ConcurrentMIMDDP }
```

Código 4.6 Esquema de paralelización sobre modelo MIMD.

4.3.3. Reduciendo las comunicaciones.

Al igual que ocurría en el caso secuencial, algunos de los valores que se computan están afectados por relaciones de dominancia. Puesto que la reducción en el número de envíos puede ayudar a minimizar la sobrecarga de comunicaciones, las posibles mejoras al esquema paralelo anterior proceden a comprobar, en cada procesador, si alguno de los valores calculados está dominado por otro. En este caso, es seguro que no puede estar incluido en ninguna secuencia óptima y puede ser desechado.

A pesar de que esta mejora no es aplicable a cualquier paralelización y no existe un esquema general de su tratamiento, se introduce el código 4.7 que para mostrar la estrategia utilizable:

```
Método: Procedure ConcurrentDominatedDP(m);
        var fmc, table;
        begin
          while not_end do begin
            fmc := receive_value;
            computed_associated(fmc, table);
            remove_dominated(table);
            mark_and_send_non_dominated(table);
          end; { while ... }
        end; { ConcurrentDominatedDP }
```

Código 4.7 Esquema de paralelización para reducir el número de comunicaciones.

En general, se trata de retener los valores óptimos necesarios hasta tener la certeza de que no son dominados por ningún otro y enviarlos (*mark_and_send_non_dominated*). Aquellos que lo sean son eliminados por el procedimiento *remove_dominated*. Al contrario que en los esquemas anteriores, no se conoce a priori el número de entradas que recibirá

cada procesador. La condición de parada, *not_end*, debe ser lanzada por el primer procesador, y a la llegada a cada uno de los siguientes comprobada y emitida en caso afirmativo. Como en el caso simple, se puede asignar más de un valor de cada variable paralelizada a cada uno de los procesadores.

Uno de los posibles inconvenientes de la reducción de comunicaciones, consiste en la posible inanición que puede afectar a los procesadores (suprareducción de mensajes o tardanza de comprobación de no dominancia). Algunos algoritmos que aplican esta optimización provocan una secuencialización no deseada, motivada por unos criterios muy rígidos en el envío de valores a los siguientes procesadores. Estos se ven obligados a estar mucho tiempo desocupados en espera de valores de entrada.

4.4. APLICACIONES.

Si bien el problema del viajante es uno de los ejemplos que más se elige de entre los problemas NP-completos (duros), no deja de ser menos importante el *problema de la mochila* (*K(napsack) P(roblem)*). Este problema de optimización combinatoria fue definido en el capítulo de ramificación y acotación. Su utilidad se debe a que aparece como subproblema en la resolución de muchos otros problemas de programación lineal entera.

Los diferentes experimentos a realizar se han obtenido variando el número de objetos posibles y la capacidad de la mochila. Los tamaños y valores de los objetos se eligen aleatorios en los intervalos correspondientes [mar90]. La comparación entre los diferentes algoritmos se realiza mediante sus tiempos de ejecución. Para los algoritmos asociados al problema de la mochila 0-1 se han generado cuatro problemas diferentes (n = 400, b = 500), (n = 800, b = 500), (n = 400, b = 2000) y (n = 800, b = 2000). Para la mochila entera se contrastan nueve experimentos. Se fijaron en este caso el número de objetos a (400, 800 y 1600) y la capacidad de la mochila a (800, 3200 y 12800).

Debido a la sincronización obvia en los algoritmos que parten de esta técnica, un esquema de array lineal o *pipeline* parece ser una de las topologías mas apropiadas. Como los parámetros asociados a los problemas (número de objetos o capacidad) pueden superar el diámetro de la red que se utilice, conviene en algunos casos, cerrar la línea con un nuevo enlace convirtiéndola en un anillo de procesadores. En este sentido, en la mayoría de los algoritmos presentados serán estas las topologías usadas, en la que el número de procesadores variará en el intervalo (2, 32). En un esquema que se encuentra a caballo entre la técnica divide y vencerás y la programación dinámica se utilizará una red hipercúbica.

El algoritmo de programación dinámica básico para resolver un problema de la mochila [hor78] se muestra en el código 4.8.

Como ya se ha citado en el apartado anterior muchos de los valores que se computan están dominados, por lo que se puede utilizar un código como el que se expreso

anteriormente, sin embargo en aras de legibilidad se ha preferido introducir este, que ha servido de partida para las primeras estrategias paralelas que surgieron en su momento. La complejidad algorítmica viene expresada por los dos bucles realizados, es decir O(nb), n por el número de objetos y b por la capacidad de la mochila.

```
for y := 0 to b do f[0][y] := 0;
for i := 1 to n do
  for y := 1 to b do
    if (y >= a[i]) then { con j = i-1 para 0-1 y j = i para entera }
      f[i][y] := max {f[i-1][y], f[j][y-a[i]] + c[i]}
    else
      f[i][y] := f[i-1][y];
```

Código 4.8 Algoritmo de programación dinámica básico para resolver un problema de la mochila.

4.4.1. Esquema de partida. Algoritmo para el modelo SIMD con memoria compartida.

La idea básica a aplicar en la paralelización de la programación dinámica debe incidir en la detección de un esquema que genere el mínimo número de sincronizaciones posibles entre los procesos que ejecuta cada procesador, manteniendo un orden de actuación en concordancia con el código secuencial de resolución. En concreto en el problema de la mochila 0-1, la formula recursiva aplicable, indica la dependencia con respecto a índices no superiores de la misma función bidimensional (misma capacidad, a lo sumo, y menor número de objetos).

$$f[i][y] = \{ f[i-1][y], f[i-1][y-a[i]] + c[i] \}$$

El código ejecutable en un modelo SIMD con memoria compartida debe seguir las restricciones de la formula anterior, esto es, con respecto a los objetos, hay que computar primero el valor óptimo excluyendo al último de ellos para la misma capacidad y otra inferior. Si el código paralelo realiza sincronizaciones cada vez que se computa el valor óptimo de la tabla para una fila, se habrán calculado los dos valores necesarios en la etapa anterior, como muestra el código 4.9, en el que se utiliza un pseudocódigo para una máquina SIMD muy parecido al lenguaje II.

```
parallel y in 0..b do f[0][y] := 0;
for i := 1 to n do
  parallel y in 1..b do
    if (y > a[i]) then
      f[i][y] := max {f[i-1][y], f[i-1][y-a[i]] + c[i]}
    else
      f[i][y] := f[i-1][y];
```

Código 4.9 Paralelización del bucle de las capacidades para la mochila 0-1 en una máquina síncrona.

Este esquema se conoce con el nombre de paralelización en las capacidades. Su complejidad se reduce a O(n), ya que el código realizado dentro del bucle paralelo es

constante y se repite n veces.

La estrategia contraria, paralelización en los objetos, se encuentra con la dificultad adicional de tener que calcular f[i-1][y] por lo menos una etapa antes. Esta nueva restricción, aparentemente indica un peor comportamiento para este segundo tipo de algoritmo. Sin embargo los resultados experimentales muestran que en las transformaciones de estos códigos a máquinas MIMD como las redes de transputers, este segundo esquema lleva a mejores comportamientos. Las cadenas de montaje que se forman producen códigos mas eficientes en el segundo caso.

A continuación se muestran aproximaciones que se basan en los dos esquemas de paralelización. En primer lugar se aborda la paralelización del bucle de capacidades, para pasar después a trabajar sobre el de los objetos.

4.4.2. Paralelización en las capacidades.

Es posible paralelizar de manera directa el bucle de las capacidades. Lin y Storer [lin91] proponen un código que paraleliza el bucle de las capacidades sobre una *Connection Machine*. El esquema de partida coincide con el código anterior (asignar el cómputo de cada capacidad a un procesador distinto), sin embargo en la práctica el número de procesadores disponibles es inferior a la capacidad de la mochila. En este caso, es necesario dividir el intervalo total en trozos que se asignan a cada procesador. Si se supone que se disponen de p procesadores, se asocia a cada uno de ellos un intervalo de tamaño $r = b/p$. En concreto, el procesador k se encarga del cómputo de los valores óptimos f[i][y] para y en un intervalo semiabierto de capacidades $[k \cdot r, (k+1) \cdot r)$[4] y para cada uno de los objetos $i = 1, \ldots, n$.

Dada la sencillez de las comunicaciones en la CM2, el algoritmo de Lin y Storer no es más que una modificación directa del algoritmo SIMD expuesto en el código 4.9. Los buenos resultados obtenidos por Lin y Storer parecen basarse, en gran medida, en la rapidez de las comunicaciones en la CM2. Después de varias aproximaciones a la paralelización del bucle de las capacidades sobre redes de transputers, hemos llegado al diseño que se describe a continuación [alm94].

El esquema que hemos implementado se ejecuta sobre un array lineal de transputers. El problema fundamental de esta estrategia surge de la dependencia que existe entre los valores f[i][y] y los valores f[i-1][y'] cuando y', menor que y, ha sido asignado a otro procesador de la red. Como ya se ha dicho el cómputo del valor f[i][y] sólo depende, a lo sumo, de los dos valores f[i-1][y] y f[i-1][y-a[i]]. Dado que el procesador que computa el valor f[i][y] ya ha computado el valor f[i-1][y], no podrá computar el valor f[i][y] hasta que reciba el valor f[i-1][y-a[i]]. En este sentido, surge una clasificación de los problemas

[4] Caso particular será el límite superior del último intervalo, si la capacidad de la mochila b no es múlti de p.

asignados a un procesador:

Se dice que el problema de la mochila (i, y) es un *problema interno* al procesador k si el problema (i-1, y-a[i]) pertenece al mismo procesador k. En otro caso, se dice que el problema (i, y) es un *problema externo*.

El algoritmo consta de una fase de inicialización y tres fases para cada objeto.

```
initial_capacity := k*r;
for y := 0 to r-1 do
begin
  f[0][y] := 0;
  if (initial_capacity + y < a[1]) then f[1][y] := 0
  else f[1][y] := c[1];
end; { for ... }
```

Código 4.10 Fase inicial del algoritmo que paraleliza el bucle de las capacidades.

Al comienzo de la computación, cada procesador procede a calcular los subproblemas dependientes de las parejas (0, y) e (1, y) tal y como se indica en el código 4.10. Obsérvese que la tabla f, en cada procesador, posee dimensiones [0..n], [0..r-1] por lo que se hace necesario realizar el desplazamiento correspondiente.

A continuación cada procesador, fijado el objeto i, computa secuencialmente los valores que dependen del objeto i en el intervalo de capacidades asignado, para lo cual se utilizan las siguientes tres fases (código 4.11):

- En primer lugar, los procesadores envían a sus vecinos los valores que otros procesadores necesitarán en el cómputo de subproblemas externos (*send_subproblems*).

- En segundo lugar, después de la recepción de los valores, los procesadores computan los valores f[i][y] correspondientes a sus subproblemas externos (*receive_and_compute_subproblems*).

- Por último, realizan el cálculo de los subproblemas (i, y) internos con y en el intervalo [0, r-1] (*compute_remaining_subproblems*). Resulta evidente que todos los subproblemas que pertenecen al procesador 0 son internos.

```
for i := 2 to n do
begin
  send_subproblems()
  receive_and_compute_subproblems();
  compute_remaining_subproblems()
end; { for ... }
```

Código 4.11 Fases que realiza el algoritmo que paraleliza el bucle de las capacidades para cada objeto.

En la i-ésima etapa cada procesador envía, con el proceso *send_subproblems()*, los valores óptimos del objeto i-1 que van a necesitar otros procesadores (código 4.12).

```
size := min (a[i], r);
last_capacity := (k+1)*r;
first_ext := last_capacity - size
for s := first_ext to (last_capacity - 1) do
  aux[s-first_ext] := f[i-1][s] + c[i];
send ! (first_ext + a[i]); [size] :: aux
```

Código 4.12 Primera fase encargada de enviar los valores que necesitan los siguientes procesadores.

Como ya se ha dicho, los procesadores consecutivos en la red tienen asignados grupos consecutivos de capacidades. Para obtener el sincronismo que se necesita, se hace circular un *token* desde el procesador 0 para marcar el final de los envíos correspondientes a cada etapa. Con el fin de evitar retrasos, se ejecuta en cada nodo un proceso *buffer* concurrentemente con el proceso principal. Así el procesador k, no empezará el cómputo de la etapa i hasta que el nodo k-1 haya computado la etapa i-1 y efectuado el correspondiente envío del *token*.

Con el procedimiento *receive_and_compute_subproblems()* (código 4.13), cada procesador recibe todos sus subproblemas externos de la etapa i.

```
while not_finish_phase do
begin
  receive ? (first_capacity; size :: packet_profit) or finish_phase
  if not finish_phase then
  begin
    if (initial_capacity <= first_capacity < last_capacity) then
    begin
      if (first_capacity + size >= last_capacity) then
      begin
        internal_size := last_capacity - first_capacity;
        send ! last_capacity; size - internal_size :: packet_profit;
        size := internal_size;
      end; { if ... }
      offset := first_capacity - initial_capacity;
      for s := offset to (offset + size) - 1 do
      begin
        f[i][s] := f[i-1][s];
        if packet_profit[s-offset] > f[i][s] then
          f[i][s] := packet_profit[s-offset];
      end; { for ... }
    end { if ... }
    else send ! first_capacity; size :: packet.profit;
          { The packet is routed to its neighbour }
  end { if ... }
  else send ! finish.phase;
  { When the token has been received it is sent and goes to the next phase }
end; { while ... }
```

Código 4.13 Fase encargada de recibir los subproblemas externos y computar los valores de la tabla asociados.

En la tercera fase, el procedimiento *compute.remaining.subproblems()* puede proceder al computo de los valores óptimos de todos sus subproblemas internos (código 4.14). El primer bucle se corresponde con aquellas capacidades más pequeñas que el peso del objeto actual. El segundo bucle computa los valores óptimos para aquellas capacidades en las que se puede introducir el objeto.

```
for y := 0 to min (a[i] - (initial_capacity+1), r-1) do
  f[i][y] := f[i-1][y];

for y := a[i] to r-1 do
  if f[i-1][y] < (f[i-1][y-a[i]] + c[i]) then
    f[i][y] := f[i-1][y-a[i]] + c[i]
  else
    f[i][y] := f[i-1][y];
```

Código 4.14 Fase encargada de computar los subproblemas internos.

Una vez finalizado este procedimiento quedarían resueltos todos los subproblemas dependientes del objeto i.

Ejecutadas las n-1 iteraciones que marca el procedimiento del código 4.11, el problema está resuelto y en el procesador p-1 se encuentra la solución.

Para estudiar el tiempo de ejecución del algoritmo, obsérvese que en la i-ésima etapa la tarea de recepción y envío de paquetes conlleva un tiempo de ejecución del orden de a[i] puesto que esta es la amplitud de los paquetes recibidos y enviados. Además el tiempo invertido en comunicación es $a[i] \cdot d[i]$, donde $d[i] = a[i]/r$ es la distancia a recorrer por los paquetes hasta llegar a su destino. La tarea de cómputo necesita un tiempo proporcional a b/p puesto que este es el tamaño del segmento de subproblemas asociado a cada procesador. Por tanto el tiempo invertido por un procesador en la etapa i es del orden de $b/p + a[i] \cdot d[i]$. El tiempo invertido en la ejecución de todo el procedimiento es:

$$O(nb/p + \sum_{i=1}^{n} a[i]a[i]/r) = O(nb/p + p(\sum_{i=1}^{n} a[i]^2)/b)$$

Si se cumple que $a[i] < G\ b/p$ para una constante G y para todo i, tenemos un algoritmo de complejidad

$$nb/p + p(\sum_{i=1}^{n} a[i]^2)/b \leq nb/p + pG^2nb^2/(p^2b) = (G^2+1)nb/p$$

que es O(nb/p) y por tanto óptimo en el sentido de que la complejidad paralela por el número de procesadores es igual a la complejidad secuencial. Por tanto, se puede concluir:

Teorema 4.4.2.1: Si $a[i] < G\ b/p$ para una constante G y para todo i, entonces el algoritmo es de complejidad O(nb/p).

Otra situación favorable se obtiene si se supone que los objetos verifican $a[i] \leq G\sqrt{b}$

para una constante G y para todo i, en tal caso, tomando $p \leq \sqrt{b}$ procesadores se tiene:

$$nb/p+p(\sum_{i=1}^{n} a[i]^2)/b \leq nb/p+G^2pnb/b \leq nb/p+G^2\sqrt{b}\,n \leq nb/p+G^2nb/p = (1+G^2)nb/p$$

que es de orden O(nb/p) y de nuevo el algoritmo paralelo propuesto es óptimo. Por tanto, se puede concluir:

Teorema 3.2.2: Si $a[i] \leq G\sqrt{b}$ para una constante G y para todo i, entonces el algoritmo es O(nb/p).

Resultados del algoritmo con cadena de montaje que paraleliza en las capacidades. *(P)ipeline (A)lgorithm with (P)arallelization on the (C)apacities.*

La tabla 4.1 muestra los resultados obtenidos con el algoritmo que paraleliza las capacidades para la serie de cuatro problemas generados. En términos generales los resultados muestran un mal comportamiento, sobre todo cuando se incrementa el número de procesadores. Para los problemas con capacidad de mochila pequeña (b = 500), la aceleración rápidamente decrece cuando usamos un gran número de procesadores. Este hecho es debido a que el tiempo dedicado a comunicaciones domina al tiempo de cómputo, puesto que el tamaño del intervalo asignado a cada procesador es pequeño. Cuando la capacidad es grande (b = 2000) este efecto se produce con un número de procesadores mayor.

Tabla 4.1 Resultados para el algoritmo que paraleliza el bucle de las capacidades PAPC.

	1 PROC.	2 PROC.		4 PROC.		8 PROC.		16 PROC.		30 PROC.	
TAM.	TIEMPO	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.
4x5	2.00	1.48	1.35	0.94	2.12	0.96	2.08	1.20	1.66	1.80	1.11
8x5	4.02	2.96	1.36	1.92	2.10	1.92	2.09	2.45	1.64	3.60	1.12
4x20	8.12	5.59	1.45	2.84	2.86	1.48	5.48	1.03	7.87	1.06	7.64
8x20	16.50	11.14	1.48	5.69	2.90	2.97	5.55	2.04	8.07	2.13	7.76

Los resultados experimentales no muestran un incremento lineal para la aceleración. La complejidad obtenida para este algoritmo es O(nb/p) cuando los valores a[i] se encuentran en el intervalo [1, b/p). Como los problemas generados no satisfacen esta condición, parece lógico este mal comportamiento. Cuando el número de procesadores es lo suficientemente grande, el segmento de capacidades asignado a cada procesador es muy pequeño. Este hecho aumenta la probabilidad de que la distancia a recorrer por los paquetes sea mayor que uno. De este modo, el tiempo invertido en comunicaciones varía como una función creciente en el número p de procesadores empleados.

Una nueva serie de experimentos (tabla 4.2) se llevaron a cabo para corroborar el hecho. En los problemas generados, se imponía la condición adicional a[i] < b/p, i = 1, ..., n. Los resultados demostraron un aumento de la aceleración, debido a la reducción de tiempo en la transmisión de datos. Al mismo tiempo se pone de manifiesto la esperada independencia respecto al número de objetos.

Tabla 4.2 Resultados para el algoritmo que paraleliza el bucle de las capacidades a[i] < b/p.

	1 PROC.	2 PROC.		4 PROC.		8 PROC.		16 PROC.		30 PROC.	
TAM	TIEMPO	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.
4x5	1.84	1.51	1.22	0.80	2.29	0.59	3.14	0.49	3.78	0.55	3.35
8x5	3.80	2.95	1.29	1.58	2.41	1.16	3.26	0.98	3.86	1.09	3.50
4x20	7.11	6.18	1.15	3.07	2.32	1.52	4.69	0.86	8.30	0.75	9.51
8x20	14.37	12.26	1.17	6.02	2.39	2.97	4.84	1.72	8.37	1.49	9.64

4.4.3. Códigos que paralelizan el bucle de los objetos. Algoritmos de canalización.

4.4.3.1. Un algoritmo con segmentación simple.

Si se observa la ecuación recursiva:

$$f[i][y] = \max \{ f[i-1][y], f[i-1][y-a[i]] + c[i] \}$$

se puede detectar que la dependencia, con respecto a los objetos, de los valores f[i][y] sólo ocurre entre valores adyacentes de i. Entonces, si se computan los valores f[i][y] en orden creciente de y, tan pronto como esté disponible f[i-1][y], también lo estará f[i-1][y-a[i]]. Este hecho permite resolver el problema usando n procesadores (1, ..., n). El procesador k se encarga del objeto k. El procesador k-ésimo recibe f[k-1][y] del procesador vecino izquierdo k-1 y envía f[k][y] al vecino derecho k+1. El procesador n-ésimo contendrá la solución cuando compute f[n][b]. Tan pronto como se computa el valor f[k][y] se envía al vecino derecho k+1 para el cómputo de su valor óptimo f[k+1][y] y la inicialización de f[k+1][y+a[k+1]]. El código 4.15 muestra las fases de recepción, cálculo y envío de valores óptimos del algoritmo que ejecuta cada uno de los n procesadores salvo el primero.

En la práctica, cuando n es grande el número de objetos supera al número de procesadores. Se hace necesario, como en el caso anterior, dividir el problema de acuerdo con el número de procesadores disponibles. El esquema de asignar intervalos de objetos consecutivos a cada procesador, no parece el más funcional en este caso, ya que provocaría retrasos en la activación de la cadena de montaje. La implementación llevada a cabo sobre una topología de anillo sugiere una distribución diferente de los objetos, en lo que se conoce como bandas [mol86]. Los procesadores primero y último se conectan a través del procesador

raíz. Este, además de su participación en la sincronización necesaria, también administra u cola donde se almacenan los mensajes que sirven de entrada al primer procesador en tratamiento de la siguiente banda.

```
f_aux[a[k]] := c[k];
for y := 0 to b do
begin
  receive ? f[k-1][y];
  if (y + a[k] <= b) then
    f_aux[y + a[k]] := f[k-1][y] + c[k];
  if (y >= a[k]) then
    f[k][y] := max {f[k-1][y], f_aux[y]}
  else
    f[k][y] := f[k-1][y];
  send ! f[k][c];
end; { for ... }
```

Código 4.15 Algoritmo simple que paraleliza el bucle de los objetos.

Se trata, como se indica en la figura 4.1, de dividir el conjunto total de objetos entre los p procesadores disponibles y formar $\lceil n/p \rceil$ etapas de ejecución en cada una de la cuales cada procesador se encarga de la resolución de los valores óptimos asociados a un de los objetos que pertenece a esa etapa (banda). Matemáticamente, el conjunto de valores

$$\{f[i][y], \text{ con } i = 1,\ldots,n \text{ e } y = 0,\ldots,b\}$$

se particiona en subconjuntos B[j], $1 \leq j \leq \lceil n/p \rceil$ con el mismo cardinal que el núme p de procesadores, de forma que:

$$f[i][y] \text{ pertenece a } B[j] \text{ si, y sólo si } \lceil i/p \rceil = j,$$

es decir, $B[j] = \{f[(j-1)p+1][y], \ldots, f[jp][y], \text{ con } y = 0,\ldots,b\}$

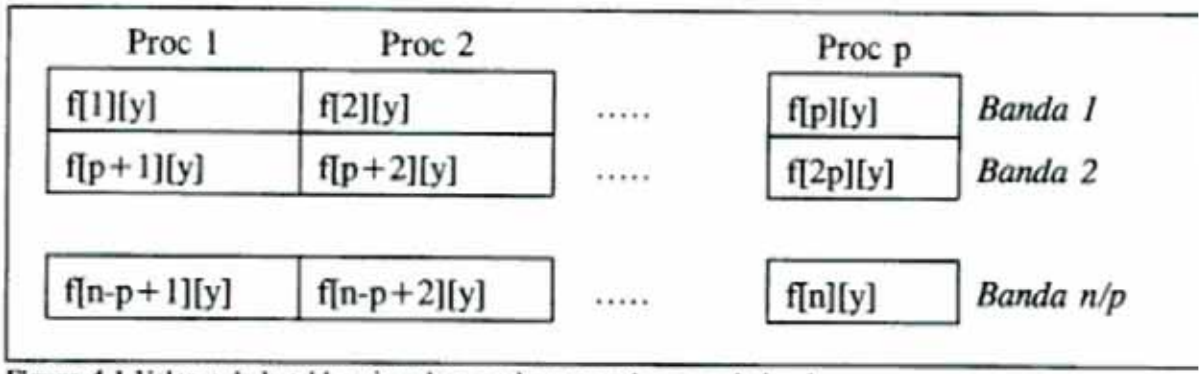

Figura 4.1 Valores de la tabla asignados a cada procesador en cada banda.

En cada banda j, el procesador k-ésimo computa secuencialmente f[(j-1)p+k][y] co y = 0, ..., b. Todas las componentes pertenecientes a la banda B[j] se calculan en paralel por los diferentes procesadores (código 4.16).

```
for j := 1 to n/p do
begin
  s := (j-1) * p + k;
  f_aux[a[s]] := c[s];
  for y := 0 to b do
  begin
    receive ? f[s-1][y];
    f[s][y] := f[s-1][y];
    if (y + a[s] <= b) then
      f_aux[y + a[s]] := f[s-1][y] + c[s];
    if (y >= a[s]) then
      f[s][y] := max {f[s][y], f_aux[y]};
    send ! f[s][y];
  end; { for ... }
end; { for ... }
```

Código 4.16 Inclusión de bandas en algoritmo simple que paraleliza el bucle de los objetos.

La cola recibe los valores finales para que el primer procesador del anillo comience el cómputo de la siguiente banda B[j+1].

La complejidad del código 4.16 en cada banda es el resultado del tiempo empleado en la computación por cada procesador O(b) más el utilizado en la propagación de los valores óptimos hasta el último procesador O(p). Puesto que el número de bandas es n/p, la complejidad total es O(nb/p+p).

	Proc 1	Proc 2	Proc 3	Proc 4
1:	(0, 0)			
2:	(a[1], c[1])	(0, 0)		
3:		(a[1], c[1])	(0, 0)	
4:		(a[2], c[2])	(a[1], c[1])	(0, 0)
5:		(a[1]+a[2], c[1]+c[2])	(a[2], c[2])	(a[1], c[1])
6:			(a[3], c[3])	(a[2], c[2])
7:			(a[1]+a[2], c[1]+c[2])	(a[3], c[3])

Figura 4.2 Esquema de las soluciones creadas a partir de los objetos sin test de dominancia.

4.4.3.2. Reduciendo el número de comunicaciones.

El costo de las comunicaciones en el algoritmo anterior es elevado frente al cálculo realizado. Se puede ver en el código 4.16 que cada procesador realiza b iteraciones, y para cada una de ellas envía un mensaje a su vecino. Por otro lado, muchas de estas soluciones enviadas no contribuyen a mejorar la solución final. Además, cuando el procesador vecino derecho recibe esta solución, genera nuevas soluciones innecesarias. Se opta, al igual que en

el caso secuencial, por la siguiente estrategia:

Se comienza el algoritmo con una solución (0,0) de capacidad 0 y beneficio 0. El primer procesador genera dos soluciones (0,0) y (a[1], c[1]). El segundo procesador genera cuatro soluciones, (0,0) y (a[2], c[2]) de la primera solución y (a[1], c[1]) y (a[1]+a[2], c[1]+c[2]) de la segunda. Este esquema se repite hasta que el último procesador alcanza la solución final. Tan pronto como el procesador k genera la solución (y, f), esta se envía al vecino k+1 para que el mismo pueda comenzar a trabajar lo antes posible. Todos los procesadores inician su ejecución casi de inmediato, puesto que la propagación de la solución (0,0) se realiza muy rápidamente.

Las soluciones forman un árbol binario (figura 4.2), donde el número de niveles es el número de objetos (n) del problema. Por lo tanto el número de soluciones parciales generadas y comunicadas por cada procesador, puede ser aún bastante grande, concretamente min {b, 2^k}. Aún siendo pequeño el número de objetos, este número crece con rapidez (exponencial) [che92]. Por esta razón, se ha investigado una nueva reducción de las comunicaciones que mantiene el mismo esquema general.

```
for y := 0 to b do
  sol[y] := -∞;
for j := 1 to n/p do
begin
  s := (j-1) * p + k;
  best_solution := -∞;
  top := 0;
  while moreCapacities do
  begin
    receive ? y, f[s-1][y];
    sol[y] := max {sol[y], f[s-1][y]};
    for u := top to y do
    begin
      if sol[u] > best_solution then
      begin
        best_solution := sol[u];
        send ! u, sol[u];
      end; { if ... }
    end; { for ... }
    top := y+1;
    if (y + a[s]) <= b then
    begin
      last_capacity := y + a[s];
      sol[y + a[s]] := f[s-1][y] + c[s];
    end; { if ... }
  end; { while ... }
  for u := top to last_capacity do
  begin
    if sol[u] > best_solution then
    begin
      best_solution := sol[u];
      send ! u, sol[u];
    end; { if ... }
  end; { for ... }
end; { for ... }
```

Código 4.17 Algoritmo que aplica dominancia en las soluciones generadas.

Muchas de las soluciones (y, f) que un procesador comunica a su vecino tienen beneficios (f) no mayores que las soluciones ya generadas con menor capacidad y no contribuyen a la solución final. De forma más precisa:

Definición: Dadas dos soluciones (y_1, f_1) e (y_2, f_2), se dice que (y_1, f_1) *domina a* (y_2, f_2) si y sólo si, $y_1 < y_2$ e $f_1 \geq f_2$.

Obviamente, se pueden reducir comunicaciones enviando sólo soluciones no dominadas. Las experiencias prueban que se obtienen soluciones dominadas frecuentemente. En el código 4.17 se muestra la implementación del algoritmo propuesto.

La variable *best_solution* mantiene, en todo instante, la mejor solución encontrada. Las soluciones se reciben en orden creciente de capacidad. En todo momento la variable *top* contiene el valor de la capacidad de la última solución recibida. La llegada de una nueva capacidad *y* conlleva el estudio de una posible mejora de *best_solution* mediante la exploración del vector de beneficios *sol*, para capacidades comprendidas entre *top* e *y*. Las soluciones con capacidades menores que *y* no pueden ser recibidas posteriormente. Además, las capacidades anteriores a *top* ya han sido analizadas.

La variable *last_capacity* almacena la última capacidad considerada hasta el momento. Después de recibir la última solución, la búsqueda de una nueva *best_solution* se reduce al intervalo de capacidades entre *top* y *last_solution*, evitando de esta manera recorrer el intervalo hasta b.

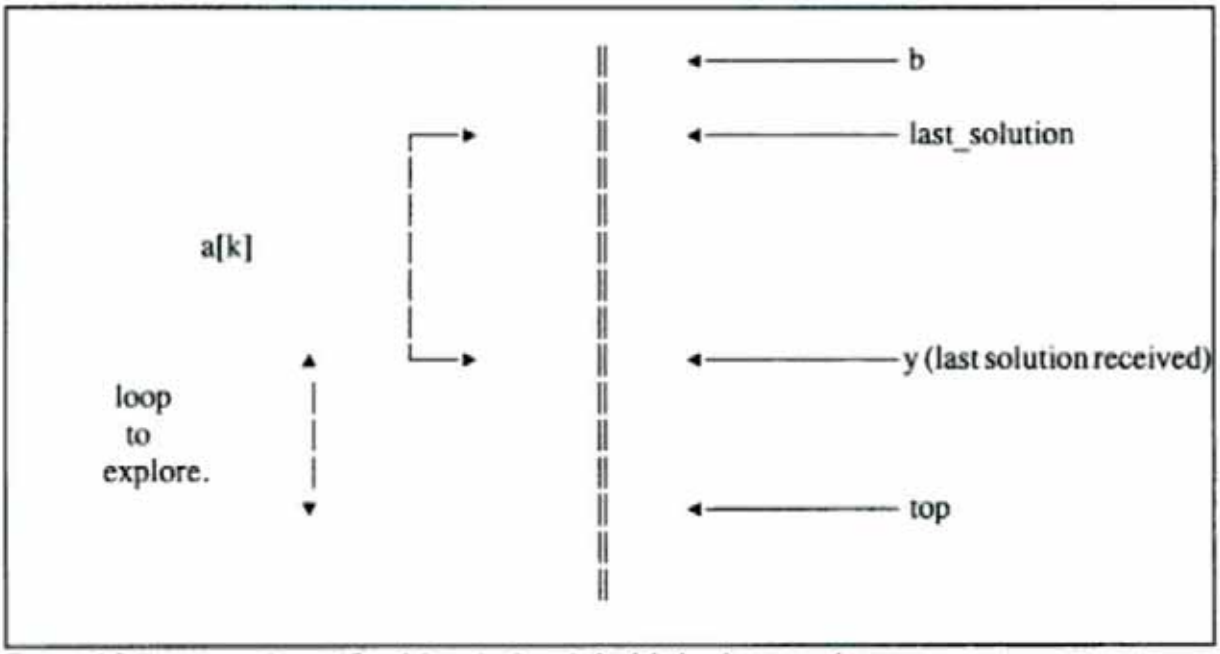

Figura 4.3 Representación gráfica del modo de trabajo del algoritmo anterior.

En la figura 4.3 se muestra gráficamente, una de las posibles situaciones en la

ejecución de este algoritmo. La complejidad del algoritmo, para el caso peor, coincide con la del secuencial O(nb), basta con tomar objetos con tamaño igual al de la capacidad de la mochila. Sin embargo, el número de comunicaciones que realiza este esquema es muy inferior al anterior de complejidad O(nb/p + p). Este esquema se puede mejorar con la inclusión de la condición del código 4.18, que garantiza que cada cierta cantidad de iteraciones (*iter*) se envía un mensaje.

```
if sol[u] > best_solution then
begin
  best_solution := sol[u];
  send ! u, sol[u];
end { if ... }
else if u mod iter = 0 then
  send ! u, sol[u];
```

Código 4.18 Inclusión de condición para mejorar complejidad del algoritmo que aplica dominancia.

Resultados para la paralelización de los objetos. Algoritmo con canalización simple *((S)imple (P)ipeline (A)lgorithm)* y algoritmo con dominancia *((P)ipeline (A)lgorithm with (D)ominancy)*.

Tabla 4.3 Resultados para el algoritmo simple que paraleliza el bucle de los objetos SPA.

	1 PROC.	2 PROC.		4 PROC.		8 PROC.		16 PROC.		31 PROC.	
TAM.	TIEMPO	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.
4x5	2.00	1.17	1.71	0.60	3.30	0.34	5.89	0.24	8.31	0.27	7.46
8x5	4.02	2.33	1.73	1.20	3.34	0.67	5.97	0.48	8.42	0.53	7.61
4x20	8.12	4.79	1.69	2.43	3.34	1.26	6.44	0.70	11.52	0.55	14.84
8x20	16.50	9.50	1.74	4.80	3.43	2.48	6.65	1.39	11.89	1.08	15.33

En la primera tabla (tabla 4.3) se muestran los resultados obtenidos para el algoritmo inicial que paraleliza el bucle de los objetos. La aceleración se mantiene creciente incluso con un número de procesadores alto. Sin embargo cuando la capacidad del problema es pequeña (b = 500) y se utiliza un gran número de procesadores no se obtiene un buen comportamiento. Para este tipo de problemas se puede producir contención, ya que la cantidad de carga asignada a cada procesador es baja mientras que la distancia a recorrer por los mensajes se vuelve alta. Por lo tanto, los procesadores pueden permanecer ociosos en espera de mensajes provenientes de la etapa anterior. Los resultados también prueban que no existe apenas dependencia de la aceleración con respecto al número de objetos.

La tabla 4.4 prueba que el algoritmo que utiliza el concepto de dominancia muestra el mejor comportamiento cuando se le compara con los restantes algoritmos aplicables al problema de la mochila 0-1 y que se describen en este capítulo. Las aceleraciones son las

más altas y se mantienen crecientes incluso cuando el número de procesadores es grande. Los resultados reflejan la reducción en el número de comunicaciones obtenida mediante este algoritmo. Se puede observar una leve dependencia con respecto al número de objetos. El algoritmo consigue su mejor comportamiento con problemas de tamaño medio. En varios casos se obtiene superlinealidad: este hecho es fácilmente explicable, si se tiene en cuenta que la versión secuencial utilizada es la implementación directa del primer algoritmo expresado en este capítulo, que no tiene en cuenta la dominancia aplicable al algoritmo secuencial.

Tabla 4.4 Resultados para el algoritmo que paraleliza el bucle de los objetos utilizando dominancia PAD.

	1 PROC.	2 PROC.		4 PROC.		8 PROC.		16 PROC.		31 PROC.	
TAM.	TIEMPO	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.
4x5	2.00	0.68	2.95	0.36	5.52	0.20	10.03	0.12	17.23	0.08	25.11
8x5	4.02	1.29	3.12	0.70	5.76	0.39	10.38	0.22	18.51	0.14	29.72
4x20	8.12	4.45	1.82	2.37	3.42	1.28	6.36	0.68	11.89	0.43	19.10
8x20	16.50	9.86	1.67	5.09	3.24	2.66	6.20	1.39	11.84	0.87	18.97

4.4.4. Un estrategia a caballo entre la programación dinámica y el divide y vencerás.

Una operación común a muchos algoritmos propuestos para hipercubos es la siguiente:

Se dispone de a_1, a_2, ..., a_p elementos situados en cada uno de los p procesadores del hipercubo y de un operador de agregación @ definido sobre dichos elementos, que es asociativo y conmutativo. Se pretende obtener el elemento $b = a_1 @ a_2 @ \ldots @ a_p$ resultante de la agregación de todos ellos en cualquier orden. El algoritmo clásico propuesto [ric90] resuelve el problema en log(p) pasos. Cada procesador k ejecuta el siguiente segmento de código (código 4.19):

```
b := ak
for dim := 0 to (logp - 1)
begin
  parallel
    send[dim] ! b
    receive[dim] ? a
  b := b @ a;
end; { for ... }
```

Código 4.19 Algoritmo de agregación para hipercubos.

El valor inicial de a_k, se corresponde en cada procesador con el valor correspondiente al procesador k. Los canales *receive[dim]* y *send[dim]* conectan al procesador con su vecino en dimensión dim para entrada y salida respectivamente. En cada iteración se agrega el elemento b resultante con el obtenido por su vecino en dimensión dim.

El algoritmo de Lee [lee88] obtiene una solución óptima al problema de la mochila 0/1 aplicando la técnica divide y vencerás y hace uso, en su fase de combinación, de una variante del algoritmo descrito anteriormente. Con ello se consigue un algoritmo de complejidad temporal $O(nb/p + b^2)$ cuando se ejecuta sobre una máquina con p procesadores. Básicamente el algoritmo consiste en una ejecución, en todos los procesadores, de cada uno de los siguientes pasos:

Paso 1.- *Descomposición:* El problema original Knap(n, O, b)[5] se particiona en p subproblemas Knap(n_k, O_k, b) con k = 0, ..., p-1, donde $O = \bigcup_{k=0}^{p-1} O_k$ y $O_k \cap O_j = \varnothing$ si $k \neq j$, y $n_k = \frac{n}{p} = s \ \forall i$. El subproblema Knap($n_k$, O_k, b) se asocia al procesador k. El tamaño del problema a resolver por el procesador k es n/p. Los subproblemas con objetos comprendidos en el intervalo semiabierto $[k \cdot n/p, (k + 1) \cdot n/p)$ serán asignados al procesador k.

Paso 2.- *Programación Dinámica*: Cada subproblema O_k se resuelve independientemente mediante el algoritmo de programación dinámica secuencial. El algoritmo de programación dinámica computa soluciones óptimas a Knap(n_k, O_k, b) para $0 \leq y \leq b$, luego cada procesador genera un vector de valores beneficio

$$C_k = (C_k[0], C_k[1], \ldots, C_k[b]) = (f_k[s][0], f_k[s][1], \ldots, f_k[s][b]).$$

Paso 3.- *Combinación*: Combinar los vectores de beneficios, obtenidos como consecuencia de resolver los p subproblemas generados, en el vector de beneficios resultado para el problema Knap(n, O, b).

Sean D y E dos vectores de beneficios para los subproblemas Knap(n_d, O_d, b) y Knap(n_e, O_e, b) tales que $O_d \cap O_e = \varnothing$. Se define la operación de combinación como:

$$F = D \ @ \ E, \text{ donde } F_y = \max \{D_z + E_{y-z} \text{ con } z = 0,..,y\} \ y = 0,..,b;$$

El vector F representa el beneficio óptimo para el conjunto de objetos en $O_d \cup O_e$. Es fácil ver que la computación de la operación de combinación de dos vectores de tamaño b conlleva una complejidad $O(b^2)$. Puede probarse también [lee88] que la operación de combinación es conmutativa y asociativa. De este hecho se deduce que el vector de beneficios final para el problema Knap(n, O, b) puede obtenerse del conjunto de vectores de beneficios (C_0, C_1, C_2, ..., C_{p-1}) obtenidos, combinándolos en cualquier orden. La aplicación directa del código 4.19 lleva a un número de operaciones igual a $O(nb/p + \log(p)b + \log(p)b^2)$. Sin embargo, puede ser mejorado ya que varios procesadores de la red están realizando simultáneamente el mismo cómputo. Balanceando la carga computacional

[5] Knap(n, O, b) se entiende como un problema de la mochila de capacidad b y un conjunto de n objetos O.

entre todos los procesadores se puede mejorar la complejidad a $O(nb/p + b^2)$.

Se ha desarrollado este algoritmo para ser ejecutado sobre una red de transputers a la que se ha dado una configuración topológica de hipercubo. Para conseguir hipercubos de dimensión superior a cuatro, se considera un hipercubo conectado en ciclos [pre81]. En este caso se han situado 2 procesadores en cada nodo.

Resultados de esta técnica doble.

Los valores representados en la tabla 4.5 corresponden a tiempos obtenidos por el algoritmo de Lee, esquema que se basa en la aplicación de dos técnicas, la divide y vencerás y la programación dinámica.

Se puede observar en la tabla que la aceleración obtenida decrece con la capacidad de la mochila, lo que fácilmente se deduce del factor b^2 que aparece en la complejidad del algoritmo. Cuando la capacidad de la mochila es $b > O(n)$, el tiempo de ejecución del algoritmo paralelo es incluso peor que el tiempo del algoritmo secuencial, independientemente del número de procesadores empleados. Por lo tanto, para que el algoritmo sea práctico, la capacidad de la mochila debe ser relativamente pequeña comparada con el número de objetos.

El tiempo de ejecución presenta un pequeño decrecimiento en la curva de aceleración para 16 y 32 procesadores. Esto es debido al hecho de que la topología usada es un hipercubo conectado en ciclos, por lo que es necesario más tiempo para atravesar el ciclo. Tal degradación podría no ocurrir si la máquina fuera un verdadero hipercubo.

Tabla 4.5 Resultados para el algoritmo de Lee.

	1 PROC.	2 PROC.		4 PROC.		8 PROC.		16 PROC.		32 PROC.	
TAM.	TIEMPO	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.	TIEMPO	ACEL.
4x5	2.00	1.27	1.57	0.92	2.18	0.75	2.66	0.91	2.20	0.96	2.08
8x5	4.02	2.28	1.76	1.43	2.81	1.01	3.97	1.04	3.84	0.94	4.25
4x20	8.12	8.25	0.98	8.38	0.97	8.47	0.96	12.06	0.67	12.09	0.67
8x20	16.50	12.45	1.33	10.45	1.58	9.50	1.74	12.55	1.31	12.03	1.37

4.4.5. Una nueva filosofía de tratamiento del problema.

Hasta ahora los códigos paralelos presentados se basaban en el principio de optimalidad de Bellman y eran válidos para los dos tipos de problema de la mochila expuestos, mochila 0-1 y mochila entera. Sin embargo, como en muchas otras áreas, este tipo de problema puede ser transformado a otro y resolverse, solucionando este último. Este es

el caso del siguiente código que se presenta, que solamente es válido para el problema de la mochila entera.

4.4.5.1. Un código secuencial convolución para el problema de la mochila entera.

Se asocia al problema de la mochila entera, un grafo dirigido acíclico G = (V, E). El conjunto de vértices V = [0, b] está formado por el conjunto de las capacidades. Un arco etiquetado con un valor beneficio c_k va desde el vértice de capacidad i al vértice de capacidad j sí, y sólo sí, existe un objeto k con tamaño a_k = j-i. Transformado el problema, se trata de encontrar el camino de longitud máxima en el grafo G [ten90]. Sea C la matriz de costos, se tiene que:

$$c_{ij} = \begin{cases} \max\{c_s \ / \ a_s = a_k\} & si\ j-i = a_k \\ 0 & si\ i = j \\ -\infty & en\ otro\ caso \end{cases}$$

Se define el operador binario MAX sobre el conjunto de enteros $Z \cup \{-\infty\}$ como:

$$n \ MAX \ m = \max\{n, m\}$$

Las propiedades que verifica este operador, asociativa, conmutativa y elemento neutro ($-\infty$), unido a la distributiva respecto al operador suma dotan al conjunto de los enteros unión el menos infinito de estructura de semianillo.

Basado en esta estructura, si se tienen dos matrices D = (d_{ij}) de dimensión nxm y E = (e_{ij}) de mxr con coeficientes en este semianillo, la matriz producto C = DxE se define del modo siguiente:

$$c_{ij} = \underset{t=0}{\overset{m-1}{MAX}}(ait + btj)$$

Entonces, si se denota por MC^s_{ij} el beneficio máximo de cualquier camino de i a j usando no más de s arcos y por C^s la potencia s de la matriz C usando la definición de matriz producto anterior, es decir:

$$C^s = C^{s/2} \times C^{s/2} \text{ para s par y } s > 2 \text{ y } C^1 = C \qquad (1)$$

ó,

$$c^s_{ij} = \underset{t=0}{\overset{b}{MAX}}(c^{s/2}_{it} + c^{s/2}_{tj}) = \underset{t=i}{\overset{j}{MAX}}(c^{s/2}_{it} + c^{s/2}_{tj}) \qquad (2)$$

se puede establecer el siguiente teorema:

Teorema 1:

Si s es una potencia de dos, se verifica la igualdad $c^s_{ij} = MC^s_{ij}$

Demostración:

La definición de la matriz C implica que el elemento c_{ij} contiene el beneficio máximo desde la capacidad i a la j usando 0 o 1 arcos. Puesto que s es par, cualquier camino de i a j que no usa más de s arcos se puede descomponer en dos caminos adyacentes que no usen mas de s/2 arcos. Por lo tanto, se tiene:

$$MC^s_{ij} = \underset{t=i}{\overset{j}{MAX}}(MC^{s/2}_{it}+MC^{s/2}_{tj})$$

razonando por inducción en s, la hipótesis es valida para s/2 y

$$MC^s_{ij} = \underset{t=i}{\overset{j}{MAX}}(MC^{s/2}_{it}+MC^{s/2}_{tj}) = c^s_{ij}$$

Este teorema conduce a un algoritmo con número de operaciones $O(b^3 \lceil \log(b) \rceil)$. El elemento c^s_{ij} contiene el beneficio máximo de pasar del vértice i al vértice j sin usar más de s arcos. De ahí sigue que, el elemento $c^{b'}_{0b}$ con $b' = 2^h$ y $h = \lceil \log(b) \rceil$, contiene el mayor beneficio de pasar de la capacidad 0 a la b usando a lo más b' arcos. Puesto que cualquier camino de 0 a b en el grafo asociado no tiene más de $b \leq b'$ arcos, $c^{b'}_{0,b}$ contiene la solución al problema. Por simplicidad se asume a partir de ahora que b es potencia de dos. En estas condiciones, el triángulo superior de la matriz C^s verifica el siguiente lema:

Lema 1:

Para todo s, y para todo $0 \leq i < j \leq b$, $0 \leq u < v \leq b$ tal que $j - i = v - u$ se tiene que $c^s_{ij} = c^s_{uv}$.

Demostración:

La demostración se hace por inducción en s. El caso $s = 1$ es consecuencia directa de la definición de C. Supuesto que la igualdad se ha probado para s/2, se tiene:

$$c^s_{ij} = \underset{t=0}{\overset{b}{MAX}}(c^{s/2}_{it}+c^{s/2}_{tj}) = \underset{t=0}{\overset{b}{MAX}}(c^{s/2}_{ut}+c^{s/2}_{tv})$$

Con este lema se puede eliminar uno de los índices y obtener la siguiente notación:

$$d^s_k = c^s_{ij} \text{ para cualquier } j\text{-}i = k.$$

De acuerdo a esta nueva notación, la fórmula (2) se puede reescribir como:

$$d^s_k = \underset{t=1}{\overset{\lfloor k/2 \rfloor}{MAX}}(d^{s/2}_t+d^{s/2}_{k-t}) \; MAX \; d^{s/2}_k, \; k = 1, \ldots, b \qquad (3)$$

Esta fórmula (3) induce un algoritmo con $O(b^2\log(b))$ número de operaciones, que mejora en un factor b el algoritmo basado en (2). Una nueva formulación que reduce la complejidad en un factor log(b) se puede alcanzar a partir del teorema 1. Como antesala son

necesarios dos lemas.

Lema 2:

Para todo $0 \leq t \leq b-k$, d^s_k es la longitud máxima de cualquier camino desde el vértice de capacidad t al de capacidad k+t sin usar más de s arcos.

Demostración:

La demostración es consecuencia directa de la definición de d^s_k.

Lema 3:

Para cualquier $s \geq k$, $d^s_k = d^k_k$ y d^s_k contiene la solución óptima para la capacidad k.

Demostración:

Se sigue de las hipótesis del problema que el tamaño de cada objeto es un entero positivo y cualquier camino en el grafo desde el vértice 0 a un vértice k no puede tener más de k arcos.

Teorema 2:

Para toda capacidad k en el intervalo 1...b y $s \geq k$, se tiene:

$$d^s_k = \underset{t=1}{\overset{\lfloor k/2 \rfloor}{MAX}} (d^s_t + d^s_{k-t}) \; MAX \; d^{s/2}_k \qquad (4)$$

Demostración:

Se prueba en primer lugar, la desigualdad siguiente:

$$d^s_k \geq d^s_t + d^s_{k-t}, \text{ para } s \geq k \text{ y } 0 \leq t \leq k/2.$$

Puesto que $s \geq k$, del lema 3 se tiene:

$$d^s_t + d^s_{k-t} = d^t_t + d^{k-t}_{k-t}$$

Sea p_1 el camino óptimo de 0 a t en el que se alcanza d^t_t y p_2 el correspondiente camino óptimo de t a k para d^{k-t}_{k-t}. La concatenación de ambos, p_1 y p_2 es un camino p de 0 a k con no más de k arcos y beneficio $d^t_t + d^{k-t}_{k-t}$. Puesto que $k \leq s$, de los lemas 2 y 3 aplicados a p y d^k_k, se sigue:

$$d^s_k = d^k_k \geq d^t_t + d^{k-t}_{k-t}$$

Para probar la desigualdad contraria, es decir, verificar que el valor d^s_k es alcanzado entre los factores de la parte derecha de la formula (4), sea p uno de los caminos de 0 a k en los que se alcanza d^s_k, es decir, un camino óptimo. Se plantean dos posibilidades: Cuando p sólo tiene un arco, $d^s_k = d^1_k = d^{s/2}_k$ y se concluye el teorema. En otro caso, p se puede dividir en dos caminos diferentes p_1 y p_2, ambos con menos de s arcos. Por lo tanto, $d^s_k =$

length(p_1) + length(p_2)

Suponiendo que p_1 va del vértice 0 al vértice r. Por el lema 2, length(p_1) y length(p_2) no superan a d^s_r y d^s_{k-r}, respectivamente. De ahí, $d^s_k = d^s_r + d^s_{k-r}$ como se quería demostrar.

Sustituyendo $d^{s/2}_k$ en la formula (4) por (3), se tiene:

$$d^s_k = \underset{r=1}{\overset{\lfloor k/2 \rfloor}{MAX}}(d^s_r + d^s_{k-r})\ MAX\ [\underset{r=1}{\overset{\lfloor k/2 \rfloor}{MAX}}(d^{s/4}_r + d^{s/4}_{k-r})]\ MAX\ d^{s/4}k,\ k = 1, \ldots, b \qquad (5)$$

de lo que sigue que la fórmula se puede simplificar a:

$$d^s_k = \underset{r=1}{\overset{\lfloor k/2 \rfloor}{MAX}}(d^s_r + d^s_{k-r})\ MAX\ d^{s/4}k,\ k = 1, \ldots, b \qquad (6)$$

y finalmente:

$$d^s_k = \underset{r=1}{\overset{\lfloor k/2 \rfloor}{MAX}}(d^s_r + d^s_{k-r})\ MAX\ d^1k,\ k = 1, \ldots, b \qquad (7)$$

De la fórmula (7), que es la convolución en el semianillo (MAX, +) del vector d^s consigo mismo, se obtiene un algoritmo que computa los valores $d^b_b = c^b_{0b}$. Se puede omitir el índice s en la formula anterior (7), lo que conduce a un algoritmo de convolución con complejidad $O(b^2 + n)$ que se muestra en los códigos 4.20 y 4.21.

```
for obj := 1 to n
  if d^b[a[obj]] < c[obj] then
    d^b[a[obj]] := c[obj];
```

Código 4.20 Algoritmo de inicialización.

La condición d[a[obj]] < c[obj] es necesaria, cuando existen diferentes objetos con el mismo tamaño (peso).

```
for k := 1 to b
  for i := 1 to k / 2
    d^b[k] := d^b[k] MAX (d^b[i] + d^b[k-i]);
```

Código 4.21 Algoritmo convolutivo secuencial.

4.4.5.2. Un algoritmo convolutivo paralelo.

Supuesto que se disponga de una cadena de montaje formada por b procesadores. A cada capacidad s se le asocia un procesador. El algoritmo para un procesador s se describe en el código 4.22. El procesador 0 realiza el código 4.20 (de inicialización). En principio el procesador s sólo debe computar los valores d^s_k para cada capacidad $k \leq s$, sin embargo, los valores para las capacidades restantes, $b \geq k > s$, son necesarios para los procesadores siguientes en la cadena de montaje, por ello cada procesador dispone de un vector local d^s

de tamaño b. La exploración de este bucle de orden b se puede dividir en cuatro intervalos bien diferenciados (figura 4.4) [alm95].

En el primer bucle (zona 1), para cada $k < s/2$, el procesador s recibe del s-1 los valores d^{s-1}_k. d^{s-1}_k contiene la solución óptima para la capacidad k. Por lo tanto, una vez almacenado en d^s_k, el procesador s sólo tiene que replicar el valor a su vecino derecho.

La zona 2 se encarga del tratamiento de los valores d^s_k asociados a las capacidades $s/2 \leq k < s$. Cada uno de ellos por el mismo razonamiento que antes, contiene la solución óptima para su capacidad. Sin embargo, también intervienen en el cómputo del valor d^s_s de acuerdo a la siguiente fórmula:

$$d^s_s = \underset{i=1}{\overset{\lfloor k/2 \rfloor}{MAX}} (d^s_i + d^s_{k-i}) \; MAX \; d^1s$$

```
-- zona 1.
for k := 1 to (s/2)-1
begin
  receive ? d^s[k]; -- reception of d^s-1[k]
  send ! d^s[k]; -- it is perfectly updated, echo to right neighbour
end;

-- zona 2
for k := s/2 to s-1
begin -- loop computing d^s[s]
  receive ? d^s[k]; -- reception of d^s-1[k]
  send ! d^s[k]; -- it is perfectly updated, echo to right neighbour
  d^s[s] := d^s[s] MAX (d^s[k]+d^s[s-k]); -- computing optimum value d^s[s]
end;

-- zona 3
receive ? temp; -- reception of d^s[s] = d^1[s]
d^s[s] := temp MAX d^s[s];
send ! d^s[s]; -- it is perfectly updated, echo to right neighbour

-- zona 4
for k := (s+1) to b
begin
  receive ? d^s[k]; -- reception of initial values d^s-1[k] = d^1[k]
  send ! d^s[k]; -- echo to right neighbour
end;
```

Código 4.22 Algoritmo convolutivo paralelo sin bandas.

Debido a que $s/2 \leq k < s$, los valores d^s_{s-k} ya fueron asignados en el primer bucle, con lo que están disponibles ambos valores, d^s_k y d^s_{s-k}.

Al finalizar el segundo bucle y recibir el valor d^1_s, se compara este último con d^s_s y se envía el valor óptimo final (zona 3).

La zona 4 (tercer bucle) actúa como un simple rutero de los valores de entrada d^1_k, con $s < k \leq b$.

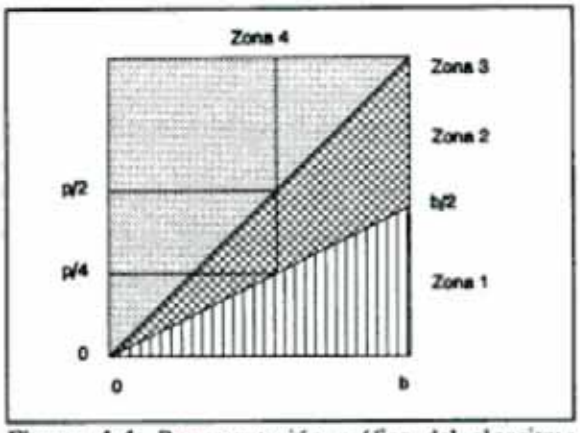

Figura 4.4: Representación gráfica del algoritmo convolutivo paralelo sin bandas.

La complejidad del algoritmo se restringe a O(b+n), obteniendo un número de operaciones equivalente al tiempo del algoritmo secuencial de partida.

Sin embargo en la figura 4.4 se muestra que en la mayor parte de la zona de exploración no se efectúa cómputo. Este hecho, unido a que normalmente no se dispone de un número tan alto de procesadores, es decir, se dispone de p+1 procesadores, con p $\leq$ b hace necesaria la utilización de una política de bandas. Para ello se conectan los procesadores siguiendo una topología de anillo. Como en el algoritmo de segmentación simple cada procesador s del intervalo p $\geq$ k $\geq$ 1 se encarga de calcular d^s, d^{s+p}, d^{s+2p}, ..., d^{s+rp} con r = b/p. Los valores emitidos, en cada banda, por el procesador p se encolan en el procesador 0. Estos valores son enviados al procesador 1, tan pronto como son requeridos. Con estas hipótesis, el algoritmo obtenido requiere b/p pasos en cada procesador, lo cual produce una complejidad total de $O(b^2/p+n)$. Cada procesador puede aprovechar ciertos cálculos realizados en bandas anteriores para computar el elemento de la banda actual, de esta forma se reduce el número de mensajes que se manejan en cada banda. Con esta nueva política aparecen seis zonas diferentes a explorar, como muestra la figura 4.5.

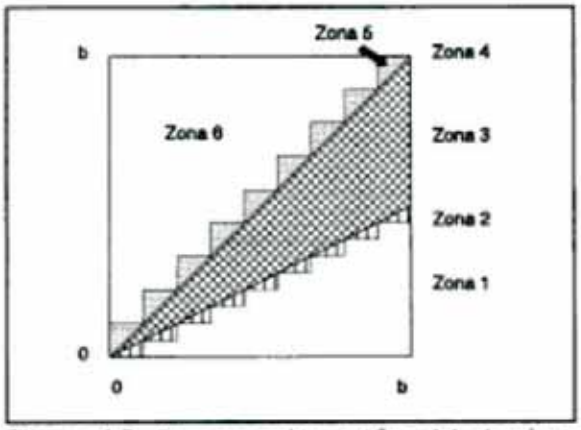

Figura 4.5: Representación gráfica del algoritmo convolutivo paralelo con bandas.

Supuesto que el análisis del algoritmo se encuentra en la etapa i y se habla del procesador s, la capacidad asignada al procesador será k = s+ip. Con esta notación y supuesto que se utiliza la variable y para identificar el intervalo a explorar, se obtiene la siguiente división:

La primera área o zona 1 comprende todos aquellos valores, que no intervienen en el cálculo de ninguna solución óptima de la banda en exploración y cuya solución óptima fue computada en bandas anteriores. El intervalo asociado de capacidades tiene como cota superior k2 = (i/2)·p.

La segunda zona, sólo necesaria para replicar en la primera banda todos los valores computados en procesadores más a la izquierda, se incluye en las demás por motivos de homogeneización. Los límites inferiores de este intervalo para una mayor legibilidad se han elegido múltiplos del número de procesadores p y sólo se realizan en las bandas pares, aunque es factible su reducción a múltiplos de p/2 realizado en todas las bandas.

```
for i := 0 to (b/p)-1 do
begin
  k := i*p + s;
  k2:= (i/2) * p;
  max := -∞;

  -- Zona 2
  for y := k2+1 to (k-1)/2 do
  begin
    receive ? d[y];
    send ! d[y];
  end;

  -- Zona 3
  if (s mod 2) = 0 then
  begin
    receive ? temp;
    send ! temp;
    d[k/2] := temp;
    max := 2 * temp;
  end;

  -- Zona 3
  for y := (k/2)+1 to (k/2)+(k-1)/2 do { k-1 }
  begin
    receive ? d[y];
    send ! d[y];
    temp := d[k-y] + d[y];
    if (temp > max) then
      max := temp;
  end;

  -- Zona 4
  receive ? d[k];
  if (d[k] > max) then
    max := d[k];
  send ! max;

  -- Zona 5
  for y = k+1 to k+(p-s) do
  begin
    receive ? temp;
    send ! temp;
  end;
end;
```

Código 4.23 Algoritmo convolutivo paralelo con bandas.

El tercer intervalo, $\lceil k/2 \rceil \leq y \leq k-1$ se reciben y se anotan los valores que serán usados para calcular $d^k[k]$. Al igual que en el caso simple, los valores adjuntos ya fueron recibidos en un bucle anterior.

La cuarta zona recibe el valor inicial $d^1[k]$ y lo compara con el valor óptimo de $d^k[k]$, enviando el máximo de ambos.

La zona 5 recibe los valores iniciales con capacidades superiores a la que se está explorando y que son calculados en algún procesador posterior de esta misma banda, para lo cual se reciben y replican. El intervalo correspondiente es $k < y \leq k + (p-s)$.

En el último intervalo o zona 6 se encuadran todas las capacidades que no serán exploradas en esta banda, y > k+(p-s).

En el código 4.23 se presenta el algoritmo correspondiente.

Resultados del algoritmo de convolución para el problema entero. *(P)arallel (C)onvolutive (A)lgorithm*. Comparaciones con los restantes algoritmos.

Tres de los algoritmos utilizados sobre la mochila 0-1 pueden ser utilizados también para el problema de la mochila entera. Para contrastar estos algoritmos con el específico de convolución, se muestran las siguiente tablas: En la primera tabla (tabla 4.6) se muestran los tiempos obtenidos con el algoritmo que paraleliza el bucle de los objetos. Se observa una aceleración creciente a medida que aumenta el número de procesadores utilizados. Sin embargo cuando el número de procesadores es pequeño, el comportamiento es malo.

Tabla 4.6 Resultados para el algoritmo SPA.

SPA	1 PROC.	2 PROC.	4 PROC.	8 PROC.	16 PROC.	32 PROC.
4x8	3.27	3.03	1.53	0.83	0.51	0.41
8x8	6.56	6.05	3.06	1.60	0.96	0.76
16x8	13.12	12.11	6.13	3.20	1.86	1.46
4x32	13.50	12.15	6.11	3.22	1.81	1.16
8x32	27.04	24.28	12.20	6.18	3.35	2.05
16x32	54.13	48.60	24.41	12.35	6.45	3.85
4x128	45.17	48.66	24.41	12.78	6.98	4.12
8x128	108.99	97.13	48.69	24.46	12.89	7.22
16x128	217.73	194.53	97.47	48.94	24.78	13.42

La segunda tabla que se muestra (tabla 4.7), corresponde al algoritmo que aplica el concepto de valores dominados. De nuevo se observan resultados superlineales (sobre todo para problemas pequeños y medianos), producidos por el mismo hecho que en el problema de la mochila 0-1.

La tabla 4.8 presenta los resultados del algoritmo que paraleliza el bucle de las capacidades. Para problemas con una capacidad grande (b = 12800), la aceleración experimenta un incremento lineal cuando aumentamos el número de procesadores. Para capacidades pequeñas, el tiempo empleado en comunicaciones domina al de cómputo.

Tabla 4.7 Resultados para el algoritmo PAD.

PAD	1 PROC.	2 PROC.	4 PROC.	8 PROC.	16 PROC.	32 PROC.
4x8	3.27	0.93	0.52	0.29	0.17	0.10
8x8	6.56	1.92	1.06	0.11	0.31	0.18
16x8	13.12	4.59	2.38	1.27	0.71	0.42
4x32	13.50	4.13	2.15	1.19	0.68	0.41
8x32	27.04	14.78	7.67	3.97	2.22	1.32
16x32	54.13	14.45	7.62	4.09	2.24	1.38
4x128	45.17	12.91	6.96	3.74	2.24	1.37
8x128	108.99	54.50	28.54	15.15	8.61	5.36
16x128	217.73	177.15	89.94	46.26	23.69	13.11

Los resultados del nuevo algoritmo (tabla 4.9) reflejan la independencia de este respecto al número de objetos. Como una consecuencia, tanto el algoritmo secuencial como el paralelo constituyen una buena opción cuando la capacidad de la mochila es del mismo orden que el número de objetos. Cuando la capacidad es muy grande (b = 12800), los resultados muestran la degradación que se produce. En todos los casos se observa una buena escalabilidad, aunque con pocos procesadores el algoritmo paralelo siempre se ve superado por el equivalente secuencial.

Tabla 4.8 Resultados para el algoritmo PAPC.

PAPC	1 PROC.	2 PROC.	4 PROC.	8 PROC.	16 PROC.	32 PROC.
4x8	3.27	2.70	1.71	1.00	0.93	0.92
8x8	6.56	5.38	3.28	3.24	1.85	1.85
16x8	13.12	10.81	6.90	4.03	3.73	3.73
4x32	13.50	9.57	4.50	2.33	1.26	0.87
8x32	27.04	18.34	8.99	4.64	2.48	1.74
16x32	54.13	36.74	18.01	9.31	5.00	3.49
4x128	45.17	36.23	17.58	8.91	4.58	2.56
8x128	108.99	72.63	35.19	17.77	9.07	5.01
16x128	217.73	145.38	70.39	35.52	18.10	9.97

Tabla 4.9 Resultados para el algoritmo PCA.

PCA	1 PROC.	2 PROC.	4 PROC.	8 PROC.	16 PROC.	32 PROC.
4x8	0.66	0.96	0.53	0.30	0.20	0.15
8x8	0.66	0.96	0.53	0.30	0.20	0.15
16x8	0.66	0.96	0.53	0.30	0.20	0.15
4x32	10.44	14.44	7.38	3.79	2.05	1.21
8x32	10.44	14.44	7.38	3.79	2.05	1.21
16x32	10.44	14.45	7.38	3.79	2.06	1.22
4x128	166.77	228.19	114.72	57.72	29.45	15.49
8x128	167.77	228.20	114.73	57.72	29.45	15.49
16x128	166.78	228.20	114.72	57.72	29.45	15.49

4.4.6. Comparaciones gráficas entre los esquemas presentados.

Como en anteriores capítulos, se concluye el análisis con una serie de figuras que contrastan las aceleraciones de los cuatro códigos siguientes (SPA, PAD, PAPC y PCA). En las primeras cuatro gráficas se presentan los resultados para cuatro problemas (16x8, 4x32, 8x128 y 16x128). Se puede observar que el mejor comportamiento corresponde al algoritmo PAD, salvo para los casos en el que la capacidad de la mochila es muy grande (12800) y el número de objetos es bastante menor que la misma. En las figuras 4.7 y 4.8 se representa también la linealidad, para destacar el factor superlineal del algoritmo PAD en uno de los casos (superlinealidad explicada anteriormente).

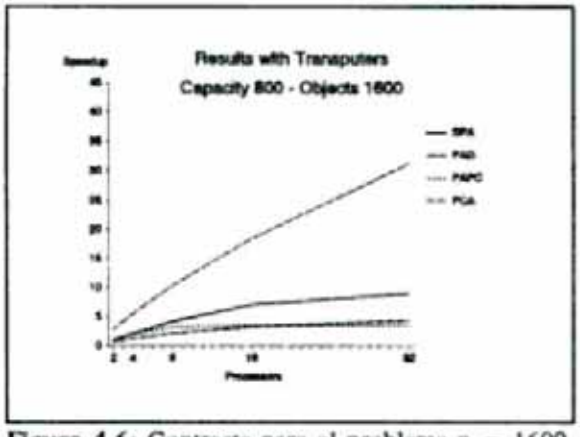

Figura 4.6: Contraste para el problema n = 1600, b = 800.

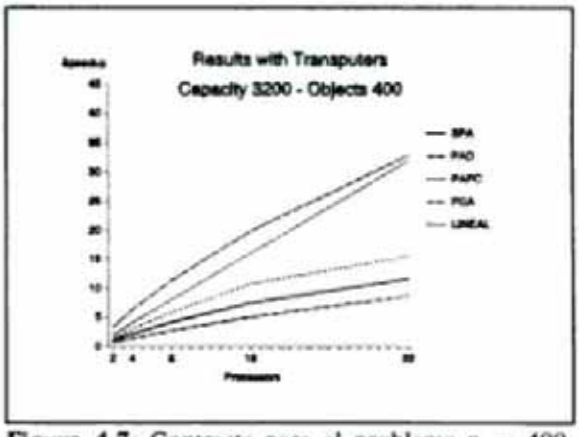

Figura 4.7: Contraste para el problema n = 400, b = 3200.

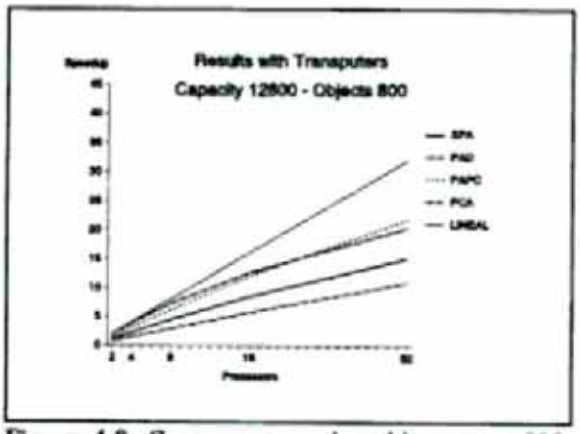

Figura 4.8: Contraste para el problema n = 800, b = 12800.

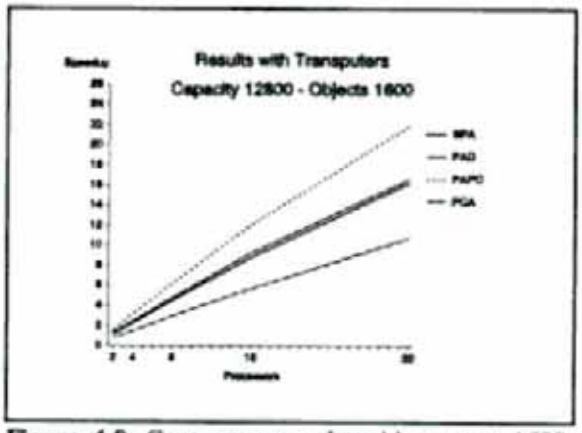

Figura 4.9: Contraste para el problema n = 1600, b = 12800.

Las siguientes cuatro figuras analizan cada uno de los cuatro algoritmos por separado. Es de destacar el factor de equilibrio necesario entre capacidad y número de objetos a la hora de obtener buenas aceleraciones.

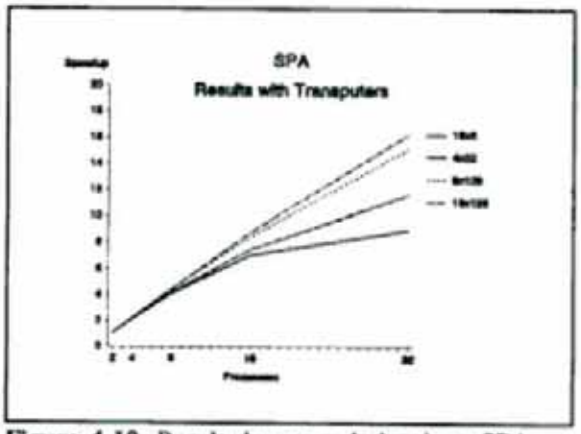

Figura 4.10: Resultados para el algoritmo SPA.

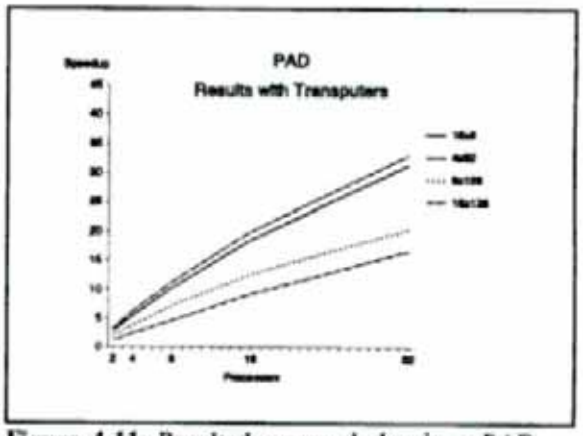

Figura 4.11: Resultados para el algoritmo PAD.

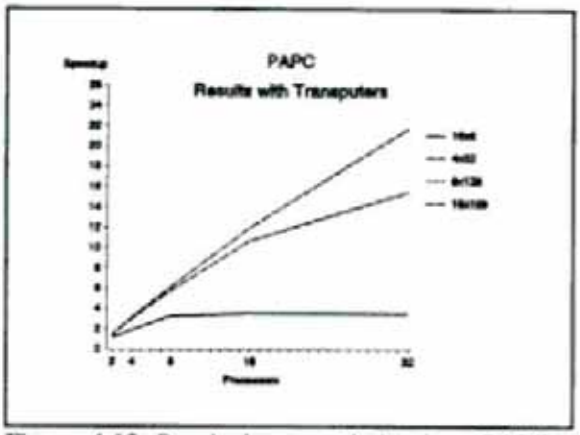

Figura 4.12: Resultados para el algoritmo PAPC.

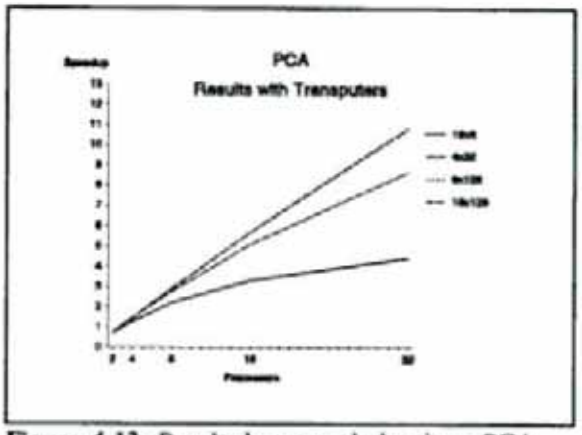

Figura 4.13: Resultados para el algoritmo PCA.

4.5. BIBLIOGRAFIA.

[alm94]. F. Almeida, D. Morales, F. García and C. Rodríguez. Dynamic programming parallel algorithms for the 0/1 knapsack problem on transputers networks. *Transputers Applications and Systems '94*, pp 817-831. Italy. IOS Press, Ohmsha.

[alm95]. F. Almeida, F. García, D. Morales and C. Rodríguez. A parallel algorithm for the integer knapsack problem for pipeline networks. to appear in *Journal of Parallel Algorithms and Applications* 6 (3/4).

[bel57]. R. E. Bellman. *Dynamic programming*. Princeton U.P.

[che92]. G. Chen and J. Jang. An improved parallel algorithm for 0/1 knapsack problem. *Parallel Computing* 18, pp. 811-821.

[hor78]. E. Horowitz and S. Sahni. *Fundamentals of computer algorithms*. Computer Science Press. Potomac, MD.

[iba87]. T. Ibaraki. *Enumerative approaches to combinatorial optimization. Part II*. J.C. BALTZER AG. Basel, Switzerland.

[kin88]. G.A.P. Kindervater and H.W.J.M. Trienekens. Experiments with parallel algorithms for combinatorial problems. *European Journal of Operational Research* 33, pp. 65-81.

[kuc82]. L. Kucera. Parallel computation and conflict in memory acces. *Information Procesing Letters* 14, pp. 93-96.

[lee88]. J. Lee, E. Shragowitz and S. Sahni. A hypercube algorithm for the 0/1 knapsack problem. *Journal of Parallel and Distributed Computing* 5, pp. 438-456.

[leo91]. C. León. Un compilador pascal paralelo para el modelo P-RAM. *Memoria de Licenciatura*. Dept. Estadística, Inv. Operativa y Computación, Univ. de La Laguna.

[li85]. G. Li and B.W. Wah. Systolic processing for Dynamic Programming. *Proceedings of 1985 International Conference on Parallel Processing*, pp. 434-441.

[lin91]. J. Lin and J.A. Storer. Processor-efficient hypercube algorithms for the knapsack problem. *Journal of Parallel and Distributed Computing* 11, pp. 332-337.

[mar90]. S. Martello and P. Toth. *Knapsack problems: Algorithms and computer implementations*. John Wiley & Sons. England.

[mol86]. D.I. Moldovan and J.A.B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers* C-35 (1), pp. 1-12.

[pre81]. F. Preparata and J. Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM* 24, pp. 300-309.

[ric90]. M. Rice, S. Seidman and P. Wang. The especification of data parallel algorithms. *Journal of Parallel and Distributed Computing* 8, pp. 191-195.

[sed83]. R. Sedgewick. *Algorithms*. Addison-Wesley. Reading, MA.

[smi91]. D.K. Smith. *Dynamic programming: A practical introduction*. Ellis Horwood. England.

[ten90]. S. Teng. Adaptative parallel algorithms for integral knapsack problem. *Journal of Parallel and Distributed Computing* 8, pp. 400-406.

[ulm92]. D.R. Ulm and P.Y. Wang. Solving a two dimensional knapsack problem on SIMD computers. *Proceedings of 1992 International Conference on Parallel Processing*.

CONCLUSIONES Y TRABAJOS FUTUROS

Según prueban todos los estudios realizados en el mercado informático, en los próximos años asistiremos a un crecimiento muy pronunciado de las máquinas paralelas orientadas al entorno industrial y comercial, mientras se mantiene el negocio de los multiprocesadores de propósito científico. Una condición indispensable para que se produzca este desarrollo es conseguir que los usuarios sean capaces de integrarse en los entornos de programación de estas máquinas, para ello se debe intentar aproximar la programación paralela al esquema secuencial ya consolidado.

Las técnicas algorítmicas ya consagradas como el divide y vencerás, la ramificación y acotación, la programación dinámica, etc. y las de expansión en los últimos años como los algoritmos evolutivos, la recristalización simulada, etc., gozan de un alto reconocimiento dentro de campos tales como la Investigación Operativa, la Inteligencia Artificial, etc. y conseguir prototipos generales eficientes para su aplicación en multiprocesadores debería ser uno de los objetivos básicos a cumplir en los próximos años. Es obvio sin embargo, que no en todas las técnicas se ha llegado a un mismo grado de generalización, en particular la programación dinámica parece a priori la que está menos desarrollada y la mayoría de los trabajos publicados se centran en el estudio de problemas particulares como el de la mochila.

En este trabajo se presentan los resultados que han ido apareciendo en la paralelización de las técnicas más extendidas y en algunas de estas se propone un nuevo esquema. En la estrategia divide y vencerás se propone un esquema general de trabajo sobre una máquina árbol, al mismo tiempo que se analizan otros dos esquemas sobre este mismo tipo de topología (esquema centralizado y esquema jerárquico). En la técnica de ramificación y acotación se evalúan tres esquemas de trabajo, en los que el control se asigna a un procesador específico o se reparte entre la red. En dos de estos algoritmos se ensaya una reducción del número de mensajes a enviar con resultados satisfactorios. En la programación dinámica se presentan tres códigos paralelos nuevos para el problema de la mochila, y además aplicando ideas ya conocidas se estudia el comportamiento de otros dos algoritmos.

Como para todas las técnicas tratadas se ha realizado un estudio sobre una máquina paralela fuertemente acoplada, como son las redes de transputers, uno de los primeros objetivos que nos planteamos es aplicar las mismas ideas sobre máquinas con un grado de comunicación más débil, como pueden ser las redes de estaciones de trabajo, y comprobar la eficiencia de las estrategias que en este caso se consigue. En este sentido, aprovechando el entorno de programación PVM hemos comenzado a realizar un estudio similar al llevado a cabo en este trabajo. Al mismo tiempo es necesario considerar otra serie de problemas a analizar, con el fin de obtener un equilibrio entre la generalidad del código a aplicar y la eficiencia que este permite.

A un plazo mayor podría ser muy interesante la creación de herramientas de trabajo para las diversas técnicas que permitieran paralelizar de forma automática los algoritmos secuenciales que se basarán en cada una de ellas, tomando como entrada los códigos específicos de cada uno de los elementos que conforman la técnica. Este objetivo está íntimamente relacionado con los intentos que se están llevando a cabo para crear compiladores que sean capaces de obtener códigos paralelos óptimos a partir de los algoritmos secuenciales ya construidos y que por lo tanto, evitan a los posibles usuarios el tener que enfrentarse con nuevos problemas específicos de la programación paralela. Aunque en esta memoria el estudio se ha centrado en tres técnicas, no existe ninguna circunstancia especial que impida realizar el mismo análisis a las estrategias que han ido tomando fuerza en los últimos años y que se han nombrado con anterioridad.

En conexión con este último apartado, el grupo de investigación que se nació hace cinco años y del que formo parte, ha desarrollado un sistema de programación paralela orientado al modelo PRAM sobre redes de transputers. Una de las perspectivas en las que ya se está trabajando, es la traslación de este sistema a otros tipos de máquinas paralelas. La unión de ambas líneas de trabajo puede llevar a la obtención de un sistema genérico de programación paralela, que a modo de cadena de montaje, pueda servir para crear y verificar códigos paralelos del modelo PRAM.

Por último quisiera dejar constancia de la importancia, tanto a nivel investigador como docente, que ha tenido para mi la realización de este trabajo. Si a nivel informático es obvio que el estudio y dominio de la programación secuencial (recursividad, punteros, etc.) no se puede llevar a cabo sin la experimentación, y que la estructuración es una metodología que necesita un cierto período de tiempo en el cual el individuo aprende a adaptar su mente, este hecho se acrecienta en la programación concurrente, donde hay que unir a la estructuración el hecho de que existen más de un de proceso que trabaja simultáneamente. No es posible entender conceptos claves como exclusión mutua, semáforos, monitores, paso de mensajes, etc., sin haber realizado algún trabajo de una cierta entidad, en el cual se aprende a llevar control de los procesos que están activos y que comunican entre si. Este último hecho me parece de vital importancia y a nivel docente se convierte en el factor principal a la hora de una buena comprensión por parte de los alumnos.

RESUMEN DE CODIGOS IMPLEMENTADOS

En este apéndice se presenta una breve descripción de los códigos implementados que se suministran en el disco flexible anexo a esta memoria. La estructura de directorios que se sigue en el disco implica que todos los ficheros de directorios ancestros en las hojas son necesarios para la ejecución de los programas. A continuación se comentan la mayoría de los ficheros incluidos en el disco.

Los ficheros .LP representan el modo de indicar a la red de transputers la topología deseada, que se carga en la placa linkputer.

TREE¿?.LP	Ficheros de descripción de conexiones en forma de árbol binario.
RING¿?.LP	Ficheros de descripción de conexiones en forma de anillo.
HYPERCUBE¿?.LP	Ficheros de descripción de conexiones en hipercubo.

Los ficheros .CFS y .PGM son los encargados de configurar la red, indicando como se conectarán los diferentes transputers de la topología entre si.

TREE¿?.CFS	Ficheros de configuración de una topología de árbol binario asociado al ANSI C de Inmos.
RING¿?.PGM	Ficheros de configuración de una topología de anillo asociado al Occam de Inmos.

Los ficheros .LNK son los encargados de indicar que ficheros objeto se desean linkar conjuntamente. Siempre estarán asociados a un fichero fuente .C.

P_ROOT.C	Fichero fuente asociado al procesador raíz de una topología árbol.
P_NODE.C	Fichero fuente asociado a los procesadores intermedios de una topología árbol.
P_LEAF.C	Fichero fuente asociado a las hojas del árbol.
P_APPL.C	Fichero fuente que contiene el método de resolución secuencial de los problemas.
P_DEF?.C	Ficheros que contienen constantes, definiciones de estructuras de datos, funciones auxiliares para la comunicación, etc.

Los ficheros .INC son ficheros includes, que contienen constantes básicas para el programa en cuestión. Al igual que los .LNK, están asociados a un fichero fuente .OCC, que como se debe intuir contienen el código Occam.

¿?_M.OCC	Fichero fuente asociado al procesador maestro de la red, aquel que se encarga de la entrada/salida y que cierra los anillos con la implementación de un proceso cola.

¿?_T.OCC	Fichero fuente asociado a cada uno de los procesadores obreros de la red.
¿?_W.OCC	Fichero fuente que contiene el algoritmo a ejecutar por cada proceso que se ejecute en un procesador obrero.

Los ficheros .TOP son ficheros asociados al entorno de trabajo del Transputer Development System e incluyen todos los programas necesarios para la ejecución del algoritmo.