UNIVERSIDAD DE LA LAGUNA

«El modelo de computación colectiva: una metodología eficiente para la ampliación del modelo de librería de paso de mensajes con paralelismo de datos anidado»

Autor: Francisco de Sande González
Director: Dr. D. Casiano Rodríguez León

Departamento de Estadística,
Investigación Operativa y Computación

Don Casiano Rodríguez León, Doctor en Matemáticas y Catedrático del Área de Lenguajes y Sistemas Informáticos adscrito al Departamento de Estadística, Investigación Operativa y Computación de la Universidad de La Laguna,

Certifica

Que la presente memoria titulada ***El Modelo de Computación Colectiva: Una Metodología eficiente para la ampliación del Modelo de Librería de Paso de Mensajes con Paralelismo de Datos Anidado***, ha sido realizada bajo su dirección por el Licenciado D. Francisco de Sande González y constituye su Tesis para optar al grado de Doctor Ingeniero en Informática.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos que haya lugar, firma la presente en La Laguna, a cinco de octubre de mil novecientos noventa y ocho.

Fdo.: Casiano Rodríguez León

A mis padres, Rosa y Pompeyo,
a quienes debo todo

A mis hermanas, Daine y Sole
por su esfuerzo en mi educación

Agradecimientos

Quisiera expresar mi agradecimiento en primer lugar al profesor D. Casiano Rodríguez León por haberme permitido integrarme en su grupo de investigación, por el entusiasmo que siempre me ha infundido y por su intensa dedicación a la realización de este trabajo.

A mis compañeros del Grupo de Paralelismo Coro, Félix, Paco, Jose Luis, Dani y Jesús Alberto por las enriquecedoras discusiones que hemos mantenido, por su ayuda desinteresada en tantas ocasiones y en particular en la revisión de esta memoria, por tantos ratos compartidos y en definitiva, por su amistad.

A los compañeros del Departamento de Estadística, I. O. y Computación y del Centro Superior de Informática por su apoyo.

A Instituciones como el C4, Ciemat, Instituto de Astrofísica de Canarias, Centro de Computación de Edimburgo y Centro Superior de Informática de la Universidad de La Laguna que nos han brindado la oportunidad de utilizar sus máquinas e instalaciones, imprescindibles en la realización de esta memoria.

Mi agradecimientos también para Oriol Riu, del Cepba, Ingrid Bárcenas, Montserrat Mestres y Joan Cambras del Cesca y Pablo López del Instituto de Astrofísica de Canarias, todos ellos administradores de las máquinas en que se han llevado a cabo los experimentos computacionales de este trabajo.

A Lola, por su comprensión, apoyo y cariño.

ÍNDICE

Capítulo III

Capítulo IV

Capítulo V

Capítulo VI

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Prólogo

Mientras que el diseño e implementación de algoritmos secuenciales está firmemente asentado en el modelo de arquitectura de Von Newmann, en el caso de las aplicaciones paralelas, la falta de un modelo universalmente aceptado de arquitectura ha conducido a una gran disparidad de aproximaciones a la hora de concebir e implementar algoritmos.

El modelo PRAM de computación paralela introducido por Fortune y Willie [For78] hace 20 años se ha mostrado durante las últimas décadas como uno de los más fructíferos a la hora de ser utilizado en la concepción de algoritmos paralelos. El poderoso mecanismo de comunicación que representa la memoria compartida, el sincronismo del modelo y la capacidad de expresar paralelismo anidado son fundamentalmente las tres características que hacen atractivo al modelo para los diseñadores de algoritmos. No obstante, el modelo PRAM ha recibido críticas en cuanto a su factibilidad por considerársele demasiado idealista y alejado de la realidad de las arquitecturas paralelas presentes en el mercado.

La programación mediante paso explícito de mensajes es probablemente la aproximación que mejores resultados proporciona en el mundo de las aplicaciones paralelas: el programador tiene en este modelo control absoluto sobre los recursos de la máquina y ha de encargarse explícitamente de la sincronización y comunicación entre procesos y de la asignación de procesos a procesadores. Hasta hace bien poco, un primer inconveniente de este modelo de programación era la falta de portabilidad de las aplicaciones. Con frecuencia las diferencias entre arquitecturas provocaban que para trasladar una aplicación paralela de una arquitectura a otra hubiera que modificar, reescribir o comprobar grandes porciones de código. No obstante, con el advenimiento de herramientas de programación mediante paso de mensajes basadas en librerías estándar (en particular PVM o MPI) este inconveniente se ha mitigado un tanto: basta ahora que el fabricante de una máquina paralela proporcione una implementación de una de estas librerías para que los códigos puedan ser portados con relativa facilidad de

una plataforma a otra. No obstante, el gran inconveniente de la programación con utilización explícita de paso de mensajes sigue siendo con diferencia, el alto grado de conocimiento que se requiere por parte del programador, quien ha de ocuparse en todo momento de lo que podríamos llamar "detalles de bajo nivel" de la programación paralela. Si se desea que las aplicaciones paralelas se asienten en el mercado y desarrollen toda su potencialidad es necesario aún mucho trabajo en el sentido de rellenar el hueco existente entre las máquinas paralelas disponibles y los programadores no expertos en este tipo de aplicaciones.

En este último sentido han surgido lenguajes como High Performance Fortran (HPF) que pretenden ser una herramienta para explotar paralelismo con mínima intervención del programador. El Fortran es el lenguaje más ampliamente difundido entre la comunidad científica, con lo que el paso a HPF no entraña demasiadas dificultades. Los compiladores de HPF tratan de extraer el paralelismo inherente a los programas y explotarlo en una arquitectura paralela. El dejar "el trabajo duro" en manos del compilador puede verse inicialmente como una ventaja para el programador, pero también merma las posibilidades de obtener resultados satisfactorios con este tipo de herramientas: la eficiencia obtenida dependerá fuertemente de la calidad del compilador. Por otra parte, mientras que este tipo de lenguajes, y en particular HPF se muestran suficientemente adecuados para abordar problemas que involucren estructuras de datos regulares: matrices, mallas regulares, vectores, etc. aún tienen carencias en el terreno de abordar problemas con estructuras irregulares asociadas: grafos, árboles, matrices dispersas, etc. En esta línea, se está trabajando activamente en dotar al lenguaje de los elementos necesarios para eliminar esta carencia.

En el seno del Grupo de Paralelismo de la Universidad de La Laguna llevamos desde 1990 trabajando en el ámbito de los lenguajes paralelos orientados al modelo PRAM. El diseño e implementación del lenguaje *ll (La Laguna)* [Leo97], [San96] es uno de los frutos más significativos de este trabajo. *ll* es un lenguaje imperativo de propósito general, extensión de un subconjunto de Pascal. El lenguaje asume la existencia de una memoria compartida por todos los procesadores. En lugar de paso explícito de mensajes *ll* admite lecturas y escrituras de la memoria compartida. La primera versión del lenguaje y su compilador aparecen a finales de 1991 y constituyeron el núcleo de la memoria de licenciatura de la profesora León [Leo92]. En la memoria de licenciatura [San93], leída en 1993, presentamos una implementación de *ll* para redes de transputers. En la Tesis Doctoral de la profesora León [Leo96a] se aglutinaba todo el trabajo que hasta ese momento se había realizado en relación a *ll*, revisando exhaustivamente las características del lenguaje y presentando un compilador para el mismo.

NESL [Ble95] es un lenguaje funcional de primer orden (las funciones no pueden ser pasadas como datos), fuertemente tipificado y libre de efectos laterales, que se ejecuta en un entorno interactivo y que genera un código intermedio. El lenguaje utiliza la secuencia como tipo de datos paralelo primitivo, y el paralelismo se obtiene exclusivamente a través de operaciones sobre estas secuencias. En este sentido, el programador ha de idear operaciones paralelas sobre conjuntos de valores más que pensar en cómo se han de asignar datos a los procesadores. NESL soporta secuencias anidadas y el paralelismo se suministra a través de un conjunto de constructos de paralelismo de datos basados en secuencias, incluyendo un mecanismo para aplicar cualquier función sobre los elementos de una secuencia en paralelo, así como un conjunto de funciones paralelas que manipulan secuencias. El lenguaje no proporciona bucles secuenciales (que pueden simularse usando recursión) para promover la utilización del paralelismo. Obviamente la presencia de un código intermedio

interpretado limita las prestaciones del lenguaje. Aunque la eficiencia de las implementaciones sobre máquinas vectoriales puede ser considerada aceptable, el rendimiento obtenido sobre sistemas distribuidos es muy pobre.

Fork95 [Kes95a], es un lenguaje paralelo imperativo fuertemente síncrono diseñado para expresar algoritmos PRAM. El lenguaje está basado en C y ofrece constructos para dividir jerárquicamente los grupos de procesadores en subgrupos y manipular subespacios de memoria privada y compartida. *Fork95* es el sucesor de FORK, que fue un diseño de tipo teórico del que nunca llegó a existir un compilador. La complejidad de FORK acabó impidiendo al lenguaje toda posibilidad de utilización práctica. No obstante, en él aparecían ya las ideas en las que se basa *fork95*, como son el concepto de grupo de procesadores y la diferenciación entre espacios de memoria privados y compartido. La crítica más importante que podemos formular a *fork95* es que nunca se ha llegado a construir un compilador del lenguaje que produzca código para una máquina paralela real.

Ya en la tesis de la profesora León se indicaba que la proyección del modelo PRAM abstracto sobre una máquina real entrañaba una importante pérdida de rendimiento debido a que el modelo ignora completamente la jerarquía de memoria de las máquinas paralelas actuales. El presente trabajo es el resultado de la continuación natural de esta línea de investigación. A partir del año 96, nuestro objetivo fue el desarrollo de ideas y herramientas que permitieran una traslación eficiente de algoritmos PRAM a máquinas paralelas comerciales. El modelo de Computación Colectiva [Rod99a] que presentaremos en el capítulo cuatro es el resultado de el esfuerzo realizado en esta dirección y *La Laguna C* [Rod97a] es la herramienta que hemos desarrollado basada en el modelo.

Para mejorar el rendimiento del equilibrado de la carga de trabajo al mismo tiempo que se mantiene la capacidad de anidar sentencias de asignación de procesadores paralelas se introducen nuevas sentencias de asignación cuya efectividad es analizada sobre el modelo PRAM y sobre sistemas distribuidos [Rod98d], [Rod98e].

Curiosamente, existen huecos o imprecisiones en la literatura sobre el modelo PRAM en cuanto se refiere a la asignación de procesadores y a la virtualización de los mismos de manera que la carga de trabajo resulte equilibrada. En esta memoria se establecen las condiciones exactas para el cumplimiento del conocido Teorema de Brent.

Para intentar paliar el principal inconveniente del lenguaje *fork95* que hemos mencionado anteriormente hemos puesto en marcha un proyecto conjunto entre nuestro grupo y el de la SB-PRAM cuyo objetivo es el desarrollo del back-end del compilador de *fork95* para el sistema operativo PRAMOS de la SB-PRAM.

El Modelo de Computación Colectiva que presentamos también en esta memoria permite la traslación eficiente de algoritmos con paralelismo de datos anidados sobre arquitecturas paralelas reales. El modelo viene caracterizado por una tripleta (*M, Div, Col)*. *M* representa la plataforma paralela (de memoria distribuida o compartida), *Div* es el conjunto de funciones de división y *Col* el conjunto de funciones colectivas. Una función se dice colectiva cuando es realizada por todos los procesadores del conjunto actual. Los conjuntos de procesadores pueden ser divididos utilizando las funciones de *Div*.

Hacemos una propuesta para una implementación eficiente de los procesos de división [Rod98d]. Aunque otras librerías como MPI ofrecen funciones de división, el coste de las mismas, como se demuestra en esta memoria, no las hace adecuadas para ser utilizadas de forma intensiva. La idea subyacente a nuestra propuesta es que cada uno de los procesadores de uno de los conjuntos producto de la escisión mantiene una

relación con uno de los procesadores en los otros subconjuntos. Esta relación determina las comunicaciones de los resultados producto de la tarea realizada por el conjunto al que el procesador pertenece. Esta estructura de división da lugar a patrones de comunicaciones que se asemejan a los de un hipercubo. La dimensión viene determinada por el número de divisiones demandadas mientras que la aricidad en cada dimensión es igual al número de subconjuntos solicitados. A semejanza de lo que ocurre en un hipercubo *k*-ario convencional, una dimensión divide al conjunto en *k* subconjuntos comunicados a través de la dimensión. Sin embargo, los subconjuntos opuestos según una dimensión no tienen porqué tener el mismo cardinal. A estas estructuras resultantes las hemos denominado Hipercubos Dinámicos.

Presentamos una clasificación de problemas paralelos en función de las características de los datos de entrada y de salida de los mismos con respecto a la visión que de ellos tienen los procesadores de la máquina. La nomenclatura introducida se utiliza para caracterizar los problemas que presentamos en la memoria.

Aportamos ejemplos de algoritmos tanto del tipo de los que hemos denominado de Computación Colectiva como de Computación Colectiva Común [Rod99b]. Este último tipo de algoritmos resuelven un tipo concreto de problemas según la clasificación introducida. Para ambos tipos de algoritmos estudiamos diferentes formas de introducir equilibrado de la carga de trabajo y los resultados que produce cada una de ellas.

Hemos diseñado una herramienta, a la que denominamos *La Laguna C* que representa una implementación concreta de las ideas subyacentes al Modelo de Computación Colectiva y exponemos los resultados computacionales obtenidos para varios algoritmos en diferentes arquitecturas.

La memoria está estructurada en seis capítulos. En el primero de ellos describimos las plataformas software y hardware en las que hemos desarrollado *llc* y en las que hemos realizado los experimentos computacionales que presentamos en el último capítulo de la memoria.

El capítulo dos está dedicado a la presentación del modelo de computación PRAM. En él introducimos el modelo, presentamos brevemente el lenguaje *ll* utilizando para ello algunos ejemplos y revisamos las diferentes implementaciones existentes. Discutimos en este capítulo la forma de implementar eficientemente una sentencia de asignación de procesadores del tipo de la que utiliza cualquier lenguaje orientado al modelo PRAM. Estudiamos también el caso de una sentencia de asignación de procesadores ponderada en la que el usuario ha de especificar una medida de la cantidad de trabajo correspondiente a cada tarea paralela. Se precisa el concepto de aceleración, se determinan las condiciones exactas para el cumplimiento del Teorema de Brent y se finaliza con una propuesta de simulación de una PRAM por una red de procesadores interconectados según una topología de mariposa.

En el tercer capítulo presentamos la SB-PRAM [Abo93] y su lenguaje de programación, *fork95* [Kes95a]. La SB-PRAM es el núcleo central de un proyecto que se está llevando a cabo en la Universidad de Saarbrücken en Alemania, cuyo objetivo es la construcción de una PRAM. A pesar de las críticas que el modelo PRAM ha recibido como poco realista, los resultados que están surgiendo de este proyecto hacen concebir esperanzas respecto a la factibilidad de una implementación realista del modelo [Grü98]. En este capítulo estudiamos tanto las características más notables de la arquitectura SB-PRAM como *fork95*, el lenguaje de programación específicamente diseñado para la SB-PRAM, que ha sido desarrollado por el profesor Kessler, como parte del proyecto. Junto con *ll*, *fork95* es el único lenguaje que conocemos que reúne todas las características necesarias para ser considerado un auténtico lenguaje orientado

al modelo PRAM. Debido a que hasta fechas muy recientes no existía el primer prototipo de la SB-PRAM, el código producido por el compilador de *fork95* sólo puede ser ejecutado a través de un intérprete.

El cuarto capítulo formaliza los conceptos de Hipercubo Dinámico y de Modelo de Computación Colectiva. Se exponen varios ejemplos que sirven para explicar las características del modelo y que son utilizados para establecer la eficiencia del mismo. Se introduce *La Laguna C* como implementación de las ideas del modelo y se presentan diferentes algoritmos desarrollados con esta librería.

El quinto capítulo está dedicado a repasar otras aproximaciones al paralelismo de datos anidado: NESL, V y Aperitif. Estas aproximaciones, al contrario que *ll* o *fork95* no están enfocadas a paralelismo PRAM, pero muestran características relevantes desde el punto de vista del paralelismo PRAM.

En el sexto y último capítulo se presentan los resultados computacionales de los experimentos realizados en este Trabajo utilizando diferentes plataformas: Cray T3E, Cray T3D, SGI Origin 2000, Digital Alphaserver, Hitachi SR2201 y una IBM SP2. La mayoría de las experiencias computacionales han sido desarrolladas utilizando las ideas del Modelo de Computación Colectiva e implementando los algoritmos en *La Laguna C*. Se presentan resultados para la Transformada rápida de Fourier, seis diferentes versiones de un algoritmo de ordenación Quicksort paralelo, un algoritmo para el cálculo de la envoltura convexa de una nube de puntos, un algoritmo de ordenación Quicksort distribuido y un algoritmo de búsqueda.

Capítulo I
PLATAFORMAS SOFTWARE Y HARDWARE

Capítulo I

Plataformas software y hardware

1.1. Introducción

En este primer Capítulo pasamos revista a las diferentes plataformas software y hardware relacionadas con los experimentos que presentaremos en el Capítulo 6 de la memoria.

Como herramientas de desarrollo software presentaremos PVM (Parallel Virtual Machine) y MPI (Message Passing Interface) dos de las librerías de programación con paso de mensajes de mayor difusión en la actualidad. No pretendemos aquí dar una descripción exhaustiva de estas librerías sino que el lector tenga constancia del tipo de servicios que proporcionan. Los resultados computacionales que presentaremos en el Capítulo 6 han sido obtenidos programando con una herramienta propia a la que hemos denominado *llc* (La Laguna C) que se describe en detalle en el Capítulo 4. Disponemos de diferentes versiones de *llc*. Las primeras versiones que desarrollamos fueron para procesadores transputers y se diseñaron utilizando Inmos C [Inm90]. También disponemos de versiones en PVM y MPI de esta herramienta, y los experimentos de esta memoria se realizaron con la versión que corre sobre MPI.

En cuanto a las plataformas hardware, hemos utilizado 7 máquinas y 6 plataformas diferentes. La Tabla 1.1 muestra el conjunto de máquinas utilizadas. La columna etiquetada "Posición" indica el puesto que la máquina ocupa en la lista de los 500 supercomputadores más potentes del mundo [Top]. Como vemos, se trata en todos los casos de arquitecturas punteras en el ámbito de la supercomputación. La arquitectura de Hitachi aparece también en esta lista, pero representada por una máquina con muchos más procesadores que la que nosotros hemos utilizado.

En la sección 1.3 de este Capítulo presentamos una breve descripción de cada una de estas plataformas exponiendo las características más notorias de cada una de ellas.

Máquina	Ubicación	# Procesadores	Posición
Cray T3E	EPCC, Edimburgo	368	27
Cray T3E	Ciemat, Madrid	32	308
Cray T3D	EPCC, Edimburgo	512	112
IBM SP2	Cesca, Barcelona	44	228
SGI Origin 2000	Cepba, Barcelona	64	277
Digital Alphaserver 8400	Cepba, Barcelona	10	-
Hitachi SR2201	EPCC, Edimburgo	8	-

Tabla 1.1 Plataformas Hardware

1.2. Plataformas Software

1.2.1. La máquina virtual paralela (PVM)

La PVM [Gei94] es un programa que permite que máquinas heterogéneas, tanto paralelas como secuenciales, colaboren como un recurso computacional concurrente único. La idea principal en que se basa PVM consiste en conectar diferentes computadoras, en principio con sistema operativo UNIX, para usarlas como un único computador paralelo. De esta forma, se podrán resolver problemas de elevado coste computacional.

La Máquina Virtual Paralela (PVM), comenzó a desarrollarse en 1989, en los Oak Ridge National Laboratories (ORNL). En el proyecto estuvieron involucrados: Vaidy Sunderam de la Universidad de Emory, Al Geist del ORNL, Robert Manchek, Weicheng Jiang y Jack Dongarra de la Universidad de Tennessee, Adam Beguelin de la Universidad de Carnegie Mellon y del Centro de Supercomputación de Pittsburg.

El modelo computacional de la PVM sigue la idea de "máquina virtual". Una máquina virtual es una colección de computadoras conectadas en red, que trabajan en un entorno de computación concurrente. Las características principales son:

- Heterogeneidad, escalabilidad, múltiple representación de datos y tolerancia ante fallos.
- Uso simultáneo de máquinas de diferentes arquitecturas con múltiples procesadores de diferentes tipos: escalares, vectoriales, etc.
- Interfaz gráfica para optimizar, mejorar, depurar y analizar los programas bajo este entorno.

La PVM está compuesta por un conjunto de primitivas de usuario y por un programa que controla la máquina virtual. La computación concurrente se consigue conectando a través de una red una serie de elementos de proceso (EP) que se comunican utilizando el sistema de intercambio de mensajes. Estos elementos de computación o de proceso están conectados en una o más redes que pueden ser de distintas topologías y tecnologías (Ethernet, fibra óptica, par trenzado, coaxial, etc.).

Una tarea es una unidad de computación equivalente a un proceso Unix. Estas tareas se podrán escribir en C o en Fortran y la única diferencia será la incorporación de las rutinas para implementar el paralelismo (creación de una nueva tarea, paso de mensajes, etc.). Denominaremos "aplicación pvm", al programa realizado por el usuario que utiliza la forma de trabajo de la máquina virtual.

La plataforma PVM se compone de dos partes:

1. Un demonio (*daemon*): programa llamado "*pvmd*" que se debe ejecutar en aquellas computadoras que constituyan la máquina virtual. Está diseñado para que cualquier persona pueda instalarlo en su máquina. Cuando un usuario quiere ejecutar una aplicación PVM ejecuta primero el demonio en

una de las máquinas, y éste se encarga de activar el resto de demonios de las máquinas que forman la PVM. A partir de aquí, el programa de aplicación podrá iniciar su ejecución desde cualquiera de las máquinas. Un aspecto importante es que se pueden estar ejecutando varias máquinas virtuales de forma solapada donde cada usuario trabaja con su propia máquina virtual.

2. Librería de rutinas: Contiene las funciones que se podrán ejecutar una vez esté activa la PVM. Entre otras, contiene rutinas para :
    - Control de procesos
    - Crear, enviar y recibir mensajes
    - Información del sistema
    - Configuraciones dinámicas (añadir y eliminar máquinas)
    - Señalización
    - Mensajes de error
    - Empaquetar datos en mensajes
    - Gestionar grupos de procesos

Cuando un usuario activa un demonio "*pvmd*" en un EP, puede especificar un fichero de entrada que contiene una lista de todos los EP que van a formar parte de la máquina virtual. Este demonio se encarga de activar cada uno de los demonios de los EP que forman la máquina virtual. A partir de este momento, se establecen los puertos de comunicaciones entre cada uno de estos demonios. Toda actividad de control entre los demonios, el control del tráfico de mensajes entre ellos y la llegada correcta de los paquetes, se realiza en estos instantes.

Los puertos de comunicaciones (*sockets*) se establecen por medio de circuitos virtuales (TCP) o datagramas (UDP) entre cada demonio existente en la máquina virtual y sus respectivas tareas locales, o también entre tareas en el mismo o diferentes EP. Estos *sockets* realizan la salida estándar y los mensajes de error estándar, que se utilizan principalmente para la depuración de los programas.

Los programas de aplicaciones están compuestos de "componentes" que son las tareas. Cuando se ejecuta una de estas componentes, múltiples instancias de cada componente podrían iniciarse. Para formar parte de la máquina virtual, un proceso PVM debe realizar una llamada a una rutina de la librería, la cual se encarga de establecer un *socket* TCP entre estas componentes y el demonio local. El demonio local se encarga entonces de informar al resto de las computadoras que conforman la máquina virtual, para que actualicen sus tablas de localización de componentes.

Una componente sólo se comunica con su demonio local. Toda petición para iniciar otro proceso en otra máquina, o peticiones de envió de mensajes a otras máquinas son coordinadas por los demonios. Como cada demonio contiene las tablas de componentes, en todo momento sabrá a qué máquina ha de enviar los mensajes. Cuando un demonio recibe un mensaje de otro demonio, pasa el mensaje a la componente adecuada en la máquina local.

El diseño global de la PVM con el uso de demonios conectados por *sockets* UDP, fue elegido por tres razones:

1. La red no se bloqueará.
2. Los *sockets* UDP implican una menor sobrecarga para configurar y mantener que los sockets TCP.
3. Las componentes no necesitan conocer la localización de ninguna otra componente ni tienen que interrumpir sus tareas para manejar los datos que llegan.

Los programas de aplicaciones ven el sistema PVM como un recurso de computación paralelo general y flexible.

1.2.1.1. Principales características de PVM

a) Interfaz de usuario

La interfaz con el usuario es una consola donde se pide información del estado de la máquina virtual. Puede indicar qué procesos están activos, qué computadoras forman la máquina, etc.

b) Identificación de las tareas

Todo proceso dentro de la PVM tiene asignado un número de identificación de tarea al que denominaremos "*tid*", identificador de tarea. Cada *tid* es único dentro de toda la máquina virtual, y está controlado directamente por los demonios, nunca por el usuario. Este sólo puede leer el *tid* de una tarea en cuestión. En PVM 3.3, existen varias rutinas de la librería para manejar los *tids* (*pvm_mytid()*, *pvm_spawn()*, *pvm_parent()*, *pvm_bufinfo()*).

c) Control de los procesos

El software PVM proporciona rutinas que permiten al proceso de usuario convertirse en una aplicación *pvm* y luego volver a ser un proceso del usuario al terminar su tarea. Existen rutinas para añadir y eliminar máquinas de la configuración de la máquina virtual. También existen rutinas para enviar señales a otras tareas y rutinas para acceder a información acerca de la máquina virtual.

d) Tolerancia ante fallos

En cualquier momento puede que una máquina integrada en la PVM deje de estar activa, con lo cual la PVM deberá actualizar sus tablas. El estado de un EP puede ser requerido por cualquier aplicación. En cualquier momento puede ser necesario añadir un nuevo EP a la máquina virtual. El programador de aplicaciones tiene toda la responsabilidad para gestionar esta característica. La PVM no realiza en ningún momento intentos de recuperación automática de tareas que fueron eliminadas debido a un fallo del EP.

e) Grupo de procesos dinámicos

Un proceso puede pertenecer a varios grupos de procesos. Estos grupos se manejan a través de funciones específicas que incorporan un proceso a un grupo o lo sacan de él. Existen muchas situaciones donde todos los procesos deben coordinar algunas operaciones.

f) Señalización

La PVM proporciona dos métodos para enviar señales entre tareas. Un método consiste en enviar una señal Unix a otra tarea. El segundo método consiste en que una tarea notifica a un grupo de tareas acerca de un evento enviando un mensaje con una etiqueta específica, definida por el usuario y que la otra tarea puede interpretar.

g) Comunicaciones

La PVM aporta rutinas para empaquetar, desempaquetar, enviar y recibir mensajes entre tareas. Existen envíos asíncronos a una tarea o a un grupo de ellas. Los *buffers* para los mensajes se habilitan de forma dinámica. El tamaño máximo de un mensaje que podrá ser enviado o recibido está únicamente limitado por la cantidad de memoria disponible en la máquina.

h) Integración de sistemas multiprocesadores

La PVM fue desarrollada en principio para unir máquinas conectadas a una red. Se han incorporado herramientas para que los sistemas multiprocesadores puedan convivir en este entorno. Los mensajes entre dos procesadores de la misma máquina viajan directamente entre ellos, en cambio los mensajes a otras máquinas de la red o de otras redes viajan a través de sus demonios. Estos se encargarán de encaminarlos al lugar apropiado ya que conocen, por sus tablas, la configuración de la máquina virtual.

1.2.1.2. Primitivas PVM

a) Control de procesos

pvm_mytid():

Esta función incorpora a la PVM al proceso que la invoca, proporcionándole un número de tarea único en toda la máquina virtual, denominado *tid*. Esta función debe ser la primera rutina que se invoque antes de cualquier otra.

pvm_exit():

Comunica al demonio local que el proceso está abandonando la PVM. Esta primitiva no elimina el proceso en sí, sino que éste continúa pero como un proceso convencional de UNIX.

pvm_spawn():

Inicializa un conjunto de tareas que se incorporarán a la PVM. Para determinar en qué EP se van a crear estas nuevas tareas, la heurística utilizada podría estar diseñada en función de las medidas de carga de los EP de la PVM y de la capacidad de cómputo de los mismos. La función devuelve el número de tareas creadas en una o más máquinas de la configuración.

Si las tareas han sido inicializadas correctamente, como parámetro de salida, devuelve un vector con los *tids* y en caso que alguna tarea no haya podido ser inicializada, devuelve en su correspondiente componente del vector, un código de error, indicando el motivo por el que no ha podido iniciarse dicha tarea.

pvm_kill():

Elimina de la PVM la tarea identificada por el parámetro *tid*.

b) Información de la máquina virtual

pvm_parent():

Devuelve el *tid* del proceso que creó la tarea que invoca a esta función. Si existe algún error devuelve un error tipo *PvmNoParent*.

pvm_pstat():

Devuelve el estado de una tarea PVM identificada por *tid*. Devuelve *PvmOk* si la tarea está en ejecución, *PvmNoTask* si no lo está, y *PvmBadParam* si el *tid* es incorrecto.

pvm_mstat()

Devuelve *PvmOk* si el EP está ejecutando tareas PVM, *PvmHostFail* si no es posible acceder al EP, o *PvmNoHost* si ese EP no está en la PVM. Esta función es útil en el diseño de aplicaciones tolerantes a fallos. En todo momento se puede interrogar por el estado de cualquiera de las máquinas involucradas en la máquina virtual.

pvm_config():

Devuelve información acerca de la PVM, incluyendo el número de EPs, tipo de arquitecturas, etc.

pvm_tasks():

Devuelve información sobre las tareas en ejecución en la PVM.

c) Configuraciones dinámicas

pvm_addhost (), pvm_delhost ()

Estas primitivas permiten añadir o eliminar un EP de la configuración inicial de la PVM.

d) Primitivas de señalización

pvm_sendsig():

Envía la señal *signum* a otra tarea PVM identificada por un *tid*.

pvm_notify():

Provoca el envío de un mensaje a una serie de tareas especificadas, al ocurrir un evento en la PVM. Los posibles eventos que pueden ocurrir son: una tarea ha finalizado, una máquina ha caído o ha sido eliminada, o si se ha añadido una nueva máquina.

e) Primitivas de mensajes de error

pvm_error():

Imprime el estado de error de la última primitiva PVM.

pvm_serror():

Seleccionar esta rutina permite enviar mensajes de error de forma automática, y de esta forma cualquier error que se produzca, automáticamente notificará el mensaje de error asociado.

f) Envío y recepción de mensajes

El envío de mensajes conlleva los pasos siguientes:

- Se debe inicializar un *buffer* de envío a través de una primitiva particular.
- Los datos que van a ser enviados deben ser empaquetados (primitivas de empaquetado).
- El mensaje se envía a otro proceso (primitivas de emisión).

Un mensaje podrá ser recibido a través de una primitiva de recepción con o sin bloqueo. Después se debe desempaquetar cada elemento del mensaje. En la PVM, sólo existe un *buffer* de envío y otro de recepción activos por cada proceso en un instante dado. El usuario es responsable de gestionar qué tipo de mensaje está activo en cada momento, pudiendo crear los que sean necesarios.

pvm_mkbuf():

Crea un *buffer* de envío vacío y especifica un código para designarlo. Existen varias opciones para la codificación de los datos en este mensaje, dependiendo del valor de la constante *encoding*:

- *PvmDataDefault*. Codificación estándar XDR.
- *PvmDataRaw*. No realizar codificación.
- *PvmDataInPlace*. Los datos se quedan en el mismo EP.

pvm_initsend():
Limpia el *buffer* de envío y lo prepara para empaquetar nuevos datos. Esta primitiva debe ejecutarse antes de empaquetar los datos. Los valores de *encoding* son los mismos que en la función anterior.

pvm_freebuf():
Elimina el *buffer* identificado por el parámetro *bufid*. Se debe ejecutar esta primitiva siempre que se deje de utilizar este *buffer*.

pvm_*getsbuf*():
Devuelve el número del *buffer* de envío activo en ese momento.

pvm_*getrbuf*():
Devuelve el número del *buffer* de recepción activo.

pvm_setsbuf():
Selecciona el nuevo *buffer* de envío activo y devuelve el identificador del anterior.

pvm_setrbuf():
Selecciona el nuevo *buffer* de recepción activo, retornando el identificador del anterior. El empaquetado de los datos se realiza con diferentes primitivas que empaquetan un conjunto de elementos del mismo tipo en el *buffer* de envío activo. Estas primitivas pueden ser llamadas múltiples veces y en cualquier orden. De esta forma, un mensaje puede contener varios vectores de datos de distintos tipos. No existen restricciones relativas a cuan complejos pueden ser estos mensajes. La única condición que existe es que se deben desempaquetar en el mismo orden en que fueron empaquetados. Se dispone de primitivas para empaquetar bytes, enteros, reales, complejos, caracteres, etc.

El envío y recepción de datos se lleva a cabo con las siguientes primitivas.

pvm_send():
Esta primitiva etiqueta un mensaje con un valor entero, *msgtag*, que indica un número de mensaje, y lo envía a una tarea especificada por un *tid*.

pvm_mcast():
Etiqueta el mensaje con un identificador entero, *msgtag*, y lo envía a las tareas especificadas por *tids*.

pvm_nrecv():
Recepción sin bloqueo. Si el mensaje solicitado no ha sido recibido, esta primitiva devuelve un cero. Esta rutina puede ser invocada repetidamente para solicitar el mensaje y comprobar si ha llegado. Mientras el mensaje llega, se puede seguir realizando otro trabajo entre dichas llamadas.
Si llega un mensaje con etiqueta específica desde una tarea *tid* concreta, entonces se colocará el mensaje en el *buffer* activo.

pvm_recv():
Primitiva de recepción con bloqueo. Espera hasta que un mensaje con la etiqueta especificada y/o con *tid* determinado haya llegado.

pvm_bufinfo():
Devuelve información acerca del mensaje con el identificador especificado.

pvm_recvf():
Uso de definiciones propias de la primitiva de recepción de mensajes.

g) Primitivas de grupos de procesos

Estas funciones permiten controlar un conjunto de procesos que por cuestiones del problema a resolver, interesa que estén agrupadas. Cuestiones como la sincronización de los procesos, se pueden realizar permitiendo que todos los procesos pertenezcan a un grupo. Dentro de un grupo, se pueden realizar diferentes funciones que aquí comentamos. Cualquier tarea PVM puede incorporarse o abandonar un grupo de procesos en cualquier momento sin tener que informar al resto de tareas del grupo.

pvm_joingroup ():
Permite que una tarea se incorpore al grupo. Crea un grupo con el nombre especificado y asigna la tarea a ese grupo. A cada tarea, al incorporarse a un grupo, se le asigna un número, a través del cual se gestionan los grupos.

pvm_lvgroup():
Una tarea abandona un grupo invocando a esta primitiva. En caso de volver a incorporarse, el número asignado será posiblemente distinto al que tenía anteriormente. Los números se asignan dinámicamente.

pvm_gettid():
Devuelve el *tid* del proceso que está en un grupo con un número determinado.

pvm_getinst():
Devuelve el número que tiene una tarea dentro de un grupo.

pvm_gsize():
Devuelve el número de miembros que existen en un grupo.

pvm_barrier():
Cuando se invoca a esta función, el proceso se bloquea en espera de que todas las tareas implicada ejecuten esta función.

pvm_cast():
Etiqueta un mensaje con un identificador entero, y envía el mensaje a todas las tareas de un grupo determinado. Si una tarea se incorpora a un grupo durante la llegada de este tipo de mensajes, podría no recibirlo.

PVM es una herramienta que tuvo un gran auge entre los años 1993 y 1997. A partir de la aparición del "estándar" de paso de mensajes MPI, los desarrolladores de PVM se integraron en la nueva plataforma constituyendo la librería de paso de mensajes más importante que existe hoy en día.

1.2.2. Message Passing Interface (MPI)

MPI (Message Passing Interface) [Mpi94] es un sistema estandarizado y portable de paso de mensajes desarrollado por un grupo de investigadores del mundo académico e industrial con soporte en una amplia variedad de computadores paralelos. El estándar define la sintaxis y la semántica de un núcleo de rutinas de librería que resultan de gran utilidad a un amplio conjunto de usuarios que desarrollan programas de paso de mensajes en C o en Fortran.

El esfuerzo de estandarización de MPI implicó a más de 80 personas de 40 organizaciones, principalmente de Estados Unidos y Europa. La mayoría de los fabricantes de computadoras paralelas del momento estuvieron implicados en el desarrollo de MPI, junto a investigadores de universidades, de laboratorios gubernamentales y de la industria. El proceso de estandarización comienza con la reunión de trabajo "Standards for Message Passing in a Distributed Memory

Environment", celebrada en abril de 1992 en Virginia. En noviembre de 1992, una reunión del grupo de trabajo de MPI propone imprimir un carácter más formal al proceso de estandarización y decide realizar reuniones periódicas con una cierta regularidad a lo largo del año 1993. El borrador de MPI se presentó en noviembre de 1993 en la conferencia Supercomputing´93. Después de un periodo de comentarios públicos se produjeron algunos cambios en la versión original de MPI, la versión 1.0 [Mpi94] apareció en junio de 1994. Este conjunto de encuentros y la discusión de la nueva plataforma, constituyó lo que se conoce como el Forum MPI, abierto a todos los miembros de la comunidad de la computación de altas prestaciones.

MPI constituye una librería de paso de mensajes, esto es, una colección de rutinas que facilita la comunicación entre los procesadores de un programa paralelo de memoria distribuida. Se presenta como la primera librería estándar y portable que ofrece buenos rendimientos. No se trata de un estándar real puesto que no fue diseñada por una organización como ANSI o ISO. Por el contrario, se trata de un estándar por consenso diseñada en el Forum MPI.

Algunas de las características más importantes que ofrece MPI son las siguientes: comunicación asíncrona, gestión eficiente de los *buffers* de mensajes, gestión eficiente de grupos, un amplio conjunto de operaciones de comunicación colectiva, topologías virtuales, creación de tipos derivados de datos y varios modos de comunicación. Estas características confieren una gran riqueza y funcionalidad al numeroso conjunto de funciones (alrededor de 125) que constituyen la librería MPI.

Las primeras implementaciones de MPI comenzaron a desarrollarse simultáneamente con la definición del estándar y sirvieron al grupo MPI como fuente de experiencias en plataformas reales. Entre las implementaciones más importantes cabe destacar las de MPICH [Gro94], LAM [Bur94] y CHIMP [Ala94]. De todas ellas MPICH goza de gran popularidad y corresponde con la versión instalada en las plataformas hardware que hemos utilizado.

1.2.2.1. La plataforma MPICH

MPICH es una implementación gratuita de MPI que contiene todas las características del estándar, donde se mantiene la portabilidad del código MPI entre las diferentes máquinas paralelas y se intenta mantener al máximo su rendimiento.

MPICH pudo implementarse rápidamente debido a la existencia de librerías bien definidas en diferentes sistemas y a la experiencia de los autores en librerías similares. Entre los desarrollos previos podemos citar P4 [But92], Chameleon [Gro93] y Zipcode [Skj94]. En P4 se incluyen funciones básicas para el paso de mensajes y para memoria compartida. En un principio, MPICH estaba implementado utilizando P4 en los casos de redes TCP/IP y memoria compartida. Chameleon consiste en una librería de paso de mensajes de alto rendimiento que se implementa como macros de C y que se utiliza en muchas librerías de fabricantes de máquinas: NX de Intel y MPL de IBM, por ejemplo. Por último, Zipcode es un sistema para desarrollar librerías escalables. En MPICH se introdujeron ideas de Zipcode, como los contextos, los grupos y los comunicadores. Zipcode también contiene muchas rutinas colectivas que trabajan con topologías virtuales, conceptos que también heredó MPICH.

Arquitectura de MPICH

En el diseño de MPICH se tuvieron en cuenta las dos características más importantes: portabilidad y eficiencia. En el desarrollo de MPICH se intentó por un lado maximizar la cantidad de código general que sirviera a una gran cantidad de plataformas sin reducir el rendimiento del sistema. Por otra parte, se ofrece una estructura que

permita que MPICH sea transportado a otra plataforma con el menor número de cambios (sólo los dependientes de la plataforma). Las características de los grupos, atributos, comunicadores son compatibles en las implementaciones de los sistemas soportados.

El mecanismo que permite la portabilidad y eficiencia es una especificación denominada ADI (Abstract Device Interface) en MPICH. Todas las funciones MPI están implementadas en términos de funciones y macros definidas en la ADI. Todas las funciones soportadas en este nivel son portables. La especificación ADI debe proporcionar cuatro tipos de funciones: funciones para especificar que se va a enviar o recibir un mensaje, para transferir datos entre la interfaz de programación del usuario y el hardware de paso de mensajes y funciones para la gestión de mensajes en cola e información del entorno de ejecución. De esta forma, la ADI consta de funciones en términos de las cuales quedan expresadas las funciones MPI. En el caso de MPICH, la ADI contiene además el código de empaquetado de mensajes, gestión múltiple de *buffers*, detección de que los mensajes entrantes pueden leerse o deben encolarse y gestión de las comunicaciones heterogéneas.

En MPICH existen diferentes implementaciones de la ADI. Una de ellas es el "Channel Interface" (CI) que está compuesto por cinco funciones como mínimo y proporciona la forma más rápida de obtener una versión MPICH bajo otra plataforma. A esta implementación se le pueden incorporar nuevas funciones específicas de las arquitecturas para obtener mayor rendimiento.

Podemos describir los niveles existentes en la implementación de MPICH de la siguiente manera. En el nivel superior están las funciones MPI. Si estas son colectivas suelen estar implementadas en un nivel inferior con funciones MPI punto a punto (MPI_Send, MPI_Recv, etc.). Debajo de este nivel está la especificación ADI. Continúan las diferentes implementaciones que dependen de la plataforma hardware. El caso más portable es el Channel Interface y existen otras implementaciones, específicas para cada una de las plataformas: Meiko, T3D, SGI.

En el nivel más bajo, el CI, lo que necesitamos es enviar datos desde el espacio de direcciones de un proceso origen a otro en el proceso destino. Esto se puede realizar con muy pocas funciones (tan sólo cinco funciones en algunos casos). La implementación de CI se realiza con Chameleon, memoria compartida o con versiones especiales propias de las plataformas.

En el CI existen tres mecanismos para implementar el intercambio de datos:

1. "Eager": en este protocolo los datos son enviados al destino de forma inmediata. Si el destino no está en espera de los datos, el receptor debe proporcional espacio para almacenar los datos de forma local. Esta opción ofrece el mayor rendimiento, sobre todo cuando los niveles inferiores proporcionan las funciones de control y almacenamiento. Tiene la desventaja que con mensajes grandes puede dar problemas al no disponer de memoria suficiente el receptor. Es la opción por defecto en MPICH.
2. "Rendezvous". Los datos son enviados al destino sólo cuando éste los solicita. Cuando se ejecuta el "receive", el receptor envía una petición al origen solicitando los datos. Este protocolo es el más robusto pero puede ser el más ineficiente dependiendo de los niveles inferiores. Para utilizar este protocolo MPICH debe configurarse con el protocolo –use_rndv.
3. "Get": los datos son leídos por el receptor. En este caso es necesario algún método para copiar los datos de la memoria de un procesador a

otro. Este mecanismo ofrece altos rendimientos pero requiere de hardware especial como memoria compartida u operaciones sobre memoria remota.

1.2.2.2. Primitivas MPI

En MPI existen aproximadamente 125 funciones divididas en diferentes grupos según sus características. A continuación detallaremos aquellas funciones que hemos considerados más importantes, entre las que destacamos las utilizadas en este trabajo.

a) Envíos y recepciones punto a punto

MPI_Get_count():
Devuelve el número de elementos recibidos por una operación. Esta función inicializa la variable denominada *status*, que permite comprobar lo que ha ocurrido con la ejecución de la función.

MPI_Send():
Envía datos con bloqueo a un procesador específico.

MPI_Recv():
Espera a recibir datos de un origen. Existe un parámetro de salida que permite diferenciar entre los tipos de mensajes que se han recibido.

MPI_Sendrecv():
Envía el contenido de un *buffer* especificado a un receptor y recibe otro mensaje de otro emisor.

b) Modos de comunicaciones en punto a punto

MPI_Bsend():
Función de envío de datos utilizando *buffers* predefinidos por el usuario.

MPI_Rsend():
Función de envío de datos cuando el receptor está esperando los datos.

MPI_Ssend():
Envío síncrono. Esta función no continuará hasta que exista el correspondiente "receive" y la llegada de los datos al destino haya comenzado.

c) Asignación de espacio (*buffer*)

MPI_Buffer_attach():
Permite informar al sistema que se debe utilizar almacenamientos para los mensajes que se van a enviar. Permite evitar bloqueos para mensajes grandes.

MPI_Buffer_detach():
Libera la memoria anteriormente asignada.

d) Comunicaciones sin bloqueos

MPI_Ibsend():
Realiza un envío sin bloqueos con asignación previa de "*buffers*".

MPI_Irecv():
Comienza una recepción sin bloqueo con asignación previa de "*buffers*".

MPI_Irsend():
Realiza un envío sin bloqueos en el estado "ready".

MPI_Isend():
Realiza un envío sin bloqueos en el estado "estándar".

MPI_Issend():
Comienza una recepción sin bloqueos en el estado "síncrono".

MPI_Test():
Comprueba si la operación no bloqueante asociada al "handler" devuelto por la función que realizó el envío ha finalizado.

MPI_Wait():
Queda pendiente de que la operación que inicializó el "handler" haya terminado.

e) Comunicaciones colectivas

Las comunicaciones colectivas hacen uso de los comunicadores. Es importante el concepto de comunicador. En la ejecución de un programa un conjunto de procesos quedan ligados en un mismo entorno. Cuando un proceso precisa enviar información a través de una función colectiva, no hace referencia al nombre de cada proceso sino al comunicador al que todos los procesos pertenecen.

MPI_Barrier():
Bloquea la ejecución del proceso que la ejecuta, hasta que todos los procesos asociados al mismo comunicador hayan ejecutado esta misma función.

MPI_Bcast():
Envía el contenido del *buffer* de datos de salida a todos los procesadores con el mismo comunicador. El parámetro *root* indica quién es el emisor de los datos.

MPI_Scatter() y MPI_Scatterv():
Envía diferentes datos desde el emisor a cada uno de los receptores con el mismo comunicador. La diferencia entre ambas funciones reside en que la primera envía la misma cantidad de datos a todos los procesos y en la segunda esta cantidad se puede personalizar.

MPI_Gather() y MPI_Gatherv():
El receptor recibe de forma no personalizada o personalizada los datos de cada uno de los procesos que forman parte del comunicador.

MPI_AlltoAll() y MPI_AlltoAllv():
En este caso, los datos no personalizados y personalizados son enviados desde todos los procesos a todos los procesos del comunicador.

MPI_Allgather() y MPI_Allgatherv():
Equivalente a MPI_Gather() y MPI_Gatherv(), pero con la característica que todos reciben de todos.

MPI_Reduce():
Combina los contenidos de los operandos de cada uno de los procesadores utilizando una función predeterminada.

MPI_Allreduce():
Todos los procesadores del grupo realizan la operación anterior.

MPI_Reduce_scatter():

Combina los contenidos de los operando de los procesos y devuelve con una operación de uno a todos personalizada, los valores que a cada uno le corresponden.

MPI_Scan():
Ejecuta una operación de prefijos en paralelo.

f) Grupos, Contextos y comunicadores

MPI_Comm_group():
Devuelve el grupo al que pertenece el comunicador.

MPI_Comm_create():
Crea un nuevo comunicador con el conjunto de procesos que se le pasa a la función.

MPI_Comm_rank():
Devuelve el número de proceso dentro del comunicador. Es el nombre lógico del procesador.

MPI_Comm_size():
Devuelve el número de procesos involucrados con el mismo grupo.

g) Gestión de información del sistema y de errores

MPI_Get_processor_name():
Devuelve el nombre del proceso donde se ha ejecutado la función.

MPI_Wtick():
Devuelve la precisión de la función MPI_Wtime.

MPI_Wtime():
Devuelve un número de doble precisión indicando el número de segundos que han pasado desde un punto determinado.

MPI_Abort():
Aborta todos los procesos del comunicador.

MPI_Finalize():
Finaliza la ejecución del programa MPI.

MPI_Init():
Inicializa MPI para comenzar la ejecución de un programa.

1.3. Plataformas Hardware

1.3.1. Redes de área local

En una red de área local pueden existir máquinas diferentes interconectadas cooperando para resolver un problema paralelo. Se puede considerar que se comportan como un único ordenador que dispone de varios nodos interconectados mediante la red. Los componentes básicos son los ordenadores conectados a la red y los dispositivos de la red. Existen muchas variantes de redes de ordenadores, las características más importante son: el tipo de medio físico, la topología y el tipo de acceso al medio.

1.3.1.1. Tipos de medio físico

Las redes de área local pueden estar conectadas utilizando diversos medios físicos. La conexión por par trenzado consiste en un par de cables de cobre trenzados

entre sí y que pueden estar apantallados o no. Este tipo de cable viene heredado de las instalaciones de la telefonía fija y en la actualidad está ampliamente extendido. Se usa en la mayoría de redes locales del tipo Ethernet. Otra forma de conectar equipos en una red es con cable coaxial, que se compone de un hilo conductor central rodeado de una malla muy fina de hilos de cobre. El espacio que queda entre el hilo y la malla está ocupado por un material aislante. Todo está cubierto por un aislante exterior. La transmisión se realiza sin modulación alguna.

Los cables coaxiales de banda ancha son muy parecidos al coaxial de banda base anterior, sin embargo, en este tipo de cable se transporta la información a diferentes frecuencias. El cable de fibra óptica se compone de varios filamentos convenientemente protegidos. Cada uno de ellos consta de un núcleo cubierto por un revestimiento. La diferencia de índices de refracción entre estos dos materiales provoca que las señales luminosas se transmitan a través del núcleo.

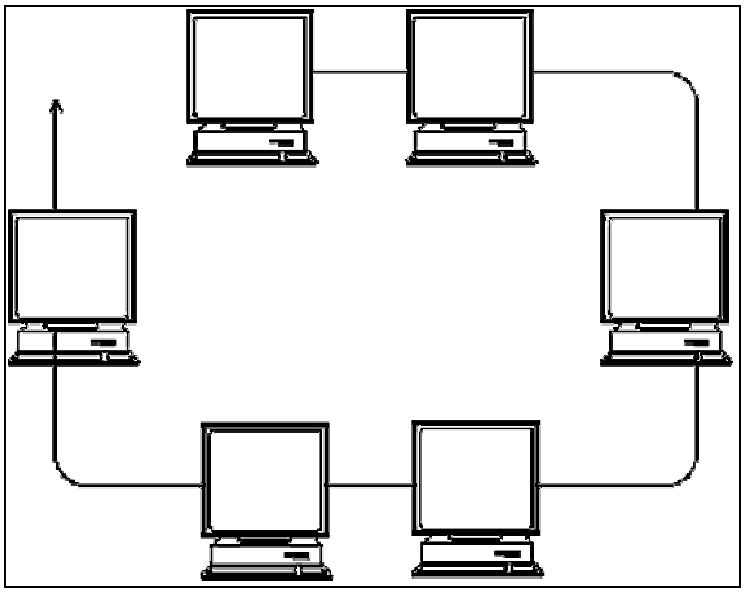

Figura 1.1 Red de estaciones de trabajo

1.3.1.2. Topologías

Con la electrónica adecuada, la topología que puede tener una red local podría ser muy variada, sin embargo, en la mayoría de las redes locales se utiliza una topología de anillo o de bus. Con el crecimiento del número de estaciones conectadas a la red y la consiguiente degradación en el rendimiento de la misma, se plantea la necesidad de segmentación de la red mediante la introducción de una técnica "*crossbar switch*". Si es necesario se puede llegar incluso a una solución "*crossbar switch*" completa en la que cada estación esta directamente conectada al "*switch*".

1.3.1.3. Modos de Acceso

Control centralizado: Polling

En las redes que utilizan control centralizado, existen dos tipos de nodos: el nodo principal que controla el acceso al medio de transmisión y los nodos secundarios. El nodo principal pregunta a cada uno de los nodos secundarios si tienen algo que comunicar. En caso afirmativo envían el mensaje y el nodo principal continúa preguntando al resto de nodos de forma secuencial.

Control Distribuido: Paso de testigo

Consiste en la circulación continua de una secuencia de bits especial llamada *testigo*. Uno de los bits del testigo indica su estado: libre u ocupado. Un nodo gana el derecho de acceso cuando recibe un testigo libre. Si tiene algún mensaje preparado lo transmite y genera un nuevo testigo que envía al siguiente nodo.

Paso de testigo en bus: Token-Bus

El paso del testigo se realiza superponiendo un anillo lógico sobre un bus físico, asignando direcciones de destino adecuadas a cada uno de los nodos. La ventaja principal de este sistema es la flexibilidad, mientras que el principal inconveniente consiste en la latencia que se introduce debido a que el testigo debe ser totalmente leído antes de comenzar a ser transmitido.

Paso de testigo en anillo: Token-Bus

El cableado físico proporciona ya el camino que va a seguir el testigo. Un testigo circula continuamente en el anillo, cuando una estación lo recibe y detecta que está libre, cambia su estado a ocupado y envía el mensaje justo después del testigo.

Acceso aleatorio

La idea de los métodos de acceso aleatorio consiste en acceder al medio siempre que se encuentre vacío. Debido a la latencia propia de la red varias máquinas pueden comenzar a enviar un mensaje sobre un medio en principio vacío, evidentemente la información se va a mezclar. Esta situación se denomina *colisión*. La diferencia entre los distintos métodos consiste en el mecanismo de contención o resolución de colisiones.

Acceso aleatorio a bus: CSMA/CD

Cada nodo inspecciona continuamente el estado del medio de transmisión e inicia una transmisión sólo si el medio se encuentra vacío. Debido a la propagación de las señales puede que otro nodo haya comenzado a transmitir también con lo que se produce una colisión. Para evitar esto los nodos esperan un tiempo aleatorio después de que el medio queda libre antes de comenzar a transmitir.

Acceso aleatorio a un anillo: Inserción de registro

Cuando un nodo tiene un mensaje que transmitir lo coloca en un registro de desplazamiento. Cuando se crea en el anillo un hueco apropiado, el registro se conecta en serie al anillo y comienza a transmitir. Si mientras tanto aparece una trama de datos por el lado de recepción, se va almacenando en el registro, esperando su transmisión posterior (excepto si éste es el nodo de destino). Por tanto, el nodo debe asegurarse que tiene espacio suficiente para almacenar los mensajes que le lleguen.

Las ejecuciones que presentamos en esta memoria se han realizado sobre un mismo segmento de la red de área local del Instituto de Astrofísica de Canarias [Iac]. Las máquinas utilizadas han sido estaciones de trabajo Sun Sparc Ultra-1 con una velocidad de reloj de 143 MHz interconectadas por un Etherswitch con acceso de tipo CSMA/CD.

1.3.2. IBM SP2

La multicomputadora IBM SP2 [Ibm] es un ordenador paralelo con memoria distribuida en el que los procesadores están interconectados mediante un subsistema de comunicaciones. Los nodos están basados en los procesadores simétricos PowerPC o P2SC. Pueden ser clasificados en tres tipos: *thin*, *wide* o *high*. IBM oferta varias posibilidades para el subsistema de comunicaciones, desde tecnología de red estándar, como Ethernet, FDDI (Fiber Distributed Data Interface) o ATM (Asynchronous

Transfer Mode), hasta desarrollos propios de IBM como el *High Performance Switch*, que ofrece los mejores rendimientos de las comunicaciones.

El subsistema de comunicaciones de altas prestaciones está compuesto por el *High Performance switch* y los adaptadores que conectan cada nodo al conmutador. Los adaptadores disponen de un microprocesador que descarga al nodo del trabajo relativo al paso de mensajes entre nodos. Además dispone de una memoria que utiliza como *buffer*. Un mecanismo de acceso directo a memoria se encarga de mover la información entre el nodo y el adaptador.

IBM describe el *High Performance switch* como "una red multietapa de conmutación de paquetes entre cualquier origen y destino, similar a una red *omega*". En esta red, el ancho de bisección crece linealmente con el tamaño del sistema, lo que garantiza la escalabilidad del mismo. El núcleo de la red es un chip crossbar que dispone de ocho puertos bidireccionales, que se puede usar para construir pequeños sistemas SP2. Los sistemas mayores deben usar tarjetas que contienen dos etapas de cuatro chips cada una, con lo que se dispone de un total de 32 puertos bidirecionales. Estos sistemas disponen además de al menos una etapa adicional para aumentar la redundancia de caminos. Existen configuraciones para 16, 48, 64 y 128 nodos.

La máquina con la que se realizaron las ejecuciones que presentamos en esta memoria fue la IBM SP2 del C4 [C4] (besiberri.sp.cesca.es), que utiliza el sistema operativo AIX 4.2.1. La máquina sufrió una actualización el 6 de junio de 1998. Hasta esa fecha la máquina estaba compuesta por 12+32 nodos (10 thin2, 2 wide y 32 thin120). Los nodos del grupo denominado 'pool9' en el que se realizaron las pruebas eran del tipo thin2 y fueron sustituídos por nodos de tipo thin160, con lo cual la configuración actual de la máquina es de 12 + 32 procesadores (42 thin160 y 2 wide), 12 GB de memoria principal, 494 GB en disco y un rendimiento punta de 27,41 Gflop/s. El cambio también afectó a la velocidad del HPS (High Performance Switch) cuya velocidad actual es de 40-80Mb/s con una frecuencia de reloj de 40MHz y un ancho de banda pico de 40MB/s. La latencia del switch con 64 nodos es de 500 microsegundos.

La Tabla 1.2 presenta algunas características técnicas de los procesadores de la IBM-SP2.

	wide	**Thin2**	**Thin120**	**thin160**
Frecuencia (MHz)	66	66	120	160
Ancho de bus	256	128	128	256
Cache de datos (Kb)	256	128	128	128
Rendimiento punta Mflops/s	266	231	406	640

Tabla 1.2 Características técnicas de los nodos de la IBM-SP2 del Cesca

1.3.3. Silicon Graphics Origin 2000

Cualquier sistema Origin [Sil] se compone de un número de nodos interconectados entre sí mediante fibra. Cada nodo consta de uno o dos procesadores, memoria, un directorio para coherencia de cache, y dos interfaces: una que conecta con el sistema de entrada/salida (XIO) y la otra con el sistema de interconexión (CrayLink).

Los módulos de procesadores

Cada módulo de procesadores contiene uno o dos procesadores R10000, cache de segundo nivel (1 ó 4 Mbytes), memoria principal, un directorio para el mantenimiento de la coherencia de la cache, un *hub*, una interfaz de entrada/salida y una interfaz para conectarse a la fibra.

Los sistemas Origin usan memoria compartida distribuida. La memoria se encuentra distribuida en los módulos de procesadores pero es accesible por todos los

procesadores. Evidentemente el coste de acceder a la memoria disponible en el módulo local es mucho más bajo que el coste de acceder a cualquier otro banco de memoria de otro módulo.

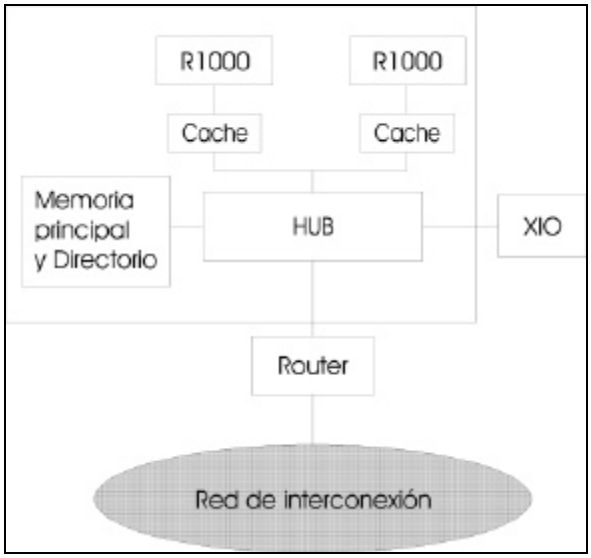

Figura 1.2 Estructura de un nodo de la SGI Origin 2000

El *hub* es un *crossbar switch* que dispone de cuatro puertos y conecta a los procesadores, la memoria principal y su memoria de directorio asociada, el sistema de intercomunicación y el subsistema de entrada/salida. Estas cuatro interfaces están interconectadas mediante un *crossbar* interno. Las interfaces se comunican enviando mensajes a través del *crossbar*. Los puertos para la memoria y los procesadores son bidireccionales y operan a 780Mb/s. Los puertos para entrada y salida y el *CrayLink* son dos half-duplex para cada uno y trabajan a 780 Mb/s. El *hub* controla tanto la comunicación intranodo entre cualquiera de los cuatro subsistemas que conecta como la comunicación con otros nodos. El *hub* se encarga de convertir los mensajes internos que usan un formato de petición/respuesta a los formatos externos que usan el sistema de entrada y salida y el CrayLink. Todos los mensajes internos son lanzados por el procesador o los dispositivos de entrada/salida. Las cuatro interfaces del *hub* actúan como controladores individuales de su propio subsistema. Cada interfaz tiene dos *buffers* FIFO, uno para los mensajes entrantes y otro para los mensajes salientes. Cuando llega un mensaje desde el exterior la interfaz lo convierte al formato *intra-hub* y lo coloca en la cola correspondiente. Los mensajes pueden ser clasificados como peticiones y respuestas. Cada *buffer* FIFO está dividido en dos a nivel lógico: uno para peticiones y otro para respuestas. El protocolo de coherencia de cache y los caminos separados para peticiones y respuestas garantizan que el sistema esté libre de interbloqueos.

El sistema de interconexión

El sistema de interconexión es un conjunto de conmutadores, llamados "*routers*", que están unidos por cables en varias configuraciones. Entre este sistema y el bus existen diferencias importantes:

- La fibra de interconexión es una malla de múltiples enlaces punto a punto conectados por conmutadores. Estos enlaces y conmutadores permiten que ocurran múltiples transacciones simultáneamente.
- Los enlaces permiten una conmutación extremadamente rápida.

- La fibra de interconexión no requiere arbitraje para su acceso ya que su uso esta limitado por contención.
- Cuando se incrementa el número de nodos se aumenta también el número de *routers* y de enlace, aumentando de esta forma el ancho de banda. En un sistema de *bus* compartido el ancho de banda es fijo.
- La topología del *CrayLink* permite que el ancho de banda de bisección crezca linealmente con el número de nodos en el sistema.

El ordenador SGI Origin 2000 en el que se realizaron las ejecuciones es karnak.cepba.upc.es, ubicado en el C4-Cepba [C4]. Durante la realización de este trabajo, la máquina sufrió una actualización con un cambio en la velocidad de los procesadores (pasaron de 196 a 250 MHz) así como en el sistema de interconexión: la máquina antigua (cuyo nombre lógico era karnak) estaba particionada en 2 máquinas de 32 procesadores, mientras que la máquina actual (karnak3) es una única máquina de 64 procesadores R10000 a 250 MHz con 8 GB de memoria principal, 288 GB en disco y un rendimiento punta de 32 Gflop/s. La máquina utiliza el Sistema Operativo IRIX 64.

1.3.4. Digital Alpha Server 8400

La Digital Alpha Server 8400 [Alp] es un sistema de bus compartido alrededor del cual se pueden conectar diversos dispositivos: uno o varios procesadores, módulos de memoria y módulos de entrada/salida.

El *bus* opera de forma síncrona con los procesadores a un submúltiplo de la frecuencia de reloj de éstos y puede llegar hasta los 100Mhz. Se compone de dos caminos separados: uno para datos de 256 bits de ancho y otro de instrucciones y direcciones de 40 bits de ancho. Para enlazar las direcciones o instrucciones que viajan por un *bus* con los datos que viajan por el otro se usa un número de secuencia. A este *bus* se conectan, a través de los *slots*, tanto los módulos que contienen procesadores como los que contienen memoria o dispositivos de entrada y salida. Dependiendo del modelo se dispone de un número diferente de *slots*, siendo nueve el mayor para el modelo 8400. Cualquier sistema debe disponer de al menos un módulo con procesadores, uno con memoria y uno de entrada/salida. El resto de *slots* se pueden completar con las siguientes limitaciones: hasta siete módulos con procesadores, hasta siete con memoria y hasta tres de entrada/salida.

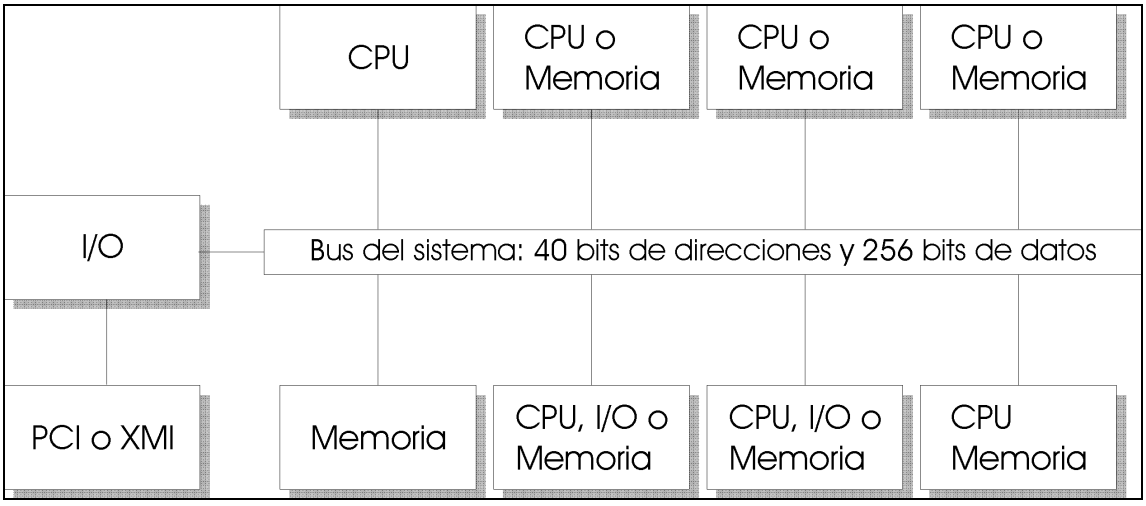

Figura 1.3 Esquema interno de la Digital Alphaserver 8400

Un modulo de procesador puede tener uno o dos microprocesadores Alpha 21164. Cada procesador dispone de su propia conexión con el *bus* y de su propia cache. Dentro

de cada procesador hay 8 Kbytes disponibles para cache de instrucciones, 8 Kbytes para cache de datos y 96 Kbytes de cache de segundo nivel. Además existe una cache de tercer nivel en el módulo para cada procesador con un tamaño de 4 Mbytes. Un sistema Alpha 8400 puede disponer de hasta siete módulos de procesadores, y por tanto, un total de 14 procesadores.

La memoria máxima que se puede conectar al sistema es de 28 Gbytes dividida en siete módulos de memoria de 4Gbytes cada uno. La memoria está dividida en bloques para permitir memoria entrelazada. Cada módulo de 2 Gbytes dispone de dos vías de acceso a memoria. Un sistema con 28 Gbytes dispone de 16 vías. El número de vías disponibles depende de la configuración de la memoria (número de módulos y tamaño de cada uno de ellos).

Existen dos tipos de módulos de entrada y salida, los que implementan buses PCI (Peripherical Component Interconnect) y los que implementan buses XMI. En ambos casos cada módulo dispone de 12 slots.

Los experimentos de esta memoria se realizaron en la Digital AlphaServer 8400 kemet.cepba.upc.es ubicado también en el C4-Cepba [C4], que tiene 10 procesadores Alpha 21164 a 440 Mz con 2 GBytes de memoria principal, 4Mb de cache, 60 GBytes en disco y una velocidad punta de 8,80 Gflop/sec.

1.3.5. CRAY T3E

El CRAY T3E [Cra] es un sistema multiprocesador de memoria compartida distribuida que soporta hasta 2048 procesadores interconectados mediante una red con topología toroidal tridimensional. Cada nodo contiene un procesador Alpha 21164, un chip de control del sistema, memoria local y un *router*. La lógica del sistema trabaja a una velocidad de 75 Mhz, mientras que los procesadores trabajan a algún múltiplo de esta velocidad (300 Mhz en el CRAY T3E o 450 Mhz en el CRAY T3E-900). Los enlaces del toro tienen un ancho de banda agregado de 600 MB/s en cada dirección y un ancho de banda útil de entre 100 y 480 MB/s dependiendo del tipo de tráfico. El sistema de entrada y salida utiliza un canal llamado *GigaRing* con un ancho de banda de 267 MB/s para cada cuatro procesadores.

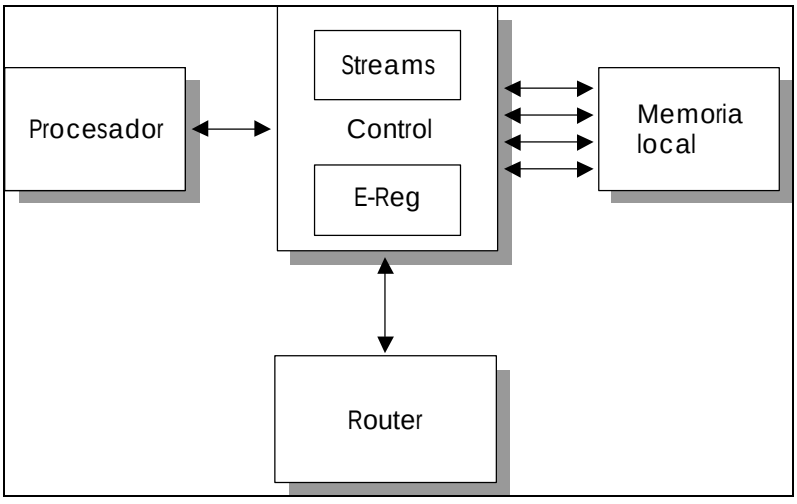

Figura 1.4 Esquema interno del Cray T3E

La memoria local consiste en un conjunto de cuatro chips controladores de memoria, que controlan directamente ocho bancos de memoria física. Cada controlador de memoria esta conectado con el chip controlador principal mediante un bus de 32 bits, con lo que dispone de una capacidad máxima de 1,2 GB/s. Este ancho de banda está mejorado además mediante un conjunto de *stream buffers*. Éstos dispositivos detectan

automáticamente referencias a direcciones consecutivas y realizan una prebúsqueda de la información en la memoria local. Un nodo de Cray T3E no dispone de memoria *cache* de segundo nivel.

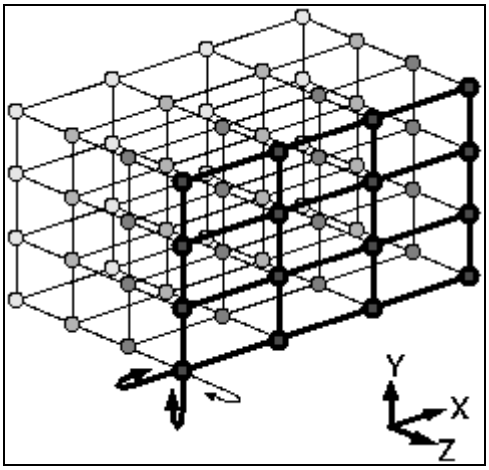

Figura 1.5 Estructura toroidal del Cray T3E

Los experimentos que presentamos en esta memoria se realizaron en los Cray T3E del Centro de Investigaciones Energéticas Medioambientales y Tecnológicas, Ciemat [Cie], crayc.ciemat.es, y en el del Centro de Computación Paralela de Edimburgo (EPCC) [Epc].

El Cray del Ciemat dispone de 32 procesadores DEC 21164 (Alpha EV-5), con una velocidad de reloj a 300 *MHz*, juego de instrucciones RISC de 64 bits, potencia de pico de 600 Mflops por procesador, Red toroidal 3D de interconexión de baja latencia, escalable hasta 136 procesadores, con l28 MBytes de memoria distribuida por procesador, escalable hasta 2Gbytes, con capacidad de l30 GBytes de disco duro SCSI y utiliza el Sistema Operativo Unicos/mk. El sistema de entrada/salida paralela está basado en la tecnología *GigaRing* de SGI/Cray.

La configuración normal de la máquina es la siguiente:

- 2 procesadores para el sistema operativo. En ellos se realizan tareas tales como el GRM (Global Resource Manager), servicio de ficheros y disco, manejo de los drivers para conexión a red, etc. Cualquier otro procesador obtiene estos servicios a través de ellos.
- 4 procesadores para comandos. Realizan tareas tales como atender las sesiones de los usuarios, procesos de compilación y comandos Unix en general. Soportan multiproceso (time sharing).
- 26 procesadores para aplicaciones paralelas. Son accesibles al usuario a través del comando mpprun. No soportan todavía multiproceso ni paginación de memoria (swap), por lo que corre en ellas un único proceso y no puede ser sustituido por otro hasta que acaba o se interrumpe.

El Cray T3E-900 del EPCC tiene procesadores Alpha (EV5.6) a 450 MHz con una potencia de pico de 900 Mflops por procesador. La mayoría de los procesadores están configurados con 128Mb de memoria o más. La configuración actual de la máquina es la siguiente:

- Un total de 368 procesadores Digital EV5.6
- 4 procesadores con 128Mbytes cada uno están dedicados en exclusiva al Sistema Operativo UNICOS/mk

- 5 Procesadores con 128 Mbytes y 8 con 256Mb se dedican a procesadores de comandos de usuario.
- 7 procesadores son redundantes y pueden mapearse para reemplazar procesadores que fallen.
- La máquina dispone de 216 procesadores para aplicaciones con procesadores de 64 y 128Mb de memoria.
- 112 procesadores para aplicaciones con mezcla de 128 y 256Mb de memoria.

Una característica del Cray T3E consiste en que el sistema operativo UNICOS/mk no soporta multitarea. Esto implica que cuando corremos una aplicación paralela en una fracción de la máquina, dicha aplicación es la propietaria de los recursos (procesadores, memoria) de esta fracción de la máquina, y ninguna otra aplicación puede hacer uso de ellos.

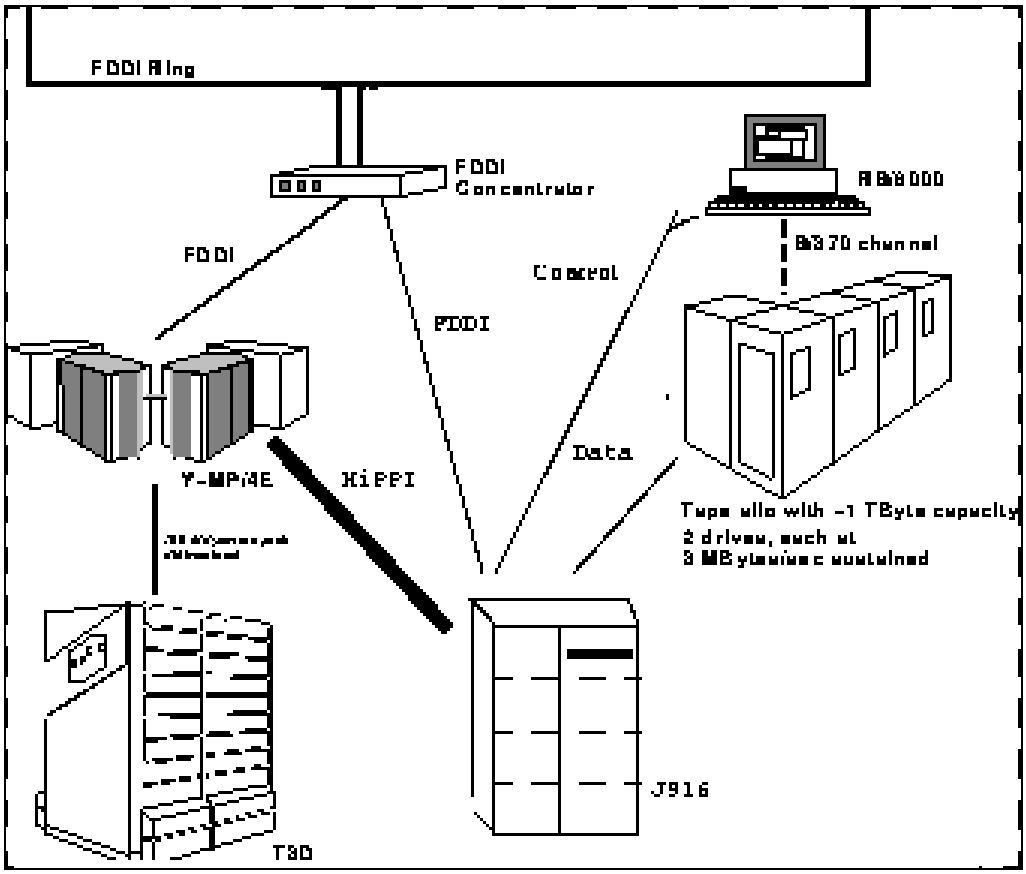

Figura 1.6 Instalación del Cray T3D en el EPCC

1.3.6. CRAY T3D

En los experimentos que presentaremos en el Capítulo 6 de esta memoria se utilizó también el Cray T3D MCN512-8 del Centro de Computación Paralela de Edimburgo (EPCC) [Epc]. Se trata de un modelo anterior al Cray T3E que describimos en esta sección.

La instalación del Cray T3D del EPCC incluye además un Cray Y-MP que actúa como front-end del T3D. El Cray T3D consiste en una matriz de 256 nodos con 2 procesadores DEC Alpha 21064 cada uno. La frecuencia de reloj de los procesadores es de 150Mhz y soportan operaciones sobre enteros de 64 bits y en punto flotante para operandos también de 64 bits conforme al estándar IEEE. La potencia de pico de los 512 procesadores es de 76.8 Gflop/s. Los procesadores incluyen caches de datos e instrucciones ambas con un tamaño de 8Kb. Cada procesador tiene 64Mb de memoria.

Los nodos están organizados en una estructura de toro tridimensional y los los 6 enlaces de cada nodo operan en paralelo soportando unos ratios de transferencia de 300Mb/s en cada enlace.

La máquina proporciona un espacio único de direcciones (cada procesador tiene acceso directo a la memoria de cualquier otro procesador, permitiendo que la memoria en su conjunto sea considerada como una única entidad.).

El T3D no se conecta directamente a ningún periférico sino que utiliza dos puertos de E/S para comunicarse con el Y-MP que proporciona todos los servicios de E/S al array. Esta conexión tiene un ancho de banda teórico de 400 Mb/s. bidireccional. La Figura 1.7 muestra la configuración del sistema.

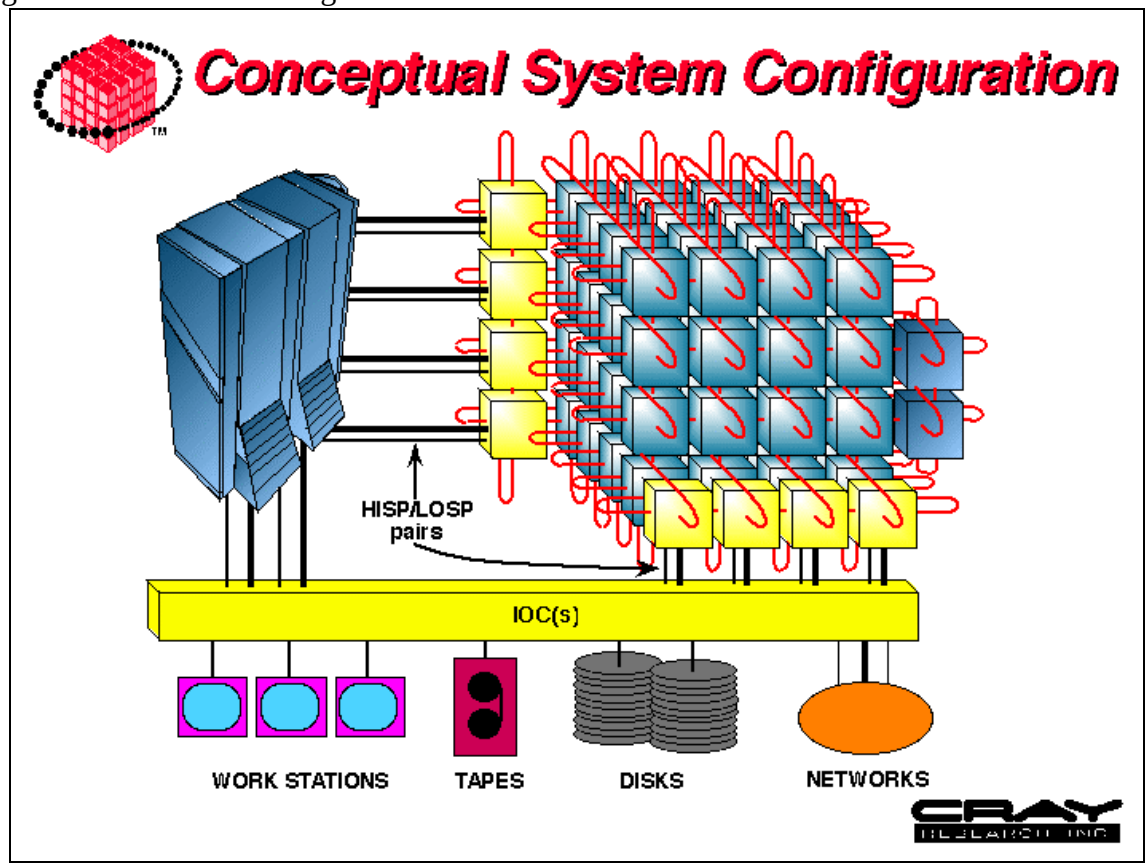

Figura 1.7 Estructura del Cray T3D

1.3.7. Hitachi SR2201

El Hitachi SR2201 es un computador paralelo de memoria distribuida con 8 procesadores HARP-1E a 150MHz. Los procesadores están basados en la tecnología PA-RISC de HP pero contienen importantes mejoras que les dotan de prestaciones que suelen estar asociadas con procesadores vectoriales sobre código vectorizable. La máquina puede expandirse hasta 2048 procesadores. Cada procesador tiene 256Mb de memoria RAM, caches primarias de datos e instrucciones de 16Kb y caches secundarias para instrucciones y datos de 512Kb.

La máquina utiliza para comunicaciones un crossbar tridimensional con un ancho de banda de 300Mb/s. La arquitectura es por tanto similar a la de otras máquinas de memoria distribuida (como los Cray T3D y T3E). La potencia pico es de 0.3Gflops en cada procesador. En la información suministrada en el EPCC se indica que su potencia es aproximadamente el doble que la del Cray T3D.

La innovadora característica de 'pseudovectorización' añadida a la arquitectura PA-RISC posibilita la carga y almacenamiento de datos entre registros y memoria

haciendo un by-pass sobre las caches. De este modo, una palabra de 8 bytes puede ser leída o escrita en memoria en cada ciclo de reloj sin interrumpir las unidades aritmético-lógicas. Hitachi añadió también en esta arquitectura TLBs (translation lookaside buffers) al diseño PA-RISC.

La máquina ejecuta una versión para multiprocesadores del HI-HX, un dialecto de UNIX.

Capítulo II
EL MODELO PRAM

Capítulo II

El modelo PRAM

2.1. Introducción

Una *(n, M)-PRAM* consta de *n* procesadores y *M* posiciones de memoria, donde cada procesador es una máquina de acceso aleatorio [For78]. Todos los procesadores comparten la memoria y se comunican a través de ella (Figura 2.1). Durante un paso dado, cada procesador puede leer un elemento de la memoria compartida en su memoria local o escribir un elemento de su memoria local a la memoria compartida o ejecutar cualquier operación *RAM* (operaciones aritmético-lógicas) sobre un dato contenido en su memoria local. En su formulación inicial es un modelo *Multiple Instruction Multiple Data* síncrono. Ello significa que los procesadores disponen de distintos contadores de programa y pueden estar ejecutando diferentes instrucciones en un instante dado, pero todos ellos tardan el mismo número de ciclos en ejecutarlas. A pesar de ello, la mayoría de los algoritmos *PRAM* propuestos explotan el modo *Single Instruction Multiple Data* en el que los procesadores ejecutan la misma instrucción sobre diferentes datos.

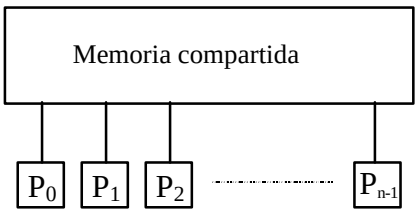

Figura 2.1 El modelo *PRAM*

2.1.1. Variantes según la resolución de los conflictos de acceso a memoria

El acceso concurrente de varios procesadores a la misma posición de la memoria compartida hace que surjan distintos modelos *PRAM*. Estas variantes son:

- La *EREW PRAM* (*Exclusive Read Exclusive Write*), donde sólo un procesador puede leer o escribir en una determinada posición de memoria a la vez.

- La *CREW PRAM* (*Concurrent Read Exclusive Write*) donde todos los procesadores pueden leer a la vez una determinada posición de memoria, pero sólo uno puede escribir en una posición de memoria en un instante.

- La *CRCW PRAM* (*Concurrent Read Concurrent Write*), donde múltiples procesadores pueden leer o escribir a la vez en una posición de memoria.

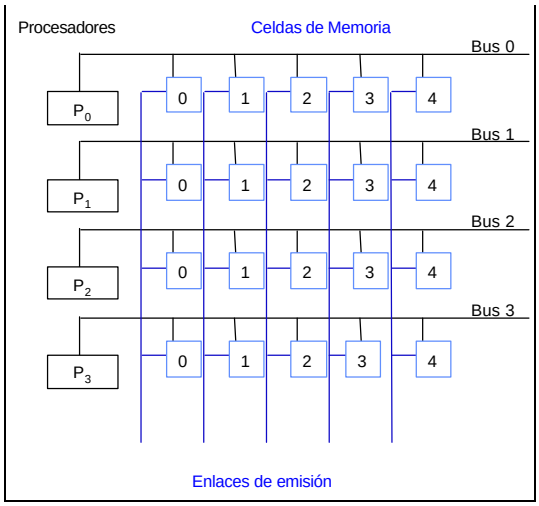

Figura 2.2 Memoria Compartida CREW de acceso constante

La *CRCW PRAM* se clasifica a su vez en cuatro variantes según su estrategia de resolución de conflicto en escritura, es decir, según qué se escribe cuando más de un procesador escribe en una misma posición de memoria en el mismo ciclo:

- *COMMON CRCW PRAM*, todos los valores escritos concurrentemente deben ser idénticos. Si no son todos iguales, se produce un error.

- *ARBITRARY CRCW PRAM*, el procesador que escribe el dato se elige arbitrariamente de entre los procesadores que escriben.

- *PRIORITY CRCW PRAM*, el procesador que escribe el dato es el procesador con mayor prioridad.

- *COMBINING CRCW PRAM*, el dato escrito es una combinación lineal de todos los valores que se escriben concurrentemente. Esta combinación se puede realizar con cualquier operación asociativa y conmutativa que sea computable en tiempo constante por una *RAM*.

2.1.2. Factibilidad del modelo PRAM

La simplicidad y generalidad del modelo *PRAM* lo han convertido en una herramienta de trabajo ampliamente aceptada, por lo que existe un gran número de algoritmos diseñados para el mismo. Sin embargo se suele cuestionar el realismo de este modelo no tanto en su calidad de modelo de programación como en su aspecto de

modelo de análisis de rendimiento. Para que la memoria compartida pueda ser accedida en tiempo constante debe existir un camino físico desde el procesador hasta la posición de memoria. Se necesitarán puertas lógicas que decodifiquen la dirección. Si la memoria tiene *M* posiciones, el coste de los circuitos de decodificación se expresaría como *f(M)*, siendo *f* alguna función de coste. Si *P* procesadores comparten esa memoria, el coste sube a *P*f(M)* lo que para valores de *M* y *P* grandes será excesivo. En la Figura 2.2 se muestra una posible organización de una memoria compartida CREW con costo *O(M*P)*, que fue propuesta en [Leo97]. Cada dirección de memoria *d* se replica en cada procesador. La columna de celdas etiquetada *d* representa la misma dirección de memoria *d*. Cuando un procesador P_i realiza un acceso de lectura simplemente lee de su réplica correspondiente usando su bus privado Bus_i. Cuando un valor es almacenado por el procesador P_i usando Bus_i en su réplica de la dirección compartida *d*, la celda receptora transmite inmediatamente el nuevo valor a las otras celdas replicantes usando el enlace de emisión vertical *d* correspondiente.

La suposición de que los accesos a memoria compartida ocurren en tiempo constante conduce en ocasiones a la concepción de algoritmos cuya eficiencia puede ser pobre en las máquinas paralelas actuales, especialmente si la labor de adaptación del algoritmo PRAM es dejada totalmente en manos de un compilador.

2.2. Activación y asignación de procesadores

Los algoritmos publicados para el modelo PRAM se describen utilizando un estilo general y algo difuso que, se considera, no necesita una definición formal para un lector con experiencia en el campo de la programación secuencial. Ejemplos de este estilo común pueden encontrarse en los libros de Quinn [Qui90] y de Gibbons y Rytter [Gib88]. La expresión del paralelismo se hace mediante una sentencia de la forma:

for all x **in** m..n **in parallel do** instruction(x)

La semántica de la sentencia consiste en:

(a) Asignar un procesador a cada elemento *x* en *{m,…,n}*.
(b) Ejecutar, en paralelo y para cada procesador asignado, todas las operaciones especificadas en *instruction(x)*.

La ejecución de la sentencia paralela termina cuando todos los procesadores implicados completan sus computaciones individuales.

Si se asume un coste logarítmico en el número de procesadores solicitados, entonces la complejidad del código en la Figura 2.3 será *O(K log (N))*. Sin embargo, y aunque los textos suelen omitir la explicación de esta cuestión, se suele cargar el código de la Figura 2.3 con una *complejidad O(K)*.

```
for i := 1 to K do
begin
  ...
  for all x in 1..N in parallel do
    instruction(x);
  ...
end;
```

Figura 2.3 Bucle secuencial con una sentencia *for all*

```
...
spawn(N);
...
for i := 1 to K do
begin
  ...
    for all x in a..b in parallel do
      instruction(x);
  ...
end;
```

Figura 2.4 Sentencias *spawn* y *for all*

Es esencial diferenciar entre los conceptos de sentencia de "activación de procesadores" y sentencia de "asignación de procesadores". En la presentación de algoritmos PRAM a menudo la activación de procesadores es denotada de la siguiente forma:

spawn(<Total Number Of Processors>)

mientras que la construcción **for all** anterior es utilizada para denotar la sentencia de asignación de procesadores. Un tiempo de arranque de orden *log(P)* es necesario para activar *P* procesadores. La idea es que el primer procesador activa al segundo y, en el paso general, el número de procesadores activos es doblado haciendo que cada uno de los procesadores activos active a uno diferente.

Puesto que al comienzo hay un sólo procesador activo, la ejecución de un programa PRAM tiene dos fases: en la primera es activado el número total de procesadores requeridos y, en la segunda, estos procesadores realizan la computación paralela ejecutando las sentencias de asignación interiores en tiempo constante (Figura 2.4).

Explicaremos cómo puede realizarse una sentencia de asignación de procesadores en tiempo constante. Consideremos que al comienzo de la computación disponemos de *M* procesadores que han sido proporcionados en tiempo *O(log M)* por la llamada a la sentencia *spawn(M)*. Suponemos la existencia de un número "físico" de índice *f*, con *f* en *H = {0,...,M-1}* asociado con cada procesador activo. El conjunto *H* de procesadores replica la ejecución de la parte secuencial del programa hasta que se alcanza una primera sentencia paralela de la forma:

for all *x* **in** *a..b* **in parallel do**
instruction(*x*)

El conjunto *H* de los *M* procesadores es particionado en $r = b\text{-}a+1 \leq M$ conjuntos $\{H_a, ..., H_b\}$.

Cada procesador, utilizando su número físico *f* decide en términos de una función que determina la **política de asignación** $Map(f) = x \in \{a,...,b\}$ a qué conjunto $H_{Map(f)} = H_x = \{f \in H / Map(f) = x\}$ se incorporará. La aplicación *Map* debe ser sobreyectiva, esto es, todo proceso debe tener asignado un procesador. La familia de conjuntos H_x constituye una partición de *H*:

$$H = \bigcup_{x \in \{a,....,b\}} H_x \; ; \; H_x \cap H_y = \emptyset \;\; si \; x \neq y; \;\; H_x \neq \emptyset \quad \forall x \in \{a,..,b\}$$

Cada procesador *f* asigna a su variable *x* el valor *Map(f)* entre *a* y *b*, cambia su valor de *M* a $M = |H_{Map(f)}|$ y su nombre físico de acuerdo con una función *F* de asignación de nombres, *f = F(f)* de tal manera que los nuevos nombres físicos de los procesadores del grupo H_x pertenecen al conjunto $\{0,...,|H_x|-1\}$. Para que *F* sea una

numeración correcta es necesario que cumpla la condición de ser inyectiva dentro de cada conjunto H_x:

$$\forall x \in \{a,..,b\}, \forall f, g \in H_x \;/\; f \neq g \Rightarrow F(f) \neq F(g)$$

Desde ese momento, los procesadores pertenecientes al conjunto H_x están todos en el mismo estado, tienen el mismo valor de la variable x y replican la misma ejecución paralela. Mientras que lo común es que un procesador soporte varias tareas (virtualización del procesador en varios procesos, como se explica en [Leo95a]), en esta etapa inicial en la que existen más procesadores que tareas, una tarea es soportada por varios procesadores (replicación de la tarea en los procesadores) [Rod98d].

$$Map: H \rightarrow \{a,...,b\}$$
$$f \rightarrow Map(f) = x$$

$$F:H \rightarrow \bigcup_{x \in \{a,...,b\}} \{0,...,|H_x|-1\}$$
$$Si\ f \in H_x \rightarrow F(f) \in \{0,...,|H_x|-1\}$$

Figura 2.5 La pareja (*Map, F)* define una política de asignación de procesadores

Una política factible para las funciones de asignación de tareas *Map* y de nombres físicos *F* es la de distribución cíclica:

$$s = M \bmod r;\ k = f \bmod r;$$

$$Map(f) = a + k = x;\ F(f) = f \operatorname{div} r; \qquad \text{(Ec. 2.1)}$$

$$M = M/r \text{ si } k \geq s \text{ y } M = M/r + 1 \text{ si } k < s$$

El tiempo invertido en la ejecución de esta política cíclica de la sentencia de asignación paralela es constante. Todas las evaluaciones únicamente implican variables privadas, sin que haya requerimientos de comunicación o accesos a memoria compartida. Cuando una nueva sentencia de asignación de procesadores es ejecutada por los procesadores en el conjunto H_x:

```
for all y in c[x]..d[x] in parallel do
  instruction(y)
```

el conjunto H_x es dividido en tiempo constante en $r_x = d[x]-c[x]+1$ subconjuntos: $\{H_{x_1},..., H_{x_{r_x}}\}$. El procesador f se incorpora al grupo $H_{Map(f)}$, asigna su identificador $y = Map(f)$ y cambia sus valores de $f = F(f)$ y $M = |H_{Map(f)}|$.

Siguiendo la política cíclica explicada anteriormente:

$$s = M \bmod r_x;\ k = f \bmod r_x;$$

$$Map(f) = c[k] + k = y;\ F(f) = f \operatorname{div} r_x; \qquad \text{(Ec. 2.2)}$$

$$M = M / r_x \text{ si } k \geq s \text{ y } M = M / r_x + 1 \text{ si } k < s$$

Este algoritmo muestra que la política de asignación de procesadores cíclica sólo requiere tiempo constante. Obviamente, el número de políticas de asignación a seguir es innumerable. Entre las alternativas a utilizar se encuentran las políticas de distribución por bloques y las mixtas por bloques y cíclicas. Una política de asignación de procesadores **óptima** es aquella que distribuye los procesadores de acuerdo con la

distribución de procesadores que requerirá la ejecución de *instrucción(x)*. Estudiaremos este tema más adelante en este mismo Capítulo.

2.2.1. Ejemplo

```
 1  ...
 2 spawn(6);
 3 ...
 4 for all x in 3..5 in parallel do
 5 begin
 6   ...
 7     for all y in 8..9 in parallel do
 8       begin
 9           ...
10      end;
11  ...
12 end;
```

Figura 2.6 Ejemplo con bucles *for all* anidados

Consideremos el código de la Figura 2.6.

En caso de éxito la llamada a *spawn* en la línea 2 activará seis procesadores físicos. Cada uno de estos procesadores tendrá sus valores de las variables $M = |H|$ y f con valores iniciales:

$M = \|H\| =$	6	6	6	6	6	6
$F(f) = f =$	0	1	2	3	4	5

La sentencia paralela de la línea 4 solicita *r=5-3+1=3* procesadores. De acuerdo con las fórmulas de la asignación cíclica, los nuevos valores son:

$M = \|H\| =$	2	2	2	2	2	2
$F(f) = f =$	0	0	0	1	1	1
$Map(f) = x =$	3	4	5	3	4	5

La sentencia de asignación paralela anidada en la línea 7 demanda *r=9-8+1=2* procesadores. Los nuevos valores para $M = |H|$, f e y computados siguiendo las fórmulas 0 son:

$M = \|H\| =$	1	1	1	1	1	1
$F(f) = f =$	0	0	0	0	0	0
$Map(f) = x =$	3	4	5	3	4	5
$y =$	8	8	8	9	9	9

```
C:\>ppc rbatcher.ll
rbatcher.eje generated
rbatcher.lst generated
C:\>_
```

Figura 2.7 Ejemplo de ejecución del compilador de *La Laguna*

2.3. El lenguaje La Laguna (ll)

El lenguaje *La Laguna* (*ll)* [Leo97], es un lenguaje orientado al modelo PRAM. Está basado en Pascal- [Han85], por lo que su aprendizaje es inmediato para cualquiera familiarizado con el lenguaje Pascal. Desde 1990 no han dejado de aparecer nuevas

versiones [Leo97], [Leo95a], [San96]. Existen versiones para diversas plataformas, tanto secuenciales como paralelas. El software y los manuales de *ll* pueden encontrarse en la referencia [Pra]. La capacidad de *ll* para expresar algoritmos PRAM es algo que ya hemos probado extensamente durante el tiempo de vida del lenguaje, tanto mediante su uso en el entorno académico (se han realizado diferentes proyectos de Ingeniería en Informática basados en el lenguaje) como a través de las diferentes publicaciones que al respecto hemos realizado.

```
program BatcherBitonicSort;
const  maxElements = 8191;  logmaxElements = 13;
type   vector = array [0..maxElements] of integer;
var  x: shared vector;   n ,  logn: shared integer;

procedure readVector;
var i: integer;
begin  readln(n); readln(logn);  for i := 0 to n-1 do read(x[i]); end;

procedure writeVector;
var i: integer;
begin   for i := 0 to n-1 do  write('x[',i,'] = ',x[i],' ');
end;

shared procedure bSort(left, right: shared integer;
                       direction: shared boolean);
begin
  if right > left then
  var center: shared integer;
  begin
     center := (left + right) div 2;
     parallel left..center do
     var i, temp1, temp2, xname, xi: integer;
     begin
         i := center+name-left+1; xname := x[name] ;  xi := x[i];
         temp1 := xname min xi;  temp2 := xname max xi;
         if direction then begin x[name] := temp1; x[i] := temp2; end
         else   begin x[name] := temp2; x[i] := temp1; end
     end;
     parallel do  bSort(left,center,direction) ||
                  bSort(center+1,right,direction);
  end
end (* bSort *);

procedure bitonicSort;
var   length, i, numOfIntervals: integer;
begin
   for i := 1 to logn do
   begin
      length := 1 shl i; numOfIntervals := n div length;
      parallel 0..numOfIntervals-1 do
      var first, last: integer;
      begin
         first := name*length;  last := first+length-1;
         bSort(first, last, (name mod 2 = 0));
      end;
   end;
end;
begin  readVector;   bitonicSort; writeVector; end.
```

Figura 2.8 Ejemplo rbatcher.ll

Considere el fichero fuente *rbatcher.ll* que se muestra en la Figura 2.8. La asignación de procesadores en *ll* se hace a través de una sentencia de la forma:

parallel <Expression1> .. <Expression2> do <Statement>

Se crean *<Expression2>-<Expression1>+1* procesos con identificadores de proceso entre *<Expression1>* y *<Expression2>*. La palabra reservada *name* denota el nombre lógico del proceso. Una segunda forma de asignar procesadores, utilizada en el procedimiento *bSort*, es a través de la sentencia de llamada paralela.

parallel do <procedure call> || ...|| <procedure call>

El cualificador *shared* es utilizado para indicar que una variable es compartida. Una variable compartida es aquella que será utilizada (accedida para lectura o escritura) no sólo por el proceso que le asigna espacio, sino por alguno de los procesos que fueron creados mediante sentencias de asignación de procesadores que fueron ejecutadas con posterioridad a la creación de la variable compartida. Cuando el cualificador *shared* se aplica a un procedimiento indica que ese procedimiento será invocado desde el ámbito de una sentencia paralela (por tanto su código debe ser compartido por varios procesadores). Por el contrario, una variable privada sólo puede ser utilizada por el proceso que la crea. Por defecto, en *La Laguna* las variables son privadas. *La Laguna* extiende el conjunto de operadores binarios de *Pascal.* Los operadores binarios *min* y *max* usados en el procedimiento *bSort* de la Figura 2.8 devuelven el mínimo y el máximo de los operandos. El código de la Figura 2.8 ordena el vector *x* utilizando el algoritmo de ordenación bitónica de Batcher [Bat82] que explicaremos más adelante en este Capítulo. En general, la semántica del lenguaje es la habitualmente utilizada en la mayoría de los artículos que utilizan el modelo PRAM. La Figura 2.7 muestra una compilación desde la línea de comandos.

```
PRAM Ver. 97.1
PRAM|T0>load 0, rbatcher
Load done.
P:    0 Name:   0    B:       0  S:     0: | PC:     0
G: 16 BT:   1023  SB:     0 SS:     0: | PROGRAM 1 2 51 1 1
PRAM|T0>go
8
3
4 5 2 7 8 9 1 6
x[0]=1 x[1]=2 x[2]=4 x[3]=5 x[4]=6 x[5]=7 x[6]=8 x[7]=9
PRAM|T1225>quit
```

Figura 2.9 Ejecución con el intérprete PRAM

El compilador de *ll* produce como salida un fichero *rbatcher.eje* y un fichero *rbatcher.lst*. El fichero *.eje* contiene el código intermedio ejecutable mediante el intérprete *pram.* El fichero *.lst* contiene el listado del código fuente y el correspondiente código ensamblador. La ejecución del fichero "*rbatcher.eje*" la realizaremos mediante una llamada al intérprete *pram* desde la línea de comandos. Es opcional especificar el nombre del programa a ejecutar en la línea de comandos:

*X>**pram rbatcher.eje***

Si no se especifica el nombre del fichero,

*X>**pram***

se puede cargar posteriormente mediante el comando *load addr, fileName*. Este comando tiene dos argumentos, en el primero se especifica la dirección y en el segundo el nombre del fichero.

En el ejemplo que nos ocupa, el comando *load 0, rbatcher* carga el ejecutable a partir de la posición 0 de memoria. Después de la carga del programa, el comando *load* muestra el estado del procesador con nombre físico 0 en el grupo inicial (Figura 2.9). Con el comando *go* se puede ejecutar el programa.

2.3.1. Implementaciones

Como se ha comentado, existen implementaciones del lenguaje para diferentes plataformas. En [San93] se puede encontrar un estudio detallado de la primera implementación paralela del lenguaje, que se realizó para redes de transputers [Car91] empleando occam [Inm88] como lenguaje de programación. En [Bar97] se implementa el compilador utilizando lex y yacc y se diseña también un intérprete secuencial del código producido por el compilador. En [Lun98] se desarrollaron intérpretes paralelos para el compilador utilizando PVM, MPI e Inmos C [Inm90]. Las últimas implementaciones del lenguaje [Lun98] tienen en cuenta las ideas del modelo de computación colectiva que presentaremos en el Capítulo 4. En estas versiones del lenguaje, el código *ll* es traducido por el compilador a código en C que es compilado a código paralelo con el apoyo de librerías para la ejecución paralela que han sido diseñadas utilizando paso de mensajes explícito con PVM, MPI o Inmos C.

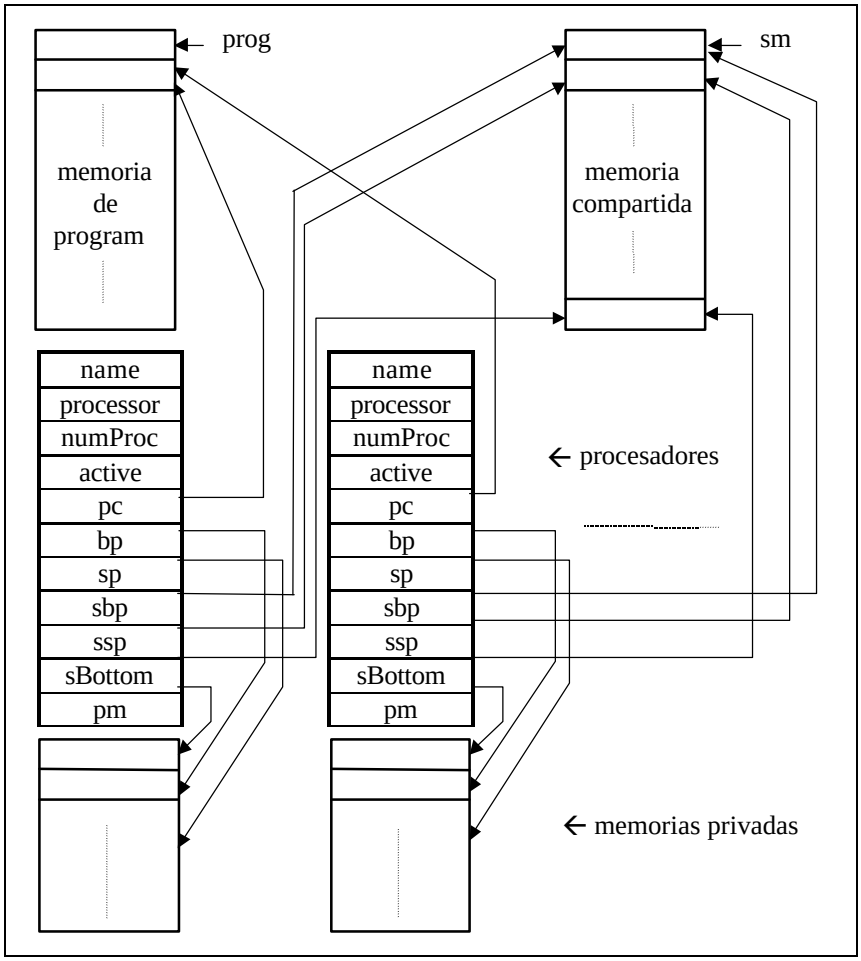

Figura 2.10 Arquitectura de la máquina objeto del compilador de *ll*

La Figura 2.10 es un esquema de la arquitectura genérica para la que el compilador de *ll* genera código. En las versiones interpretadas, el compilador genera código orientado a pila para esta máquina genérica. La máquina objeto del compilador consta de un número de procesadores que se puede cambiar a través de un fichero de

configuración, una memoria de programa común a todos ellos (todos los procesadores ejecutan el mismo programa), una memoria compartida y una memoria privada para cada procesador que son manejadas en forma de pila. Cada procesador está caracterizado por el conjunto de registros que aparecen en la figura, entre los que destacamos:

- *name:* número lógico del procesador en el programa;
- *numProc:* número de procesadores por grupo;
- *active*: flag indicador de si el procesador se encuentra activo o inactivo;
- *pc*: contador de programa;
- *bp*, *sp: puntero de base* y *puntero de pila* para acceder a la memoria privada del procesador;

```
 1 parallel <Expr1> .. <Expr2> do
 2   parallel <Expr3.. <Expr4> do
 3     begin
 4     S;
 5     end;

valor1 := Evaluación(Expr1)
Push(valor1)
valor2 := Evaluación(Expr2)
Push(valor2)
PATTERN
valor3 := Evaluación(Expr3)
Push(valor3)
valor4 := Evaluación(Expr4)
Push(valor4)
PATTERN
Código(S)
ENDPATTERN
ENDPATTERN
```

Figura 2.11 La traducción de sentencias paralelas anidadas

- *sbp*, *ssp*, *sBottom*: puntero de base, de pila y fondo compartidos para el acceso al segmento de la memoria compartida correspondiente a cada procesador.
- *pm*: memoria privada para cada procesador.

Las primeras versiones del compilador utilizaban un analizador sintáctico descendente recursivo, mientras que las últimas han sido escritas utilizando lex y yacc. El compilador reconoce una gramática muy similar a la de Pascal, que ha sido ampliada para soportar las sentencias paralelas, sentencias de semibloque, así como la declaración de variables y procedimientos compartidos entre otros elementos específicos.

En las versiones interpretadas del lenguaje (ya sea sobre máquinas secuenciales o paralelas) el compilador produce instrucciones orientadas a pila de la forma

$$OPCODE\ arg_1\ arg_2\ \ldots\ arg_N$$

la mayoría de estas instrucciones son ejecutadas por cada procesador de la máquina de forma individual, sin precisar la cooperación con otros. Así por ejemplo la instrucción ADD toma dos operandos de la pila (privada) del procesador, y los suma almacenando el resultado también en la pila. Las instrucciones PATTERN o SHASSIGN que se emiten como resultado de la traducción de una sentencia *parallel,* o una escritura a memoria compartida respectivamente, son ejemplos de instrucciones que sí precisan de comunicación entre los procesadores para su ejecución.

La Figura 2.11 muestra el esquema de traducción que el compilador utiliza para traducir las sentencias *parallel.* La instrucción PATTERN es la que cambia el estado paralelo de un procesador. Como consecuencia de la ejecución de la primera sentencia *parallel* (linea 1 en la Figura 2.11) el conjunto de procesadores de la máquina se particiona en *valor2-valor1+1* subconjuntos siguiendo una determinada política. La Figura 2.12 presenta un pseudocódigo de la implementación de la política cíclica que ya hemos presentado anteriormente.

```
1 Pop(A)
 2 Pop(B)
 3 n = B - A + 1
 4 IF n > NUMPROC THEN
 5    RunTimeError()
 6 IF (ACTIVE) THEN
 7   BEGIN
 8   S := NUMPROC mod n
 9   aux := PROCESSOR mod n
10   NAME := A + aux
11   PROCESSOR := PROCESSOR div n
12   NUMPROC := NUMPROC div n
13   IF (aux < S) THEN
14     NUMPROC := NUMPROC + 1
15   END
```

Figura 2.12 La implementación de la política cíclica de distribución de procesadores

En las primeras implementaciones del lenguaje se virtualizaban procesadores: un mismo procesador físico se encargaba de la simulación de un cierto número de procesadores virtuales o lógicos [Leo95a], [Leo96a] pero debido a la ineficiencia que esta simulación por software introduce, las últimas versiones producen un error si el número de procesadores demandados es superior al número de procesadores físicos disponibles, como vemos en el código de la figura (líneas 4 y 5).

En las implementaciones paralelas del lenguaje siempre hemos tomado la aproximación de una memoria compartida que está totalmente replicada en los procesadores de la máquina. Un acceso de lectura a memoria compartida queda así convertido en un simple acceso a la memoria local del procesador, mientras que un acceso para escritura a la memoria compartida ha de traducirse en el envío de mensajes para actualizar la copia de la dirección modificada que reside en otros procesadores.

Una sentencia condicional o una sentencia de repetición con una condición que es privada (que puede tener diferente valor en diferentes procesadores) provoca que haya procesadores que ejecuten el cuerpo de la sentencia, mientras que otros no lo ejecutarán. El registro *active* de los procesadores almacena este estado del procesador. Para mantener la sincronización PRAM entre los procesadores, si hay algún procesador que haya de ejecutar una parte del código, todos los procesadores de la máquina la ejecutarán, con la diferencia que unos lo harán como procesadores *activos* y otros como procesadores *inactivos*. De este modo, cuando se produce una escritura en memoria compartida todos los procesadores envían el mensaje correspondiente al valor que han escrito y todos reciben mensajes del resto de procesadores. Sólo aquellos procesadores que tienen activo su flag *active* realizarán actualizaciones de la memoria compartida. Con este modo de trabajo se garantiza en todo momento la coherencia de la memoria compartida.

```
 1 void replicateSHmem(int ACTIVE, int len, int addr, int ptr) {
 2   int i, j, k;          /* Contadores. */
 3   int **data;           /* informacion recibida de cada procesador. */
 4   int msglen;           /* Longitud de los mensajes. */
 5   int nummsg;           /* # de mensajes a enviar */
 6   int vecino;
 7   int atid, atag, alen;
 8   int a;                /* Direccion donde se actualizará la memoria. */
 9
10   msglen = len + 2;
11   /* Memoria para los datos a enviar y recibir */
12   data = (int **)malloc(NUMPROCESSOR*sizeof(int *));
13   for (i = 0; i < NUMPROCESSOR; i++) data[i] = (int *)malloc(msglen*sizeof(int));
14
15    /* Inicializar los datos del procesador */
16   data[0][0] = ACTIVE;  data[0][1] = addr;
17   for (i = 2; i < msglen; i++)
18     data[0][i] = reg.pm[ptr + i - 2];
19
20   /* Envío y recepción */
21   for (i = 0; i < dim; i++) {
22     nummsg = 1 << i;
23     vecino = reg.PhysicName ^ (1 << i);
24     for (j = 0, k = nummsg; j < nummsg; j++, k++) {
25       pvm_initsend(ENCODING);
26       pvm_psend(tids[vecino], MSG_REPL, (char *)data[j], msglen, PVM_INT);
27       pvm_precv(tids[vecino], MSG_REPL, (char *)data[k], msglen, PVM_INT, &atid,
28       atag, &alen);
30     }
31   }
32   /* Actualización de la memoria */
33   for (i = 0; i < NUMPROCESSOR; i++)
34     for (j = 0; j < len; j++)
35       if (data[i][0] == TRUE) {        /* Msg valido */
36         a = data[i][1];                /* addr */
37         sm[j + a] = data[i][j + 2];    /* Actualizacion */
38       }
39   for (i = 0; i < NUMPROCESSOR; i++)
40     free(data[i]);
41     free(data);
42 }
```

Figura 2.13 Replicación de la memoria compartida en PVM

El código de la Figura 2.13 presenta la implementación utilizando PVM de la rutina *replicateSHmem()* que ejecutan todos los procesadores como consecuencia de una escritura en memoria compartida. Entre las líneas 20 y 31 se producen las comunicaciones y el bucle de la línea 33 realiza las actualizaciones de la memoria compartida supuesto que el mensaje es válido (ha sido enviado por un procesador *activo*.

2.3.2. Ejemplos

2.3.2.1. Suma de prefijos

Dado un vector inicial *Mk* de tamaño *n*, el problema de la suma de prefijos consiste en obtener las sumas parciales *sum[j]* definidas según la fórmula:

```
const MAXN = ...;
type VectInt = array [1..MAXN] of integer;
var   n: shared integer;
      S, sum, Mk: shared VectInt;
```

Figura 2.14 Declaraciones

$$sum[j] = \mathring{a}_{i=1,j}\ Mk[i] \quad j\ \hat{I}\{1,...,n\}$$

Supongamos las declaraciones de la Figura 2.14.

El problema puede ser resuelto en paralelo teniendo en cuenta que para hacer la suma de prefijos de un intervalo *sum[i...j]* podemos hacer en paralelo la suma de prefijos de la primera mitad *sum[i..(i+j)/2]* y de la segunda mitad *sum[(i+j)/2+1..j]*. Luego sólo tendremos que sumar el total de los elementos de la primera mitad a cada una de las sumas de prefijos de la segunda mitad. Esto último puede hacerse también en paralelo. El código *ll* de la Figura 2.16 es la codificación directa de esta idea. La Figura 2.15 es el resultado de la ejecución del algoritmo sobre una red de cuatro estaciones de trabajo. La versión del intérprete utilizada es la desarrollada en PVM.

```
shared procedure prefix(i,j: shared integer);
var k: shared integer;
begin
  if i < j then begin
    parallel 1..2 do
    var a, b: integer;
    begin
      if name = 1 then begin a := i; b := (i+j) div 2; end
      else  begin  a := ((i+j)div 2)+1; b := j;   end;
      prefix(a, b);
    end;
    k := (i+j) div 2;
    parallel k + 1..j do  sum[name] := sum[name]+sum[k];
  end
  else sum[i] := Mk[i] (* only one item in i..j *)
end; { prefix }
```

Figura 2.16 Suma de prefijos

```
ll Interpreter Ver 1.0   12.Feb.98 (c) PCGULL 1998
Version for LAN Architecture.

Reading config file...
        Number of processors:       4
        Hipercube dimension:        2
        Program size:            4096 bytes.
        Private memory size:     4096 bytes.
        Shared memory size:      8192 bytes.
Loading program...
Sending program to the nodes...

Executing...
introduzca el array:
a[4]=2
a[5]=5
a[6]=1
a[7]=3
resultados:
b[4]=2
b[5]=7
b[6]=8
b[7]=11
```

Figura 2.15 Resultado de la ejecución de la suma de prefijos

2.3.2.2. Ordenación bitónica

El algoritmo de ordenación bitónica fue propuesto por Batcher en [Bat82]. Una secuencia bitónica *x* de longitud *n* es aquella que, al considerarla como un ciclo (esto es, el siguiente del último elemento es el primero) se puede descomponer en una subsecuencia no decreciente seguida de una secuencia no creciente:

$$\exists\, i, j \in \{0,\ldots, n-1\} / x_j \leq x_{j+1} \leq \ldots \leq x_i \ y \ x_i \geq x_{i+1} \geq \ldots \geq x_{j-1}$$

donde todas las operaciones en los índices se realizan en aritmética módulo *n*.

El procedimiento *bSort* de la Figura 2.8 explota la siguiente propiedad de las secuencias bitónicas:

Si la secuencia $x = (x_0, \ldots, x_n)$ es bitónica, entonces las subsecuencias $y=(y_0,\ldots,y_{n/2-1})$ y $z = (z_0,\ldots,z_{n/2-1})$ dadas por:

$$y_i = min\{x_i, x_{i+n/2}\};\ z_i = max\{x_i, x_{i+n/2}\} \text{ con } i \in \{0,..n/2-1\}$$

son también bitónicas y ocurre que

$$y_j \leq z_i \ \forall\ i, j \in \{0,..n/2-1\}$$

Esta propiedad es consecuencia de que la secuencia de las y_i tiene a lo sumo un único punto de corte con la secuencia de las z_i. El procedimiento *bitonicSort* ordena sucesivamente intervalos de longitud creciente usando *bSort*. Puesto que cualquier secuencia de longitud dos es bitónica, puede ser ordenada mediante una llamada a *bSort*. El intervalo total es particionado en secuencias de longitud dos que se ordenan en paralelo. Las secuencias pares se ordenan de manera creciente y las impares de manera decreciente. Ahora los subintervalos de longitud cuatro son bitónicos y se les puede ordenar usando *bSort*. El resultado será que los subintervalos de longitud ocho son bitónicos. Después de *log(n)* iteraciones repitiendo este procedimiento el vector *x* está ordenado.

2.3.2.3. Un Quicksort paralelo

El siguiente programa *ll* constituye un ejemplo paradigmático de los requerimientos de los algoritmos desarrollados para el modelo de programación PRAM. Utiliza tres procedimientos paralelos que se anidan en sucesivas llamadas, hace un uso intenso de la memoria compartida y de la capacidad de sincronización que ofrece el modelo. El algoritmo, propuesto en [Jaj92], paraleliza las llamadas recursivas del conocido procedimiento Quicksort [Hoa61], también paraleliza el procedimiento de partición y para ello hace a su vez uso de un procedimiento paralelo para la suma de prefijos. Después de leerlo es aconsejable preguntarse si un algoritmo como este puede ser implementado eficientemente en un computador distribuido.

Sea *S* un vector que contiene los *n* elementos a ordenar (declaraciones de la Figura 2.14) y denotemos por *lft* y *rgt* los extremos izquierdo y derecho del vector a ordenar. Se toma aleatoriamente un elemento pivote *S[pp]* del vector de entrada *S*. La estrategia del Quicksort es la de reorganizar utilizando un procedimiento de división *part(lft, rgt, pp)* los elementos de *S* en dos subvectores *S[lft..pp-1]* y *S[pp+1..rgt]*, de tal forma que cada elemento del primer segmento *S[lft..pp-1]* es menor que el pivote y cualquier elemento del segundo intervalo *S[pp+1..rgt]* es mayor o igual que el pivote. Las ordenaciones recursivas de los dos subintervalos resultantes pueden ser realizadas en paralelo. La Figura 2.17 contiene el código *ll* correspondiente.

```
shared procedure sort(lft, rgt: shared integer);
var pp: shared integer;.
begin
    if lft < rgt then begin
      pp := (lft + rgt) div 2; (* pivot position *)
      part(lft, rgt, pp);         (* pp contains new pivot position *)
      parallel 1..2 do
      var a, b: integer;  (* sort in parallel the two subintervals *)
      begin
        if (name = 1) then begin   a := lft; b := pp - 1;  end
        else  begin  a := pp+1; b := rgt;   end;
        sort(a, b);
      end
    end;
end; { sort }

begin
  inputArray(S); sort(1, n); writeArray(S);
end.
```

Figura 2.17 Procedimiento *sort* y programa principal

El procedimiento *part* en la Figura 2.18 realiza la división. En paralelo se compara *S[pp]* con cada elemento de *S*, y se marcan usando la variable *Mk* los elementos menores con un 1 y los mayores con un 0. Ahora necesitamos trasladar los elementos menores al principio de *S*, y los elementos mayores al final de *S*, de tal forma que el elemento *S[pp]* esté entre los dos subvectores. Esto se logra a través de una llamada al procedimiento *prefix* de la Figura 2.16 que nos devuelve en el vector *sum* las sumas parciales. La variable *NM = sum[rgt]* contiene el número de elementos menores que el pivote. Para un procesador *name* dado, la variable *sum[name]* contiene el número de elementos que están a la izquierda de *S[name]* que son menores que el pivote *S[pp]*. Esta información es suficiente para decidir en qué posición *pos* colocar cada elemento sin que existan conflictos de escritura.

2.4. Tiempo y número de procesadores

Las medidas de rendimiento de un algoritmo paralelo L más evidentes son el *Número de Procesadores* utilizados P y el *Tiempo* invertido $T_{Par}(L, P)$ por el algoritmo L cuando se usan P procesadores.

2.4.1. Suma de prefijos

Retomemos el procedimiento L = *prefix* de la Figura 2.16 que realiza la suma de prefijos. ¿Cuántos procesadores son necesarios, supuesto que no exista virtualización de procesadores? Observe que *n* procesadores son suficientes si aplicamos el algoritmo de distribución cíclica explicado en la sección 2.2. El algoritmo distribuye la mitad de los procesadores a cada una de las dos llamadas recursivas. Estos procesadores son suficientes para la demanda que realiza la sentencia *for all* utilizada para actualizar los elementos de la parte derecha del vector *sum.*

```
shared procedure part(lft,rgt: shared integer; var pp:shared integer);
var NM, oldPp: shared integer;

shared procedure prefix(i,j: shared integer);
       .... (*ver código en la Figura 2.16 *);

begin (* part *)
  parallel lft..rgt do
    if S[name] < S[pp] then Mk[name] := 1 else Mk[name] := 0;
    prefix(lft, rgt);
    oldPp := pp;
    NM := sum[rgt];
    parallel lft..rgt do
    var pos: integer;
    begin
       if name = pp then begin
            pp := lft+NM; pos := pp
       end
       else if Mk[name] = 1 then   pos:=sum[name]+lft-1
       else begin
         pos :=   name-sum[name]+NM;
         if name < oldPp then pos:=pos+1
     end;
     S[pos] := S[name]
  end  { parallel }
end; { part }
```

Figura 2.18 Procedimiento para la partición del vector

Como vimos en la sección 2.2, el algoritmo de asignación de procesadores sólo requiere tiempo constante. Lo mismo ocurre con las operaciones implicadas de asignación a los parámetros *a* y *b* antes de la llamada, así como la actualización de la parte derecha del vector *sum.* Por tanto el tiempo empleado $T_{Par}(Prefix, n)$ es $C*log\ n$ para alguna constante *C*.

2.4.2. Ordenación Bitónica

El perfecto equilibrio de la carga de trabajo entre las tareas generadas en el algoritmo de ordenación bitónica de la Figura 2.8 hace que el algoritmo de asignación cíclica de procesadores de la sección 2.2 funcione con *n* procesadores. El procedimiento *bitonicSort* realiza *log(n)* iteraciones en las que se llama a *bSort*. Por su parte, *bSort* realiza *log(n)* llamadas paralelas recursivas. Por tanto el tiempo $T_{Par}(bitonicSort, n) = D*log^2(n)$, donde *D* es una constante.

2.4.3. Quicksort Paralelo

El tiempo invertido por el algoritmo *L = Sort* de la Figura 2.17 depende del número de llamadas paralelas recursivas que necesite. En cada paso recursivo se llama al procedimiento de partición. El número de etapas de partición depende del ritmo al que se reduce el tamaño de los subvectores intermedios. En un ambiente pesimista, se pueden tener particiones de tamaño muy desigual, y esto puede ocasionar que el tamaño del subvector más largo se vea decrementado en sólo una unidad después de cada iteración. Por lo tanto, en el peor caso, el algoritmo necesita *O(n)* iteraciones. Puesto que el procedimiento *part* necesita tiempo *O(log n)* para realizar la suma de prefijos, el tiempo de ejecución de *sort* puede llegar a ser tan malo como el del algoritmo secuencial *O(n log n).* Sin embargo, si los intervalos resultantes de las llamadas a *part* son de tamaño similar, el procedimiento *sort* alcanzará sólo una profundidad

logarítmica. Se puede esperar entonces que en promedio el vector esté ordenado en tiempo $O(\log^2 n)$.

Puede verse fácilmente que, el algoritmo de asignación de procesadores cíclico, explicado en la sección 2.2, hace en este caso una mala gestión de los procesadores disponibles. Esto es consecuencia de que el número de procesadores requeridos en la división recursiva es proporcional al tamaño del intervalo a ordenar. Al asignar el mismo número de procesadores a los dos procesos, el algoritmo de asignación cíclica desperdicia los recursos. En el peor caso la profundidad del árbol de llamadas es *n* y se necesitarán *n(n-1)/2* procesadores.

2.5. Asignación de Procesadores Controlada por el Programador

Es claro que en el ejemplo del Quicksort paralelo una política de asignación correcta sólo requiere *n* procesadores. Existen varias alternativas para mejorar la política cíclica determinada por las ecuaciones (Ec. 2.1) y (Ec. 2.2).

```
parallel <Exp1>..<Exp2> with weight <Exp(name)> do
  <Statement>
```

Figura 2.19 Sentencia de Asignación de Procesadores Ponderada

```
parallel <Exp1>..<Exp2> map <Processor Mapping> do
  <Statement>
```

Figura 2.20 Sentencia parallel con control de asignación de procesadores

La Figura 2.19 muestra una sentencia de asignación de procesadores "ponderada". En ella el programador especifica mediante pesos w_{name} la proporción de procesadores que se requerirán para el proceso *name*. La Figura 2.21 muestra su utilización en el procedimiento *sort*. En este caso el número de procesadores a asignar a cada grupo es proporcional a la longitud del intervalo.

Otra posibilidad es hacer que el programador especifique la política de asignación de procesadores. Ello significa determinar para cada procesador los elementos que definen la política (Figura 2.5). La Figura 2.20 muestra una extensión de la sentencia *parallel* del lenguaje *La Laguna* con una cláusula *map* en la que el programador en términos del valor del nombre físico del procesador (denotado por *physical*) decide a qué grupo pertenecerá cada uno de los procesadores. Utilizando esta sentencia, los códigos de la Figura 2.23 y de la Figura 2.22 computan una distribución de procesadores adaptada a la carga de trabajo. Sólo dentro del ámbito de una sentencia <*Processor Mapping*> le es permitido al programador de *ll* cambiar explícitamente las variables *numprocessors*, *name* y *physical.*

```
shared procedure sort(lft, rgt: shared integer);
var pp: shared integer;.
      a, b: shared interval;
begin
    if lft < rgt then begin
      pp := (lft + rgt) div 2; (* pivot position *)
      part(lft, rgt, pp);         (* pp contains new pivot position *)
      a[0] := lft; b[0] := pp; a[1] := pp; b[1] := rgt;
      parallel 0..1 with weight b[name]-a[name] do
        sort(a[name], b[name]);
   end;
end; { sort }
```

Figura 2.21 Sort con asignación ponderada de procesadores

```
processor mapping proportional(left, right, pp: integer);
begin
    if (left+physical <= pp) then begin
       numprocessors := pp-left+1;
       name := 1;
    end
    else begin
      numprocessors := rgt -pp;
      name := 2;
      physical := physical-pp;
    end;
end;
```

Figura 2.22 Definiendo una política de asignación

```
shared procedure sort(lft, rgt: shared integer);
var pp: shared integer;.
begin
    if lft < rgt then begin
      pp := (lft + rgt) div 2; (* pivot position *)
      part(lft, rgt, pp);     (* pp contains new pivot position *)
      parallel 1..2 map proportional(left,right,pp)   do
      var a, b: integer;
      begin
        if (name = 1) then begin   a := lft; b := pp - 1;  end
        else  begin  a := pp+1; b := rgt;   end;
        sort(a, b);
      end
    end;
end; { sort }
```

Figura 2.23 Distribuyendo los procesadores explícitamente

2.6. Las Sentencias Condicionales como Sentencias de Asignación de Procesadores

En el marco de la programación PRAM existe una extensión natural de la semántica de una sentencia condicional *if-then-else* cuando ésta tiene las dos ramas:

if <*Expression*> then < *Statement1* > else <*Statement2*>

El significado de esta extensión es que todos los procesadores en el proceso evalúan la expresión. El grupo actual de procesadores se divide en dos grupos. Los procesadores para los que la condición es verdadera forman el primer grupo y ejecutan la sentencia <*Statement1*>. Aquellos para los que la condición es falsa forman el segundo grupo y ejecutan <*Statement2*>. La sentencia condicional se convierte así en una sentencia de asignación de procesadores. Para no penalizar la eficiencia, se impone que los dos nuevos procesos se ejecutan asíncronamente (aunque dentro de cada grupo se mantiene la sincronía). Como veremos en el Capítulo 3, esta es la semántica utilizada para la sentencia condicional en *fork95*.

2.7. Aceleraciones y paradojas

Resulta interesante comparar la velocidad alcanzada por un algoritmo paralelo con la del mejor algoritmo secuencial *BestSeq* que resuelve el mismo problema. Se define la

Aceleración de un algoritmo paralelo $S(L, P)$ como el cociente entre el tiempo $T_{BestSeq}$ del mejor algoritmo secuencial y el tiempo del algoritmo paralelo $T_{Par}(L, P)$ utilizando P procesadores.

$$S(L, P) = T_{BestSeq}/T_{Par}(L, P)$$

Por ejemplo, el algoritmo para la suma de prefijos de la Figura 2.16 requiere tiempo $O(log(n))$ y utiliza n procesadores. Su aceleración es de orden

$$S(Prefix, n) = O(n/log(n))$$

El algoritmo de ordenación paralela basado en el Quicksort que aparece en la Figura 2.19 utiliza n procesadores para obtener una aceleración que, en el peor caso será de orden constante, mientras que en promedio cabe esperar una aceleración de orden $n\ log(n)/log^2(n) = n/log(n)$. Sin embargo, el algoritmo de ordenación bitónica tiene una aceleración de orden $n/log(n)$ incluso para el caso peor. No obstante, no se debe menospreciar la influencia de las constantes asociadas a los órdenes de complejidad. En general, aspiramos a desarrollar algoritmos paralelos que alcancen una aceleración tan grande como sea posible.

2.7.1. Límites en la Aceleración alcanzable

Un algoritmo paralelo es *óptimo*, cuando alcanza una aceleración de orden igual al número de procesadores que utiliza. Se dice entonces que el algoritmo alcanza una *aceleración lineal.*

Teorema 2.1 Un algoritmo paralelo no puede alcanzar una aceleración superior a la lineal:

$$S(L, P) \leq P$$

Para demostrarlo, consideremos un algoritmo paralelo L que se ejecuta en $T_{Par}(L, P)$ pasos usando P procesadores. El algoritmo L da lugar a un algoritmo secuencial L' que se ejecuta en una máquina secuencial en $P*T_{Par}(L, P)$ pasos. Basta que por cada instrucción de la máquina paralela la máquina secuencial ejecute un bucle de tamaño P simulando cada uno de los procesadores. Si alguno de los procesadores simulados no participa en el paso, L' simplemente incrementa el contador. Por definición, el mejor algoritmo secuencial *BestSeq* debe tardar menos que este algoritmo L' (si no fuera así, ¡*BestSeq* no sería el mejor algoritmo secuencial!):

$$T_{BestSeq} \leq T_{L'} = P*T_{Par}(L, P)$$

Despejando P se sigue que $T_{BestSeq}/T_{Par}(L, P) = S(L, P) \leq P$.

2.7.2. Aceleración Superlineal

Sin embargo, la experiencia nos muestra que los fenómenos de aceleraciones superiores a la lineal ocurren ... ¡y no tan infrecuentemente como cabría esperar después de demostrada la desigualdad anterior!. ¿Qué ocurre? ¿Es falsa la demostración anterior?

```
program SuperLinearSpeedup;
const N = 512; N1 = 511; P = 4;
type vector = array [0..N1] of integer;
var A:  shared  vector;
    x, posx, proc: shared  integer;
    found: shared boolean;
 begin
    ... (* array initialization *); x := A[(P-1)*(N div P)];
    parallel 0..P-1 do
    var first, last, k: integer;
    begin
      first := name*(N div P);
      last := first + (N div P)-1;
      k := first;  found := false;
      while (not found) and (k <= last) do
      begin
         if (A[k] = x) then begin
            found := true; posx := k; proc := name
         end;
         k := k+1;
       end;
    end;
    if found then
      writeln('found in position = ',posx,' by processor ',proc);
end.
```

Figura 2.24 ¿Puede ser la aceleración mayor que el número de procesadores?

Considere el programa *ll* de la Figura 2.24 que busca un elemento *x* en un vector desordenado. Se sabe que *x* aparece en el vector a lo sumo una vez. Bajo esta hipótesis, el algoritmo de la Figura 2.24 trabaja en una CREW PRAM. Si ocurre la especial circunstancia que el elemento *x* esta situado en la posición (*N div P)*(P-1)*, el procesador *P-1* encontrará el elemento en la primera iteración, pondrá *found* a *true* y todos los procesadores abandonarán el bucle de búsqueda. El mejor algoritmo secuencial de búsqueda (recorrer el vector de izquierda a derecha), requiere (*N div P)*(P-1)* iteraciones bajo esas hipótesis. Por ejemplo para *N = 512* y *P = 4* procesadores, el algoritmo secuencial ejecuta *384* iteraciones por una sola del paralelo. Cuando *N* es suficientemente grande la aceleración será mayor que el número de procesadores. Este fenómeno se conoce con el nombre de *Superlinealidad* o *Aceleración Superlineal.* La Figura 2.25 muestra la ejecución del correspondiente algoritmo secuencial (el programa es cargado mediante la orden *pram seq* en la línea de comandos y posteriormente ejecutado mediante la orden *go*). Son ejecutadas *17993* instrucciones del lenguaje máquina (el número que aparece tras la etiqueta *T* indica en cada instante el número de instrucciones PRAM ejecutadas). Después de reiniciar la PRAM mediante la orden *ini*, procedemos a cargar el programa ejecutable generado para el código *ll* de la Figura 2.24 (orden *load 0, speedup*). Se ejecutan *2682* instrucciones. La aceleración obtenida es por tanto igual a *17993/2682 = 6,7* que es mayor que el número *4* de procesadores utilizados. ¿Dónde está el fallo del razonamiento anterior?

Podría sospecharse que el problema es causado por la falta de realismo del modelo CREW PRAM. ¿Qué ocurre sin se trabaja con un modelo EREW?. En tal caso, la lectura concurrente a la variable *found* al comienzo del bucle de la Figura 2.24 está prohibida. Sin embargo sería posible declarar *found* como privada y sustituir la condición de finalización del bucle por el resultado de una "reducción" paralela por la operación lógica *OR* sobre las variables *found*. Tal operación de reducción puede ser realizada en tiempo logarítmico usando un algoritmo similar al explicado en la sección

2.3.2.1. El algoritmo de búsqueda resultante sigue ofreciendo una conducta superlineal en el caso particular considerado.

La verdadera respuesta reside en la imprecisión con que hemos utilizado el concepto de *Mejor Algoritmo Secuencial.* ¿En qué sentido estamos hablando del mejor algoritmo? ¿Nos referimos al caso promedio? Si es así, puede ocurrir que la versión secuencial L' del algoritmo paralelo sea más rápida para ciertos casos que se "desvían" del promedio y el razonamiento del Teorema 2.1 resultaría falso para esos casos. Lo mismo ocurre si estamos hablando del mejor algoritmo secuencial para "el peor caso del secuencial". Puede que dicho "peor caso" sea bastante improbable que ocurra y que la versión secuencial L' del algoritmo paralelo sea más veloz que el mejor algoritmo secuencial para un buen número de casos. De nuevo, para esos casos la aceleración podría ser superlineal. Esto es de hecho lo que ocurre en el ejemplo de la Figura 2.24.

Obsérvese que si un algoritmo paralelo L es óptimo, el orden de complejidad de su secuencialización L' es igual a la del mejor algoritmo secuencial *BestSeq*. Por tanto, cada vez que un diseñador de algoritmos paralelos descubre un algoritmo paralelo óptimo, nos está también ofreciendo un algoritmo secuencial que iguala en rendimiento (salvo quizá un factor constante) al mejor algoritmo secuencial conocido. Esta propiedad nos habla de la dificultad de encontrar algoritmos paralelos óptimos... y también nos sugiere que si una tabla de aceleraciones de cierto algoritmo paralelo está llena de anomalías superlineales, es bastante probable que los problemas que se utilizaron en su confección no fueran los más adecuados para el correspondiente mejor algoritmo secuencial.

El Teorema 2.1 es cierto si se fijan adecuadamente los casos del problema. Por ejemplo, si se define la aceleración como el cociente entre el tiempo del Mejor Algoritmo Secuencial sobre su Peor Caso dividido por el tiempo del Algoritmo Paralelo sobre su correspondiente Peor Caso:

$$S(L, P) = T_{BestSeq}(\text{Worst BestSeq Case}) / T_{Par}(L, P, \text{Worst L Case})$$

o si se define la aceleración sobre el peor caso del paralelo:

$$S(L, P) = T_{BestSeq}(\text{ Worst L Case}) / T_{Par}(L, P, \text{ Worst L Case})$$

o como el cociente de las esperanzas de las variables aleatorias $T_{BestSeq}$ y T_{Par}:

$$S(L, P) = E(T_{BestSeq}) / E(T_{Par})$$

```
C:\ >pram seq
Parallel Random Access Machine.
PRAM|T0>go
Sequential search: found in position = 384
PRAM|T17993>ini
PRAM|T0>load 0,speedup
Loaded.
PRAM|T0>go
found in position = 384 by processor 3
PRAM|T2682>_
```

Figura 2.25 Superlinealidad: Observe el contador de tiempos junto a la etiqueta *T*

2.8. El Coste, La Eficiencia y el Trabajo

El *coste C(L,P)* de un algoritmo PRAM L se define como el producto del número de procesadores P por el tiempo empleado $T_{Par}(L, P)$

$$C(L,P) = P*T_{Par}(L, P).$$

La fórmula para $C(L,P)$ puede ser interpretada diciendo que el coste de un algoritmo paralelo es el tiempo o complejidad del correspondiente algoritmo secuencial L' descrito en el Teorema 2.1 El cociente $E(L,P)$ entre el tiempo del mejor algoritmo secuencial y el coste del algoritmo paralelo se denomina *eficiencia*.

$$E(L,P) = T_{BestSeq} / C(L, P)$$

La eficiencia nos da una medida del aprovechamiento de los P procesadores requeridos por el algoritmo. Un valor de $E(L,P)$ próximo a uno indica una aceleración lineal y un aprovechamiento eficiente de los procesadores.

Es obvio que el algoritmo L' puede ser optimizado a un algoritmo L'' cuyo bucle de simulación solo recorra el *número de procesadores que participan* en ese paso en vez de recorrer hasta *el número total P de procesadores necesitados*. Esto supone una mejora con respecto a L' porque como consecuencia de la ejecución de sentencias condicionales y bucles puede ocurrir que algunos de los procesadores no deban participar en la ejecución de ciertas instrucciones. La complejidad del algoritmo L'' se denomina *el trabajo*, $W(L)$, realizado por el algoritmo paralelo L. Obviamente el trabajo es menor que el coste: $W(L) \pounds C(L, P)$.

Consideremos un programa PRAM L que se ejecuta en $T(P) = T_{Par}(L, P)$ pasos y requiere un trabajo $W = W(L)$. Denotaremos por $W(k)$ el número de procesadores que participan en el paso k, con $k = 0,...,T(P)-1$. Puesto que asumimos tiempo constante para las instrucciones de la máquina, el trabajo realizado en un paso k es igual al número de procesadores $W(k)$ que participan en el paso. En cada paso k, el programa L'' ejecuta la correspondiente instrucción $i_{k,p}$ de la máquina PRAM para cada procesador p que participa en el paso, determina si el procesador p permanecerá *participante* o no en el siguiente paso $k+1$ e incrementa o no el contador de procesadores *participantes* para el siguiente paso. Por tanto, admitiendo una complejidad constante para cada una de las instrucciones de la máquina PRAM, la complejidad W de L'' es:

$$W = W(0) +...+W(T(P)-1)$$

2.8.1. Reduciendo el Número de Procesadores

Si disponemos de una máquina PRAM con P' procesadores, podemos dividir los pasos de un algoritmo paralelo L que usa P procesadores en dos clases de pasos, $Seq(P')$ y $Par(P')$ definidos como:

$$Seq(P') = \{k \hat{I} \{0,...,T(P)-1\} / W(k) < P'\}$$
$$Par(P') = \{k \hat{I} \{0,...,T(P)-1\} / W(k) \, ^3 \, P'\}$$

El conjunto $Seq(P')$ es el conjunto de pasos en los que participan menos de P' procesadores en su ejecución. Por el contrario, el conjunto $Par(P')$ está formado por los pasos que requieren más de P' procesadores. Podemos intentar ejecutar el algoritmo PRAM L que utiliza P procesadores en la máquina PRAM con $P' \pounds P$ procesadores, haciendo que en cada paso $k \hat{I} Par(P')$, cada uno de los P' procesadores simule aproximadamente el mismo número $W(k)/P'$ de procesadores. Así el tiempo invertido en la máquina PRAM con P' procesadores será:

(Ec. 2.3) $$T(P') = |Seq(P')| + \mathring{a}_{k \hat{I} Par(P')} ceil(W(k)/P')$$

La deducción de la fórmula (Ec. 2.3) supone cierta la siguiente hipótesis:

Hipótesis de Asignación Rápida:

Para cada paso *k* es posible calcular *W(k)* en tiempo constante y asignar los *W(k)* procesadores virtuales que resultaron participantes en el paso *k-1* a los procesadores físicos en tiempo constante.

Supuesta esta hipótesis, si se tiene en cuenta que es consecuencia directa de la definición que $|Seq(P')| \leq T(P)$, se sigue de (Ec. 2.1) la desigualdad:

$$T(P') \leq T(P) + \sum_{k \in Par(P')} ceil(W(k)/P') \leq T(P) + ceil(W/P')$$

La inecuación resultante:

$$T(P') \leq T(P) + ceil(W/P')$$

es denominada "Teorema de Brent" por algunos autores [Gib88], [Qui90] y "Work-Time Scheduling Principle" [Jaj92] por otros. Si se asume que normalmente $T(P) \leq ceil(W/P')$, se sigue que $T(P')$ es de orden $O(W/P')$. Por tanto, medido en órdenes de complejidad, la reducción de procesadores puede hacerse sin introducir ineficiencia.

2.8.2. Teorema de Brent Generalizado

La condición de que el número de procesadores *W(k)* que participan en el paso *k* pueda ser computado en tiempo constante y los *W(k)* procesadores virtuales que resultaron participantes en el paso *k-1* puedan ser asignados a los procesadores disponibles en tiempo constante no se cumple en todos los algoritmos PRAM. ¿Bajo qué condiciones es falsa esta hipótesis? Un típico ejemplo es la situación en la que un cierto conjunto síncrono de procesadores que están *participando* en un paso *k* comienzan a ejecutar el código correspondiente a una sentencia condicional sin cláusula *else*:

if <Boolean Expression> then <Statement>;

la semántica PRAM obliga a los procesadores que evalúan *<Boolean Expression>* a *FALSE* a permanecer "*ociosos*" hasta el final de la sentencia condicional. Aquí las expresiones *participante* y *ocioso* tienen significado diferentes de los de activación y desactivación utilizados en la sección 2.2. Los procesadores *ociosos* continúan "implicados" en la computación. Por ejemplo, en algunas versiones de *La Laguna*, los procesadores continúan ejecutando el código de las instrucciones sólo que los contenidos de la memoria no son modificados. Esto les sirve para seguir el "rastro" de la computación y reincorporarse sin coste alguno en el momento en el que les corresponde volver a participar. Tenemos pues tres conceptos diferentes:

1. ***Activación,***
2. ***Asignación de Procesadores y***
3. ***Participación***.

Para ilustrar lo que ocurre cuando no se cumple la hipótesis de asignación rápida consideremos el siguiente esquema de traducción para la sentencia condicional anterior (este esquema es el utilizado por algunas versiones del compilador de *La Laguna*):

```
<Boolean Expression>.Translation
PUSHPATTERN
COPYTOPS
GOIFZERO LABEL
<Statement>.Translation
LABEL: POPPATTERN
```

Después de la traducción de la expresión lógica, <*Boolean Expression*> con la llamada a la instrucción *PUSHPATTERN*, cada procesador *p* guarda en la pila su *registro de participación*. El registro de participación es una estructura de datos en el que cada procesador "físico" almacena el "patrón" actual de los procesadores virtuales de los que se encarga que "participan" en el paso. Por ejemplo, en las mencionadas versiones de *La Laguna*, el registro de participación incluye, entre otras subestructuras, un vector de unos y ceros que indica la participación o no de los procesadores virtuales, el número de procesadores virtuales activos y un vector *NAME* conteniendo los nombres lógicos de los procesadores virtuales activos. Después de la traducción de la expresión lógica, la instrucción *COPYTOPS* "retira" los procesadores participantes que evaluaron la expresión <*Boolean Expression*> a *FALSE*. Esta instrucción *COPYTOPS* deberá contar el nuevo número $W(k+1)$ de procesadores que participan en el paso siguiente y repartir la carga de procesadores virtuales de manera equitativa entre los P' procesadores disponibles. El cálculo de $W(k+1)$ y el reparto de la carga puede hacerse mediante una suma de prefijos de las cargas individuales $W_p(k+1)$ resultantes en cada uno de los procesadores físicos $p \hat{I} \{0,...P'-1\}$. Esta suma de prefijos permite realizar posteriormente una escritura exclusiva de los nombres de aquellos procesadores virtuales que participarán en el paso $k+1$. Se trata de un algoritmo similar al utilizado en el procedimiento *part* de la Figura 2.18.

<table>
<tr><td colspan="7">——————————— tiempo ———————————→</td></tr>
<tr><td colspan="3">Antes de COPYTOPS</td><td></td><td></td><td colspan="2">Después</td></tr>
<tr><td>PHYS</td><td>NAME</td><td>BOOL</td><td>SUM</td><td>PREFIX</td><td>PHYS</td><td>NAME</td></tr>
<tr><td rowspan="4">0</td><td>2</td><td>1</td><td rowspan="4">2</td><td rowspan="4">2</td><td rowspan="3">0</td><td>2</td></tr>
<tr><td>4</td><td>0</td><td>7</td></tr>
<tr><td>7</td><td>1</td><td>9</td></tr>
<tr><td>8</td><td>0</td><td rowspan="2">1</td><td>10</td></tr>
<tr><td rowspan="4">1</td><td>9</td><td>1</td><td rowspan="4">2</td><td rowspan="4">4</td><td>15</td></tr>
<tr><td>10</td><td>1</td><td>19</td></tr>
<tr><td>11</td><td>0</td><td rowspan="2">2</td><td>21</td></tr>
<tr><td>13</td><td>0</td><td>33</td></tr>
<tr><td rowspan="4">2</td><td>15</td><td>1</td><td rowspan="4">4</td><td rowspan="4">8</td><td>35</td></tr>
<tr><td>19</td><td>1</td><td rowspan="3">3</td><td>46</td></tr>
<tr><td>21</td><td>1</td><td>51</td></tr>
<tr><td>33</td><td>1</td><td>62</td></tr>
<tr><td rowspan="4">3</td><td>35</td><td>1</td><td rowspan="4">4</td><td rowspan="4">12</td></tr>
<tr><td>47</td><td>1</td></tr>
<tr><td>51</td><td>1</td></tr>
<tr><td>62</td><td>1</td></tr>
</table>

Figura 2.26 Ejecución de COPYTOPS en una PRAM con 4 procesadores

La Figura 2.26 muestra una traza de COPYTOPS en una PRAM con $P' = 4$ procesadores que virtualizan $P = 16$ procesadores lógicos. La columna denominada BOOL contiene el resultado de la evaluación de <*Boolean Expression*>. Las entradas de la columna SUM muestran los números de procesadores lógicos por procesador físico (los nombres de estos últimos aparecen en las columnas etiquetadas PHYS) que participarán en el siguiente paso. Por ejemplo, el procesador físico *1* está al cargo de la simulación de un cierto número de procesadores lógicos, de los cuáles están participando los procesadores lógicos *9, 10, 11* y *13*. Los procesadores lógicos *9* y *10* evalúan la expresión a cierta y los otros dos a falsa. Después de la suma de prefijos

(columna PREFIX) de los elementos del vector SUM, cada procesador físico compacta en el vector NAME los nombres de sus procesadores lógicos participantes. Lo hace, como indican las flechas en la Figura 2.26 en las posiciones a partir de PREFIX[PHYS-1]. Mediante un cálculo sencillo los procesadores físicos pueden determinar cuántos procesadores lógicos corresponden a cada uno y de qué segmento del vector NAME se deben de encargar. **Así pues hay un coste adicional *O(log(P'))* en aquellos pasos que implican un descenso en el número de procesadores que participan. Este tiempo *O(log(P'))* es necesario para poder calcular *W(k+1)* y asignar las tareas para el paso *k+1* a los procesadores.** La ecuación (Ec. 2.4) se escribe bajo esta consideración como:

$$T(P') \leq |Seq(P')| + \sum_{k \in Par(P')} log(P') + ceil(W(k)/P')$$

de donde se deduce

$$T(P') \leq |Seq(P')| + log(P') * |Par(P')| + ceil(W/P')$$

El teorema de Brent sigue siendo válido si se expresa en términos de órdenes de complejidad, esto es *T(P')* es de orden

$$O(T(P) + ceil(W/P'))$$

Como resultado de la acción de *COPYTOPS* puede ocurrir que la evaluación de la expresión lógica resulte falsa para todos los procesadores y que ningún procesador resulte participante. En tal caso la instrucción *GOIFZERO LABEL* hace saltar a todos los procesadores a la instrucción *POPPATTERN* en la que se recupera el antiguo patrón de procesadores participantes. En caso contrario, se ejecuta el código resultante de la traducción de la sentencia.

2.8.3. La Ley de Amdahl Generalizada

De la definición de los conjuntos *Seq(P')* y *Par(P')* se sigue que

$$T_{Par}(L, P') \leq |Seq(P')| + |Par(P')| / P'$$

Sustituyendo en la definición de la aceleración tenemos:

$$S(L, P') = T_{BestSeq} / T_{Par}(L, P') \leq T_{BestSeq} / \{|Seq(P')| + |Par(P')| / P'\}$$

Como $|Par(P')| = W - |Seq(P')| \geq T_{BestSeq} - |Seq(P')|$, resulta que:

$$S(L, P') \leq 1 / \{|Seq(P')| / T_{BestSeq} + (T_{BestSeq} - |Seq(P')|) / (T_{BestSeq} * P')\}$$

Si denotamos por *f* el porcentaje $f = |Seq(P')| / T_{BestSeq}$ de operaciones que requieren menos de *P'* procesadores con respecto al número de operaciones que realiza el mejor algoritmo secuencial, obtenemos la inecuación:

$$S(L, P') \leq 1/(f + (1-f)/P')$$

La desigualdad anterior se conoce con el nombre de ley de Amdahl. Obsérvese que como corolario se sigue que $S(L, P') \leq 1/f$. Por tanto, la aceleración alcanzable por un algoritmo *L* sobre una máquina con *P'* procesadores está acotada por el porcentaje $W / |Seq(P')|$ de operaciones (con respecto al trabajo) de pasos que requieren menos de *P'* procesadores. En particular $W / |Seq(1)|$ es una cota inferior de la aceleración alcanzable por *L* sobre cualquier máquina paralela. Por ejemplo, si el *25%* de las operaciones de un algoritmo *L* están en *Seq(1)*, la aceleración nunca podrá ser mayor que *4*, ya que

$$4 = 100/25 = W/|Seq(1)| \geq T_{BestSeq} / |Seq(1)|$$

2.9. Simulación del Modelo PRAM por una Mariposa

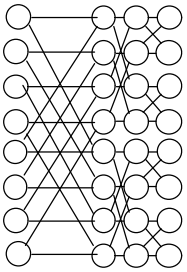
Figura 2.27

En este apartado discutiremos algunos de los problemas que surgen en el diseño de los accesos a variables compartidas en la realización de un lenguaje orientado al modelo PRAM en un sistema distribuido. Estos problemas fueron tratados en [Leo97] y [Leo95a]. La discusión se realiza sobre una mariposa, pero los razonamientos son válidos para la mayoría de las redes "hipercúbicas".

Una mariposa o *"butterfly" N*-dimensional tiene $(N+1)*2^N$ nodos y $N*2^{N+1}$ aristas. Los nodos se corresponden con pares (w,i) donde i es el *nivel* o *dimensión* del nodo y w es un número que denota la *fila* del nodo.

Dos nodos (w,i) y (w',i') están conectados si, y sólo si

$$(i' = i+1 \wedge w = w') \vee (i' = i+1 \wedge w = w' \text{ XOR } 2^i)$$

Es costumbre convertir la mariposa en una topología cilíndrica haciendo que las columnas inicial y final coincidan.

Existe una correspondencia natural entre cada nodo del hipercubo y cada fila de la mariposa. La Figura 2.27 muestra una mariposa con 3 dimensiones y 8 filas. Se puede obtener un hipercubo de una mariposa simplemente colapsando los nodos en la misma fila y observando cómo se solapan los enlaces proyectados. Recíprocamente, se puede obtener una mariposa de un hipercubo desplegando cada nodo en una fila de $N+1$ nodos y haciendo que cada uno de estos $N+1$ procesadores se ocupe de un enlace en una dimensión diferente. Cada uno de los nodos desplegado de una fila se enlaza con el nodo siguiente en la fila correspondiente a la dimensión de la que se hace cargo. De aquí se sigue que un hipercubo con N nodos puede ser simulado en $log(N)$ pasos por una mariposa con $N*(log(N)+1)$ nodos. La mariposa tiene un grado de *4*, un diámetro $O(log(N))$ y un ancho de bisección $Q(N/log(N))$.

Teorema 2.2 /Construcción

> **Cualquier algoritmo L que se ejecuta en tiempo $T = T_{Par}(L,P)$ en una *(P,M)-EREW PRAM* con P procesadores y M celdas de memoria compartida puede traducirse a un algoritmo L^b que se ejecuta en tiempo $O(T*(P/P'+log(P')))$ en una máquina mariposa (butterfly) con $P' \pounds P$ procesadores con una probabilidad próxima a uno.**

Esquema de la demostración/implementación: Supongamos que $P' = 2^r$ y que $M = 2^s$. Cada uno de los P' procesadores en la mariposa emula a un conjunto P/P' de los P procesadores de la *(P,M)-PRAM*. Cada procesador se encarga también de la emulación de M/P' celdas de memoria compartida usando para ello parte de su memoria privada. El problema principal es ¿Qué hacer si algunos de los P procesadores PRAM acceden a la memoria compartida?. La solución pasa por disponer de una función h^L

$$h^L: [0,M-1] \text{ ¾ ®} [0,P'-1] \acute{} [0,M/P'-1]$$
$$h^L(m) = (h^L{}_P(m), h^L{}_M(m)) \hat{I} [0,2^r-1] \acute{} [0,2^{s-r}-1]$$

Esta aplicación puede ser generada por el compilador a partir del programa PRAM L El valor $h^L(m)$ para una dirección m de memoria compartida dada, nos determina el procesador $h^L{}_P(x)$ que la guarda y la dirección $h^L{}_M(x)$ en la que está almacenada en dicho procesador $h^L{}_P(x)$. Entonces cada procesador físico, según avanza en el bucle de tamaño P/P' de virtualización va lanzando las demandas del correspondiente procesador virtual utilizando la función h^L. De esta manera se solapan

cómputo y comunicaciones. El paralelismo de segmentación que se produce hace que el tiempo de latencia *O(log(P'))* marcado por el diámetro de la mariposa sólo tenga que ser contabilizado una vez. La función h^L puede construirse a partir una función g_L biyectiva en el espacio de las direcciones de memoria compartida:

$$g_L: [0, M-1] \rightarrow [0, M-1]$$

y definiendo las funciones componentes h_P y h_M de h en el espacio de nombres de procesador y de direcciones de memoria como los números que cumplen la ecuación:

$$gL\ (m) = hM(m)*2r + hP(m)$$

o lo que es lo mismo:

$$hM(m) = gL\ (m)\ div\ 2r \quad y\ hP(m) = gL\ (m)\ mod\ 2r$$

La función g_L es construida por el compilador adaptada a los patrones de acceso del algoritmo L de manera que los accesos resulten lo más uniformemente distribuidos y la congestión sea escasa. Una mejora adicional es hacer que el compilador calcule una función $g_{L,A}$ distinta adaptada a los patrones de acceso a la variable A declarada como compartida en el programa PRAM L. Lo ideal es que el cómputo de $g_{L,A}$ pueda hacerse en tiempo constante. Normalmente, la construcción de la función $g_{L,A}$ está basada en técnicas de "hashing". En ocasiones una combinación de "mappings" cíclicos y por bloques como las utilizadas en algunos lenguajes paralelos es suficiente [Hpf93].

El problema de encontrar la asignación de una variable a los procesadores de manera que se minimice la congestión en los procesadores puede formalizarse como sigue. Para un paso PRAM s y un procesador i denotamos por $h_{s,i}$ el valor:

$$h_{s,i}(g_{L,A}) = max\{in_{s,i}(g_{L,A}), out_{s,i}(g_{L,A})\} \qquad \text{(Ec. 2.5)}$$

donde

- $in_{s,i}(g_{L,A})$ es el número de peticiones a la memoria compartida administrada por el procesador i de la mariposa en el paso s para la distribución de memoria fijada $g_{L,A}$
- $out_{s,i}(g_{L,A})$ es el número de peticiones a la memoria compartida realizadas por el procesador i de la mariposa en el paso s, para la distribución de memoria $g_{L,A}$

El operador de máximo en (Ec. 2.5) puede ser sustituido por el de suma según sea el número de puertos y la capacidad de entrada/salida paralela que tenga la red.

El problema es buscar la función g dentro de una familia Y de funciones analíticas cuyo cómputo pueda ser implementado de manera muy eficiente (en tiempo constante) y tal que minimice el desequilibrio de la congestión en los procesadores de la mariposa:

$$min_{g \in Y} \sum_{s=0,..,R} max_{i=0,..,P-1} |h_{s,i}(g) - E(h_s(g))| \qquad \text{(Ec. 2.6)}$$

donde R denota el número de pasos del programa PRAM y $E(h_s(g))$ es el promedio de peticiones a memoria en el paso s.

$$E(h_s(g)) = (\sum_{i=0,..,P-1} h_{s,i}(g))/P$$

Si la variable A implicada en los accesos a memoria en el paso es relativamente pequeña, puede ser factible renunciar a que g se pueda expresar en forma analítica y duplicar la cantidad de memoria requerida para A, utilizando para su implementación una estructura de punteros a procesadores.

La resolución del problema (Ec. 2.6) resuelve también parte del problema de equilibrio de la carga de trabajo entre los procesadores planteado en la sección 2.5.

Obsérvese que en esta simulación la asignación de los procesadores PRAM a los procesadores de la mariposa es estática y se determina al comienzo de la computación. Puede ocurrir que los procesadores asignados a algunos procesadores físicos de la mariposa dejen de participar temporalmente en la ejecución como consecuencia de la evaluación de condiciones lógicas en sentencias condicionales y bucles, mientras los procesadores PRAM de otros procesadores físicos de la mariposa se mantienen participando. De esta manera se produciría un desequilibrio en la carga de trabajo. El problema de encontrar una buena asignación de los procesadores PRAM a los procesadores físicos de la mariposa está asociado al problema de la asignación de las direcciones de memoria compartida a los procesadores.

Se puede demostrar que para cada programa PRAM L y cada variable EREW A existen funciones $g_{L,A}$ que garantizan que el envío de las peticiones y las respuestas está libre de congestiones con una probabilidad próxima a uno [Lei92]. Por tanto los accesos pueden realizarse en tiempo $O(P/P' + log(P'))$. Como el algoritmo PRAM simulado requiere T pasos, tenemos una complejidad $O(T*(P/P'+log(p')))$.

Capítulo III
FORK95 Y LA SBPRAM

Capítulo III

fork95 y la SB-PRAM

3.1. Introducción

Cuando a principios de la década de los 90 comenzamos a trabajar en el diseño del lenguaje *ll*, éste era la única implementación de un lenguaje con todas las características exigibles a un lenguaje orientado al modelo PRAM [Leo92]. En 1995 surge *fork95*. Junto con *ll*, este lenguaje soporta recursividad y paralelismo. Las soluciones ofrecidas en su diseño e implementación constituyen alternativas para el caso de memoria compartida, a las propuestas que hacemos en este trabajo. El proyecto en el que está inscrito *fork95* implica también el desarrollo de un prototipo de arquitectura PRAM, denominado SB-PRAM.

En la sección 3.2 de este Capítulo presentamos las principales características de la SB-PRAM y el epígrafe 3.3 está dedicado al estudio de las características de *fork95* como lenguaje orientado al modelo PRAM.

3.2. La SB-PRAM

La SB-PRAM [Sbp], [Abo93], es un multiprocesador MIMD escalable, masivamente paralelo fuertemente síncrono, de memoria compartida con tiempo de acceso a memoria uniforme. El proyecto está siendo desarrollado en el Departamento de Informática de la Universidad de Saarbrücken (Alemania) por el equipo del profesor W. J. Paul y lleva en marcha desde principio de los años 90. El objetivo del proyecto es conseguir una máquina de 64 procesadores físicos (2048 virtuales) de tipo RISC y una memoria compartida de 2Gb.

El proyecto SB-PRAM ha sido concebido como un proyecto universitario de bajo presupuesto que pretende probar que es posible la implementación de una PRAM, es por ello que en este estudio haremos hincapié en explicar las ideas en las que se basa la implementación más que en una presentación de las prestaciones que es posible lograr. En la actualidad está disponible un prototipo, la 4-SB-PRAM con cuatro procesadores

físicos y una velocidad de procesador de 0.25 MFlops. Los autores del proyecto indican en [Bac97] las modificaciones necesarias para pasar a otro prototipo esta vez con 128 procesadores físicos que se encuentra actualmente en construcción y en [For96] presentan HPP, una tercera implementación utilizando la última tecnología VLSI y enlaces de alta velocidad así como otras alternativas tecnológicas. La nueva máquina, HPP correría a una velocidad de 96MHz y alcanzaría un incremento de un factor 10 con respecto a una SB-PRAM de 128 procesadores. Con las prestaciones de los desarrollos futuros de la SB-PRAM los autores han comprobado mediante simulaciones que es posible obtener para algunos benchmarks significativos unas prestaciones sostenidas similares a las de multiprocesadores de memoria distribuida comerciales actuales.

La SB-PRAM está formada por *n* procesadores y un número igual de módulos de memoria. Como en todas las máquinas de memoria compartida, los procesadores están interconectados con los módulos de memoria por una red de interconexión (ver Figura 3.1) que en el caso de la SB-PRAM tiene topología de mariposa. Un acceso a la memoria en un procesador se transforma en un paquete de datos que es inyectado en la red de interconexión y viaja hasta el módulo de memoria adecuado. En caso que se trate de una operación de lectura de la memoria, un paquete de datos de respuesta viaja en

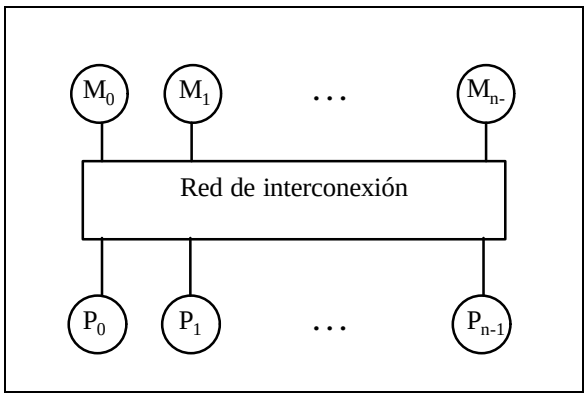

Figura 3.1 Implementación de una PRAM

sentido inverso desde el módulo de memoria hasta el procesador.

Para conseguir altas prestaciones en una arquitectura de este tipo se debe poner especial cuidado en el diseño de ciertos elementos. En particular, disminuir (u ocultar) la latencia de la red de interconexión es un objetivo crucial del diseño. La latencia se oculta mediante dos estrategias: en primer lugar mediante la virtualización de procesadores, por la cual cada procesador físico de la máquina simula el comportamiento de un cierto número de procesadores virtuales (procesos o hebras de ejecución). En segundo lugar, se utiliza una instrucción LOAD retardada: la red no proporciona inmediatamente la respuesta a una instrucción LOAD. Otro aspecto importante es limitar la congestión de los módulos de memoria (puntos calientes). Si en un determinado instante del cómputo todas las referencias a memoria están soportadas en el mismo módulo, los accesos a este módulo de memoria se convertirán en un cuello de botella. La congestión de los módulos se evita también usando dos estrategias: en primer lugar, las direcciones de memoria producidas en los procesadores se transforman mediante una función hash que las distribuye uniformemente entre los diferentes módulos de memoria, y por otra parte, los nodos de la red combinan paquetes de datos con la misma dirección de destino en un único paquete, disminuyendo el tráfico de información en la red. Con esta estrategia de la combinación también se implementa en hardware las operaciones de prefijos.

Las ideas subyacentes al desarrollo de la SB-PRAM no han sido desarrolladas por primera vez en este proyecto. El concepto de procesadores virtuales en hardware se utilizó en el Denelcor HEP [Smi78] o en el Tera MTA [Alv90]. Las ideas del hashing y de la combinación de paquetes se habían utilizado ya en el NYU Ultracomputer [Got83] y la IBM RP3 [Pfi85]. El NYU Ultracomputer, el IBM RP3, la Tera MTA y el Stanford DASH [Len92] (un multiprocesador con memoria virtual compartida y coherencia de cache) también tienen soporte hardware para operaciones de prefijos. El propósito de la SB-PRAM consiste en implementar todas estas ideas en una única máquina.

La SB-PRAM consta de *n* procesadores físicos que implementan una priority CRCW-PRAM con $p=v*n$ procesadores, siendo *v* el número de procesadores virtuales por procesador físico. La arquitectura está basada en la de la Fluent Machine de Ranade [Ran88]. La Figura 3.2 presenta una visión global de la arquitectura de la SB-PRAM. El diseño de la máquina ha precisado del desarrollo de tres circuitos integrados específicos: el procesador, los nodos de la red de interconexión y el chip de ordenación que en la Figura 3.2 aparece con la etiqueta 'sorter'. En nuestra descripción prestaremos atención a aquellas características de la arquitectura que posibilitan la implementación eficiente de una PRAM, y no abordaremos otras decisiones de diseño que son ampliamente utilizadas en otro tipo de máquinas.

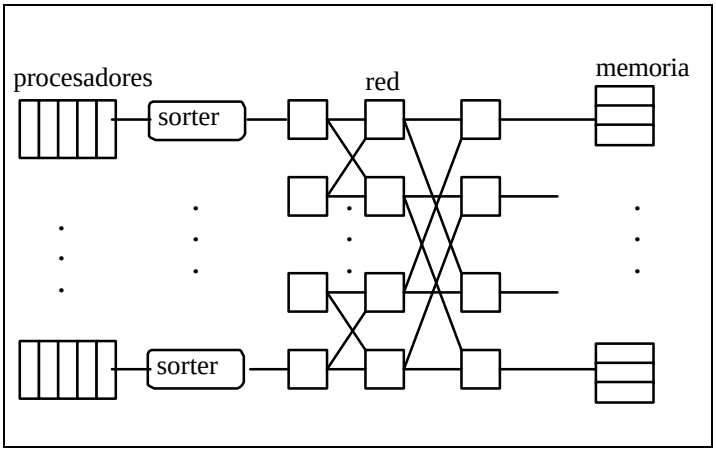

Figura 3.2 Arquitectura de la SB-PRAM

3.2.1. Los procesadores

El procesador [Kel94] implementa una arquitectura de tipo multihilo (multithreaded) Berkeley-RISC-1 con formato de instrucciones de 32 bits. Soporta en hardware operaciones aritméticas y lógicas sobre operandos enteros de 32 bits, así como suma y producto (no división) en punto flotante en simple precisión de acuerdo al estándar IEEE. Las instrucciones son de tres direcciones y las aritméticas pueden operar solamente sobre registros. La SB-PRAM suministra también como instrucciones máquina operaciones de prefijo para suma, máximo, Y lógico, y O lógica que se ejecutan en 2 ciclos de CPU.

Cada procesador físico soporta 32 procesadores virtuales. Cada procesador posee su propia memoria de programa (arquitectura Hardvard), con lo que se reduce el tráfico que introduciría en la red el almacenar el programa en la memoria compartida. Con una memoria de programa por cada procesador físico, todos los procesadores virtuales ejecutan el mismo programa; no obstante, es posible hacer bifurcaciones dependientes del número del procesador virtual.

Un procesador virtual está caracterizado por el contenido de sus registros específicos mientras que la unidad aritmético lógica y la unidad de control y

decodificación son compartidas por todos los procesadores virtuales de un procesador físico. Los procesadores virtuales se planifican de modo cíclico. Después de la ejecución de una instrucción, el control se transfiere al siguiente procesador virtual (proceso), independientemente de la ocurrencia de accesos a memoria. Este tipo de control requiere un cambio de contexto después de la ejecución de cada instrucción (para el cual hay hardware específico) y exige que todas las instrucciones conlleven el mismo tiempo de ejecución. Llamaremos un *ciclo de procesador* a una vuelta en el bucle de planificación de los 32 procesadores virtuales de un procesador físico. Cada procesador virtual posee su propio conjunto de registros: 32 registros de propósito general que se ubican fuera del chip y 6 de propósito específico. Además, cada procesador físico (y por tanto todos sus procesadores virtuales) tienen acceso a cuatro registros de propósito especial que se utilizan en la transformación de las direcciones generadas por el procesador. El procesador utiliza operaciones LOAD y STORE diferentes para acceder a su memoria local o global. Los accesos locales se utilizan para accesos a memoria y periféricos ubicados en la misma placa que el procesador, mientras que los globales son enviados a la red de interconexión. La instrucción LOAD global se implementa con un retardo unitario, es decir, el registro destino de una instrucción LOAD se actualiza después que ha sido ejecutada la instrucción siguiente al LOAD. Los autores del diseño afirman que en la mayoría de los casos no es necesaria la inserción de instrucciones de no-operación después de las LOAD. No existen caches en el chip y el procesador opera a una frecuencia de reloj de 8MHz.

Cuando se realiza un acceso a la memoria compartida, las direcciones lógicas se transforman en direcciones físicas en dos pasos. En primer lugar, si el bit más significativo de la dirección lógica es uno, se suma a ésta el contenido de un registro especial *hi_base*; en caso contrario se suma el contenido del registro *lo_base*. La finalidad de esta operación es asignar diferentes espacios de memoria a datos privados y compartidos. El segundo paso consiste en aplicar una función hash de la forma $H(x) = a\ x \bmod m$ donde x es la dirección transformada que se obtuvo en el paso anterior, y m, el tamaño de la memoria se supone que es una potencia de dos. El factor a es un entero impar en el rango [1, m-1] que se elige aleatoriamente antes del comienzo de la ejecución de la aplicación y se almacena en un registro especial dentro del chip. El módulo de memoria $h(x)$ que contiene la dirección x viene determinado por

$$h(x) = \lfloor H(x)/(m/p) \rfloor$$

siendo p el número de módulos de memoria (que coincide con el número de

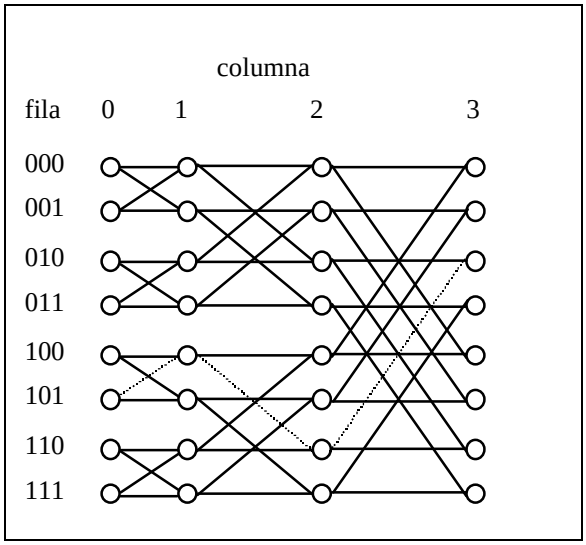

Figura 3.3 Red Mariposa de 4 etapas

procesadores físicos). Por lo tanto, el número del módulo de memoria se almacena en los *log p* bits más significativos de la representación binaria de *H(x)*. Los bits menos significativos indican la dirección dentro del módulo. La función hash utilizada tiene las ventajas de ser muy fácil de computar (el módulo se limita a la extracción de un cierto número de bits de un registro, y el producto se realiza en registros específicos) y de ser una función biyectiva, lo cual facilita la implementación del algoritmo de encaminamiento de información en la red de interconexión. Los autores aseguran que con esta función hash simple consiguen unos resultados suficientemente buenos [Eng93].

Los procesadores no utilizan ningún tipo de memoria cache (que es la aproximación que utiliza el DASH o la KSR [Alm94] debido a que ello provoca variaciones en el tiempo de acceso a memoria, con la dificultad añadida del mantenimiento de la coherencia de las caches.

La finalidad del chip de ordenación (sorter) consiste en ordenar por sus direcciones las peticiones de acceso a memoria que realiza cada uno de los procesadores, de forma que estas peticiones se inyectan en la red de interconexión en orden ascendente de direcciones. El sorter ejerce de interface entre el procesador y la red y hay uno por cada procesador físico. El chip de ordenación implementa un array lineal sistólico de 8 entradas que puede ser llenado y vaciado simultáneamente. El chip se encarga de mezclar en uno sólo todos los accesos a una misma posición de memoria que se produzcan dentro del mismo *ciclo de procesador*. El chip trabaja a una frecuencia de reloj de 32MHz, que es también la frecuencia de trabajo de los nodos de la red de interconexión.

3.2.2. La red de interconexión

La red de interconexión de la SB-PRAM tiene una topología de mariposa. Como ya estudiamos en el epígrafe 2.9 del Capítulo 2, una red mariposa de grado 2 consiste en una red de *n(1+log n)* nodos. Cada nodo tiene asignado un número unívoco *<col, fil>* con *0 £ col £ log n, 0 £ fil £ n-1*. *<col, fil>* puede verse como la concatenación de las representaciones binarias de los números *col* y *fil.* El nodo *<col, fil>* está conectado con los nodos *<col+1, fil>* y *<col+1, fil*$\otimes 2^{col}$*>* donde $\otimes$ denota la o-exlusiva. La Figura 3.3 presenta una red mariposa de 4 etapas. La Figura 3.4 representa la red de mariposa de la SB-PRAM, con cuatro entradas y cuatro salidas.

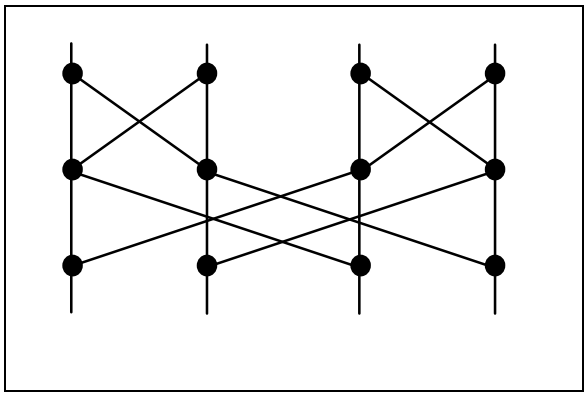

Figura 3.4 Topología de la red de la 4-SB-PRAM

Los nodos de la red de interconexión [Cro93] cuyos chips han sido específicamente diseñados para la máquina implementan un nodo de encaminamiento de una mariposa con 2 entradas, in_0 e in_1 y 2 salidas out_0 y out_1 (en la Figura 3.6 se presentan dos de estos nodos).

Supondremos que in_0 y out_0 son los enlaces que para un determinado nodo aparecen en la dimensión horizontal mientras que in_1 y out_1 son los de la dimensión no-horizontal. La forma de encaminar datos en una red mariposa es simple: en un determinado nodo se compara el bit de la columna de ese nodo con el mismo bit del nodo destino; si son iguales, el mensaje ha de encaminarse por out_0, en caso contrario por out_1. En la Figura 3.3 se representa con trazo discontinuo el camino que seguiría un mensaje que partiera del nodo <5, 0> con destino al nodo <2, 3>.
Los procesadores envían peticiones de lectura, escritura o prefijos a la memoria, y estas peticiones son encaminadas hacia los módulos de memoria a través de la red de interconexión. Si la petición es una lectura (LOAD) o una operación de prefijo será necesario enviar una respuesta a través de la red al procesador que realizó la petición. Los paquetes de datos que circulan por la red consisten en un modo (READ, WRITE o PREFIX_OP), una dirección y en el caso de WRITE o PREFIX_OP un dato. Para obtener una longitud de paquete única se inserta un valor que no será utilizado (dummy) en los paquetes READ.

Cada procesador está conectado a un nodo de entrada de la red a través de un chip de ordenación. El sorter recolecta las peticiones de memoria de 8 procesadores virtuales y las inyecta en la red ordenadas por su dirección de destino. Después de enviar las peticiones el sorter inyecta en la red un paquete especial con modo EOR (End of Round) y dirección de valor infinito, que indica el final de lo que llamaremos un *ciclo de red*. El paquete EOR se inyecta también en la red en el caso que no haya peticiones de acceso a memoria en un determinado *ciclo de procesador*. Dado que cada procesador físico emula 32 procesadores virtuales, en un *ciclo de procesador* tienen lugar cuatro *ciclos de red* y se inyectarán en la red 4 paquetes EOR por procesador. Si se requiere un paquete de datos respuesta, se almacena información de encaminamiento en una cola FIFO en cada nodo de la red. En el sentido desde los procesadores hacia la memoria, el encaminamiento se determina por la dirección de la petición, mientras que en el sentido de la memoria hacia los procesadores, el camino se determina mediante la información que se almacenó en las colas en el camino 'de ida'.

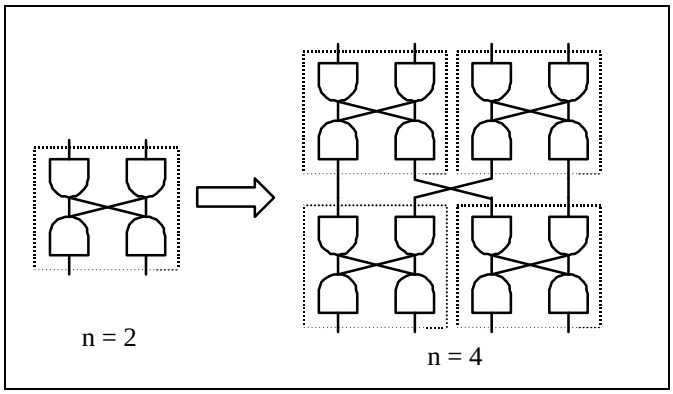

Figura 3.5 Particionado de la red

La red ha de tener una frecuencia de reloj al menos doble de la de los procesadores debido a que las partes de direcciones y datos de un paquete de datos están multiplexadas. Por simplicidad se eligió una frecuencia de reloj cuatro veces más alta que la de los procesadores. Los nodos de la red se han diseñado en la misma tecnología que el procesador (tecnología *sea of gates*) y tal como se muestra en la Figura 3.5, partes de diferentes nodos se han integrado en un único chip. En la parte izquierda de la figura se presenta un esquema del contenido de un chip, mientras que en la parte derecha se muestra cómo conectando entre sí los chips se obtiene la red de la 4-SB-PRAM (Figura 3.4). Esta decisión del diseño ha venido motivada por el esfuerzo en

reducir el coste de cableado fuera del chip así como la reducción del número de circuitos a utilizar [Cro93]. Los autores del diseño indican [Bac97] que una primera versión del chip de los nodos de la red que se llegó a construir se descubrió que era incorrecto debido a una incorrecta implementación del algoritmo de encaminamiento. Debido a ello, en el prototipo disponible de la 4-SB-PRAM la red es emulada con un diseño FPGA (utilizan la FPGA XC4013 de Xilinx) con una frecuencia de reloj de 5MHz. Ello les ha permitido comprobar la corrección del diseño, y cuando construyan el chip, éste tendrá la frecuencia prevista de 32MHz.

3.2.2.1. El algoritmo de ruteo

El algoritmo de encaminamiento que utiliza la SB-PRAM [Kel95] es una versión segmentada del algoritmo de Ranade [Ran91] con colas FIFO de tamaño 16. La idea fundamental de este algoritmo (que es la misma que la que utiliza la Fluent Machine de Ranade) consiste en conseguir que la red preserve el orden de las peticiones de acceso a memoria que produjo el chip de ordenación a la salida del procesador. Cada nodo de la red selecciona de sus dos buffers de entrada (ver Figura 3.6) el paquete con menor dirección para ser emitido a la salida, manteniendo el orden de los paquetes de entrada (dos secuencias ordenadas que se mezclan produciendo una única secuencia, también ordenada). Si en un mismo *ciclo de red* hay peticiones de memoria del mismo tipo (lectura, escritura, prefix_op) para la misma dirección de memoria, las peticiones se combinan y una sola petición se pasa a la siguiente etapa de la red. Esto es posible porque la red, como hemos indicado, preserva la ordenación establecida por el chip de ordenación. En el camino de vuelta (sentido desde memoria hacia el procesador) las peticiones que fueron combinadas se re-duplican utilizando la información que se almacenó en las colas. Debido a la combinación de peticiones a memoria, a lo sumo 4 peticiones a una misma dirección de memoria alcanzan un determinado módulo de memoria incluso si todos los procesadores virtuales de la máquina acceden simultáneamente la misma posición de memoria. Dado que la SB-PRAM implementa

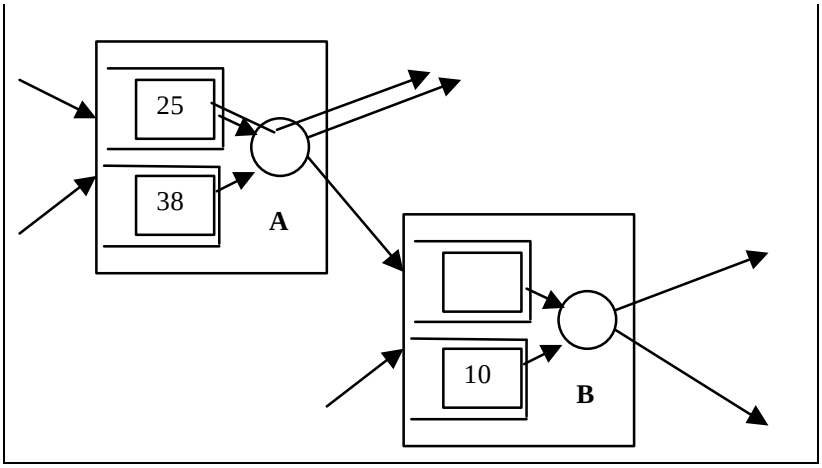

Figura 3.6 Utilización de los paquetes fantasma en los nodos de la red

una priority-CRCW PRAM, una petición de escritura concurrente se resuelve seleccionando el operando correspondiente al procesador con mayor número, que se puede determinar a partir del enlace por el que llega la petición al nodo. Las operaciones de prefijo (multiprefijo) se implementan en la red de interconexión como una forma especial de combinación. En el camino desde procesadores a memorias dos operandos *i* y *j* se procesan de acuerdo a su operación de prefijo *op* $\hat{I}$ *{and, or, max, add}* y el

resultado se envía en el sentido hacia la memoria. Además, el operando, *op* y el operador izquierdo *i* se almacenan en otra cola FIFO. En el camino desde la memoria hacia el procesador, el resultado entrante en un nodo, *s* y *s op i* son enviados hacia los procesadores (uno por cada enlace en el sentido memoria hacia procesadores). Los nodos de la red de interconexión disponen de una unidad aritmético lógica para llevar a cabo estas operaciones.

El paquete de datos seleccionado en un nodo se transmite al siguiente nivel de la red a través del enlace de salida adecuado. Sólo los paquetes con modo EOR se emiten por las dos salidas del nodo para garantizar la separación de los *ciclos de red*. La dirección infinita de los paquetes EOR garantiza que un paquete EOR sólo será seleccionado si ambos buffers de entrada contienen paquetes EOR.

Consideremos la situación representada en la Figura 3.6 en la que el nodo B no puede transmitir el paquete con dirección 10 porque debe asegurar que no recibirá dentro del mismo *ciclo de red* un paquete con dirección menor que 10. Cuando el nodo A selecciona el paquete con destino 25 para transmitirlo a través del enlace de salida superior, puede transmitir esa información al nodo B enviándole un paquete con un modo especial GHOST (fantasma) y dirección 25 por el otro enlace de salida. Cuando el nodo B recibe el paquete fantasma sabe que debido al orden establecido, todos los paquetes que recibirá en el futuro por el enlace por el que ha recibido el paquete fantasma tendrán direcciones mayores que 25, por lo tanto, un paso después el nodo B puede transmitir el paquete con destino 10. Si un paquete fantasma llega a un nodo que contiene más de un paquete o bien un paquete que ha de esperar, entonces el paquete fantasma puede ser eliminado de la red.

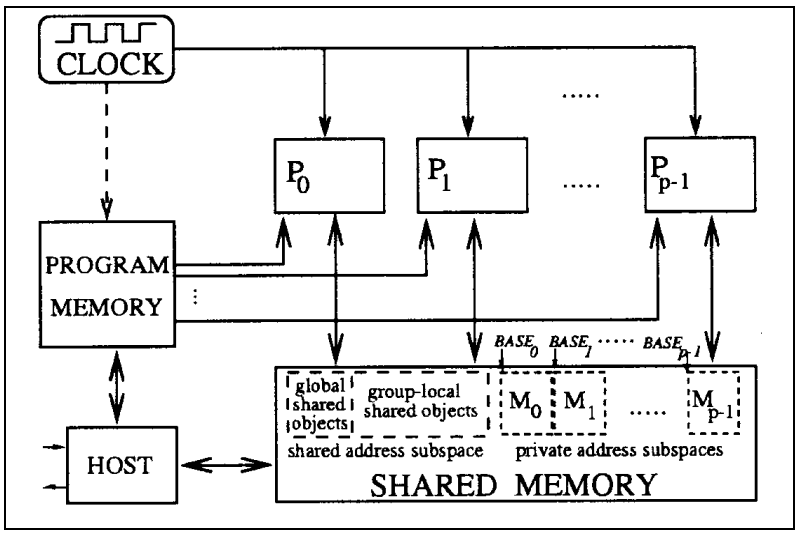

Figura 3.7 Estructura de la SB-PRAM desde el punto de vista del programador

3.3. Fork95

Fork95 [For], [Kes95a], [Kes95b], [Kes97a], es un lenguaje experimental paralelo imperativo fuertemente síncrono diseñado para expresar algoritmos PRAM. El lenguaje está basado en ANSI C [Ans90] y ofrece constructos para dividir jerárquicamente los grupos de procesadores en subgrupos y manipular subespacios de memoria privada y compartida. Fork95 es el sucesor de FORK [Hag92], [Kes94], que fue fundamentalmente un diseño de tipo teórico y del que nunca llegó a existir un compilador. En FORK los punteros, arrays dinámicos, estructuras de datos complejas y

las sentencias de control de flujo no estructuradas fueron sacrificadas en aras de facilitar el diseño y comprobar la corrección del lenguaje. Estas limitaciones acabaron convirtiendo a FORK en un lenguaje sin posibilidad alguna de utilización práctica, no obstante, en FORK aparecían ya las ideas en las que se basa Fork95 como son el concepto de grupo de procesadores y la diferenciación entre espacios de memoria privados y compartido. En la actualidad, los autores de Fork95 trabajan en una nueva versión del lenguaje, que se llamará ForkLight y que pretenden que sea más adecuado para máquinas asíncronas de memoria compartida.

En Fork95 están presentes todas las características habituales en los lenguajes de programación secuencial. El lenguaje es en este caso un superconjunto de C, y las partes secuenciales son idénticas a las escritas en ese lenguaje. Aparte de librarle de las carencias que impedían una implementación útil de FORK, las mayores novedades de Fork95 con respecto a su antecesor es la introducción de cómputos localmente asíncronos para evitar los puntos de sincronización y permitir una mayor libertad a la hora de elegir el modelo de programación. Por otra parte, se ha abandonado la virtualización de procesadores limitando el número de procesos a los recursos hardware, y se han introducido punteros en el lenguaje.

Tanto FORK como Fork95 están muy ligados en su génesis al proyecto de construcción de la SB-PRAM. La Figura 3.7 presenta la estructura de la SB-PRAM desde el punto de vista del programador. Los subespacios de direcciones privadas de memoria de cada procesador están incrustados en la memoria compartida, y para su acceso se utilizan direcciones relativas a un puntero BASE para cada procesador. Todos los procesadores reciben la misma señal de reloj, de modo que la máquina es síncrona a nivel de instrucciones. El tiempo de acceso a memoria es uniforme para todos los procesadores y posiciones de memoria. El almacenamiento de una palabra de 32 bits en memoria consume un ciclo de CPU (es decir, lo mismo que una operación aritmética entera o en punto flotante), mientras que una lectura de un dato de este tipo consume dos ciclos de CPU. Como hemos estudiado, esta relación ideal de coste de comunicaciones/cómputo se alcanza gracias a la utilización de técnicas de hashing, ocultamiento de la latencia, combinación inteligente de nodos red y virtualización hardware de procesadores.

Fork95 proporciona dos modos de programación diferentes: el modo síncrono (que era el único disponible en FORK) y el asíncrono. Cada función se clasifica como síncrona o asíncrona. Dentro del modo síncrono los procesadores forman grupos que pueden ser divididos recursivamente en subgrupos, dando lugar a una estructura jerárquica de grupos en árbol. Las variables compartidas tienen asignada una memoria única para todo el grupo de procesadores que las creó, mientras que las variables privadas tienen asignada posiciones de memoria independientes en cada procesador. Todos los procesadores pertenecientes a un grupo hoja operan síncronamente. En el modo asíncrono, la librería de ejecución de Fork95 proporciona diferentes rutinas para implementar varias clases de semáforos, barreras, bucles paralelos equilibrados y colas paralelas, estructuras todas que son de interés en la implementación de algoritmos asíncronos. Por defecto, el modo de ejecución de los programas paralelos es el asíncrono. Dado que el lenguaje está basado en C, se puede incluir en los programas Fork95 código C (secuencial) estándar con muy pocos cambios.

En las siguientes secciones presentaremos con mayor detalle algunas de las características más importantes de Fork95, así como ejemplos de implementación de programas y su ejecución.

```
#include <fork.h>
#include <io.h>


void main(void) {
  if (__PROC_NR__ == 0)
    printf("Programa ejecutado por %d
procesadores\n",_STARTED_PROCS__);
  barrier;
  prS("Hola\n");
  barrier;
}
```

Figura 3.8 Un primer programa simple en Fork95

3.3.1. La ejecución de programas Fork95

Dado que la SB-PRAM, máquina objeto del compilador Fork95 está aún a nivel de pre-prototipo en periodo de pruebas, los miembros del equipo de trabajo Fork95/SB-PRAM han diseñado un simulador de la SB-PRAM con el que se pueden comprobar los algoritmos diseñados con el lenguaje. A este simulador se le ha llamado *pramsim* y con él hemos ejecutado todos los códigos fork95 que se presentan en esta memoria. El compilador de Fork95, *fcc* [Kes98] versión 1.7 y el simulador de la SB-PRAM se ejecutan en estaciones de trabajo SUN bajo SunOS o Solaris. La simulación de un gran número de procesadores requiere bastante memoria principal. El compilador de Fork95, *fcc*, genera código para el simulador de la SB-PRAM y su diseño está basado en el del compilador lcc 1.9 [Fra91], [Fra95], un compilador de ANSI C de una pasada. El compilador genera código ensamblador (ficheros *.s) que luego se procesan para convertirlos en formato COFF (ficheros *.o). El cargador de enlaces *plink* produce

```
***** PRAM-SIMULATOR (v2.0) *****
          >>>>>>>> DAMN FAST <<<<<<<<
       (c) by hirbli & stefran  07.10.94
You have 2 physical processors with 2 vP's each

Relocating file hola... done
Loading file /var/tmp/aaaa001pd.cod...Doing realloc
Doing realloc
 done
PRAM P0 = (p0, v0)> g
Programa ejecutado por 4 procesadores
Hola
Hola
Hola
Hola
EXIT: vp=#0, pc=$000001fc
EXIT: vp=#1, pc=$000001fc
EXIT: vp=#2, pc=$000001fc
EXIT: vp=#3, pc=$000001fc
Stop nach 12194 Runden, 812.933 kIps
01fc 18137FFF  POPNG   R6, ffffffff, R1
PRAM P0 = (p0, v0)> q
```

Figura 3.9 La ejecución del programa de la Figura 3.8

código ejecutable que correría en la SB-PRAM y también en el simulador.

La Figura 3.8 presenta un primer ejemplo de programa Fork95. Todos los programas han de incluir el fichero *fork.h*, que contiene las cabeceras de las rutinas que

soportan paralelismo así como algunas macros. Como es habitual, el fichero *io.h* contiene las declaraciones de las funciones de entrada salida como *printf()* o la macro *prS()* que permite imprimir en pantalla una cadena de caracteres. Si este programa se graba con el nombre *hola.c* y se compila:

```
fcc -FORK -m -I$FORKDIR/include hola.c
```

y se ejecuta mediante el simulador *pramsim* se obtiene el resultado que aparece en la Figura 3.9, que indica al final que la ejecución del programa consumió 12194 ciclos (eso significa que cada procesador virtual de la SB-PRAM ejecutaría 12194 instrucciones, y la ejecución se llevó a cabo a 812.933 kIps (kilo instrucciones por segundo), lo cual es una medida de la rapidez de ejecución del intérprete. Los autores del simulador utilizan 'rounds' en lugar de ciclos de cpu porque en la implementación de la SB-PRAM, cada procesador físico soporta un máximo de 32 procesadores virtuales. El comando 'g' que aparece en la Figura 3.9 después del prompt del simulador es el que se utiliza para iniciar la ejecución del programa que se ha cargado. Si se escribe 'help' aparece una lista completa de los comandos disponibles que permiten, entre otras cosas, cambiar el número de procesadores (físicos y virtuales) que se van a utilizar por defecto. En nuestro ejemplo, la ejecución la llevaron a cabo cuatro procesadores virtuales, ejecutados por dos físicos, como se indica al comienzo de la ejecución.

```
#include <fork.h>
#include <io.h>

#define N 10
sh int sq[N];
sh int p;

void main(void) {
  pr int i;

  p = __STARTED_PROCS__;
  for (i = __PROC_NR__; i < N; i += p)
    sq[i] = i * i;
  barrier;
  for (i = __PROC_NR__; i < N; i += p)
    pprintf("i: %d sq: %d\n", i, sq[i]);
  barrier;
}
```

Figura 3.10 Un bucle de paralelismo de datos para calcular los cuadrados de N números

En Fork95 el programa de usuario (la función main()) es ejecutada por todos los procesadores disponibles en la PRAM en la que el programa haya sido arrancado. Este número de procesadores arrancados puede conocerse en tiempo de ejecución a través de la constante global entera y compartida *__STARTED_PROCS__*. Cada procesador mantiene una constante entera global y privada *__PROC_NR__* que es el identificador del procesador en tiempo de ejecución. Los procesadores se numeran consecutivamente desde 0 a *__STARTED_PROCS__-1*. Utilizando estos identificadores se puede escribir el típico bucle de paralelismo de datos como se muestra en la Figura 3.10 en la que el bucle se utiliza para calcular los cuadrados de N números. El programador siempre puede, a través del preprocesador utilizar nombres diferentes de *__STARTED_PROCS__* o *__PROC_NR__* para estos identificadores. La función *pprintf()* que aparece en la Figura 3.10 es completamente análoga al clásico *printf()* de C, con la salvedad que imprime en pantalla el identificador (*__PROC_NR__*) del

procesador que la ejecuta, de forma que el resultado es el que aparece en la Figura 3.11. La sentencia *barrier*, que aparece en los dos programas que hemos presentado provoca una sincronización por barrera de todos los procesadores que están ejecutando el programa, de forma que todos han de alcanzar ese punto del código para poder proseguir la ejecución.

```
#0000# i: 0 sq: 0
#0001# i: 1 sq: 1
#0002# i: 2 sq: 4
#0003# i: 3 sq: 9
#0000# i: 4 sq: 16
#0001# i: 5 sq: 25
#0003# i: 7 sq: 49
#0002# i: 6 sq: 36
#0001# i: 9 sq: 81
#0000# i: 8 sq: 64
```

Figura 3.11 Resultado de la ejecución del programa de la Figura 3.10

3.3.2. Variables privadas y compartidas

Como hemos visto en la Figura 3.7, la memoria compartida de la PRAM se particiona, de acuerdo a lo que el usuario pretenda, en un espacio de direcciones privadas para cada procesador y un espacio de direcciones compartidas, comunes a todos los procesadores que podrá ser a su vez dinámicamente subdividido entre los diferentes grupos de procesadores. De acuerdo a esta división, las variables se clasifican en Fork95 como privadas (se han de declarar con el cualificador *pr*) o compartidas (*sh*). Por defecto, si una variable se declara sin cualificador de compartición, se asume que es privada. El término compartida se refiere en todo momento al grupo de procesadores que ha definido la variable. Definiremos más adelante con precisión el concepto de grupo de procesadores. Por ahora es suficiente saber que una variable compartida es accesible por todos los procesadores del grupo que la ha declarado y tiene asignada una posición única en memoria.

No es parte del lenguaje el definir lo que ocurre si varios procesadores escriben concurrentemente la misma variable compartida. Fork95 adopta el método de resolución de conflictos de la máquina objeto. En el caso de la SB-PRAM, triunfará el procesador con mayor *__PROC_NR__* (es decir, se trata de una *priority* CRCW-PRAM). De cualquier modo, dado que pueden utilizarse otros esquemas de resolución de conflictos de escritura, un buen programa no debería depender de estos esquemas, sino que debería utilizar elementos del propio lenguaje (como las operaciones de prefijos) para definir el comportamiento deseado.

IDENTIFICADOR	SIGNIFICADO	TIPO	COMENTARIO
__STARTED_PROCS__	Número de procesadores disponibles	*sh int*	sólo de lectura
__PROC_NR__	Identificador físico del procesador	*pr int*	sólo de lectura
@	Identificador lógico del grupo	*sh int*	puede redefinirse
$	Identificador lógico del procesador	*pr int*	puede redefinirse

Tabla 3.1 Identificadores especiales en Fork95

El resultado de una función que devuelva algún resultado será siempre privado. Obviamente, un valor retornado por una función puede hacerse accesible a todos los procesadores asignándolo a una variable compartida.

La variable especial ***$*** mantiene en cada momento el identificador lógico de cada procesador relativo a su grupo (en FORK este identificador se representaba por #). Inicialmente, *$* toma el valor del identificador físico del procesador (*__PROC_NR__*), pero puede cambiarse durante la ejecución del programa. La Tabla 3.1 presenta los identificadores especiales de Fork95.

La Tabla 3.2 resume la utilización de los cualificadores para el tipo de almacenamiento (privado o compartido de Fork95.

CUALIFICADOR	SIGNIFICADO	NOTA	EJEMPLO DE USO
`sh`	Declara una variable compartida		`sh int a=7, *p, v[10];`
`pr`	Declara una variable privada	defecto	`pr int i, *q, s[2];`

Tabla 3.2 Cualificadores de tipo de almacenamiento

3.3.3. Operaciones de prefijos

La SB-PRAM proporciona a nivel de instrucciones máquina operaciones de prefijos para las operaciones de suma, máximo, and y or lógicas que se realizan en dos ciclos de CPU para un máximo de 4096 procesadores físicos. Fork95 proporciona al programador estas operaciones como operadores atómicos para expresiones (no como funciones). Consideremos por ejemplo, la expresión

```
k = mpadd(ps, expresion);
```

Supongamos que esta sentencia es ejecutada simultáneamente por un conjunto *P* de procesadores. *ps* deberá ser un puntero (puede ser privado) a una variable entera compartida, *s*, y *expresion* ha de ser una expresión entera. Así pues, diferentes procesadores pueden apuntar a diferentes variables compartidas, *s* (usando punteros privados) de forma que diferentes sumas de prefijos se computen simultáneamente. Sea $Q_s \subseteq P$ el subconjunto de procesadores de *P* cuyos punteros apuntan a la misma dirección compartida *s*. Supongamos que los procesadores $q_{s,i} \in Q_s$ están indexados en orden creciente de su identificador físico, *__PROC_NR__* (este orden es heredado del hardware y no depende del identificador lógico, *$*). En primer lugar cada procesador $q_{s,i}$ de cada conjunto Q_s evalúa la expresión localmente dando lugar a un valor entero privado, $e_{s,i}$. Para cada posición de memoria compartida *s* direccionada, sea v_s el valor de *s* antes de la ejecución de *mpadd*. La instrucción *mpadd* computa simultáneamente para cada procesador $q_{s,i} \in Q_s$ el valor entero privado $v_s + e_{s,0} + e_{s,1} + \ldots + e_{s,i-1}$ y en el ejemplo considerado, este valor es asignado a la variable *k*. Inmediatamente después de la ejecución de *mpadd*, cada dirección de memoria compartida *s* contiene la suma de prefijos:

$$v_s + \sum_{j \in Q_s} e_{s,j}$$

de todas las expresiones participantes.

Si se ignora el valor devuelto por la instrucción mpadd(), ésta puede utilizarse para realizar una reducción global. Sin embargo, si se está interesado solamente en este efecto lateral se pueden utilizar las funciones *syncadd*, *syncmax*, *syncand* y *syncor* que llevan los mismos parámetros pero no retornan ningún valor. Por ejemplo, el número de

procesadores disponibles puede calcularse en la variable *p* ejecutando síncronamente el código:

```
sh int p = 0;
syncadd(&p, 1);
```

La misma idea puede utilizarse para conseguir una renumeración consecutiva de los procesadores *0, 1, ..., p-1* almacenada en un identificador privado *name*:

```
sh int p = 0;
pr int name = mpadd(&p, 1);
```

```
 1 #define LIBRE 0
 2 #define BLOQUEADO   1
 3
 4 sh int lock = LIBRE;
 5
 6 void main(void) {
 7   if (__PROC_NR__ == 0)
 8     printf("%d Procesadores ejecutando\n", __STARTED_PROCS__);
 9   barrier;
10   printf("Procesador %d\n", __PROC_NR__);
11   while (mpadd(&lock, BLOQUEADO) != LIBRE)
12     ;
13   printf("Hola del procesador %d\n", __PROC_NR__);
14   lock = LIBRE;
15   barrier;
16 }
```

Figura 3.12 Una implementación simple del acceso a una sección crítica

La instrucción *mpadd* que se ejecuta cuando se traduce una expresión *mpadd()* se ejecuta atómicamente. En la SB-PRAM consume 2 ciclos de CPU. Por ello, esta instrucción (y el resto de operaciones de prefijos) pueden utilizarse para implementar el acceso exclusivo a secciones críticas de código como en el ejemplo que presentamos en la Figura 3.12.

```
PRAM P0 = (p0, v0)> g
4 Procesadores ejecutando
PPPPrrrroooocccceeeessssaaaaddddoooorrrr    1230



Hola del procesador 1
Hola del procesador 2
Hola del procesador 3
Hola del procesador 0
Stop nach 27152 Runden, 905.067 kIps
```

Figura 3.13 El programa de la Figura 3.12 ejecutado por 4 procesadores

En el programa de la Figura 3.12, el recurso compartido es la pantalla, y hemos de impedir el acceso concurrente a la misma, si queremos que las escrituras en pantalla no se entremezclen. De hecho, la escritura concurrente en pantalla que todos los procesadores que ejecutan este programa producen al alcanzar la llamada a *printf()* de la línea 10, provoca que estas escrituras aparezcan entremezcladas unas con otras, tal como muestra la Figura 3.13. En cambio, las escrituras producidas por la llamada a *printf()* de la línea 13 aparecen separadas en pantalla porque el acceso a pantalla ha sido

convenientemente protegido mediante el uso de un 'cerrojo' (lock). Los procesadores producen su escritura según el número de procesador 1, 2, 3, 0, es decir, se considera el último procesador al que tiene *_PROC_NR_=0*.

Cuando la variable *lock* tiene un valor no nulo, la expresión *mpadd()* se evalúa a un valor no nulo en todos los procesadores, de modo que los procesadores ejecutan el bucle vacío de espera de las líneas 11 y 12 hasta que obtienen un valor cero de la expresión *mpadd()* (lo cual ocurre cuando se convierten en $q_{l,0}$ en la notación que utilizamos anteriormente). Puesto que el resto de procesadores que ejecutan el bucle de espera (los $q_{l,i}$ con i>0) obtienen valores no nulos, se garantiza que sólo un procesador accede a la sección crítica. Fork95 proporciona diferentes tipos de cerrojos que se implementan en ensamblador porque se utilizan muy frecuentemente en cómputos asíncronos. El código de la Figura 3.19 ilustra la utilización en Fork95 de un cerrojo simple mediante el tipo de datos *simple_lock*. Las funciones *simple_lock_init()*, que inicializa el cerrojo así como *simple_lockup()* y *simple_unlock()* que enmarcan la sección crítica, se definen en el fichero *fork.h* y están implementadas en ensamblador.

Fork95 soporta también los operadores *mpmax()* (máximo), *mpand()* (y lógica) y *mpor()* (o lógica). Estas operaciones están disponibles sólo para enteros debido a limitaciones hardware de la SB-PRAM.

Muchos algoritmos paralelos pueden implementarse elegantemente utilizando como bloques básicos las operaciones de prefijo [Ble89], de ahí la importancia de una eficiente implementación de las mismas. La librería PAD [Kes96] proporciona rutinas multiprefijo para tipos de datos arbitrarios con una complejidad temporal $O((n/p) + \log p)$, siendo p el número de procesadores y n el tamaño de los datos.

3.3.4. Zonas síncronas y asíncronas en los programas Fork95

Fork95 proporciona dos modos diferentes de programación asociados estáticamente (en tiempo de compilación) con zonas de código: el modo síncrono y el asíncrono. En el modo síncrono, los procesadores se mantienen sincronizados a nivel de sentencia, es decir, todos los procesadores ejecutan la misma sentencia en el mismo instante de tiempo y cumplen el que los autores de Fork95 llaman invariante de sincronicidad, que indica que los contadores de programa son iguales en cada instante de tiempo para todos los procesadores pertenecientes al mismo grupo activo. En modo asíncrono esto no tiene porqué ser cierto. En modo asíncrono la estructura del grupo de procesadores no puede alterarse (es de sólo lectura), no pueden alojarse (declararse) variables compartidas ni se puede alojar objetos en el heap compartido. En modo asíncrono no existen puntos de sincronización implícito entre los procesadores, pero pueden provocarse utilizando la sentencia *barrier*.

Al comienzo de un programa Fork95, todos los procesadores en los que el programa ha sido arrancado por el usuario ejecutan en paralelo el código de inicialización. A continuación, todos los procesadores comienzan la ejecución del programa en modo asíncrono invocando a la función *main()*.

Las funciones se clasifican en síncronas o asíncronas, y por defecto son asíncronas (ver Tabla 3.3). La función *main()* se considera asíncrona por defecto.

CUALIFICADOR	SIGNIFICADO	NOTA	EJEMPLO DE USO
`sync`	Declara (puntero a) función síncrona		`sync int tonta(sh int a) {...}`
`async`	Declara (puntero a) función asíncrona	defecto	`sh void(*tarea[a])(void);`

Tabla 3.3 Cualificadores de tipo de funciones y punteros

Una función síncrona se ejecuta en este modo (síncrono), excepto los bloques que comiencen con una sentencia *farm*:

```
farm <sentencia>
```

La sentencia *farm* provoca la entrada en modo asíncrono para la ejecución de <*sentencia*> y restablece el modo síncrono al final de su ejecución mediante una sincronización por barrera de todos los procesadores que han ejecutado el cuerpo.

La funciones asíncronas se ejecutan en este modo excepto los bloques que comiencen con una sentencia *start*:

```
start <sentencia>
```

La sentencia *start*, sólo se permite en modo asíncrono y cambia el modo de ejecución a síncrono para la sentencia que constituye su cuerpo. Provoca que todos los procesadores disponibles se sincronicen y ejecuten la sentencia simultáneamente en modo síncrono con identificadores lógicos *$* únicos numerados consecutivamente entre 0 y *__STARTED_PROCS__-1*.

Para poder mantener esta clasificación estática del código en síncrono o asíncrono, desde dentro de una zona asíncrona sólo se pueden invocar funciones asíncronas. Como contrapartida, si se invoca una función asíncrona desde una zona síncrona de programa, se provoca una entrada implícita en modo asíncrono. Es superfluo utilizar una sentencia farm desde un contexto asíncrono y ello puede provocar un bloqueo (el compilador genera un aviso).

Las funciones asíncronas no deben alojar variables locales compartidas. Esto implica que están prohibidos los parámetros formales compartidos para las funciones asíncronas. La implementación actual del compilador de Fork95 sólo permite sentencias start en el nivel más externo del programa. No se permite el anidamiento de sentencias start, lo cual es una debilidad del lenguaje que los autores de Fork95 tratan de obviar mediante una generalización de la sentencia start (la sentencia *join*) que permite el anidamiento arbitrario de zonas de código síncronas y asíncronas [Kes97].

La sentencia *seq*:

```
seq <sentencia>
```

funciona de forma similar a la sentencia *farm*. Su cuerpo es una zona asíncrona y se ejecuta por tanto en modo asíncrono, pero al contrario que *farm*, es ejecutada únicamente por un sólo procesador elegido arbitrariamente de entre el grupo de procesadores que ejecuta la sentencia *seq*. La sentencia *seq* trata de conseguir una implementación eficiente por parte del back-end del compilador que podría mantener a todos los procesadores del grupo ocupados en la ejecución de <sentencia> mediante la

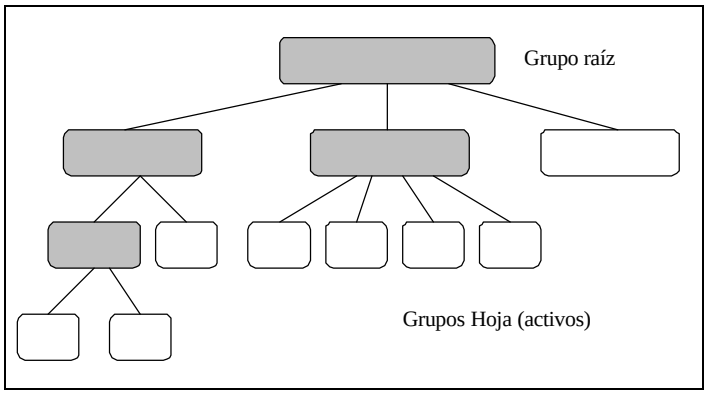

Figura 3.14 La jerarquía de grupos de Fork95

extracción de paralelismo a nivel de instrucción. El compilador emite un aviso si se utiliza una sentencia *seq* en una zona de código asíncrono.

3.3.5. El concepto de grupo en Fork95

Los programas Fork95 son ejecutados por grupos de procesadores, en lugar de como es habitual por procesadores individuales. Inicialmente hay sólo un grupo que contiene todos los procesadores disponibles y los grupos pueden ser subdivididos recursivamente. En cualquier punto de la ejecución del programa, todos los grupos existentes en ese momento constituyen una jerarquía de grupos que se estructura en forma de árbol (ver Figura 3.14).

Los subgrupos de un grupo dado se distinguen por su identificador de grupo. El identificador de grupo del grupo hoja al cual pertenece un procesador se accede a través de la variable compartida @. Inicialmente @ vale 0 pero su valor puede cambiarse de forma adecuada durante la ejecución del programa. Tanto el identificador de procesador $ como el de grupo, @ se guardan automáticamente cuando un grupo es dividido y se restablecen cuando el grupo se reactiva, no obstante, es responsabilidad del programador asignarle a estas variables valores adecuados.

En cualquier punto de la ejecución de un programa en modo síncrono Fork95 cumple el invariante de sincronicidad, de forma que todos los procesadores pertenecientes al mismo grupo de procesadores activos siguen el mismo flujo de control y ejecutan las mismas instrucciones en los mismos instantes. Además, todos los procesadores del grupo tienen acceso a un subespacio común de direcciones compartidas, de forma que las variables compartidas alojadas tienen asignadas una dirección de memoria única para cada grupo que las declara. Un procesador puede conocer el número de procesadores que pertenecen a su grupo a través de la rutina síncrona *groupsize()*.

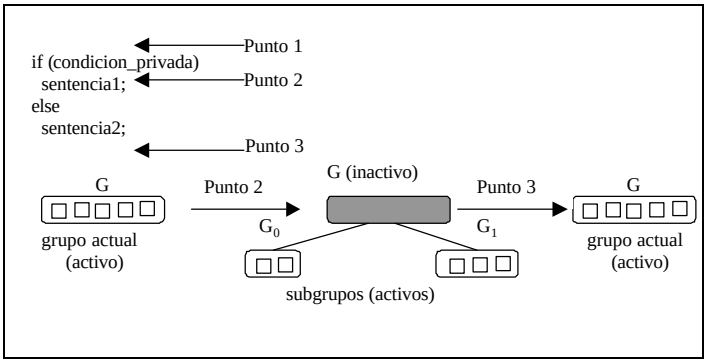

Figura 3.15 La división de los grupos en modo síncrono con una sentencia condicional con condición privada

A la entrada en un segmento de código síncrono, los procesadores forman un único grupo. No obstante, es posible que el flujo de control diverja en ramas cuyas condiciones dependan de valores privados. Para poder mantener el invariante de sincronicidad, el grupo hoja activo ha de dividirse en subgrupos. Se considera que una expresión es privada si no está garantizado que se evalúe al mismo valor en todos los procesadores (o lo que es lo mismo, si contiene una variable privada, una llamada a función o un operador multiprefijo). Las condiciones de sentencias condicionales o de bucle que sean compartidas no afectan a la sincronicidad, dado que la rama que se tome será la misma para todos los procesadores que la ejecuten.

Una condición privada en una sentencia condicional (if) provoca que el grupo actual de procesadores sea dividido en dos subgrupos (ver Figura 3.15): los procesadores para los que la condición se evaluó a un valor no nulo forman el primer subgrupo y ejecutan la rama correspondiente al *if*, mientras que los procesadores

```
 1 #include <fork.h>
 2 #include <io.h>
 3
 4 void main(void) {
 5   pr int i = $;
 6
 7   start {
 8     if ($ < 4) {
 9       @ = 23;
10       $ = 50-$;
11       if (i == 0)
12         farm pprintf("Rama if, $: %d @: %d\n", $, @);
13       else
14         farm pprintf("Rama else, $: %d @: %d\n", $, @);
15     }
16     else farm pprintf("No hace nada\n");
17   }
18 }
```

Figura 3.16 Un grupo inicial de procesadores se divide varias veces

restantes ejecutan la parte correspondiente al *else*. El grupo actual (padre) se desactiva y su espacio de direcciones compartidas disponible es dividido entre los nuevos subgrupos. Cada uno de los subgrupos establece una pila y un heap compartidos para su utilización para permitir que se declaren y alojen objetos compartidos relativos a cada subgrupo. Los subgrupos heredan sus identificadores de procesador y grupo (@ y *$*) del grupo padre, pero pueden redefinirlos localmente. Cuando ambos subgrupos han finalizado la ejecución de la sentencia condicional, se liberan y se reactiva el grupo padre mediante una sincronización explícita de todos sus procesadores. En el programa de la Figura 3.16, la sentencia condicional de la línea 8 divide el conjunto de procesadores que la ejecute en dos subconjuntos: aquellos con identificador menor que 4 y el resto. Los procesadores con identificador menor que 4 vuelven a ser divididos en dos subgrupos por la sentencia condicional de la línea 11, puesto que ésta también depende de una variable privada. La Figura 3.17 muestra el resultado de la ejecución del programa con 8 procesadores, y la Figura 3.18 muestra la jerarquía de grupos en el instante en que se ejecutan las dos alternativas correspondientes al *if* de la línea 11. En la Figura 3.18, los grupos que aparecen sombreados están inactivos en el instante de la computación que estamos considerando.

```
#0004# No hace nada
#0005# No hace nada
#0006# No hace nada
#0007# No hace nada
#0000# Rama if, $: 50 @: 23
#0001# Rama else, $: 49 @: 23
#0002# Rama else, $: 48 @: 23
#0003# Rama else, $: 47 @: 23
```

Figura 3.17 Ejecución del programa de la Figura 3.16

Un efecto similar tiene la ejecución de una sentencia de repetición si tiene condiciones privadas de salida. Supongamos que los procesadores de un grupo hoja, G alcanzan en modo síncrono una sentencia de repetición. Todos los procesadores que han de ejecutar la primera iteración de la sentencia formarán un grupo G' y se mantienen pertenecientes a G' mientras estén dentro del bucle de repetición. Una vez que la condición del bucle haya sido evaluada a cero por algún procesador de G', ese procesador abandona el grupo G' y espera al final de la sentencia de repetición para sincronizarse con todos los procesadores del grupo original, G. Dado este comportamiento, en la implementación de las sentencias de repetición no es necesario dividir la memoria compartida de G dado que los procesadores que abandonan la ejecución del bucle, simplemente esperan en una sincronización a que finalice el bucle el resto de procesadores.

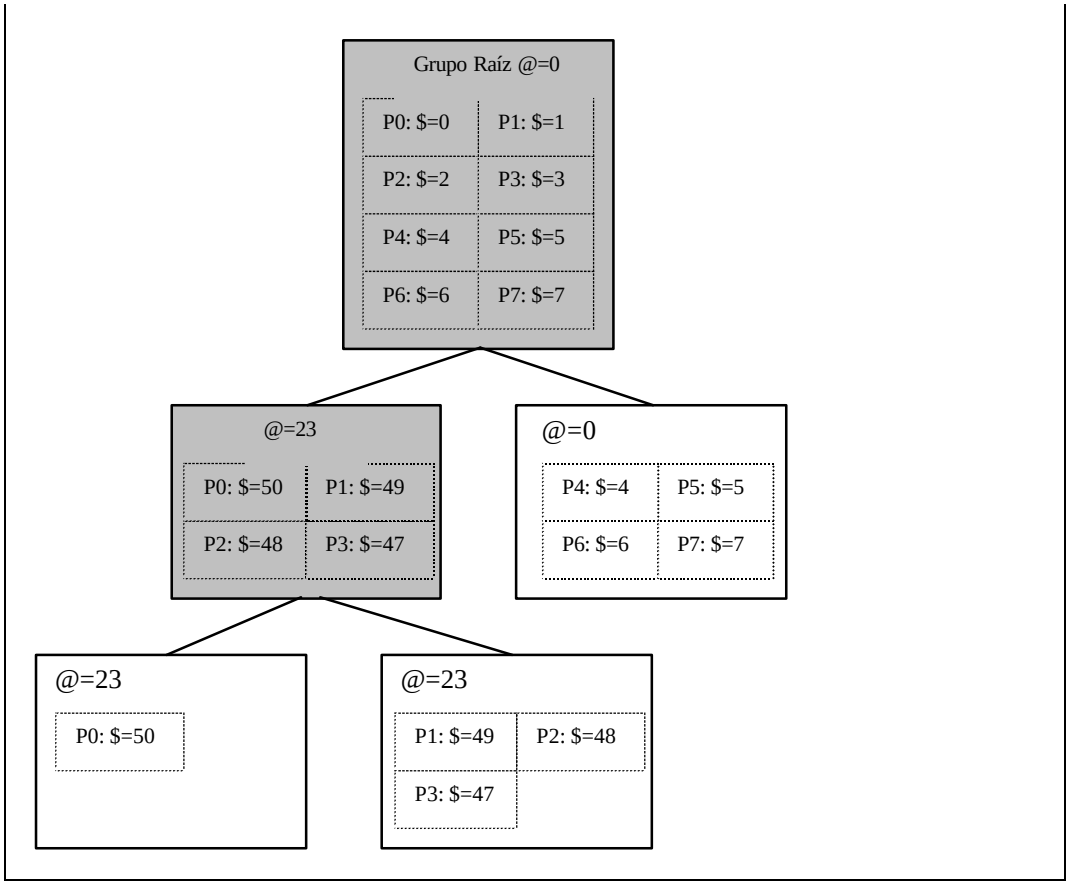

Figura 3.18 La jerarquía de grupos correspondiente a la ejecución del programa de la Figura 3.16

El siguiente código:

```
sh int p = groupsize();
pr int i;
for (i = $; i < n; i += p)
  a[i] += c * b[i];
```

ilustra la situación que hemos expuesto: los procesadores van abandonando el grupo que se crea para la ejecución del bucle for conforme finalizan la ejecución del bucle (cada procesador recorre un número diferente de iteraciones). Este típico bucle de paralelismo de datos se puede escribir como:

```
p = groupsize();
forall(i, 0, n, p) a[i] += c * b[i];
```

porque en el fichero *fork.h* está definida la macro:

```
#define forall(i,inf,sup,p)
     for (i=$+(inf); i<(sup); i+= (p))
```

que ejecuta su cuerpo con variable de bucle *i*, variando en el rango *[inf, sup)* utilizando todos los procesadores *p* del grupo hoja.

La división de grupos puede forzarse explícitamente en Fork95 mediante la sentencia fork (que da nombre al lenguaje). La ejecución de:

```
fork(e1; @=e2; $=e3) <sentencia>
```

significa lo siguiente: en primer lugar, se evalúa la expresión compartida e_1, que indica el número de subgrupos a crear, y el grupo hoja que ha ejecutado la sentencia se divide en e_1 subgrupos. Cada procesador evalúa e_2 para determinar el número del grupo hoja recién creado al que pertenece. El valor de e_3 indica para cada procesador el nuevo valor de su identificador dentro del subgrupo al que pertenece. El subespacio de memoria compartida del grupo padre se subdivide en tantas partes iguales como subgrupos se han creado y asigna una de ellas a cada subgrupo de forma que cada subgrupo disponga de su propia zona de memoria compartida. A continuación, cada subgrupo ejecuta <*sentencia*> de forma que los procesadores dentro de cada subgrupo ejecutan síncronamente, pero diferentes subgrupos pueden ejecutar diferentes flujos de control. Una vez que se finaliza la ejecución de <*sentencia*>, los procesadores de todos los subgrupos se sincronizan, las zonas de memoria compartida se mezclan, el grupo padre se reactiva como el grupo hoja en curso y todos los procesadores ejecutan síncronamente la siguiente sentencia que siga a la sentencia *fork*. Es posible que una sentencia *fork* cree subgrupos vacíos, aunque en ese caso, el trabajo de un grupo vacío finaliza inmediatamente. Al disponer de libre acceso a los nombres físicos en la sentencia *fork*, es el programador quien diseña las aplicaciones *Map* y *F* de asignación de tareas a procesadores que estudiamos en el Capítulo 2.

SENTENCIA	MODO	TIPO	PARÁMETROS	SIGNIFICADO
`groupsize`	sync	`int`	void	Retorna el número de procs. en el grupo
`async_groupsize`	async	`int`	void	Igual que groupsize pero para llamadas desde zonas asíncronas
`parentgroupsize`	sync	`int`	void	Retorna el número de procs. en el grupo padre.

Tabla 3.4 Inspección de la estructura de grupo

La Tabla 3.4 muestra las sentencias de Fork95 que permiten inspeccionar la estructura del grupo al que un procesador pertenece.

3.3.6. Ejecución síncrona y asíncrona. La sentencia `join`

El modo de ejecución síncrono elimina la necesidad de proteger variables compartidas mediante semáforos o cerrojos porque son accedidas de forma determinista: el programador puede confiar en un tiempo fijo de ejecución para cada operación, que es el mismo en todos los procesadores durante la ejecución del programa. No es necesario tomar precauciones adicionales para evitar comportamientos no deterministas. Si consideramos el código de la Figura 3.19 en el que la mitad de los procesadores ejecutarán la rama *if* de la sentencia condicional de la línea 9 y la otra mitad la rama del *else*, tenemos que no está determinado si la variable *y* acabará almacenando el valor antiguo o el nuevo de *a*, pero la semántica síncrona garantiza que

todos los procesadores que escriben su variable privada *y* en la rama *else* del condicional le asignarán *el mismo* valor de *a*. No será necesario garantizar este hecho mediante una exclusión mutua.

El modo de ejecución estrictamente síncrono se corresponde con el estilo de programación PRAM, en la que el programador no ha de preocuparse por comportamientos no deterministas puesto que cada operación conlleva el mismo tiempo para todos los procesadores.

En las zonas de código asíncronas no existen puntos de sincronización implícitos. El mantener el invariante de sincronismo requiere un coste significativo, incluso para los casos en los que el grupo de procesadores que ejecuta el código contiene un único procesador, o cuando es innecesario el invariante de sincronismo porque no hay dependencias de datos. El convertir esas regiones de código en asíncronas conlleva un incremento sustancial en la velocidad de ejecución de los programas. Los autores de Fork95 indican que a través de la utilización de *farm* y funciones asíncronas consiguieron reducir los tiempos de ejecución (siempre medidos con el simulador *pramsim*) en un 50% [Kes97b].

La sentencia *join*, recientemente introducida en el lenguaje [Kes97b], es una generalización de la sentencia *start* que permite cambiar de modo de ejecución asíncrono a síncrono más fácilmente, así como el anidar arbitrariamente zonas de código síncronas y asíncronas. La sentencia tiene la siguiente sintaxis:

```
join(sentencia_retardo; condición_salida; SM_size)
  <sentencia>
else trabajo_util()
```

```
 1 sh int a = 1, x = 0;
 2 pr int y = 7;
 3 sh simple_lock lock;
 4
 5 void main(void) {
 6   simple_lock_init(lock);
 7   start {
 8     seq printf("%d Procesadores ejecutando\n", __STARTED_PROCS__);
 9     if (__PROC_NR__ % 2 == 0)
10       a = x;
11     else
12       y = a;
13   }
14   simple_lockup(&lock);
15   printf("Proc %d a: %d y: %d\n", __PROC_NR__, a, y);
16   simple_unlock(&lock);
17   barrier;
18 }
```

Figura 3.19 El código entre las líneas 9 y 15 es síncrono

La sentencia que aparece en el cuerpo de un *join* es una zona de código síncrono. En la sentencia *join*, varios procesadores se unen formando un grupo síncrono que ejecutan el cuerpo del *join*. Los procesadores que se unen en el *join*, pueden proceder de diferentes grupos hoja. El primer argumento de la sentencia *join*, la *sentencia_retardo* especifica una sentencia que es ejecutada por un procesador distinguido de los que forma el grupo del *join*, y modeliza una condición o un intervalo de tiempo que debe cumplirse para iniciar la ejecución de *sentencia*. La *condición_salida* es una expresión booleana suministrada por el programador, que puede ser diferente para diferentes procesadores y representa la condición que debe cumplirse para que un procesador abandone el grupo

síncrono. El tercer parámetro de la sentencia, *SM_size* especifica el tamaño de memoria compartida que es alojada para disponer de una pila y heaps compartidos por los miembros del grupo que ejecutan el *join*. Esta memoria se libera cuando finaliza el cuerpo del *join*. La parte *else* de la sentencia *join* es opcional y especifica una sentencia asíncrona, *trabajo_util()* que es ejecutada por los procesadores que llegan tarde a la sentencia *join* (la ejecución del *join* ya se ha iniciado cuando un procesador la alcanza) y también por los procesadores que han abandonado el grupo del *join*. Una sentencia *retry* dentro del cuerpo de *trabajo_util()*, provoca que los procesadores que la ejecutan traten de ganar acceso al grupo síncrono del *join*. La sentencia *retry* funciona de forma similar a *continue* en los bucles.

En un programa no puede haber más de una sentencia *join*. Si el programador lo desea, puede encapsular la llamada dentro de una función e invocar la función desde diferentes puntos del código. A nivel de implementación hay una serie de variables de las que el programador podría hacer uso según sus intereses.

La semántica de la sentencia *join* se explica bien con una analogía que utiliza un autobús: los procesadores que se unen a un *join* son pasajeros que tratan de acceder a un autobús. Los pasajeros del autobús forman un grupo síncrono durante la ejecución del cuerpo del *join* (duración del trayecto). Los pasajeros han de acceder y salir del autobús en puntos concretos del recorrido (paradas del autobús), y si no lo consiguen, pueden volver a intentarlo la próxima ocasión que el autobús pase de nuevo por la parada (sentencia *retry*) pero mientras esperan, pueden también realizar algún trabajo alternativo (*trabajo_util()*).

SENTENCIA	SIGNIFICADO	EJEMPLO	MODO	CUERPO
`start`	Todos los procesadores entran en modo síncrono	`start{v[$]=$;}`	asíncrono	síncrono
`farm`	Entrada en modo asíncrono	`farm puts("Hola");`	síncrono	asíncrono
`fork`	Divide el grupo actual en subgrupos	`fork(4;@=@%4;$=$/4)...`	síncrono	síncrono
`seq`	Un procesador entra en modo asíncrono	`seq printf(...);`	síncrono	asíncrono
`join`	Forma un nuevo grupo raíz	`join(wait();0;1000)...`	asíncrono	síncrono
`retry`	Volver a intentar el 'join'	`else{otra(); retry;}`	asíncrono	-
`barrier`	Sincronización por barrera local al grupo	`barrier;`	asíncrono	-

Tabla 3.5 Sentencias

La implementación disponible actualmente de la sentencia `join` en Fork95 es una versión algo restringida que se realiza a través de una macro:

```
join(joinID, sentencia_retardo, condicion_salida,
     SM_size, reintentar, sentencia, trabajo_util);
```

en la macro, *reintentar* es un entero que indica si el procesador que ejecuta la macro ha de reintentar acceder al *join* si no lo consigue en primera instancia o bien si lo ha abandonado, y el primer argumento es un entero distinto para cada *join* que aparezca en el programa (en nuestra analogía sería el identificador del autobús).

Los autores utilizan las sentencia *join* para implementar lo que denominan secciones críticas síncronas paralelas como contraposición a las secciones críticas asíncronas secuenciales [Kes95b], no obstante no exploraremos en profundidad esta posibilidad dado que se aparta del modelo de programación PRAM.

La Tabla 3.5 resume las sentencias más importantes de Fork95 desde el punto de vista del paralelismo, indicando el modo de ejecución de la misma y su cuerpo, y un ejemplo de utilización de cada una.

3.3.7. Punteros y heaps

Al contrario que su antecesor, FORK, Fork95 ofrece al programador punteros. El uso de punteros es análogo al que de ellos hace C. No es preciso distinguir entre

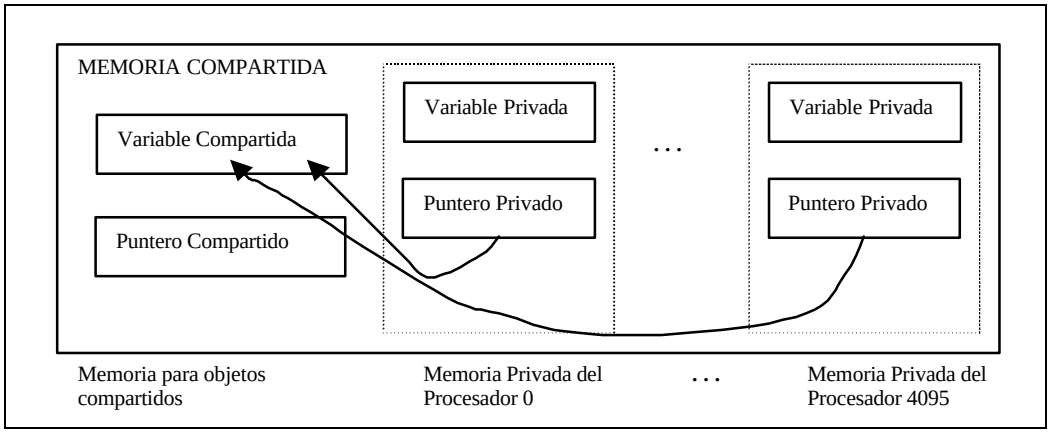

Figura 3.20 Diferentes punteros privados apuntando a la misma variable compartida

punteros a variables compartidas y privadas sino que las variables de tipo puntero compartidas pueden apuntar a variables privadas y viceversa. Es responsabilidad del programador establecer las relaciones convenientes en cada momento. La declaración

```
sh int *puntero_compartido;
```

corresponde a un puntero que puede apuntar a memoria compartida o privada.

El fragmento de código:

```
pr int privada, *puntero_privado;
sh int compartida;
puntero_privado = &compartida;
```

hace que las variables puntero (privados) *puntero_privado* de los procesadores que ejecuten la asignación apunten todas a una variable compartida (ver Figura 3.20). De forma análoga, si todos los procesadores ejecutan en paralelo la siguiente asignación:

```
puntero_compartido = &privada;
```

se produce una escritura concurrente, y de acuerdo a la semántica que hemos expuesto, la variable puntero (compartida) apuntará a la variable privada del procesador con mayor índice que participe en la asignación.

Las variables privadas pueden hacerse accesibles globalmente a través de los punteros, puesto que una variable puntero compartido que apunte a una variable privada hace que ésta sea accesible (a través del puntero) a todos los procesadores que declararon el puntero compartido (ver Figura 3.21).

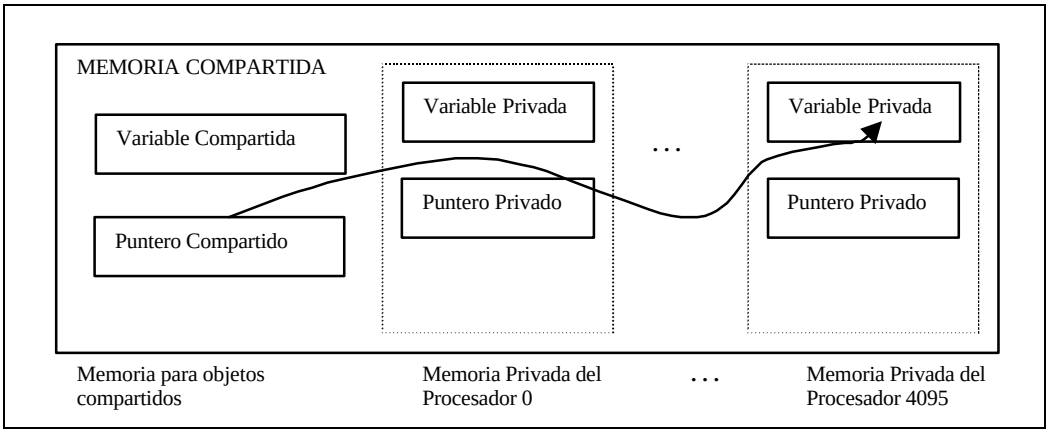

Figura 3.21 Una variable privada accesible a todos los procesadores a través de un puntero compartido

Fork95 dispone de tres tipos de heaps para el almacenamiento de objetos dinámicos: un heap privado para cada procesador, un heap compartido automático para cada grupo y un heap global compartido y permanente. Para alojar y liberar memoria en los heaps privados se utilizan las funciones (asíncronas) habituales de C, *malloc()* y *free()*. La memoria del heap compartido permanente se maneja utilizando las funciones asíncronas *shmalloc()* y *shfree()*. El heap automático compartido está pensado como una forma de suministrar memoria rápida y temporal a los procesadores de un grupo. El tiempo de vida de las entidades alojadas en el heap automático compartido mediante la función síncrona *shalloc()* está limitado al tiempo de vida del grupo de procesadores que ejecutó la función. Estas entidades se eliminan de la memoria si el grupo que las alojó finaliza su existencia. La función síncrona *shallfree()* libera todas las entidades dinámicas alojadas mediante *shalloc()* en la función (síncrona) que se esté ejecutando.

FUNCION	MODO	TIPO	PARÁMETROS	SIGNIFICADO
`malloc` `free`	`async` `async`	`char *` `void`	`pr uint` `pr char *`	Aloja memoria en el heap privado Libera memoria del heap privado
`shmalloc` `shfree`	`async` `async`	`char *` `void`	`pr uint` `pr char *`	Aloja memoria en el heap global compartido Libera memoria del heap global compartido

`shalloc`
`shavail`
`shallfree`
	`sync`
	`async`
	`sync`
		`char *`
		`uint`
		`void`
			`shared uint`
			`void`
			`void`
				Aloja memoria del heap compartido automático
				Tamaño libre del heap compartido automático
				Libera todos los bloques alojados con `shalloc`

Tabla 3.6 Rutinas para el manejo de memoria dinámica

Fork95 también permite punteros a funciones. Dado que a través de un puntero privado a función, cada procesador podría invocar síncronamente a una función

diferente (y es imposible conocer estáticamente a cuál), las llamadas a funciones síncronas a través de un puntero privado son automáticamente convertidas por el compilador en llamadas asíncronas. Los punteros privados sólo pueden apuntar a funciones asíncronas.

La Tabla 3.6 resume las funciones de Fork95 para manejar los diferentes heaps disponibles. En esa tabla, *uint* es una abreviatura para *unsigned int*.

3.3.8. Precauciones a la hora de programar en Fork95

Si el programador coloca muchas sentencias condicionales anidadas con condiciones privadas, ello provocará sucesivas divisiones implícitas del grupo de procesadores hoja, con la consiguiente ineficiencia producida por el hecho de que hay que dividir la memoria compartida disponible entre los nuevos subgrupos creados.

La situación se hace especialmente crítica si las condiciones forman parte de una función que se invoca recursivamente: al cabo de unas pocas llamadas recursivas, la fragmentación que se produce en la memoria compartida hará que los bloques disponibles sean de un tamaño tan pequeño que a efectos prácticos, serán inservibles. Es responsabilidad del programador el evitar esta forma de programación.

Una buena práctica al programar en Fork95 si se pretende producir programas eficientes, consiste en conmutar a ejecución asíncrona desde el momento en que se sabe que todos los grupos hoja de la jerarquía de grupo contienen un único procesador. El pasar a modo asíncrono evita la sobrecarga asociada con la creación de subgrupos y la fragmentación de la memoria compartida.

```
 1 #include <fork.h>
 2 #include <io.h>
 3
 4 sync void output_array( sh int*, sh int );
 5 sync void prefix_sum( sh int*, sh int, sh int *, sh int );
 6
 7 sh int *a, *b;
 8 sh int n = 100;
 9
10 sync void prefix_sum(sh int *in, sh int n, sh int *out,
11                       sh int initsum) {
12   pr int i;
13
14   for (i=$; i<n; i+=p)
15     out[i] = mpadd(&sum, in[i]);
16 }
17
18 void main(void) {
19   pr int i;
20   start {
21     a = (int *) shalloc(n);
22     b = (int *) shalloc(n);
23     seq  prS("Vector inicial:\n");
24     farm for (i=$; i<n; i+= __STARTED_PROCS__)
25       a[i] = 1;
26     output_array(a, n);
27     prefix_sum(a, n, b, 0);
28     seq  prS("Vector de salida:\n");
29     output_array(b, n);
30   }
31 }
```

Figura 3.22 La suma de prefijos en Fork95

3.3.9. Algunos ejemplos

3.3.9.1. La suma de prefijos

```
Relocating file prefix_sum... done
Loading file /var/tmp/aaaa002M1.cod...Doing realloc
Doing realloc
 done
PRAM P0 = (p0, v0)> g
Vector inicial:
1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1

Vector de salida:
0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
16  17  18  19  20  21  22  23 24

EXIT: vp=#0, pc=$00000377
EXIT: vp=#1, pc=$00000377
EXIT: vp=#2, pc=$00000377
EXIT: vp=#3, pc=$00000377
Stop nach 74366 Runden, 1239.433 kIps
0377 18137FFF  POPNG   R6, ffffffff, R1
```

Figura 3.23 Ejecución del programa de la Figura 3.22

Basándonos en la instrucción mpadd(), la implementación de la suma de prefijos en Fork95 resulta tan trivial como el código que presentamos en la Figura 3.22. La función *main()* de la línea 18 aloja memoria del heap compartido para los vectores de entrada y de salida. El bucle *for* de la línea 24 inicializa el vector de entrada con un uno en cada componente mediante una ejecución asíncrona a través de la sentencia *farm*, e invoca en la línea 27 a la función *prefix_sum()*. La función toma como argumentos el vector de entrada, el número de componentes, el vector de salida, y un valor inicial, *initsum*, que se sumaría a todas las componentes del vector de salida (que en nuestro ejemplo inicializaremos a cero). La función divide el vector de entrada en N/p segmentos y el bucle de la línea 14 recorre todos los segmentos asignando un procesador a cada componente del segmento. Cada procesador, mediante la operación *mpadd()* de la línea 15 calcula la suma de prefijos de su componente en el vector de salida. El primer argumento de *mpadd()* va 'transmitiendo' la suma acumulada de cada uno de los segmentos hacia el siguiente. La función *prefix_sum()* supone que los identificadores *$* de los procesadores son números consecutivos en el rango [0..groupsize()-1]. En el código de la Figura 3.22 esto es cierto por el contexto en que la función es invocada, pero si no lo fuera, se podría conseguir fácilmente mediante otra llamada a *mpadd()*. La Figura 3.23 presenta el resultado de la ejecución del programa, utilizando un vector de 25 componentes y 4 procesadores.

3.3.9.2. El Quicksort

El programa que presentamos en la Figura 3.24 es una implementación en Fork95

```
 1 sync void qs(sh int *array, sh int n , sh int p, sh int *temparray) {
 2 sh int lowerprocs;           /* No. de procs. para los subgrupos */
 3 sh int lowersize, uppersize; /* Tamaño de los vectores lower[], upper[] */
 4 sh int equalsize;            /* No. de elementos iguales al pivot */
 5 sh int *lower, *upper;       /* Subvectores a ordenar recursivamente */
 6 sh int *equal, l, e, u;
 7 sh int pivot = 0;
 8 sh int aux = 0;
 9 pr int j, k, mygroup;
10
11   if ($ >= n) return;    /* Nunca se necesitan más de n procesadores */
12   farm if ($ == 0) pprintf(" qs(%d,%d)\n", n, p);
13   if (n <= 1)  return;                                        /* trivial */
14   if (n == 2)  { sort2(array[0], array[1], pivot); return; }  /* simple */
15   if (p == 1)  { farm qsort(array, n, 1, cmp); return; }      /* secuencial */
16   renumber(aux);                                 /* Poner $ con valores de 0 a p-1 */
17   lowersize = uppersize = equalsize = 0;
18   lowerprocs = 0;
19   pivot = array[0];         /* Elegir un pivote */
20
21   /* En paralelo, hallar los tamaños de los subvectores lower[], equal[], upper[]: */
22   farm
23     for (j = $; j < n; j += p) /* Recorrer el vector en paralelo */
24       if (array[j] < pivot) mpadd(&lowersize, 1);
25       else
26       if (array[j] > pivot) mpadd(&uppersize, 1);
27       else                  mpadd(&equalsize, 1);
28
29   /* Asignar memoria a los subvectores lower[] y upper[] en temparray[]: */
30   lower = temparray;
31   equal = lower + lowersize;
32   upper = equal + equalsize;
33
34   /* En paralelo, copiar los elementos del vector hacia lower[]/equal[]/upper[] */
35   l = e = u = 0;
36   farm
37     for (j = $; j < n; j += p)   /* Recorrer el vector en paralelo */
38       if (array[j] < pivot)  { k = mpadd(&l, 1); lower[k] = array[j]; }
39       else
40       if (array[j] > pivot)  { k = mpadd(&u, 1); upper[k] = array[j]; }
41       else                   { k = mpadd(&e, 1); equal[k] = array[j]; }
42
43   farm
44     for (j = $; j < n; j += p)
45       array[j] = temparray[j];    /* Copiar de nuevo temparray en array */
46   /* Ahora temparray[] se puede usar como temporal en llamadas recursivas */
47
48   if (lowersize > 1 && uppersize > 1) {   /* Caso general */
49     /* Subdividir los p procesadores en relación al tamaño de los subvectores */
50     farm lowerprocs = (int)((float)(lowersize*p)/(float)(lowersize+uppersize));
51     if (lowerprocs == 0) lowerprocs = 1;    /* corrección */
52     if (lowerprocs == p) lowerprocs = p-1;  /* corrección */
53     farm mygroup = ($ < lowerprocs) ? 0 : 1;
54     fork (2; @=mygroup; $=$) {
55       if (@ == 0) qs(array, lowersize, lowerprocs, temparray);
56       else      qs(array + lowersize + equalsize,
57                    uppersize, p - lowerprocs, temparray + lowersize + equalsize);
58     }
59   }
60   else
61   if (lowersize > 1)  qs(array, lowersize, p, temparray);
62   else
63   if (uppersize > 1)  qs(array + lowersize + equalsize,
64                       uppersize, p, temparray + lowersize + equalsize);
65   /* else nada que hacer; */
66 }
```

Figura 3.24 El Quicksort en Fork95

del algoritmo de ordenación Quicksort [Hoa61] que ya presentamos en el Capítulo 2. Para ordenar un vector, el algoritmo elige un elemento arbitrario del vector como

elemento pivote y subdivide el vector en tres subvectores conteniendo los elementos menores, iguales y mayores que el pivote. A continuación procede recursivamente ordenando los subvectores que contienen los elementos menores y mayores que el pivote.

Estudiemos la implementación de la Figura 3.24 con más detalle: la función qs() es una función síncrona (línea 1) que toma como parámetros un puntero al vector a ordenar, *array*, el número de elementos del vector, *n*, el número de procesadores que se van a utilizar en la ordenación, *p* y un vector auxiliar, *temparray*, del mismo tamaño que el vector a ordenar. En el programa principal, la función se invocaría con *__STARTED_PROCS__* como tercer parámetro, indicando el número de procesadores a utilizar. En las líneas 13-15, el código trata los casos en que el número de elementos a ordenar es uno (en cuyo caso no hay nada que hacer), dos (en cuyo caso, si no están ordenados, se intercambian mediante una llamada a la macro *sort2* definida por el usuario) y el caso al que en general se debiera llegar en último término, en el que el número de procesadores a utilizar es uno, en cuyo caso se procede a ordenar el vector secuencialmente mediante una llamada a la función estándar *qsort* definida en *stdlib.h*. Si no nos encontramos ante ninguno de estos casos (es decir, hay más de un procesador disponible y el número de elementos es mayor que dos) el algoritmo asigna valores entre *0* y *p-1* a los *p* procesadores disponibles, mediante la llamada de la línea 16 a la macro definida por el usuario *renumber()* que utiliza una llamada a *mpadd()*. En la línea 19 se elige el primer elemento del vector a ordenar como elemento pivote (se podría elegir cualquier elemento) y en las líneas 22-27 el código halla en paralelo el número de elementos menores, iguales y mayores que el pivote elegido. Para ello se recorre todo el vector en bloques de tamaño igual al número de procesadores *p* utilizados asignando un procesador a cada posición del subvector considerado. Mediante las sentencias *mpadd()*de las líneas 24, 26 y 27 los procesadores incrementan en paralelo la variable correspondiente (*lowersize*, *uppersize* o *equalsize*) al caso del elemento considerado.

```
 1 #include <fork.h>
 2 #include <assert.h>
 3 #include <io.h>
 4 #define Nmax 8
 5 sync void quicksort( sh int *);
 6 sh int a[Nmax] = {4, 5, 7, 2, 3, 5, 6,
1};
 7 pr int x, pos = 0;
 8 main() {
 9    start (Nmax) {
10       sh int numElem;
11       x = a[$];
12       quicksort( &numElemt );
13       a[pos-1] = x;
14    }
15 }
```

Figura 3.25 El programa principal para el Quicksort de la Figura 3.26

Una vez conocido el número de elementos menores, mayores e iguales al pivote, en las líneas 30-32 se asignan direcciones dentro del vector temporal *temparray* a los vectores *lower*, *equal* y *upper* que van a contener los elementos menores, iguales y mayores al pivote. Los elementos del vector *array* son copiados hacia su array correspondiente mediante el código de las líneas 35-41. Ese código utiliza una técnica similar: se recorre el array en paralelo mediante el típico bucle de paralelismo de datos y cada procesador se encarga de analizar una posición del vector fuente y de actualizar la correspondiente posición del vector de destino. La componente del vector de destino que ha de actualizar

un determinado procesador se calcula mediante las sumas de prefijos que se calculan en las líneas 38, 40 y 41. El código de las líneas 43-45 copia el vector *temparray* de nuevo en su vector original, *array*. De este modo el vector temporal podrá ser reutilizado en las llamadas recursivas. El código de las líneas 48-59 trata el caso general en que hay elementos menores y mayores que el pivote (los casos particulares en que sólo hay elementos menores o sólo mayores se trata en el código de las líneas 61-64). En este caso general se divide el grupo de procesadores disponibles (recordemos que hay más de un procesador en este grupo) en dos subgrupos de forma que el número de procesadores asignados a cada subgrupo sea proporcional al tamaño del vector a ordenar. En el código de las líneas 50-52 se calcula el número de procesadores que se asignará al subgrupo que ordenará el subvector de menores. La llamada a *fork* de la línea 54 divide explícitamente en dos el grupo de procesadores, preservando para los nuevos grupos creados el identificador lógico de procesador y cambiando su identificador de grupo a 0 o 1 en función del identificador del procesador (línea 53). Los procesadores del grupo 0 ordenan recursivamente el vector de elementos menores que el pivote (línea 55) mientras que los del grupo 1 ordenan el vector de elementos mayores

```
 1  sync void quicksort( sh int *numElement) {
 2  sh int pivot, numLeft = 0, numRight = 0, totalPivots = 0;
 3  pr int left, right, pivotPos;
 4   pivot = x;
 5   left =  (x < pivot);
 6   right = (x > pivot);
 7   if (x == pivot) pivotPos  = mpadd(&totalPivots,1);
 8   else if (left)  quicksort( &numLeft );
 9   else quicksort( &numRight );
10   if (x == pivot)  pos = numLeft + pivotPos;
11   if (right) pos += totalPivots + numLeft;
12   *numElement = numLeft + numRight + totalPivots;
13  }
```

Figura 3.26 Otra implementación del Quicksort en *fork95*

que el pivote (líneas 56-57).

3.3.9.3. Otra implementación del Quicksort

Los códigos *fork95* de la Figura 3.25 y de la Figura 3.26 presentan otra elegante implementación del Quicksort en Fork95 basada en la creación implícita de grupos a través de la sentencia condicional. Cada uno de los procesadores copia mediante la asignación de la línea 11 (Figura 3.25) un elemento distinto del vector compartido *a* en la variable privada *x*. En la línea 4 de la Figura 3.26 los procesadores realizan una escritura concurrente en la variable compartida *pivot*. Uno de ellos, que podría ser elegido de manera no determinista, el programador no sabe cuál, logrará que su valor de *x = a[$]* sea escrito. Como consecuencia de la semántica de la sentencia condicional, las sentencias condicionales de las líneas 7, 8 y 9 dividen el grupo en 3 grupos. Uno formado por los procesadores cuya *x = a[$]* es menor que el pivote, otro formado con aquellos cuya *x* es mayor que el pivote y otro con los procesadores con *x = pivot*. Los dos primeros grupos proceden recursivamente a computar en la variable privada *pos* la posición del elemento *$* relativa al subvector ordenado. El tercer grupo realiza una suma de prefijos mediante la llamada a la función *mpadd*. Como consecuencia de esta llamada, en *pivotPos* queda la posición relativa del procesador dentro del grupo y en *totalPivots* el número de elementos en el subvector iguales al pivote. La posición relativa al subgrupo actual de los elementos iguales al pivote es computada en la línea 10. Para los elementos de la parte derecha es computada en la línea 11. Por último, la

asignación de la línea 13 en la Figura 3.25 coloca cada elemento en la posición que le corresponde en el vector ordenado.

3.3.9.4. Mergesort

```
 1 sync void mergesort(int *array, int n, int *sortedarray) {
 2   sh int p = groupsize();
 3
 4   seq pprintf(" mergesort(array=%p, n=%d, p=%d)\n", array,n,p);
 5   if (n <= THRESHOLD || p == 1)
 6     seq seq_sort(array, n, sortedarray);
 7   else {
 8     sh int *temp = (int *)shalloc(n);
 9     fork (2; @= $ % 2; $ = $ / 2)
10        mergesort(array+@*(n/2),(1-@)*(n/2)+@*(n-n/2),temp+@*(n/2));
11     merge(temp, n/2, temp+n/2, n-n/2, sortedarray);
12   }
13 }
```

Figura 3.27 El mergesort en Fork95

La función *mergesort* que presentamos en la Figura 3.27 toma como parámetros un puntero a un vector de entrada, *array*, su tamaño, *n*, y otro puntero a un vector de salida ordenado, *sortedarray*. La función asume que la memoria para el vector de salida ha sido alojada antes de la llamada. La estrategia del programa consiste en dividir el

```
 1 sync void merge(int *src1, int n1, int *src2, int n2, int *dest) {
 2   sh int p = groupsize();
 3   sh int iter;
 4   sh int *pos12, *pos21;  /* Vectores temporales */
 5   pr int i;
 6
 7   farm assert(p > 1);
 8   pos12 = (int *)shalloc(n1);
 9   pos21 = (int *)shalloc(n2);
10   iter = 0;
11   farm
12     for (i = mpadd(&iter, 1); i < n1; i = mpadd(&iter, 1))
13       pos12[i] = get_pos(src1[i], src2, n2);
14   iter = 0;
15   farm
16     for (i = mpadd(&iter, 1); i < n2; i = mpadd(&iter, 1))
17       pos21[i] = get_pos(src2[i], src1, n1);
18   farm {
19     /* Copiar hacia dest usando la informaci¢n de posicion */
20     for (i=$; i<n1; i+=p)  dest[i+pos12[i]] = src1[i];
21     for (i=$; i<n2; i+=p)  dest[i+pos21[i]] = src2[i];
22   }
23 }
```

Figura 3.28 La función merge

vector a ordenar en dos mitades de igual tamaño y ordenar recursivamente cada una de ellas dividiendo el grupo de procesadores disponibles en dos subgrupos (sentencia fork de la línea 9), cada uno de los cuales se encarga de una de las dos mitades. Ambos subgrupos de procesadores realizan la misma llamada recursiva a *mergesort*, pero utilizan datos de entrada y de salida diferentes porque los parámetros de la llamada se calculan en función de @, el identificador del grupo que hace la llamada.

Cuando el tamaño del vector a ordenar es suficientemente pequeño o bien el número de procesadores disponibles es sólo 1 (línea 5) se invoca a la función *seq_sort*, que ordena el vector secuencialmente. La función *merge* que se llama en la línea 11 se encarga de mezclar en paralelo los dos subvectores ordenados dejando el resultado almacenado en el vector *sortedarray.*

```
***** PRAM-SIMULATOR (v2.0) *****
          >>>>>>>> DAMN FAST <<<<<<<<
       (c) by hirbli & stefran  07.10.94
You have 5 physical processors with 1 vP's each

Relocating file mergesort... done
Loading file /var/tmp/baaa002o5.cod...Doing realloc
Doing realloc
 done
PRAM P0 = (p0, v0)> g
Enter N = 32
#0000#  mergesort(array=620cb, n=32, p=5)
#0000#  mergesort(array=620cb, n=16, p=3)
#0001#  mergesort(array=620db, n=16, p=2)
#0000#  mergesort(array=620cb, n=8, p=2)
#0001#  mergesort(array=620db, n=8, p=1)
#0002#  mergesort(array=620d3, n=8, p=1)
#0003#  mergesort(array=620e3, n=8, p=1)
#0000#  mergesort(array=620cb, n=4, p=1)
#0004#  mergesort(array=620cf, n=4, p=1)
#0000# Array 620ab of size 32:
 -932 -812 -799 -691 -688 -656 -555 -539 -523 -437 -391 -366 -201 -182
-171 -151 -132 -51 -2 38 198 203 291 297 368 395 673 694 703 855 926
931
```

Figura 3.29 Resultado de la ejecución del mergesort con 5 procesadores

El código de la función *merge* aparece en la Figura 3.28. La función mezcla los dos vectores ordenados *src1* y *src2* de tamaños *n1* y *n2* respectivamente en el vector *dest*. La función utiliza dos vectores auxiliares, *pos12* y *pos21* para calcular en ellos la posición de cada elemento de cada uno de los dos vectores en el otro vector (*pos12[i]* almacenará el número de elementos de *src2* que son menores que el elemento *src1[i]*). A partir de esta información, los bucles de las líneas 20 y 21 de la Figura 3.28 colocan cada uno de los elementos de *src1* y *src2* en su posición definitiva en *dest*. El cálculo de los valores de los vectores *pos12* y *pos21* se realiza mediante una búsqueda binaria secuencial, a través de la función *get_pos()*. Los bucles de las líneas 12 y 16 utilizan la sentencia *mpadd()* para paralelizar el cómputo de los vectores pos. La inicialización e incremento de la variable del bucle hace que cada procesador calcule la posición de un elemento distinto.

La implementación que presentamos del mergesort asume que todos los elementos a ordenar son distintos y la Figura 3.29 presenta el resultado de la ejecución del programa utilizando 5 procesadores con un vector de 32 elementos generado aleatoriamente. Vemos que en la primera división del vector, el grupo de 5 procesadores se divide en dos subgrupos con 3 y 2 procesadores, y posteriormente el grupo con 3 procesadores se dividirá en dos subgrupos con 2 y 1. Los procesadores individuales finalmente ordenan secuencialmente segmentos del vector original de tamaños 8, 8, 8, 4 y 4.

3.3.9.5. La FFT

Dada una serie de números $\{a_0, a_1,..., a_{n-1}\}$ su transformada discreta de Fourier es la secuencia $\{b_0, b_1,..., b_{n-1}\}$ siendo

$$b_j = \sum_{k=0}^{n-1} a_k * w^{kj} \quad para \quad j = 0,1,\cdots n-1$$

$$siendo \quad w = e^{2pi/n} = \cos\left(\frac{2p}{n}\right) + i\, sin\left(\frac{2p}{n}\right)$$

$$i = \sqrt{-1}$$

Esto es, $w^n = e^{2\pi i} = 1$ y 1, w, w^2, w^3, ..., w^{n-1} coincide con las raíces n-ésimas de la unidad en el plano complejo. Si n es una potencia de dos, $n=2^s$ para algún entero s, la expresión para los b_j Se puede reescribir como:

$$b_j = \sum_{m=0}^{2^{s-1}-1} a_{2m}\, e^{2pijm/2^{s-1}} + w^j \sum_{m=0}^{2^{s-1}-1} a_{2m+1}\, e^{2pijm/2^{s-1}}$$

```
 1 sync cplx *fft(sh cplx *a, sh int n, sh cplx *w) {
 2   sh cplx *ft;                          /* Vector para el resultado */
 3   sh cplx *even, *odd, *fteven, *ftodd;   /* Punteros temporales */
 4   sh int p = 0;
 5   sh int ndiv2;
 6   pr int i;

 7   $ = mpadd(&p, 1);  /* Asegurar numeración consecutiva de los procesadores */
 8   seq  prS("fft\n");
 9   if (n==1)  {
10     seq {
11       ft = shmalloc(1);
12       ft[0] = cnum(a[0]->re, a[0]->im);
13     }
14     return ft;
15   }
16   if (p==1) return seq_fft(a, n, w);
17   seq ft = (cplx *) shmalloc(n);  /* Memoria para el resultado */
18   ndiv2 = n >> 1;
19   even = (cplx *) shalloc(ndiv2);
20   odd  = (cplx *) shalloc(ndiv2);
21   for(i = $; i < ndiv2; i += p) { /* Bucle paralelo */
22     even[i] = a[2 * i];           /* Cálculo de componentes pares e impares */
23     odd[i]  = a[2 * i + 1];
24   }
25  if ($ < p/2)
26    fteven = fft(even, ndiv2, w);
27  else
28    ftodd = fft(odd, ndiv2, w);
29  farm
30    for(i = $; i < ndiv2; i += p) {     /* Bucle paralelo */
31      pr cplx t     = cmul(w[i], ftodd[i]);
32      ft[i]         = cadd(fteven[i], t);
33      ft[i + ndiv2] = csub(fteven[i], t);
34      freecplx(t);
35    }
36  seq shfreecplxarray(fteven, ndiv2);
37  seq shfreecplxarray(ftodd, ndiv2);
38  shallfree();  /* Liberar los vectores even[] y odd[] */
39  return ft;
40 }
```

Figura 3.30 La Transformada rápida de Fourier en Fork95

Al algoritmo que calcula la transformada discreta de Fourier mediante esta expresión se le conoce como transformada rápida de Fourier (FFT) [Akl89]. Se trata de un algoritmo recursivo, dado que en la expresión anterior, cada una de las sumas es a su vez una transformada discreta de Fourier. La primera suma corresponde a los términos pares y otra a los impares de la secuencia de entrada.

La FFT es un algoritmo de amplísimo uso en un gran espectro de campos, entre los que podemos reseñar el procesamiento digital de señales, teoría de la codificación, transmisión del habla, proceso de imágenes, predicción meteorológica, etc. De ahí la importancia del diseño de algoritmos rápidos para realizar este cómputo.

La función *fft* de la Figura 3.30 es solución recursiva para el cálculo de la transformada rápida de Fourier del vector de complejos *a* que recibe como primer parámetro. El segundo parámetro, *n* es el número de componentes del vector, que se asume que es una potencia de dos, mientras que el tercer parámetro es un vector que contiene las potencias de la raíz n-ésima de la unidad en el plano complejo, *1, w, w^2, .., w^{n-1}* siendo $w = e^{2pi/n}$.

El código entre las líneas 9-15 trata el caso trivial en que el número de componentes del vector es uno. En la línea 16, mediante la llamada a la función *seq_fft()* se resuelve el caso en que el número de procesadores disponibles es sólo uno, en cuyo caso el problema se resuelve secuencialmente. El resto del código (líneas 17-40) resuelve el caso general en que el número de componentes y el número de procesadores disponibles es mayor que uno. En este caso el algoritmo procede recursivamente en paralelo dividiendo el vector original en sus componentes pares e impares (bucle paralelo de las líneas 21-24) que almacena en los vectores *even* y *odd*. A continuación el grupo de procesadores disponibles se divide en dos subgrupos de igual tamaño (sentencia condicional de las líneas 25-28) de forma que la mitad de los procesadores (aquellos con identificador lógico menor que la mitad del número de procesadores disponibles) llaman recursivamente a *fft()* sobre el subvector de componentes pares, mientras que el resto opera sobre las componentes impares. El bucle paralelo de las líneas 29-35 se encarga de la combinación de los resultados de la transformada de componentes pares e impares generando el vector de resultados. Las funciones *cadd(), cmul()* y *csub()* que aparecen en este bucle se encargan de realizar la suma, producto y sustracción de números complejos. La función finaliza liberando la memoria alojada para resultados temporales.

3.3.9.6. La Quickhull

La función *qh()* que presentamos en la Figura 3.32 calcula la envolvente convexa de una nube de puntos. Se trata de una versión paralela del algoritmo Quickhull [PRE85] para hallar la envoltura convexa de una nube de puntos. El algoritmo *quickhull* recibe este nombre por su gran parecido con el conocido algoritmo de ordenación Quicksort. Dados N puntos en el plano, el problema consiste en hallar cuales de estos puntos se encuentran en el perímetro de la menor región convexa que contenga a todos los puntos. Se trata de un problema que tiene multitud de aplicaciones, especialmente en el campo de la geometría computacional.

El conjunto de puntos al que se ha de calcular la envolvente convexa se puede dividir en dos subconjuntos de puntos: aquellos que quedan en cada semiplano de los definidos por una línea que une dos puntos con máxima distancia entre sí, *p1* y *p2* en la Figura 3.31. La función *qh()* se puede utilizar para calcular la envolvente de cada uno de los dos subconjuntos. El primer parámetro de la función, *pt[]* es un vector de índices al vector de puntos, en cuyas dos primeras componentes se encuentran los dos puntos *p1* y *p2* que definen el problema que se está resolviendo, y el segundo parámetro indica el número de puntos del vector *pt*.

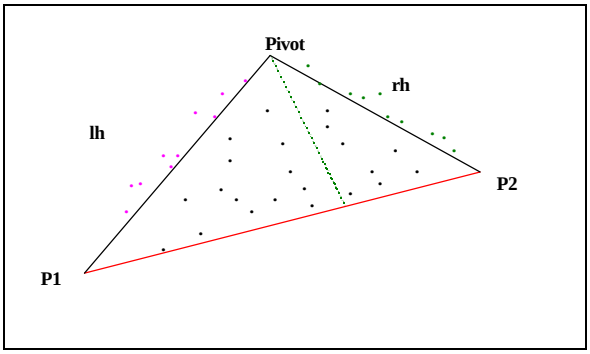

Figura 3.31 Cálculo de la envoltura convexa de una nube de puntos

En las líneas 10-15 del código de la Figura 3.32 se abordan los casos simples: si el número de puntos es dos, no hay nada que hacer, si son tres, hay que marcar el que se encuentra entre p1 y p2 como perteneciente a la envolvente y si el número de procesadores disponibles es sólo uno, el problema se resuelve mediante la función secuencial *seq_qh()* que se invoca en la línea 15..

```
 1 sync void qh(sh int *pt, sh int n) {
 2   sh int pivot;
 3   sh int p = 0;            /* Número de procesadores disponibles */
 4   sh int p1=pt[0], p2=pt[1];   /* Linea (p1, p2) */
 5   sh int *lh, *rh, n1, n2;    /* Para almacenar los subproblemas */
 6   sh int firstrightproc;
 7
 8   farm pprintf("qh(%d)\n", n);
 9   $ = mpadd(&p, 1);   /* Renumera $ y determina p */
10   if (n==2) return;
11   if (n==3) {  /* El punto entre p1 y p2 pertenece al hull */
12     belongs_to_hull[pt[2]] = 1;
13     return;
14   }
15   if (p==1) { seq_qh(pt, n); return; }
16
17  pivot = pivotize(pt, n, p);
18  belongs_to_hull[pivot] = 1;
19
20   /* Hallar subproblemas: */
21   n1 = n2 = n;
22   lh = delete_right(pt, &n1, p1, pivot);
23   rh = delete_right(pt, &n2, pivot, p2);
24
25   /* Determinar cantidad de procesadores en cada subproblema */
26   firstrightproc = (int) ((float)p * est_work(n1) / est_work(n));
27   if (firstrightproc == 0 && n1 > 0) firstrightproc = 1;
28   if ($ < firstrightproc)   /* Dividir el grupo de procesadores: */
29     qh(lh, n1);
30   else
31     qh(rh, n2);
32 }
```

Figura 3.32 La Quickhull en Fork95

En el caso general, el algoritmo elige un punto como pivote (línea 17) mediante la llamada a *pivotize()* y lo marca como perteneciente a la envolvente (línea 18). El vector global *belongs_to_hull[]* almacena los índices de los puntos pertenecientes a la envoltura convexa. Como pivote, la función *pivotize()* devuelve el punto de mayor

distancia a la recta P1P2. Las llamadas a *delete_right()* de las líneas 22 y 23 se utilizan para descomponer el problema en dos subproblemas, que se almacenan en los vectores *lh* y *rh* y que corresponden a los puntos a la izquierda de la línea *P1-pivot* y a la derecha de la línea *Pivot-P2* (ver Figura 3.31). La función *delete_right()* devuelve un vector del que se han eliminado los puntos de *pt* que quedan a la derecha de los puntos indicados por sus dos últimos parámetros. A continuación, en las líneas 26 y 27 el programa calcula el número de procesadores que se dedicará a la solución de cada subproblema mediante una llamada a la macro *est_work()* definida por el usuario, que proporciona una estimación de la dificultad del problema en cuestión. El grupo de procesadores disponibles será dividido en dos subgrupos y el número de procesadores de cada grupo dependerá de esta estimación. En las líneas 29 y 31 del código, la función *qh()* es llamada recursivamente para resolver los dos subproblemas resultantes. El primer subgrupo de procesadores opera sobre *lh* y el segundo sobre *rh*.

Capítulo IV
EL MODELO DE COMPUTACIÓN COLECTIVA

Capítulo IV

El modelo de Computación Colectiva

4.1. Introducción

En este Capítulo se exponen las ideas fundamentales de lo que denominamos Computación Colectiva. Se definirán conceptos como variables paralelas, variables comunes, funciones de división, jerarquía de procesadores etc., que nos permitirán presentar una clasificación de problemas, así como disponer del lenguaje necesario para representar claramente las ideas del modelo que desarrollaremos. En la sección 4.5 se introduce el concepto de Hipercubo Dinámico. Este concepto viene inducido por los patrones de comunicaciones que se producen entre conjuntos de procesadores. Luego se aplican en el epígrafe 4.6 los conceptos introducidos a la implementación de algoritmos, proporcionando ejemplos como el de la Transformada rápida de Fourier, el cálculo de la envoltura convexa, el Quicksort y un algoritmo de búsqueda. El siguiente punto a tratar es el equilibrado de la carga de trabajo entre el conjunto de procesadores, tanto en general como en el modelo de computación colectiva común. El modelo de Computación Colectiva se extiende con nuevas componentes para ser utilizado como modelo de predicción del tiempo de cómputo de los algoritmos, y ese aspecto del modelo es el que se estudia en el penúltimo epígrafe del Capítulo, que finaliza con la exposición de algunos aspectos de la herramienta *llc* que no se abordan hasta ese momento.

La mayoría de los conceptos que se introducen en este Capítulo aparecen en él por primera vez en esta memoria y son fundamentales para comprender el modelo de Computación Colectiva. El modelo viene caracterizado por una tripleta (*M, Col, Div)*. *M* representa la plataforma paralela (de memoria distribuida o compartida), *Col* el conjunto de funciones colectivas y *Div* es el conjunto de funciones de división. Una función se dice colectiva cuando es realizada por todos los procesadores del conjunto actual. Los conjuntos de procesadores pueden ser divididos utilizando las funciones de *Div*.

El modelo aporta una metodología para la traslación eficiente de algoritmos con paralelismo de datos anidados sobre arquitecturas paralelas reales. Otras aproximaciones a este problema como pueden ser las de *fork95* o NESL (que presentaremos en el siguiente Capítulo) o bien no utilizan como plataforma objeto máquinas paralelas reales o bien no obtienen la eficiencia que se consigue en nuestro modelo.

Hacemos una propuesta para una implementación eficiente de los procesos de división. Aunque otras librerías como MPI ofrecen funciones de división, el coste de las mismas, como demostraremos, no las hace adecuadas para ser utilizadas de forma intensiva.

Existen muchos problemas en los que la implementación de las funciones de división se puede realizar de forma eficiente utilizando exclusivamente información local al conjunto de procesadores que la realiza. De esta manera se evita el coste asociado a las comunicaciones. En este Capítulo se introducen familias de problemas (como las que hemos denominado Común-Común, Privado-Privado, etc.) en las cuales es posible utilizar esta aproximación.

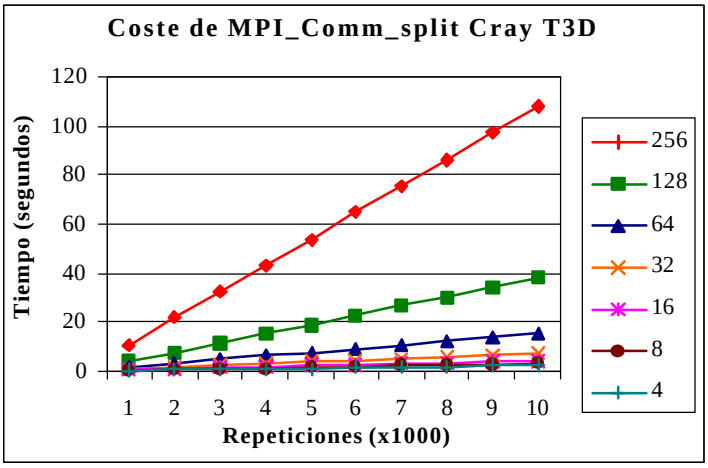

Figura 4.1 El coste de las funciones de división

La Figura 4.1 muestra una comparación en un Cray con 256 procesadores del coste de las funciones de división de MPI (etiqueta MPI) y las que proponemos en este Capítulo (etiqueta PAR). Se observa que mientras que el coste de las funciones de MPI crece con el número de divisiones realizadas, el de las que proponemos permanece comparativamente constante. En el Capítulo 6 de esta memoria presentamos más resultados computacionales en este sentido.

La idea subyacente a nuestra propuesta es que cada uno de los procesadores de uno de los conjuntos producto de la escisión mantiene una relación con uno de los procesadores en los otros subconjuntos. Esta relación determina las comunicaciones de los resultados producto de la tarea realizada por el conjunto al que el procesador pertenece. Esta estructura de división da lugar a patrones de comunicaciones que se asemejan a los de un hipercubo. La dimensión viene determinada por el número de divisiones demandadas mientras que la aricidad en cada dimensión es igual al número de subconjuntos solicitados. A semejanza de lo que ocurre en un hipercubo *k*-ario convencional, una dimensión divide al conjunto en *k* subconjuntos comunicados a través de la dimensión. Sin embargo, los subconjuntos opuestos según una dimensión no tienen porqué tener el mismo cardinal. A estas estructuras resultantes las hemos denominado Hipercubos Dinámicos.

Presentamos una clasificación de problemas paralelos en función de las características de los datos de entrada y de salida de los mismos con respecto a la visión

que de ellos tienen los procesadores de la máquina. La nomenclatura introducida se utiliza para caracterizar los problemas que presentamos en la memoria.

Aportamos ejemplos de algoritmos tanto del tipo de los que hemos denominado de Computación Colectiva como de Computación Colectiva Común [Rod99b]. Este último tipo de algoritmos resuelven un tipo concreto de problemas (los de la clase Común-Común). Para ambos tipos de algoritmos estudiamos diferentes formas de introducir equilibrado de la carga de trabajo y los resultados que produce cada una de ellas.

Muchos ejemplos del Capítulo se presentarán utilizando un pseudocódigo de tipo *La Laguna C* (*llc*) [Rod97a]. *Llc* es una herramienta basada en el Modelo de Computación Colectiva que extiende el modelo de librería de paso de mensajes con las ideas de la computación colectiva y que será introducida en la sección 4.9 de este Capítulo. La sintaxis y la semántica de las sentencias secuenciales de *llc* son similares a las de C mientras que la semántica de las construcciones paralelas concretas utilizadas será aclarada en cada caso.

4.2. Generalidades

A lo largo del Capítulo utilizaremos intensivamente los conceptos de bloque, nivel de profundidad y ámbito usuales en lenguajes de programación [Aho90]. Se hará uso de estos conceptos en el sentido usual que tienen en lenguajes imperativos con ámbito léxico tales como C o Pascal y que describimos brevemente a continuación

Las reglas de ámbito de un lenguaje determinan qué declaración de un identificador se ha de utilizar cuando el identificador aparece en el texto del programa. La *regla de ámbito estático* determina la declaración que se aplica a un identificador con sólo examinar el texto del programa. La *regla de ámbito dinámico* determina la declaración aplicable a un identificador durante la ejecución del programa, considerando las actividades en curso. En este Capítulo utilizaremos la regla de ámbito estático, puesto que nos centraremos en lenguajes que utilizan este tipo de regla.

Un *bloque* es una sentencia que contiene sus propias declaraciones de datos locales. Una característica de los bloques es su estructura de anidamiento. El ámbito de una declaración en un lenguaje con estructura de bloques viene dado por la regla del *anidamiento más cercano*:

1. El ámbito de una declaración en un bloque B incluye B.
2. Si un identificador x no está declarado en un bloque B, entonces una referencia a x en B está en el ámbito de una declaración de x en un bloque abarcador B' tal que:
   i) B' tiene una declaración de x, y
   ii) B' está anidado más cerca alrededor de B que cualquier otro bloque con una declaración de x.

La estructura de bloques se puede implementar utilizando asignación de memoria por medio de una pila. Dado que el ámbito de una declaración no se aplica fuera del bloque en que aparece, se puede asignar la memoria para el identificador declarado cuando se entra al bloque y desasignar cuando el control abandona el bloque. Este punto de vista considera al bloque como un "procedimiento sin parámetros", llamado únicamente desde el punto inmediatamente anterior al bloque y regresando únicamente al punto inmediatamente posterior al bloque. Así pues, se puede mantener el entorno no local para un bloque utilizando las mismas técnicas que se utilizan para los procedimientos. Una implementación alternativa consiste en asignar de una sola vez memoria para un cuerpo de un procedimiento completo. Si hay bloques dentro del

procedimiento, entonces la asignación se hace para la memoria necesaria para las declaraciones dentro de los bloques.

Con ámbito léxico y sin procedimientos anidados (como ocurre por ejemplo en C), cualquier identificador que sea no local a un procedimiento es no local a todos los procedimientos. Su dirección estática puede ser utilizada por todos los procedimientos, independientemente de cómo se activen.

En un lenguaje con procedimientos anidados, una referencia a un identificador no local *a* en un procedimiento, está dentro del alcance de la declaración anidada más cercana de *a* en el texto del programa estático.

La noción de *profundidad de anidamiento* de un procedimiento se utiliza para implementar el ámbito léxico. Se considera que el identificador del programa principal está a profundidad de anidamiento 1 y se suma 1 a la profundidad de anidamiento al pasar de un procedimiento abarcador a un procedimiento abarcado.

Una implementación directa del ámbito léxico para procedimientos anidados se obtiene añadiendo un puntero, denominado *enlace de acceso*, a cada registro de activación. Si el procedimiento *p* está anidado inmediatamente dentro de *c* en el texto fuente, entonces el enlace de acceso en un registro de activación para *p* apunta al enlace de acceso para la activación más reciente de *c*.

Supóngase que el procedimiento *p* a profundidad de anidamiento n_p hace referencia a un identificador no local *a* con profundidad de anidamiento $n_a <= n_p$. El registro de activación en el que ha sido asignada memoria para *a* se alcanza recorriendo n_p-n_a enlaces de acceso a partir del registro de activación que se encuentra en el tope de la pila de ejecución. El valor n_p-n_a se puede obtener en tiempo de compilación.

4.3. Definiciones

Consideremos un computador paralelo *M* (en general de memoria distribuida) constituido por un conjunto de procesadores $P=\{p_1, p_2, ..., p_N\}$ interconectados entre sí mediante algún tipo de red de interconexión y dotados cada uno de ellos de una memoria. Asumiremos además que todos los procesadores de la máquina ejecutan un mismo programa (es decir, un modelo de programación SPMD). Dado que todos los procesadores de la máquina ejecutan el mismo programa, asumiremos que todos ellos tienen acceso a un espacio común de identificadores. Supondremos también que los procesadores están organizados en *conjuntos de procesadores*. Al comienzo del cómputo todos los procesadores *P* de la máquina paralela *M* pertenecen a un mismo conjunto de procesadores, al que llamaremos *conjunto raíz*. Todos los procesadores de un determinado conjunto se caracterizan porque ejecutan la misma tarea.

Cada procesador tiene asignado un identificador lógico, NAME y una variable NUMPROCESSORS que almacena en todo instante el número de procesadores presentes en el conjunto al cual pertenece el procesador.

Definición 4.1: Computación Colectiva. Definiremos computación colectiva sobre una máquina paralela *M* como una forma de cómputo en la que el conjunto de procesadores de la máquina sólo puede realizar tres tipos de operaciones:

1. Cualquier cómputo secuencial: asignaciones, bucles, llamadas a función, etc.
2. Operaciones colectivas $C \hat{I} Col$
3. Funciones de división $D \hat{I} Div$

Una instanciación del modelo de computación colectiva queda determinado dando la tupla

$$(M, Col, Div)$$

La clase de funciones *Div* y *Col* deben cumplir las propiedades que se establecerán en los párrafos que vienen a continuación.

Por ejemplo, *M* podría ser cualquier máquina paralela (IBM-SP2, Cray T3E, una red de estaciones de trabajo ejecutando PVM, etc.), *Col* podría ser el subconjunto de funciones colectivas de MPI (*MPI_Reduce*, *MPI_Allreduce*, *MPI_Scan*, etc.) y *Div* podría ser la función de división de comunicadores de MPI (*MPI_Comm_split*, *MPI_Cart_sub*).

Otros ejemplos de operaciones colectivas y de división los encontramos en el conjunto de funciones colectivas y de división de *La Laguna C* respectivamente. Podemos tomar también como conjunto de operaciones colectivas, *Col* el conjunto de operaciones multiprefijo de *fork95* y sus sentencias de asignación a variables compartidas ejecutadas en modo síncrono. Como funciones de división *Div* en *fork95* podemos tomar el conjunto formado por las sentencias condicionales, *fork*, y de repetición.

Definición 4.2: Funciones de división y jerarquía de procesadores: Sea $P=\{p_1, p_2, ..., p_N\}$ un conjunto de procesadores. Sea $Q \acute{I} P$ un conjunto hoja de procesadores. Una función de división $D(T_1, T_2, ..., T_s) \hat{I} Div, s \hat{I} N$ es una función que permite particionar el *conjunto hoja* Q en un cierto número $r £ s$ de subconjuntos disjuntos $Q_1, ..., Q_r$:

$$Q = \grave{E}_{1 £ i £ r} Q_i ; \quad Q_i \Ç Q_j = Æ \ \ si\ i \neq j \ \ " \ 1 £ i, j £ r$$

Div es el conjunto de funciones de división definidas y toda división de un conjunto de procesadores deberá realizarse utilizando alguna función de división, $D \in Div$.

Cada uno de los conjuntos de procesadores Q_i mantiene las mismas propiedades que el conjunto de partida Q y los procesadores de cada uno de los subconjuntos Q_i se renombran con valores distintos de su identificador lógico NAME en el rango $[0, |Q_i|-1]$. NAME es una *variable paralela privada*. Cada uno de los conjuntos de procesadores Q_i en los que se divide Q trabajará en paralelo en una cierta *tarea* T_i y a la finalización de la misma, los conjuntos Q_i se reintegrarán en el conjunto original, Q. En lo que sigue, el término *tarea* se referirá a cada uno de los flujos (threads) T_i definidos por los parámetros de la función de división $D \in Div$.

La ejecución de una división de un conjunto de procesadores consta de los siguientes pasos:

1. División del conjunto original Q de procesadores
2. Asignación de subconjuntos Q_i de procesadores a las tareas T_i
3. Asignación de datos de entrada In_i de las tareas paralelas T_i a los procesadores de Q_i
4. Ejecución de las tareas paralelas T_i
5. Distribución de los datos generados Out_i por las tareas paralelas T_i
6. Reconstrucción del conjunto de procesadores inicial Q

La división de los conjuntos de procesadores produce una jerarquía a la que llamaremos *jerarquía de conjuntos* que puede ser representada mediante un árbol como el que presentamos en la Figura 4.2. En la figura, los nodos del árbol representan conjuntos de procesadores, mientras que los puntos interiores a los nodos representan los procesadores mismos.

Los conjuntos de procesadores más recientemente creados en un determinado instante del cómputo los denominaremos *conjuntos hoja*. Inicialmente, el conjunto raíz es el único conjunto hoja, mientras que en cualquier instante de la ejecución del

programa, todos los procesadores de la máquina pertenecen a algún conjunto hoja, como muestra la Figura 4.2. En un instante determinado, los nodos en la jerarquía de conjuntos corresponden a conjuntos de procesadores *Q* que están ejecutando una misma tarea *T*. Los nodos hoja corresponden a tareas *T* que están siendo ejecutadas, mientras que los nodos interiores corresponden a tareas *T'* que están a la espera de la finalización de sus subtareas para su completación.

Las funciones de división deberán ser ejecutadas por todos los procesadores de un conjunto hoja. La variable paralela común NUMPROCESSORS, que almacena el número de procesadores en el conjunto hoja, deberá ser reevaluada al mismo valor por todos los procesadores asignados a la misma tarea T_i. Todos los procesadores asignados a una misma tarea constituyen un conjunto hijo del conjunto actual dentro de la jerarquía.

Nótese que, si entendemos por virtualización de procesadores el mecanismo por el cual varios procesadores lógicos son simulados por un único procesador físico, aquí tendríamos lo que podríamos llamar "antivirtualización de procesadores", es decir, varios procesadores físicos juegan el papel de un único procesador lógico, puesto que

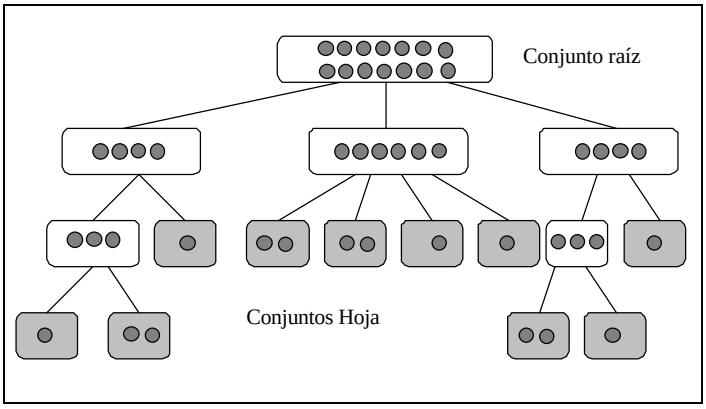

Figura 4.2 La estructura jerárquica de los procesadores

todos los procesadores de un conjunto de la jerarquía realizan el mismo cómputo y se pueden considerar como un único procesador virtual.

```
1 int g(int x) {
 2   . . .
 3 }
 4 int s(int z) {
 5   . . .
 6 }
 7
 8 main(void) {
 9   . . .
10   PAR(r(), s());
11 }
```

Figura 4.3 Una función de división en *La Laguna C*

En el código de la Figura 4.3, la llamada PAR(*r()*, *s()*); que aparece en la línea 10 es un ejemplo de una función de división en *La Laguna C*. El conjunto de procesadores hoja que ejecuta la llamada se divide en dos subconjuntos y cada uno de ellos ejecuta una de las funciones que se pasan como parámetro en la llamada (funciones *r()* y *s()*). En este ejemplo, las tareas a ejecutar T_1 y T_2 están representadas por las funciones *r()* y *s()* respectivamente.

El constructo

PARBOOL(*C, f1(), f2()*);

Es otra forma de función de división en *llc*. En este caso, *C* es una condición booleana y el conjunto de procesadores hoja ejecuta la función *f1()* o *f2()* dependiendo del valor resultante de la evaluación de *C*.

SPLIT($T_1()$, $T_2()$);

es otra de las funciones de división de *llc* en la que el conjunto de procesadores se divide también en dos subconjuntos para ejecutar las tareas $T_1()$ y $T_2()$. La diferencia entre SPLIT() y PAR() radica en que el quinto paso de una función de división (el de

```
 1 . . .
 2 group = NAME % 2;
 3 MPI_Comm_split(comm, group, NAME, &newcomm);
 4 MPI_Comm_size(newcomm, &NUMPROCESSORS);
 5 MPI_Comm_rank(newcomm, &NAME);
 6 if (group)  /* Primer grupo */
 7   t1();
 8 else
 9   t2();
10 MPI_Comm_free(&newcomm);
11 MPI_Comm_size(comm, &NUMPROCESSORS);
12 MPI_Comm_rank(comm, &NAME);
```

Figura 4.4 Una función de división

distribución de los datos generados) no se produce en el caso de la función SPLIT().

La Figura 4.4 presenta otro ejemplo de una función de división implementado en MPI. Un conjunto hoja de procesadores es dividido en dos subconjuntos. En la figura se pueden reconocer algunas de las fases involucradas en la ejecución de una función de división, así las líneas de código 2-5 constituyen la fase de división del conjunto, la sentencia condicional de la línea 6 realiza la asignación de subconjuntos de procesadores a las tareas paralelas, las llamadas a las funciones *t1()* y *t2()* de las líneas 7 y 9 representan las dos tareas paralelas a ejecutar, y el código de las líneas 10-12 constituyen la fase de reunificación del conjunto original.

En un lenguaje más orientado al modelo de computación colectiva como puede ser *llc*, la expresión de una función de división de un conjunto de procesadores resulta más simple, como veremos en más ejemplos a lo largo de este mismo Capítulo.

Definición 4.3: Conjuntos complementarios de un conjunto hoja Q_i. Son todos los conjuntos hojas $Q_j, j \neq i$ que fueron creados por la misma función de división $D\hat{I}Div$ que dio lugar a la creación de Q_i.

Definición 4.4: Profundidad de activación: Fijado un instante de la ejecución de un programa, a un identificador *a* se le asigna *profundidad de activación* 1 cuando tiene lugar la primera activación del bloque que contiene su declaración. La *profundidad de activación* se refiere a la dirección de memoria que se asigna a un identificador en un nuevo registro de activación, indicando que en ese momento tiene lugar la primera asignación de memoria a la variable. Se incrementa la *profundidad de activación* en uno, por cada activación posterior del bloque que contiene la declaración del identificador *a*. Si el bloque que contiene la declaración de *a* no se activa en el instante de la ejecución fijado, la *profundidad de activación* de *a* es 0. A esta regla la denominaremos *Regla de Profundidad de Activación Dinámica* ya que nos permite

determinar la *profundidad de activación* aplicable a un identificador durante la ejecución, considerando las activaciones en curso.

```
 1 int b;
 2 int p(void) {
 3   int a;
 4   . . .
 5 }
 6 int q(void) {
 7   int x;
 8   if (...) q();
 9 }
10 int main() {
11   b = 7;
12   p();
13   q();
14 }
```

Figura 4.5 Profundidad de activación

Por ejemplo, en la línea 11 del código de la Figura 4.5 la variable global *b* tiene profundidad de activación 1, puesto que a dicha variable en ese momento de la ejecución se le ha asignado memoria una vez, al comenzar la ejecución del programa. En la línea 12 cuando se invoca a la función *p()*, la variable *a* declarada en la línea 3 tiene profundidad de activación 1, puesto que su memoria se asignará en el registro de activación de *p()* que se empuja en el tope de la pila de ejecución. A la variable *x* declarada en la línea 7, se le asigna profundidad de activación 1 cuando tiene lugar la primera llamada a *q()* (línea 13). Si tiene lugar una segunda llamada recursiva a *q()* (línea 8), se empuja un nuevo registro de activación para *q()* y por lo tanto, la profundidad de activación de *x* se incrementa y pasa a ser dos. Las sucesivas llamadas recursivas a *q()* modifican el valor de la profundidad de activación de *x*.

Definición 4.5: Variables paralelas: Sea $Q=\{p_0, p_1, ... p_{m-1}\}$ un conjunto de la jerarquía de conjuntos. Asumamos que en un procedimiento *B* se declara una variable local *v*. Si un procesador $p_i \hat{I} Q$ ejecuta una llamada al procedimiento *B*, conceptualmente empuja un registro de activación en el tope de su pila de ejecución y asigna una dirección d_i a la variable *v*. Fijemos un instante de la ejecución del programa, en el que los procesadores $p_i \hat{I} Q$ pueden ejecutar una llamada al procedimiento *B*. Sea $a_i, i \hat{I} \{0, ..., m-1\}$ la profundidad de activación de la variable *v* en el procesador p_i en ese momento. Diremos que la tupla de direcciones de memoria $d_0, d_1,\ldots,d_{m-1}$ define *una variable paralela* si y solo si $a_0 = a_1 =\ldots= a_{m-1}$.

Al comienzo de la ejecución de un programa, todas las variables globales se asignan de forma estática, por lo tanto ya tienen sus direcciones de memoria asignadas a un cierto registro de activación, así pues su *profundidad de activación* será uno en cualquier momento posterior de la ejecución del programa. En conclusión, todas las variables globales de un programa son variables paralelas. Abusando del lenguaje nos referiremos a la variable paralela por el identificador *v* de la declaración de variable a la que están ligadas. Obsérvese que debe existir una instancia de la variable paralela por cada procesador del conjunto *Q*. Aunque las direcciones d_i estén ligadas al mismo identificador *v* con la misma profundidad de activación, si ocurre que en alguno de los procesadores p_j no existe la correspondiente d_j, diremos que el conjunto de direcciones define una *variable incompleta.*

```
 1 int z;
 2 int f(void) {
 3   int a;
 4   ...
 5 }
 6
 7 main() {
 8   float x;
 9   x := 1.0;
10   if (NAME < 4)
11     f();
12 ...
13 }
```

Figura 4.6 Una variable incompleta

Supongamos que en la Figura 4.6 el conjunto de procesadores que ejecuta el programa y la función *main()* es un conjunto raíz formado por 6 procesadores. Tanto la variable *z* declarada en la línea 1 como la variable *x* declarada en la línea 8 son variables paralelas durante la ejecución de todo el programa. Sin embargo, la variable *a* declarada

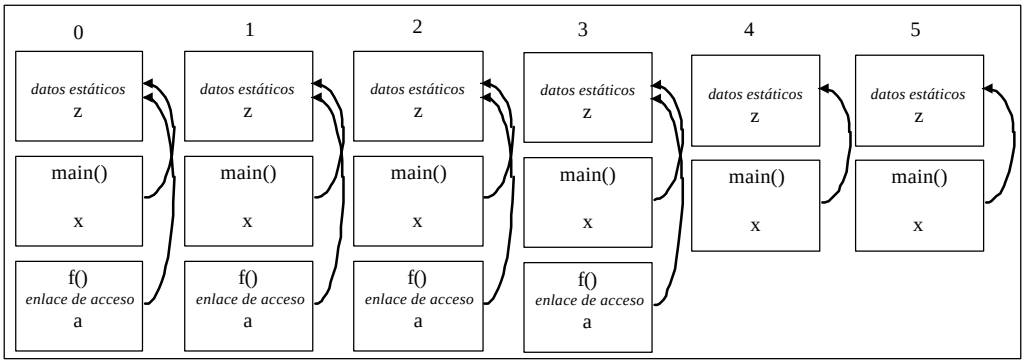

Figura 4.7 Variables paralelas y no paralelas en el momento de la ejecución de la línea 4 del programa de la Figura 4.6. Las etiquetas de la parte superior son los nombres lógicos de los procesadores

en la línea 3 sólo existe en un subconjunto de los procesadores que ejecutan el programa: aquellos que invocan a la función *f()* en la línea 11, y por ello no la consideraremos una variable paralela sino que se trata de una variable incompleta. La variable *a* tiene profundidad de activación 1 en los procesadores con NAME<4 mientras

```
 1 int g(int x) {
 2   int b;
 3
 4   while (x > 0)
 5     g(x-1);
 6   ...
 7 }
 8
 9 main() {
10   PAR(g(NAME), h());
11 }
```

Figura 4.8 No todas las instancias de *b* definen una variable paralela

que no es instanciada por el resto de procesadores. Aunque la variable existe en el conjunto de procesadores con NAME<4, este conjunto de procesadores no constituye un

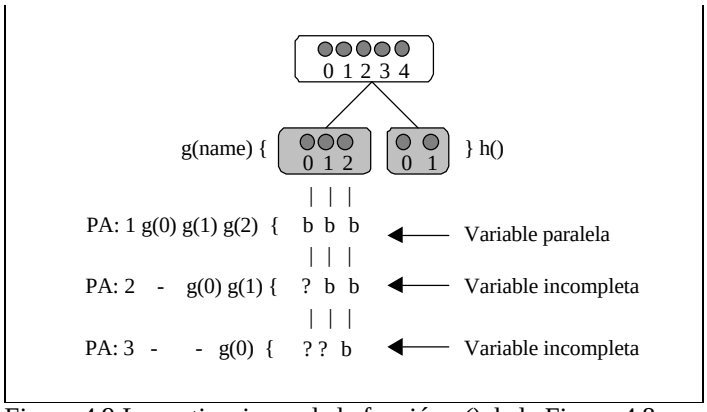

Figura 4.9 Las activaciones de la función *g()* de la Figura 4.8

conjunto hoja puesto que, a diferencia de *fork95*, la semántica de la sentencia condicional que estamos considerando es tal que no divide el conjunto hoja actual de procesadores. Por el contrario, la variable *x* declarada en la línea 8 sí es paralela durante la ejecución de todo el programa, puesto que la declaración de la línea 8 es ejecutada por todos los procesadores del conjunto raíz.

La Figura 4.7 muestra una posible representación de la pila de ejecución del programa de la Figura 4.6 utilizando enlaces de acceso.

Nótese que la variable *b* declarada en la línea 2 del código de la Figura 4.8, tiene niveles de activación diferentes para cada procesador del conjunto hoja que ejecuta la función *g()*, por lo tanto no todas las instancias de *b* definen una variable paralela.

La llamada a la función de división PAR *Î Div* de la línea 10

PAR(g(NAME), h());

divide al conjunto raíz de procesadores en dos subconjuntos hoja (ver Figura 4.9), uno de ellos ejecutando la tarea *g(NAME)* y el otro *h()*. Existe una instancia de *b* por cada activación de la función *g()*, pero hay un número diferente de activaciones de la función recursiva *g()* para cada procesador del primer conjunto hoja. Esto es consecuencia de que en la llamada paralela realizada en la línea 10, el parámetro actual del que depende el número de activaciones de la función es diferente para cada procesador. La primera llamada a la función *g()* en la línea 10 es realizada por todos los procesadores del primer

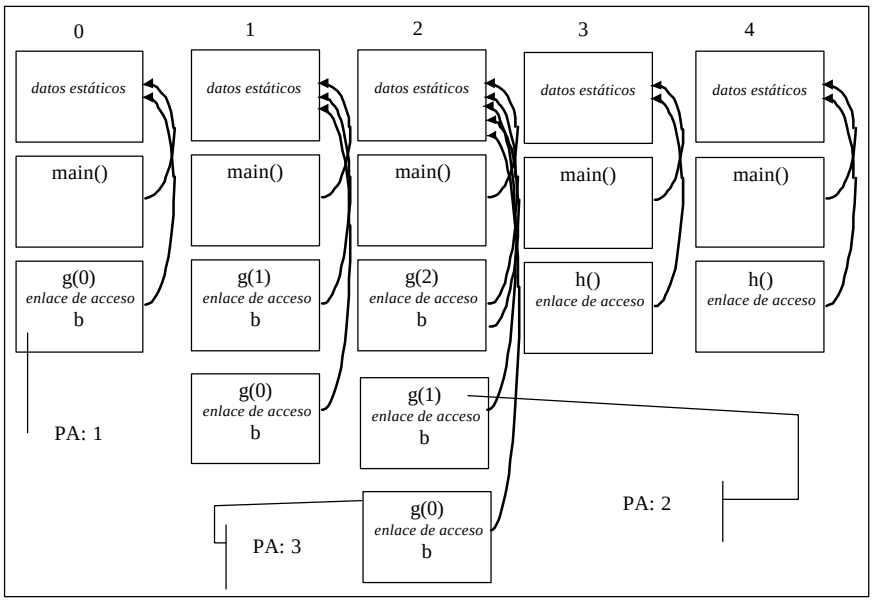

Figura 4.10 La pila de ejecución para el programa de la Figura 4.8

subconjunto y da por tanto lugar a una variable paralela en la instanciación de la declaración de la línea 2 con profundidad de activación 1. Dado que el procesador con nombre lógico NAME=0 no realiza la llamada recursiva de la línea 5, las siguientes instanciaciones de *b* con profundidades mayores que uno, definen una variable que no es paralela (incompleta). En la parte izquierda de la Figura 4.9, la etiqueta PA indica la profundidad de activación de cada instanciación de las variables *b*.

Definición 4.6: Variable Común: Sea $Q = \{p_0, p_1,\ldots,p_{m-1}\}$ un conjunto de la jerarquía de conjuntos. Fijemos un nivel de profundidad de anidamiento, *n*. Sea *v* una variable paralela con profundidad de anidamiento $n_v \leq n$ y v_k el valor de *v* en el procesador p_k, $k \in \{0,...,m-1\}$. Diremos que *v* es una *variable común* a profundidad de anidamiento *n*, si y solo si, $v_i = v_j \ \forall\ i, j \in \{0,...,m-1\}$

Nótese que una variable paralela para ser considerada común al conjunto *Q* debe mantener el mismo valor en todos los procesadores del conjunto *Q* mientras dicho conjunto permanece activo, esto es, mientras es un conjunto hoja. La variable sigue siendo común incluso si las componentes $v_0, v_1, \ldots v_{m-1}$ de la variable paralela *v* dejan de tener el mismo valor cuando *Q* se divide como consecuencia de la ejecución de una función *DIDiv*. Sin embargo, para ser común es necesario que después de la reunificación del conjunto *Q*, las componentes de *v* tengan el mismo valor.

```
 1 int x, y;
 2
 3 void f(void) {
 4   x = 7;
 5   y = 5;
 6 }
 7 void g(void) {
 8   x = 6;
 9   y = 5;
10 }
11 main(void) {
10   y = 1;
11   SPLIT(f(); g())
12   printf("x: %d y: %d\n", x, y);
13 }
```

Figura 4.11 Variables comunes y no comunes

La variable *x* declarada en la línea 1 de la Figura 4.11 es una variable paralela no común en el ámbito de su declaración (nivel de profundidad 1) puesto que las tareas paralelas *f()* y *g()* creadas por la función de división *SPLIT()* le asignan valores diferentes. Por el contrario, la variable *y* declarada en la misma línea sí es común a nivel de profundidad 1 puesto que en ese ámbito mantiene todo el tiempo el mismo valor. Como consecuencia de este hecho, en la escritura que tiene lugar en la línea 12 del código, todos los procesadores del conjunto hoja escribirán el mismo valor (5) para *y*, y diferentes valores para *x*.

Nótese que los conceptos de variables comunes y compartidas son diferentes: estamos considerando un sistema distribuido en el que cada procesador tiene su propio espacio de direcciones de memoria. Las direcciones de memoria de una variable paralela son en general distintas en cada procesador. Utilizando las variables comunes es posible emular en un sistema distribuido las facilidades que proporciona una memoria compartida.

Definición 4.7: Variable Privada: Diremos que una variable paralela *v* del programa es *privada* en un determinado ámbito estático cuando no existe garantía de que los valores

de la variable, v_k sean iguales en los distintos procesadores $k \hat{I} Q \acute{I} P$ en dicho ámbito estático. Dicho de otro modo, una variable paralela es privada si no es común.

Por extensión de los conceptos de variables privadas y comunes hablaremos también de valores privados o comunes como los valores correspondientes a variables privadas y comunes respectivamente.

```
1 common  a, c;
2   {
3   c := 1;
4   a := 0;
5   b := NAME;
6   REDUCEBYADD(c, b);
7   }
```

Figura 4.12 Variables privadas y comunes

En el código de la Figura 4.12 las variables *a* y *c* llevan el cualificador *common*. En el pseudocódigo que estamos utilizando, el cualificador *common* indica al compilador que se trata de variables comunes en el ámbito de ese cualificador, mientras que la variable *b* es privada en dicho ámbito. Como se deduce de la asignación de la línea 5 cada procesador asignará un valor diferente a *b*. La operación de reducción de la línea 6 provocará que todos los procesadores almacenen en *c* el mismo valor: $c=\Sigma_{i \hat{I} P} b_i$ siendo *P* el conjunto hoja de procesadores. En el ámbito dinámicamente creado por la llamada a la operación REDUCEBYADD(*c, b*) la variable *c* puede eventualmente dejar de ser común. No obstante, *c* es común con respecto al ámbito estático de la declaración de la línea 1.

Definición 4.8: Variable local: La creación ordinaria de una variable por un procesador que ejecuta el código correspondiente a una declaración da lugar a una *variable local.* Cada una de las variables v_i componentes de una variable paralela (v_0, ... v_{m-1}) y también las de una variable incompleta son variables locales. Utilizaremos el término *variable local* para indicar que sólo existen garantías de que dicha variable existe en un procesador determinado.

Definición 4.9: Variables resultado: Llamaremos resultado de la ejecución de un bloque de código al conjunto de variables paralelas externas al ámbito del bloque que resultan modificadas por la ejecución del mismo, tanto si le son pasadas como parámetro como si son modificadas como consecuencia de un efecto lateral.

Por ejemplo, en el código de la Figura 4.12, las variables resultado del bloque de código son *a*, *b*, y *c*.

```
int h;
void m(int *z, int w) {
  int x, y;
  x =7;
  *z = 24;
  h = 3;
}
main(void) {
  int a, b;
  m(&a, b);
  . . .
}
```

Figura 4.13 Variables resultado

Asumiendo que todas las variables que aparecen en el código de la Figura 4.13 son paralelas, las variables resultado de la función *m()* son *a* (parámetro modificado) y *h* (variable no local a *m()* que resulta modificada).

Definición 4.10: Operación colectiva: Una operación colectiva $C \hat{I} Col$ realizada por un conjunto hoja de procesadores $Q=\{p_1, p_2, ... p_s\} \acute{I} P$ es una operación $C(z_1, v_2, ... w_s)$ sobre unos datos de entrada realizada por todos los procesadores de *Q*. *Col* es el conjunto de operaciones colectivas definidas. La realización de una operación colectiva conlleva una comunicación entre todos los procesadores que la realizan y representa un punto de sincronización en la ejecución del programa: los procesadores del conjunto hoja no prosiguen la ejecución de código hasta que todos los procesadores de *Q* han completado la operación.

La llamada a REDUCEBYADD() que aparece en la Figura 4.12 es un ejemplo de operación colectiva, puesto que la llamada a REDUCEBYADD() ha de ser realizada por todos los procesadores del grupo hoja. Es un error invocar a una operación colectiva si esa llamada no la realizan todos los miembros del grupo hoja. Esta situación se ilustra en el código de la Figura 4.14 en el que sólo la mitad de los procesadores del grupo invocaría a REDUCEBYADD().

```
main() {
  int y, x = NAME;
  if (NAME < NUMPROCESSORS / 2)
    REDUCEBYADD(y, x);
  . . .
}
```

Figura 4.14 Error en la llamada a una operación colectiva

Definición 4.11: Llamada colectiva: Una llamada a una operación *f* se dice colectiva si $f \hat{I} Col$ y cada uno de sus argumentos es una variable local de cada uno de los procesadores participantes.

Definición 4.12: Operación colectiva común: Una operación colectiva $C \hat{I} Col$ se dice *común* si todas las variables resultado y todos los valores resultado de la operación son comunes. Una operación colectiva común garantiza que si es invocada con variables resultado paralelas, éstas serán variables comunes después de su ejecución.

La operación de reducción que aparece en la línea 6 de la Figura 4.12 es un ejemplo de operación colectiva común: cada procesador del conjunto hoja aporta el valor de su variable *b* y como resultado de la operación, todos los procesadores del conjunto obtienen en *c* la suma de las variables *b* aportadas por cada uno de ellos.

La función de MPI *MPI_Reduce()* es una operación colectiva, pero no común: la función realiza una reducción (utilizando un operador predefinido) en la que participan todos los procesadores de un comunicador, quedando el resultado de la operación almacenado sólo en uno de los procesadores participantes (que se especifica). La función *MPI_Allreduce()* sí es colectiva común puesto que en este caso, el resultado de la reducción se almacena en todos los procesadores que realizan la función.

Definición 4.13: Bloque de código colectivo común: Diremos que un bloque de código es colectivo común si todas las variables paralelas que eran comunes a la entrada del bloque lo son también en el ámbito estático del bloque, y por tanto a la salida del mismo.

El código de la Figura 4.12 determina un bloque de código colectivo común puesto que las variables comunes *a* y *c* lo son en todo el ámbito del bloque, mientras que la variable *b*, es privada.

Supongamos que en la Figura 4.12 sustituyéramos la llamada a REDUCEBYADD de la línea 6 por una llamada

PREFIXBYADD(c, b);

PREFIXBYADD(c, b) produce que cada procesador obtenga en la variable *c* la suma de los valores aportados por cada procesador en las variables *b*, pero los valores de *b* también quedan actualizados, en este caso con un valor diferente para cada procesador NAME correspondiente a la suma parcial, $b_0+b_1+...b_{NAME}$. Aún con este cambio el código resultante obtenido de la Figura 4.12 sería un bloque de código colectivo común en el supuesto de que la variable paralela *b* sea privada y la variable *c* sea común.

Consideremos un conjunto de procesadores hoja con cuatro procesadores que ejecutan el código de la Figura 4.15 donde *y* es una variable paralela común, *x* y *z* son variables locales. El código de la figura es colectivo común, puesto que después de la ejecución la variable resultante, $y=(y_0, y_1, y_2, y_3)$ contiene el mismo valor común en todos los procesadores del conjunto hoja.

```
. . .
if NAME > 2
  REDUCEBYADD(y, x);
else
  REDUCEBYADD(y, z);
```

Figura 4.15 Una operación colectiva común

Por el contrario, el código de la Figura 4.16, en el que suponemos *z* e *y* variables paralelas comunes no es colectivo común puesto que las variables paralelas *z* e *y* son comunes a la entrada del código, pero no a la finalización del mismo. El vector de direcciones resultados (y_0, y_1, z_2, z_3) no constituye una variable paralela porque no se corresponde con un mismo identificador del programa.

```
. . .
if NAME > 2
  REDUCEBYADD(z, x);
else
  REDUCEBYADD(y, u);
```

Figura 4.16 Una operación colectiva no común

Definición 4.14: Función de división común: Diremos que una función de división es común si sus variables de entrada y de salida son variables comunes. Esto es, todas las variables resultado de una función de división común son comunes supuesto que lo eran antes de la ejecución de la misma.

Las diferentes tareas paralelas T_i de una función de división pueden generar resultados potencialmente diferentes, pero en una función de división común la distribución de estos resultados entre los procesadores participantes es tal que al finalizar la división todos los procesadores obtienen los mismos resultados.

Cuando se produce una división de un conjunto de procesadores, *Q*, las variables que eran comunes de *Q* hasta el instante de la división podrían perder su condición de comunes durante la ejecución de las subtareas por los subconjuntos Q_i en los que se

divide *Q*. A la finalización de la división, cuando se recupera el conjunto de partida, *Q*, estas variables pueden volver o no a ser comunes de *Q* dependiendo del esquema que se haya seguido en la distribución de los resultados de las tareas paralelas. Si la función de división es común, deberán volver a ser comunes.

El constructo *fork* de *fork95* puede considerarse una función de división común si asociamos el concepto de variable compartida con el de variable común: toda variable compartida tiene el mismo valor en todos los procesadores después de la ejecución de un *fork*. Ni en MPI ni en *llc*, las funciones de división de conjuntos de procesadores se pueden calificar de comunes puesto que el hecho de que lo sean o no, depende de las tareas paralelas que la división genere. El código de la Figura 4.4 es un ejemplo de código MPI en el que la función *MPI_Comm_split()* puede dar lugar a una división no común. De modo similar, en *llc* cuando se utiliza el constructo PAR como en:

PAR(*t1()*; *t2()*)

la división será común si las tareas paralelas t_1 y t_2 preservan común el estado de las variables que lo fueran antes de su ejecución.

Podríamos imaginar un lenguaje orientado al modelo de computación colectiva que garantizara la existencia de funciones de división comunes, independientemente de la tareas paralelas que se ejecutaran. Toda variable debería estar cualificada como común o no común a partir de su declaración y la implementación del lenguaje se encargaría de mantener comunes después de la ejecución de una función de división aquellas variables que lo eran antes de su ejecución. Esta idea se podría implementar por ejemplo en *llcc98* [Lun98] del siguiente modo: en la generación de código para el cuerpo de una función de división (una sentencia *parallel* por ejemplo) se enlazarían en una lista todas las variables que hubieran sido declaradas comunes cuyo valor fuera modificado en el cuerpo de la función. La última fase de la generación del código para la función consistiría en la actualización en todos los procesadores del conjunto hoja original de las variables comunes que hubieran resultado incoherentes.

Definición 4.15: Computación colectiva común: Definiremos computación colectiva común sobre una máquina paralela *M* como una computación colectiva (*M*, *Col*, *Div*) en la que todos los bloques de código son comunes.

Presentaremos ejemplos de computación colectiva común en la sección 4.6.1.

4.4. Clasificación de problemas

Denominamos problema al diseño de un programa o rutina que compute una especificación de una función de entrada/salida. En el caso paralelo, presentamos la siguiente clasificación de problemas atendiendo a la naturaleza de las variables (común, local o privada):

- **Común-Común (CC)**: Definiremos un problema de tipo Común-Común como aquél cuyas variables paralelas tanto de entrada como de salida son comunes.

- **Común-Privado (CP)**: En un problema de este tipo, las variables paralelas de entrada son comunes, mientras que las de salida son privadas.

- **Común-Local (CL)**: Un problema se define como de tipo Común-Local cuando sus variables paralelas de entrada son comunes, mientras que la solución deberá almacenarse en una variable local a un procesador prefijado.

- **Privado-Común (PC):** En este caso, las variables paralelas de entrada del problema son privadas, y las de salida comunes.

- **Privado-Privado (PP)**: Para este tipo de problema, las variables de entrada son privadas, y la solución se almacena también en variables privadas.

- **Local-Local (LL)**: Tanto las variables de entrada como las de salida son locales.

Esta clasificación incluye solamente aquellos problemas con interés computacional.

4.5. Hipercubos dinámicos

Dada una partición $P = \{Q_0, .., Q_{m-1}\}$ de un conjunto Q decimos que los conjuntos Q_j con $j \neq i$ son los *complementarios* del conjunto Q_i. Denotaremos por $\mathbf{P}(A)$ al conjunto de todos los subconjuntos de A (partes de A).

Una **relación de *vecindad, sociedad* o *partnership*** en P es cualquier correspondencia $G = (G_i)_{i \in \{0,...,m-1\}}$ donde

$$G : Q \to P_{i=0,...,m-1}\, \mathbf{P}(Q_i)$$
$$G_j : Q \to \mathbf{P}(Q_j)$$

Definición 4.16: Al par $(P, (G_i)_{i \in \{0,...,m-1\}})$ lo denominamos *dimensión* si la familia de funciones $(G_i)_{i \in \{0,...,m-1\}}$ cumple las condiciones de Existencia, Identidad, Sobreyectividad Generalizada, Inyectividad Generalizada y Simetría que se exponen a continuación. Cuando ocurre que $q \in G_j(q')$ decimos que q es vecino de q' en la dimensión determinada por el par $(P, (G_i)_{i \in \{0,...,m-1\}})$.

1) **Existencia:** Todo elemento $q \in Q$ tiene al menos un *socio, partner* o *vecino* en cada uno de los conjuntos complementarios Q_j. Esto es, para todo $q \in Q$, la imagen mediante la función G_j cumple que

$$Q_j \cap G_j(q) \neq \emptyset.$$

Sea $q \in Q$. Puesto que P es una partición de Q, ocurre que existe un único i tal que $q \in Q_i$.

2) **Identidad**: El socio de un elemento en el conjunto al que él pertenece es él mismo.

$$G_i(q) = q \text{ si y sólo si } q \in Q_i$$

3) **Sobreyectividad Generalizada**: Para todos $i, j \in \{0, ..., m-1\}$, todo elemento en Q_j tiene un socio en Q_i.

$$\forall i,j \in \{0, ..., m-1\}: \cup_{q \in Q_i} G_j(q) = Q_j$$

4) **Inyectividad Generalizada**: La condición de Inyectividad generalizada implica que dos vértices q y q' en el mismo conjunto de la partición no pueden tener un *vecino* común.

$$\forall i, j \ \forall q, q' \in Q_i : q \neq q' \text{ se verifica que } G_j(q) \cap G_j(q') = \emptyset$$

5) Simetría: $\forall i, j \in \{0, ..., m-1\}$ y $\forall q \in Q_i : |G_j(q)| \geq 1$, esto es, $G_j(q) = \{q_0, q_1, ..., q_k\} \subseteq Q_j$ entonces existe un único $q_l \in G_j(q)$ tal que $G_i(q_l) = q$ y además se cumple que los

restantes elementos de $G_j(q)$ carecen de socio en el complementario i-ésimo. Esto es, *" s¹ l Gi(qs)= Æ*.

El elemento q_l recibe el nombre de *líder* de los socios de *q* en el complementario *j*.

Definición 4.17: |*P*| se denomina aricidad de la dimensión.

Definición 4.18: Un árbol o jerarquía de dimensiones se dice que constituye un **hipercubo dinámico** si, y sólo si se cumple que:

1) La raíz del árbol es la "dimensión trivial" (P_0, G_0) tal que $P_0 = \{Q\}$ siendo *Q* el conjunto total y G_0 es la función identidad.

2) Si el nodo *T* está etiquetado con la dimensión $(P', (G'_i)_{i \hat{\imath} \{0,...,r-1\}})$ tal que $P' = \{Q'_0, .., Q'_{r-1}\}$ es una partición del conjunto *Q'*, los hijos de *T* están etiquetados con dimensiones que particionan los conjuntos Q'_i de *P'*. Eventualmente algunos de los conjuntos Q'_i de *P'* pueden permanecer sin particionarse (en cuyo caso son hojas de la jerarquía).

Definición 4.19: Sea *H* un hipercubo dinámico. Se denomina *dimensión* del hipercubo *H* a la profundidad del árbol o jerarquía de dimensiones.

Definición 4.20: Asociado a un hipercubo dinámico es posible construir un *árbol de conjuntos*. En el árbol de conjuntos, cada nodo está etiquetado con un conjunto de alguna de las particiones de la jerarquía que determina el hipercubo dinámico. Las reglas que definen al árbol de conjuntos son:

1) La raíz del árbol de conjuntos está etiquetada con el conjunto total *Q*. El nodo raíz tiene tantos hijos como elementos hay en la primera partición hija (P, G) del nodo (P_0, G_0) de la jerarquía de dimensiones del hipercubo dinámico. Los nodos hijos están etiquetados de izquierda a derecha con los elementos Q_i de $P = \{Q_0,..., Q_{r-1}\}$ en el orden dado en la partición *P*.

2) Dado un nodo etiquetado por un conjunto *Q* que viene de una particion (P, G) del hipercubo dinámico, esto es, *Q* es uno de los elementos de *P*. Este nodo tiene tantos hijos como elementos existen en la particion *P'* que refina *Q* en *P* (si es que existe). Los nodos hijos están etiquetados con los elementos Q_i de $P' = \{Q'_0,..., Q'_{s-1}\}$ en el orden dado en la partición *P'*.

Definición 4.21: Llamamos *árbol de pesos normalizado* al que resulta de etiquetar cada nodo *T* del árbol de conjuntos etiquetado con el conjunto *Q*, con el número real $|Q|/|father(Q)|$ siendo *father(Q)* el conjunto que etiqueta al nodo padre de *T*. (Se asume que *father(Root) = Root*).

La secuencia de pesos $w_0,..., w_{r-1}$ de los hijos etiquetados con conjuntos $Q_0,..., Q_{r-1}$ de un nodo dado etiquetado con conjunto *Q* verifican la ecuación:

$$\text{(Ec. 4.1)} \qquad w_0 + ... + w_{r-1} = 1$$

y contiene por tanto las proporciones w_i de elementos de *Q* que pertenecen a Q_i.

Definición 4.22: Si los pesos elegidos w_i en el árbol de pesos mantienen la proporción:

$$\text{(Ec. 4.2)} \qquad w_0 / |Q_0| = ... = w_i / |Q_i| = ... = w_{r-1} / |Q_{r-1}|$$

sin que se tenga que cumplir la ecuación de normalización (Ec. 4.1) hablaremos simplemente de un *árbol de pesos*.

Conocida la cardinalidad del conjunto inicial *Q*, el árbol de pesos contiene toda la información necesaria para construir las particiones de la jerarquía asociada con el hipercubo dinámico. La primera partición del hipercubo estaría constituída por *Q*. Dada una partición, se refina cada uno de los conjuntos de la misma en subconjuntos de tamaño proporcional a los pesos de los nodos hijos del correspondiente nodo en el árbol de pesos.

Definición 4.23: Cuando construimos el hipercubo dinámico *H* asociado a un árbol de pesos *W*, decimos que *H* es el *hipercubo dinámico ponderado* por *W*.

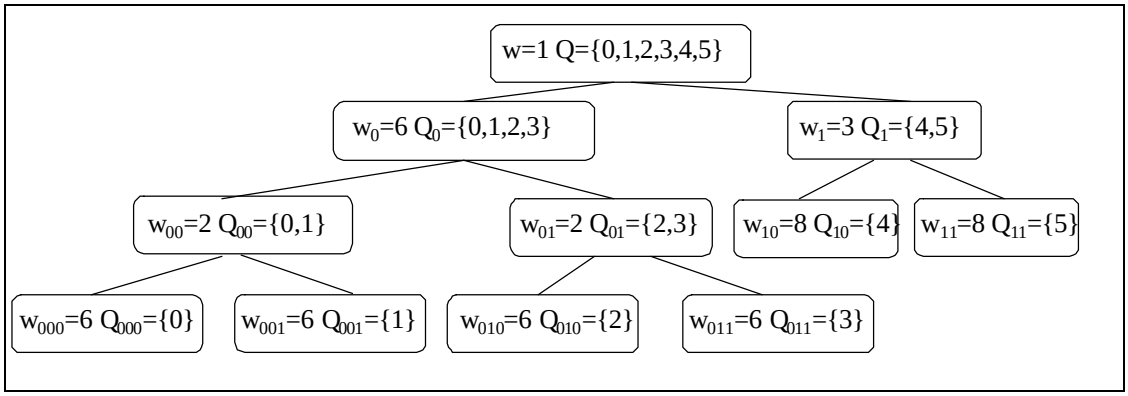

Figura 4.17 Un árbol de pesos

La Figura 4.17 muestra un árbol de pesos. Los nodos están etiquetados con los pesos w_i y los conjuntos del árbol de conjuntos.

4.5.1. Hipercubo binario

Como es bien conocido, un hipercubo binario de dimensión *d* es un grafo con $p=2^d$ nodos. Esto es, $Q = \{0,1,..., 2^d-1\}$. La dimensión *0* define una partición $P^0=\{Q^0_0, Q^0_1\}$ del conjunto $Q=\{0, 1, ... 2^d-1\}$ en dos subconjuntos Q^0_0, y Q^0_1 formados respectivamente por los nodos pares e impares. En general, cada dimensión (en el sentido clásico del término) $i \in \{0..d-1\}$ determina una partición $P^i = \{Q^i_0, , Q^i_1\}$ donde

$Q^i_0 = \{n \in Q$ / el bit i-ésimo de n es cero$\}$. $Q^i_1 = \{n \in Q$ / el bit i-ésimo de n es uno$\}$

El nodo *q* está conectado en dimensión $i \in \{0..d-1\}$ con el nodo cuya representación binaria difiere de *q* en el i-ésimo bit. Esto es,

$G^i : Q \to P_{s=0,1}\ Q_s\ i = 0,...,d-1$.

$G^i_s : Q \to Q_s$

$G^i_s(n) = n_d n_{d-1} ... s ... n_o$

Obsérvese que en cualquier dimensión *i* un hipercubo binario la aricidad es $|P^i|=2$.

Es trivial comprobar que, cualquiera que sea $i \in \{0..d-1\}$, los pares $(P^i, (G^i_s)_{s \in \{0,1\}})$ cumplen las condiciones de existencia, identidad, sobreyectividad generalizada, inyectividad generalizada y simetría que caracterizan el concepto de dimensión.

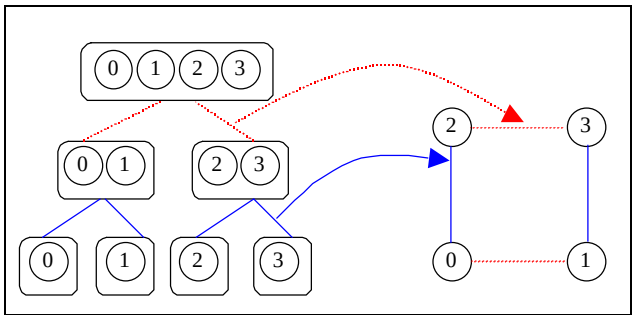

Figura 4.18 Una jerarquía de dimensiones para un hipercubo binario de dimensión 2

La Figura 4.18 muestra una jerarquía de dimensiones correspondiente a las particiones $\{Q\}$, P^1, P^0 y el correspondiente hipercubo dinámico, que en este caso se corresponde con un hipercubo binario de dimensión dos. Obsérvese que la Definición 4.16 de "dimensión" de hipercubo dinámico coincide con el concepto clásico de dimensión de un hipercubo binario.

4.5.2. Hipercubo k-ario

Un hipercubo k-ario, siendo k un número natural, de dimensión d es un grafo con $p=k^d$ nodos. La dimensión *0* define una partición $P^0=\{Q^0_0,..., Q^0_{k-1}\}$ del conjunto $Q=\{0, 1, ... k^d-1\}$ en k subconjuntos Q^0_s formados respectivamente por los nodos cuyo último dígito k-ario es s. En general, cada dimensión (en el sentido clásico del término) $i \in \{0..d-1\}$ determina una partición $P^i = \{Q^i_0 ,..., Q^i_{k-1}\}$ donde

$$Q^i_s = \{n \in Q \,/\, el\ bit\ i\text{-}ésimo\ de\ n\ es\ s\} \text{ con } s \in \{0..k-1\}.$$

El nodo q está conectado en dimensión $i \in \{0..d-1\}$ con los nodos cuya representación binaria difiere de q en el i-ésimo k-dígito. Esto es,

$G^i : Q \to P_{s=0,..k-1}\ Q_s \quad i = 0,...,d-1.$

$G^i_s : Q \to Q_s$

$G^i_s(n) = n_d n_{d-1} ... s ... n_o$

Obsérvese que en cualquier dimensión i un hipercubo k-ario la aricidad es $|P^i| = k$.

Es trivial comprobar que, cualquiera que sea $i \in \{0..d-1\}$, los pares $(P^i, (G^i_s)_{s \in \{0,...,k-1\}})$ cumplen las condiciones de existencia, identidad, sobreyectividad generalizada, inyectividad generalizada y simetría que caracterizan el concepto de "dimensión".

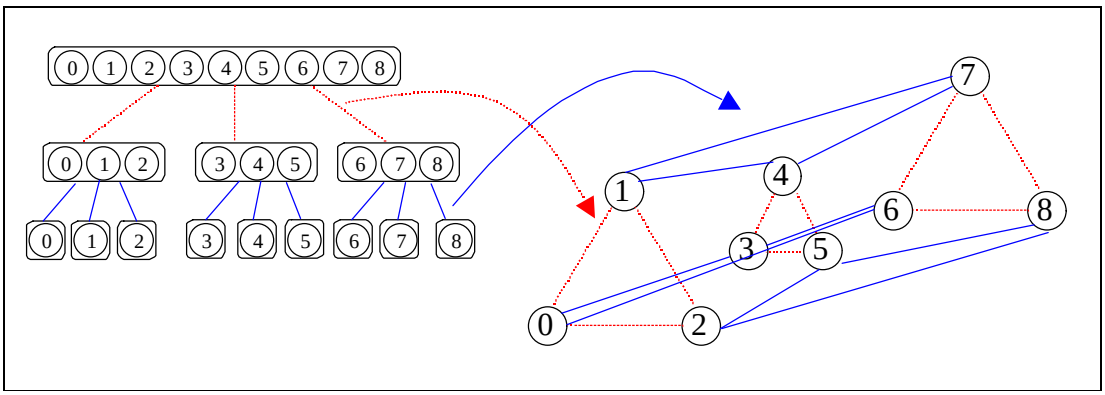

Figura 4.19 Una jerarquía de dimensiones para un hipercubo ternario de dimensión 2

La Figura 4.19 muestra una jerarquía de dimensiones correspondiente a las particiones $\{Q\}$, P^1, P^0 y el correspondiente hipercubo dinámico, que en este caso se corresponde con un hipercubo ternario de dimensión 2. Las aristas de trazo discontinuo corresponden a la dimensión uno (P^1, G^1) y las de trazo continuo a la dimensión cero

(P^0, G^0). Observamos que cada nodo tiene 3 vecinos en cada dimensión (uno de los cuales es él mismo). Cada nodo tiene dos aristas de trazo discontinuo y dos de trazo continuo. Obsérvese que la Definición 4.16 de "dimensión" de hipercubo dinámico coincide con el concepto clásico de dimensión de un hipercubo *k*-ario.

4.5.3. Hipercubo dinámico

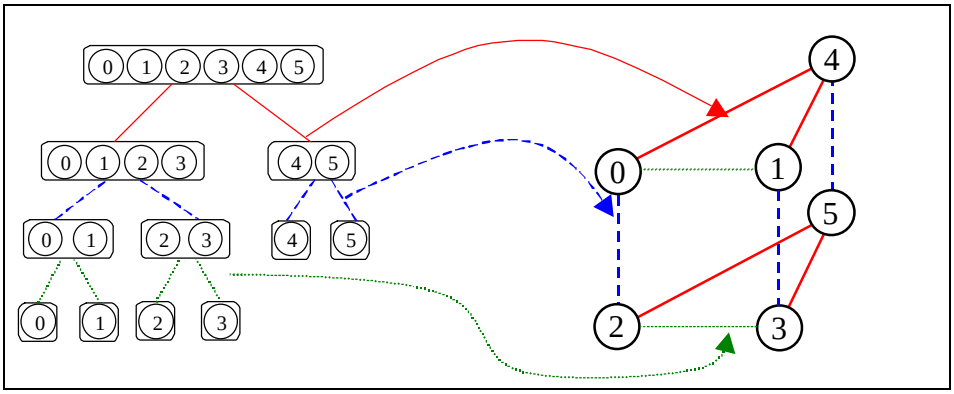

Figura 4.20 Una jerarquía de dimensiones para un hipercubo dinámico de dimensión 3

La Figura 4.20 muestra una jerarquía de dimensiones $(P^i, (G^i_s)_{s \hat{I} \{0,1,2\}})$ dada por las particiones P^i (en la parte izquierda de la figura) y las funciones de vecindad $(G^i_s)_{s \hat{I} \{0,1,2\}}$ (en la parte derecha) de un hipercubo dinámico de dimensión 3 (profundidad de la jerarquía de dimensiones). Se trata del hipercubo dinámico asociado con el árbol de pesos que presentamos en la Figura 4.17. En este ejemplo, como en los hipercubos binarios, los nodos $R^i \hat{I} P^i$ son particionados en dos conjuntos $R^i = Q^i_o \grave{E} Q^i_1$. Sin embargo, a diferencia de los dos ejemplos anteriores, los conjuntos Q^i_o y Q^i_1 pueden diferir en sus cardinales. La consideración de este tipo de topologías se justifica por la aparición de esta clase de jerarquías en problemas irregulares. Las funciones G^i representadas corresponden a una de las familias de funciones de vecindad utilizadas en la librería *La Laguna C*. En la figura las aristas representan las funciones de vecindad. Las aristas de trazo continuo corresponden al primer nivel de la jerarquía, la de trazo discontinuo grueso al segundo nivel de la jerarquía y la de trazo discontinuo fino al tercer nivel. Más explícitamente las funciones G^i vienen dadas por las ecuaciones:

Supuesto $|Q^i_o| \geq |Q^i_1|$ y que $|Q^i_o|$ es múltiplo de $|Q^i_1|$

$$G^i_1(n) = (n - first(Q^i_0))/|Q^i_1| + first(Q^i_1) \text{ si } n \hat{I} Q^i_0$$
$$G^i_1(n) = n \text{ si } n \hat{I} Q^i_1$$
$$G^i_0(n) = \{ j \text{ tales que } (j - first(Q^i_0))/|Q^i_1| + first(Q^i_1) = n \} \text{ si } n \hat{I} Q^i_1$$
$$G^i_0(n) = n \text{ si } n \hat{I} Q^i_0$$

Caso contrario, $|Q^i_1| \geq |Q^i_0|$, asumiendo también que que $|Q^i_1|$ es múltiplo de $|Q^i_0|$ las funciones G^i se definen como:

$$G^i_1(n) = (n - first(Q^i_1))/|Q^i_0| + first(Q^i_0) \text{ si } n \hat{I} Q^i_0$$
$$G^i_1(n) = n \text{ si } n \hat{I} Q^i_1$$
$$G^i_0(n) = \{ j \text{ tales que } (j - first(Q^i_1))/|Q^i_0| + first(Q^i_0) = n \} \text{ si } n \hat{I} Q^i_1$$
$$G^i_0(n) = n \text{ si } n \hat{I} Q^i_0$$

En el caso en que la hipótesis de divisibilidad exacta entre los cardinales de los conjuntos no se cumpla, las expresiones de las funciones de vecindad pueden

modificarse de manera conveniente. Sin embargo en aras de la claridad de la exposición, omitimos los detalles de esta modificación.

Obsérvese que a diferencia de los ejemplos anteriores en los que, fijada la dimensión, el socio de un elemento en un complementario dado es único, en la Figura 4.20 los nodos 4 y 5 tienen ambos dos socios en la primera dimensión en el conjunto complementario. La propiedad de inyectividad no es violada, ya que, por ejemplo, los vértices 0 y 1 tiene como socio común a 4 y los vértices 2 y 3 al 5.

4.6. Implementación de algoritmos divide y vencerás en Computación Colectiva Común (CCC) mediante Hipercubos Dinámicos

La Figura 4.21 presenta el esquema general de un algoritmo divide y vencerás secuencial binario en el que para obtener la solución de un problema *x*, el problema es dividido en subproblemas x_0 y x_1 a los cuales se les aplica recursivamente el mismo procedimiento de resolución. Este proceso recursivo finaliza cuando los problemas tienen un tamaño suficientemente pequeño como para ser considerados sencillos, en cuyo caso se utiliza otro procedimiento (*conquer* en la figura) para su resolución.

```
procedure DC(x: Problem; r: Result);
...
if trivial(x) then conquer(x, r)
else
   divide(x, x0, x1);
   DC(x0, r0);
   DC(x1, r1);
   combine(r, r0, r1);
endif;
...
```

Figura 4.21 Esquema general de un algoritmo divide y vencerás secuencial

Este tipo de problemas es particularmente adecuado para ser abordados en paralelo porque frecuentemente los subproblemas en que se divide el problema original pueden ser resueltos de forma independiente en paralelo.

```
1 procedure pDC(x: Problem; r: Result);
2 ...
3 if trivial(x) then conquer(x, r)
4 else
5    divide(x, x0, x1);
6    PARALLEL(pDC(x0, r0), r0, pDC(x1 , r1), r1);
7    combine(r, r0, r1);
8 endif;
9 ...
```

Figura 4.22 Esquema general de un algoritmo divide y vencerás paralelo

El modelo de computación colectiva común es fácilmente aplicable a la resolución en paralelo de problemas siguiendo la aproximación divide y vencerás. En la Figura 4.22 se presenta esta aproximación. Supongamos que se desea resolver el problema *x* en paralelo de forma que todos los procesadores obtengan la solución *r* del problema. También asumiremos que el problema a resolver es de tipo Común-Común, por lo tanto, todos los procesadores disponen de los datos de entrada. Así la variable paralela *x* que contiene los datos de entrada del problema es una variable común.

La llamada a la función de división $PARALLEL \in Div$ de la línea 6 provoca la activación en paralelo de dos tareas del mismo tipo (*pDC*) para resolver cada uno de los

dos subproblemas en que se ha dividido el problema original. La llamada divide el conjunto de procesadores hoja actual en dos subconjuntos. Cada uno de los subconjuntos de procesadores resuelve en paralelo un subproblema x_i, y al finalizar la división todos los procesadores del conjunto de partida obtienen la solución del problema en la variable paralela común r por combinación de las soluciones parciales r_0 y r_1. Esta forma de proceder se aplica recursivamente hasta que no existen procesadores disponibles. En este caso el subproblema en cuestión se resuelve secuencialmente.

Dado que se trata de un problema de tipo Común-Común, todos los procesadores disponen del problema inicial. Todos los procesadores comienzan perteneciendo al conjunto raíz, y replicando la ejecución del código hasta que se alcanza la primera función de división.

En general, cada vez que un conjunto de procesadores se divide como consecuencia de la ejecución de la función de división PARALLEL se crea una nueva dimensión (P^i, G) en el sentido de la Definición 4.16. La aricidad de la dimensión (Definición 4.17) es igual al número de tareas en paralelo asignadas por la función PARALLEL. La creación de la partición o dimensión puede realizarse en tiempo proporcional al número de tareas solicitado (aricidad). En el ejemplo de la Figura 4.22 el conjunto inicial Q de procesadores se divide en dos subconjuntos $P=\{Q_0, Q_1\}$ en tiempo constante. Los procesadores en Q_0 realizan la tarea *pDC(x0, r0)*, y los de Q_1 ejecutan *pDC(x1, r1)*. Al tratarse de un problema de tipo Común-Común, cada uno de los procesadores q de Q_0 tiene almacenado en la variable común r_0 el resultado de *pDC(x0, r0)*. Al finalizar la ejecución de las tareas paralelas, los procesadores de Q_0 y Q_1 utilizan la relación de sociedad definida por G para comunicar a sus procesadores socios los resultados r_0 y r_1 generados por las tareas *pDC(x0, r0)* y *pDC(x1, r1)*, que figuran como parámetros en la llamada a la función PARALLEL en la línea 6 del código de la Figura 4.22. Concretamente, cada procesador q de Q_0 envía el resultado r_0 a sus procesadores socios en $G_1(q)$. Simétricamente, cada procesador q' de Q_1 envía el resultado r_1 a sus socios en $G_0(q')$. El coste de esta fase viene dominado por el tiempo de comunicaciones:

$$D*max\{|r1|, |r2|\}$$

donde $|r1|$ y $|r2|$ denotan la longitud (por ejemplo en bytes) de los resultados.

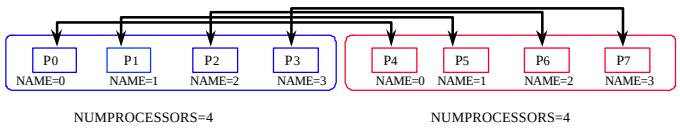

Figura 4.23 Fase de división. Cada uno de los *8* procesadores elige un *socio* en conjunto complementario

Las diferentes dimensiones del hipercubo dinámico generado por el problema divide y vencerás de la Figura 4.22 se corresponden con las sucesivas llamadas a la función de división PARALLEL. Supuesto que existe un número suficiente de procesadores, la dimensión del hipercubo resultante dependerá de la profundidad de anidamiento de dichas llamadas.

El tiempo *Time(pDC(x,r))* invertido en una llamada a *pDC(x,r)* cumple la fórmula recursiva:

$$Time(pDC(x,r)) = Time(divide(x, x0, x1)) + \\ + max(Time(pdC(x0, r0), Time(pdC(x1, r1)) + D*max\{|r1|, |r2|\}.$$

La Figura 4.23 muestra un conjunto con *8* procesadores que se divide en dos subconjuntos de *4* como consecuencia de la primera llamada a la función de división *PARALLEL*, y las relaciones *de sociedad* establecidas entre los procesadores de los dos grupos cuando se utiliza la política explicada en el epígrafe 4.5.1. La segunda llamada recursiva a la función *PARALLEL* crea una nueva dimensión cuyas relaciones de sociedad se muestran en la Figura 4.24. Obsérvese que la tercera llamada recursiva da lugar a un conjunto de relaciones entre socios que se conforman según un hipercubo binario perfecto.

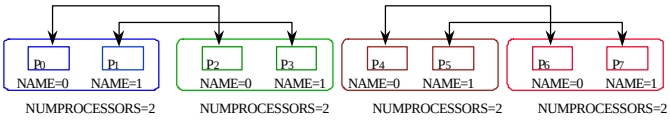

Figura 4.24 Los dos grupos se escinden de nuevo en otros dos

En la Figura 4.25 presentamos el pseudocódigo correspondiente a la implementación de una función de división PARALLEL. Las llamadas PUSHPARALLELCONTEXT y POPPARALLELCONTEXT que aparecen en las líneas 2 y 19 de la figura se encargan de almacenar y recuperar respectivamente el

```
 1 if (NUMPROCESSORS > 1) {
 2     PUSHPARALLELCONTEXT;
 3     /* Fase de división del conjunto */
 4     Calcular:
 5       NUMPROCESSORS,
 6       NAME,
 7       NUMPARTNERS, PARTNERS(G),
 8       SUBSETS(P={Q0, Q1})
 9      /* Intercambio de resultados */
10     if (IN Q0) {
11       pDC(x0, r0);
12       SWAP(PARTNER, r1, r0);
13     }
14     else {
15       pDC(x1, r1);
16       SWAP(PARTNER, r0, r1);
17     }
18     /* Fase de reunificación */
19     POPPARALLELCONTEXT;
20 }
21 else {
22   pDC(x0, r0);
23   pDC(x1, r1);
24 }
```

Figura 4.25 Expansión de una función de división PARALLEL binaria

contexto paralelo de la ejecución. Este contexto está definido por los valores de las variables NAME, NUMPROCESSORS, y la función de sociedad, *G*.

La fase de división del conjunto hoja de procesadores conlleva la asignación de los procesadores del conjunto hoja a los subconjuntos creados Q_0 y Q_1. Existen diferentes políticas para realizar esta asignación, como ya comentamos en el Capítulo 2 de esta memoria.

Dependiendo del conjunto (Q_0 ó Q_1) al que pertenezca el procesador considerado, éste realizará la llamada $pDC(x_0, r_0)$ ó $pDC(x_1, r_1)$. A la finalización de la llamada, el

procesador intercambia los resultados con su socio según *G* en el conjunto complementario.

Si no hay suficientes procesadores para proceder a la división del conjunto hoja (NUMPROCESSORS es uno), las dos llamadas recursivas correspondientes a las tareas, se ejecutan en secuencia (líneas 22 y 23 de la Figura 4.25).

Una alternativa a este comportamiento cuando se agotan los procesadores disponibles consiste en que el programador suministre un código secuencial para ser

```
 1 procedure pDC(x: Problem; r: Result);
 2 ...
 3 if (NUMPROCESSORS > 1) then
 4   if trivial(x) then conquer(x, r)
 5   else
 6      divide(x, x0, x1);
 7      PAR(pDC(x0, r0), r0, pDC(x1 , r1), r1);
 8      combine(r, r0, r1);
 9   endif;
10 else
11   sDC(x, r);
12 endif;
13 ...
```

Figura 4.26 Un algoritmo divide y vencerás paralelo con alternativa secuencial

utilizado cuando esto ocurre (línea 11 de la Figura 4.26).

En *llc* existen las dos formas de función de división, que aquí hemos presentado como PARALLEL y PAR. En *La Laguna C* se les llama respectivamente PARVIRTUAL y PAR.

La condición de Sobreyectividad Generalizada exigida a la función de sociedad, *G* garantiza que todo procesador recibe el resultado de la tarea realizada por los procesadores del conjunto complementario.

Cada resultado correspondiente a una tarea es recibido por cada procesador sólo una vez como consecuencia de la propiedad de Inyectividad Generalizada

4.6.1. Ejemplos

En los siguientes subepígrafes presentamos la implementación en *llc* de tres algoritmos divide y vencerás, la Transformada rápida de Fourier, el cálculo de la envolvente convexa de una nube de puntos y un algoritmo de ordenación. Se mostrará también la implementación siguiendo las ideas del modelo de computación colectiva común de un algoritmo que realiza la búsqueda de una clave en un vector desordenado.

Los resultados computacionales de algunos de estos algoritmos se presentan y comentan en el Capítulo 6 de esta memoria.

4.6.1.1. La transformada rápida de Fourier

La Figura 4.27 presenta una solución divide y vencerás del cálculo de la transformada rápida de Fourier (FFT) implementada utilizando *La Laguna C*.
El algoritmo toma como entrada un vector de complejos que se almacena en la variable común *a* y el número de elementos en dicho vector, *N*, devolviendo en el vector común *b* la señal transformada. El algoritmo se basa en que el cómputo de la transformada de los términos pares y los impares es independiente y puede realizarse por tanto en paralelo. En *llc*, la variable común NUMPROCESSORS almacena en cada instante el número de procesadores disponibles en el conjunto de procesadores hoja actual. Así, el

```
1 void parFFT(Complex *a, Complex *b, int N) {
 2   Complex *a2, *A2;       /* Even terms */
 3   Complex *a1, *A1;       /* Odd terms */
 4   int n, size;
 5
 6   if(NUMPROCESSORS > 1) {
 7     if (N == 1) {           /* Trivial problem */
 8       b[0].re = a[0].re;
 9       b[0].im = a[0].im;
10     }
11     else {
12       n = N / 2;
13       Odd_and_even(a, a2, a1, n);
14       size = n * sizeof(Complex);
15       PAR(parFFT(a2,A2,n), A2, size, parFFT(a1,A1,n), A1, size);
16       Combine(b, A2, A1, n);
17     }
18   }
19   else
20     seqFFT(a, b, N);
21 }
```

Figura 4.27 La Transformada rápida de Fourier en *llc*

```
void Odd_and_even(Complex *a, Complex *a2, Complex
*a1, int n) {
  int size;

  size = n * sizeof(Complex);
  a2 = (Complex *)mymalloc(4 * size);
  A2 = a2 + n;
  a1 = A2 + n;
  A1 = a1 + n;
  for (i = 0, j = 0; i < n; i++, j = 2 * i) {
    a2[i] = a[j];
    a1[i] = a[j+1];
  }
}
```

Figura 4.28 La fase de división

algoritmo comienza por determinar en la línea 6 si hay más de un procesador en el conjunto hoja. En caso contrario, se invoca un procedimiento secuencial *seqFFT* que ha de ser suministrado por el usuario. En caso de disponer de más de un procesador en el conjunto, se comprueba en la línea 7 si el problema a resolver es trivial, en cuyo caso la señal transformada es igual a la original. Si el problema no es trivial, la llamada de la línea 13 a la función *Odd_and_even()* (Figura 4.28) descompone la señal inicial *a*, en sus componentes pares e impares, almacenándolas en los vectores *a2* y *a1* respectivamente. A continuación se utiliza el constructo PAR de *llc*, que es una función de división, para realizar las dos llamadas paralelas y recursivas que se encargan de calcular las transformadas de los términos pares e impares, devolviéndolas en los vectores comunes *A2* y *A1* respectivamente, ambos de tamaño *size*. Cuando un conjunto de procesadores hoja ejecuta la llamada PAR de la línea 15 se crean dos nuevos conjuntos hojas que ejecutarán en paralelo cada una de las dos llamadas recursivas a *parFFT()*. El constructo PAR se encarga en *llc* de actualizar el valor de NUMPROCESSORS para los dos subconjuntos de acuerdo con la política de distribución de procesadores en grupos implementada.

La Figura 4.28 presenta el código de la función *Odd_and_even()* que divide el vector original en sus componentes pares e impares mientras que la Figura 4.29 muestra el código de *Combine()*, la función que combina los resultados parciales.

```
void Combine(Complex *b, Complex *A2, Complex *A1, int n) {
  int i;
  Complex wi;

  for (i = 0; i < n; i++) {
    wi.re = cos(2.0L * PI * i / N);
    wi.im = sin(2.0L * PI * i / N);
    A[i].re = A2[i].re + (wi.re * A1[i].re - wi.im * A1[i].im);
    A[i].im = A2[i].im + (wi.re * A1[i].im + wi.im * A1[i].re);
    A[i+n].re = A2[i].re - (wi.re * A1[i].re - wi.im * A1[i].im);
    A[i+n].im = A2[i].im - (wi.re * A1[i].im + wi.im * A1[i].re);
  }
}
```

Figura 4.29 La fase de combinación

4.6.1.2. Cálculo de la envoltura convexa: Quickhull

La Figura 4.30 presenta la implementación en *llc* del algoritmo Quickhull [Pre85] para el cálculo de la envoltura convexa de una nube de puntos que ya se introdujo en el Capítulo 3.

```
 1 void qh(int *h, int *N, int max) {
 2   int lmax, rmax, ls, rs, *lh, *rh;
 3
 4   if (trivial(h, N, max))
 5     *N = solve(h, max);
 6   else {
 7     divide(h, *N, max, &lh, &ls, &lmax, &rh, &rs, &rmax);
 8     PARVIRTUAL(qh(lh,&ls,lmax), lh, ls, qh(rh,&rs,rmax), rh, rs);
 9     *N = combine(h, lh, ls, rh, rs);
10   }
11 }
```

Figura 4.30 Quickhull en *llc*

La Figura 4.31 muestra un ejemplo gráfico del modo de proceder del algoritmo, que comienza por determinar dos puntos, P1 y P2 con máxima distancia entre sí que con seguridad pertenecerán a la envoltura convexa. El problema original se puede dividir en dos subproblemas simétricos, consistentes en hallar la envoltura superior e inferior con respecto a la recta determinada por P1P2. Consideremos sólo uno de estos subproblemas. El algoritmo determina el punto P3, con máxima distancia a la recta P1P2. Este punto P3 ha de pertenecer a la envoltura convexa dado que una recta paralela a P1P2 que desde el infinito se acercara a ésta intersectaría P3 como primer punto de la nube. Por otra parte, los puntos que quedan en el interior del triángulo P1P3P2 no pertenecen a la envoltura convexa, mientras que los puntos que quedan fuera de este triángulo sí son candidatos a pertenecer a la misma. Así pues el algoritmo procede a dividir el problema original en dos subproblemas (almacenados en las variables comunes *lh* y *rh*) definidos por las rectas P1P3 y P2P3 y procede recursivamente. La envoltura superior estará constituida por los puntos: P1, los puntos de *lh* que pertenezcan a la envoltura, el punto P3, los puntos de *rh* que pertenezcan a la envoltura y el punto P2. De modo totalmente simétrico se computa la envoltura inferior.

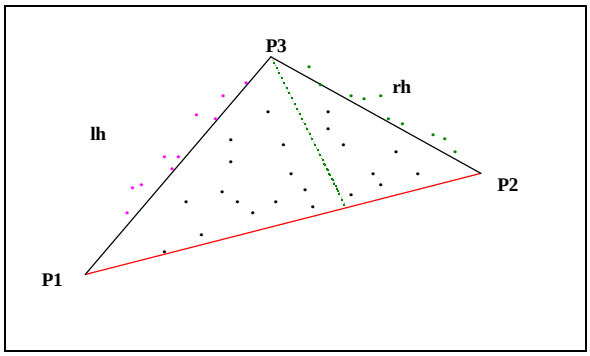

Figura 4.31 Cálculo de la envoltura convexa de una nube de puntos

En el código de la Figura 4.30, el vector común *h* almacena los índices de la nube de puntos. Inicialmente, *h[0]* y *h[N]* contienen los índices correspondientes a los puntos P1 y P2, mientras que *max* almacena el índice del punto con máxima distancia a la recta P1P2. Al retornar de la llamada a *qh*, el vector *h* contendrá los puntos pertenecientes a la envolvente, y el valor de *N* habrá cambiado, actualizándose con el número de puntos de la misma. La llamada a la función *divide()*, almacena los *ls* puntos que quedan por encima de la recta P1P3 en el vector *lh*, y devuelve en *lmax* el índice del punto con mayor distancia a esa recta. De modo análogo almacena en *rh* los *rs* puntos que quedan por encima de P2P3 y devuelve en *rmax* el índice del punto más alejado de esa recta. La función *combine()* se encarga de colocar en el vector *h* los puntos correspondientes a la envolvente izquierda y derecha, así como de actualizar el valor de *N*.

En el ámbito de la función *qh()* la variable *N* es común a todos los procesadores del conjunto hoja que ejecuta la función. Cada llamada a PARVIRTUAL en la línea 8 es una función de división que divide el conjunto hoja en dos subconjuntos para ejecutar como subtareas las dos llamadas recursivas a *qh()* que toma como parámetros. A cada subtarea se le pasan nuevos valores de *N* (*lmax* y *rmax* respectivamente) que son valores comunes para cada nuevo conjunto hoja respectivamente.

La llamada a la función de división común PARVIRTUAL utilizada en la línea 8 del código es otra forma de implementar una sentencia de asignación de procesadores en *llc*, y se encarga de realizar en paralelo las dos llamadas recursivas para calcular las envolventes izquierda y derecha. Al realizar la llamada a PARVIRTUAL, si hay suficientes procesadores disponibles en el conjunto de procesadores hoja (más de uno), la ejecución de la llamada dividirá el conjunto en dos subconjuntos. Los procesadores del primer subconjunto ejecutarán la llamada a *qh* correspondiente a la envolvente izquierda (primer parámetro de la llamada a PARVIRTUAL) y los del segundo subconjunto realizarán la llamada correspondiente a la envolvente derecha (cuarto parámetro de la llamada). En caso de no contar con suficientes procesadores para dividir el conjunto de procesadores, la implementación de PARVIRTUAL se encarga de realizar las dos llamadas recursivas en secuencia. Los parámetros *lh* y *ls* de la llamada se utilizan para indicar que la envolvente izquierda está formada por los *ls* puntos almacenados en la dirección apuntada por *lh*. De forma análoga se utilizan *rh* y *rs* para devolver la envolvente derecha. Al finalizar una llamada a PARVIRTUAL los dos subconjuntos de procesadores intercambian sus resultados y se reintegran al conjunto de procesadores original del que formaban parte.

La virtualización de procesadores llevada a cabo en *llc*, permite al programador ignorar el número de procesadores realmente disponibles en la máquina con la que trabaja. Los programas diseñados en *llc* son independientes de esta característica de la

arquitectura y pueden ser ejecutados sin cambio alguno en una máquina monoprocesador o en una máquina masivamente paralela. No obstante, la característica de virtualización pudiera introducir alguna ineficiencia en las prestaciones del algoritmo, así pues, el programador puede optar por no utilizar esta característica.

4.6.1.3. Optimizaciones en la utilización de la memoria en el cálculo de la envoltura convexa

Una implementación directa del algoritmo Quickhull alojará dinámicamente vectores para las envolturas izquierda y derecha. Dado que el número de puntos que pertenecerán a dichas envolventes es desconocido a priori, el tamaño solicitado para los vectores debe prever el peor caso. Esto significa que habrá que alojar vectores de tamaño igual al de la nube de puntos de entrada. Por ejemplo, en el caso de que los *N* puntos de la nube inicial estén uniformemente distribuidos en un semicírculo, la cantidad de memoria requerida por el algoritmo será:

$$2N + 4N/2 + 8N/4 + ... + 2N = 2(logN)N$$

Es posible reducir el consumo de memoria del algoritmo a un solo vector de tamaño *N* sin introducir ningún cambio en la complejidad computacional del algoritmo. La idea para esta mejora consiste en reutilizar el vector inicial. Operando de manera análoga a como lo hace el conocido procedimiento *partition* para el Quicksort, los puntos que caen a la izquierda de la recta P1-P3 (ver Figura 4.31) se compactan en la parte izquierda del vector, mientras que los puntos a la derecha de la recta P3-P2 se colocan en la parte derecha del vector. Hay un bucle de recorrido del vector para cada una de estas compactaciones. Cuando en el recorrido del bucle se encuentra un punto que debe ser colocado en la nube opuesta, se intercambia con el correspondiente punto 'saliente' del otro bucle. De esta manera el consumo se reduce a un vector de tamaño *N*.

La Figura 4.32 compara las aceleraciones de los algoritmos paralelo y secuencial utilizando alojamiento dinámico de memoria (etiqueta malloc()) y minimizando el consumo de memoria a través de la reutilización del vector de la nube de puntos (etiqueta vector). El algoritmo que reutiliza el vector obtiene mejor aceleración debido a la sobrecarga computacional que supone la readministración del vector. Para tamaños mayores que 1M en los datos de entrada no fue posible ejecutar el algoritmo que utiliza memoria dinámica en el Cray T3D, mientras que ello sí fue posible para el otro algoritmo.

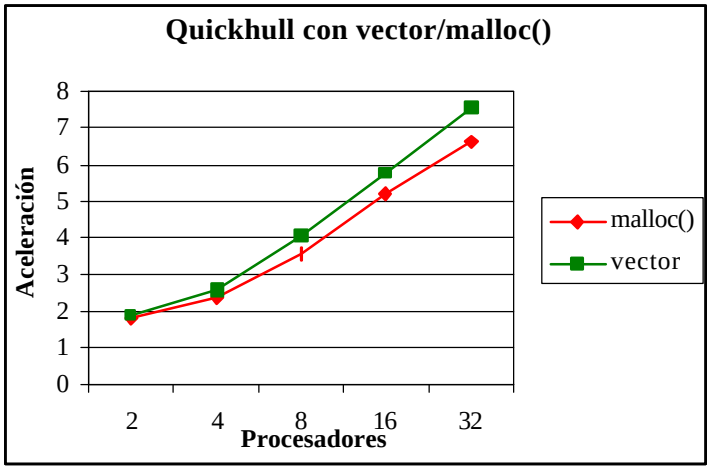

Figura 4.32 Quickhull con vector/malloc Cray T3D. Tamaño: 1M puntos

4.6.1.4. Ordenación: Quicksort

Veamos a continuación un tercer ejemplo de algoritmo divide y vencerás colectivo común. Presentamos en este caso una paralelización del Quicksort [Hoa61], en dos versiones: con virtualización de procesadores (Figura 4.34) y sin esta posibilidad (Figura 4.33).

```
 1 void qs(int first, int last) {
 2   int i, j, size1, size2;
 3
 4   if (NUMPROCESSORS > 1) {
 5     if (first < last) {
 6       partition(&i, &j, first, last);
 7       size1 = (j - first + 1) * sizeof(int);
 8       size2 = (last – i + 1) * sizeof(int);
 9       PAR(qs(first, j), v + first, size1,
10           qs(i, last), v + i, size2);
11     }
12   }
13   else
14     seqquicksort(first, last);
15 }
```

Figura 4.33 El Quicksort en *llc* usando la función de división PAR

La función *qs* que se muestra en ambas figuras toma como parámetros (*first* y *last*) los índices del vector global y común *v* a ordenar (si el tamaño del vector es SIZE, la llamada desde el programa principal se realiza como *qs(0, SIZE-1)*). El procedimiento *partition* [Hoa61] se encarga en ambos códigos de dividir el vector a ordenar en dos subvectores. El procedimiento elige al azar un elemento como pivote, y divide el vector en dos segmentos de modo que en el primero, todos los elementos serán menores que el pivote, mientras que en el segundo, todos los elementos serán mayores que él. Las variables comunes *i* y *j* que aparecen como parámetros del procedimiento *partition* junto con los extremos del intervalo a particionar (*first* y *last*) marcan los límites de la partición del vector de entrada. Las llamadas a las funciones de división común PAR y PARVIRTUAL en las figuras se encargan de realizar recursivamente y en paralelo las llamadas a *qs* que se encargan de ordenar las dos mitades en que ha sido dividido el vector. Observamos de nuevo en el caso del algoritmo sin virtualización (Figura 4.33) que el programador ha de suministrar un procedimiento secuencial *seqquicksort* para el caso en que el sistema no disponga de procesadores suficientes para realizar las dos llamadas en paralelo.

```
void qs(int first, int last) {
  int i, j, size1, size2;

  if (first < last) {
    partition(&i, &j, first, last);
    size1 = (j - first + 1) * sizeof(int);
    size2 = (last – i + 1) * sizeof(int);
    PARVIRTUAL(qs(first, j), v + first, size1,
               qs(i, last), v + i, size2);
  }
}
```

Figura 4.34 El Quicksort en *llc* usando virtualización de procesadores

La capacidad de utilizar un algoritmo secuencial cuando no hay virtualización de procesadores, posibilita que el usuario pueda suministrar un algoritmo secuencial cuyo comportamiento sea mejor que sucesivas llamadas secuencializadas al algoritmo

paralelo, que es lo que en último término realiza el constructo con virtualización, PARVIRTUAL.

```
 1 INITCRONO(start);
 2 i0 = 0;
 3 rate = SIZE / NUMPROCESSORS;
 4 remaining = SIZE % NUMPROCESSORS;
 5 first = rate * NAME;
 6 last = first + rate;
 7 if (IAMTHELASTPROCESSOR)
 8   last = SIZE;
 9 for (i = first; ((i < last) && (array[i] != KEY)); i++);
10 if (i < last)
11   REDUCEBYMIN(i0, i);
12 else
13   REDUCEBYMIN(i0, INFINITY);
14 ENDCRONO(finish);
15 GPRINTF("%d: time: (%lf)\n", NAME, finish-start);
16 if (i0 == INFINITY) GPRINTF("%d: NOT Found\n", NAME);
17 else GPRINTF("%d: Found Position: %d\n", NAME, i0);
```

Figura 4.35 Búsqueda de KEY en un array desordenado

4.6.1.5. Un algoritmo de búsqueda

El algoritmo *llc* que presentamos en la Figura 4.35 cuando es ejecutado por un conjunto hoja de procesadores, realiza una búsqueda del valor *KEY* en un vector desordenado de tamaño *SIZE*. Cada uno de los procesadores del conjunto hoja divide el vector en segmentos de tamaño *rate* (línea 3) y realiza una búsqueda secuencial (bucle de la línea 9) en el segmento que le corresponde. La macro booleana IAMTHELASTPROCESSOR devuelve 1 sólo para el procesador con mayor identificador lógico (NAME) del conjunto hoja. La llamada REDUCEBYMIN(*i0, x*) actualiza la variable común *i0* con el mínimo de los valores *x* aportados por los procesadores del conjunto. REDUCEBYMIN() es una operación colectiva, y es invocada por todos los procesadores del conjunto hoja que ejecutan el código (los que hayan encontrado el valor *KEY* invocan la operación en la línea 11 y los que no, en la línea 13). De este modo, cuando se alcanza la línea 14, todos los procesadores del conjunto hoja habrán almacenado en la variable común *i0* el valor del menor índice del array en el que se encuentra la clave buscada o el valor *INFINITY* en caso que la clave buscada no se encontrara en el vector y cuando se ejecuta la función GPRINTF, todos los procesadores del conjunto escribirán el mismo resultado indicando que han encontrado (o no) la clave buscada en el vector.

4.6.2. Equilibrado de la carga en Computación Colectiva Común mediante Hipercubos Dinámicos Ponderados

Conseguir que la carga de trabajo que realiza cada uno de los procesadores en una aplicación paralela sea homogénea es un aspecto crucial para obtener programas eficientes. En el modelo de computación colectiva, las sentencias de asignación de procesadores ponderadas permiten al programador equilibrar la carga de trabajo entre los procesadores del sistema. En los algoritmos considerados anteriormente (Figura 4.27, Figura 4.30, Figura 4.33, Figura 4.34), la ejecución de la función de división común PAR (o PARVIRTUAL) conlleva la división del conjunto de procesadores que la ejecuta en tantos subconjuntos como tareas hayan de realizarse en paralelo. El algoritmo de división del conjunto de procesadores correspondiente a estas sentencias de asignación de procesadores divide el conjunto hoja en subconjuntos de igual cardinal, de forma que en general ello puede contribuir a desequilibrios en la carga de trabajo

```
void find(int first, int last, int middle) {
  int left, right, i, j;

  left = first;
  right = last;
  while (left < right) {
    partition(&i, &j, left, right);
    if (middle <= j) right = j;
    else if (i <= middle) left = i;
      else left = right;
  }
}
```

Figura 4.36 El procedimiento *find*

asignada a cada procesador, especialmente en el caso de algoritmos irregulares. En el caso del código correspondiente al cálculo de la FFT de la Figura 4.27, esta política de división del conjunto de procesadores no provoca desequilibrio de la carga, puesto que se asigna un número igual de procesadores al cálculo de la transformada de las componentes pares de la señal que al de las impares (y hay la misma cantidad de ambas). No obstante, si se considera de nuevo la implementación del Quicksort de la Figura 4.34, se observa que el procedimiento *partition()* dividirá al vector original en dos intervalos cuyos tamaños pueden ser considerablemente distintos dependiendo de la elección que se realice del elemento pivote (que puede elegirse al azar). Asignar el mismo número de procesadores a la ordenación de cada uno de los subintervalos ocasiona un desigual reparto del trabajo entre los procesadores, con la consiguiente pérdida de eficiencia. El mismo problema se presenta en el caso del algoritmo del cálculo de la envolvente convexa (Figura 4.30), en el que la complejidad del cómputo de las envolventes izquierda y derecha puede ser diferente.

El programador puede remediar este tipo de desequilibrios, haciendo que los

```
void qs(int first, int last) {
  int middle, size;

  if (first < last) {
    middle = (first + last) / 2;
    size = (middle - first + 1) * sizeof(int);
    find(first, last, middle);
    PARVIRTUAL(qs(first, middle), v + first, size,
               qs(middle + 1, last), v + middle + 1, size);
  }
}
```

Figura 4.37 El Quicksort en *llc* usando el procedimiento *find*

subproblemas que se van a resolver en paralelo conlleven el mismo coste. Por ejemplo, en el caso del Quicksort paralelo, el programador podría garantizar que los dos subvectores que se van a ordenar en paralelo sean del mismo tamaño, con lo cual sería razonable asignar el mismo número de procesadores a cada uno de los dos subintervalos. El procedimiento *find()* debido a Hoare [Hoa71] que se presenta en la Figura 4.36, y que se basa en repetidas llamadas al procedimiento *partition()* realiza esta tarea con un coste que se puede demostrar es el doble del correspondiente a *partition()*.

Para utilizar este procedimiento, bastaría invocarlo en lugar de *partition()* en los códigos presentados en la Figura 4.33 y Figura 4.34 como podemos ver en la Figura 4.37. El algoritmo calcula el elemento mitad del vector a ordenar en la variable común *middle*, y mediante la llamada a *find()* divide el vector en dos mitades de igual tamaño, que serán ordenadas recursivamente y en paralelo mediante la llamada a PARVIRTUAL.

De este modo se garantiza que la carga de trabajo correspondiente a cada llamada paralela de *qs* será la misma.

```
 1 void qs(int first, int last) {
 2   int i, j, size1, size2;
 3
 4   if (first < last) {
 5     partition(&i, &j, first, last);
 6     size1 = (j - first + 1) * sizeof(int);
 7     size2 = (last – i + 1) * sizeof(int);
 8     WEIGHTEDPARVIRTUAL(size1, qs(first, j), v + first, size1,
 9                        size2, qs(i, last), v + i, size2);
10   }
11 }
```

Figura 4.38 El Quicksort utilizando la función de división WEIGHTEDPARVIRTUAL

Una alternativa a esta posibilidad consiste en utilizar una sentencia de asignación de procesadores ponderada, mediante la cual el programador puede controlar el número de procesadores que se asigna a cada tarea paralela. La Figura 4.38 muestra el código correspondiente a la implementación del Quicksort paralelo utilizando la función de división común ponderada WEIGHTEDPARVIRTUAL (existe también la función de división WEIGHTEDPAR, que sería la contrapartida ponderada de PAR). En este caso, además de los parámetros de la llamada a PARVIRTUAL cuya utilización hemos considerado, el primer y quinto parámetro de la llamada a WEIGHTEDPARVIRTUAL son valores que el programador ha de suministrar en la llamada y que indican la cantidad de trabajo asociada con cada una de las tareas paralelas. Al suministrar estos pesos o valores, el programador proporciona al sistema un árbol de pesos para la jerarquía resultante de la ejecución del algoritmo divide y vencerás. El sistema asignará los procesadores según el Hipercubo Dinámico Ponderado asociado al árbol de pesos. Como explicamos en el epígrafe 4.5, esto da lugar a subconjuntos de procesadores de tamaños proporcionales a los pesos suministrados por el programador.

En el caso del Quicksort, una buena medida de la cantidad de trabajo representado por cada llamada es el tamaño del subintervalo a ordenar. No siempre el cálculo de la cantidad de trabajo que conlleva cada uno de las llamadas paralelas es tan simple de obtener. Por ejemplo, en el caso de la envoltura convexa, la cantidad de trabajo que corresponde a cada uno de los subproblemas en que se divide el problema original no depende de la cantidad de puntos de cada subproblema sino de la distribución en el

```
 1 void qh(int *h, int *N, int max) {
 2 int lmax, rmax, ls, rs, *lh, *rh, w1, w2;
 3
 4 if (trivial(h, N, max))
 5   *N = solve(h, max);
 6 else
 7   if (NUMPROCESSORS == 1)
 8     seqqh(h, N, max);
 9   else {
10     divide(h, *N, max, &lh, &ls, &lmax, &rh, &rs, &rmax);
11     getWeights(&w1, &w2, lh, ls, lmax, rh, rs, rmax)
12     WEIGHTEDPAR(w1, qh(lh, &ls, lmax), lh, ls,
13                 w2, qh(rh, &rs, rmax), rh, rs);
14     *N = combine(h, lh, ls, rh, rs);
15   }
16 }
```

Figura 4.39 El Quickhull usando WEIGHTEDPAR

plano de estos puntos. La Figura 4.39 muestra la aplicación de la función de división WEIGHTEDPAR al problema de la envoltura convexa. En el código que aparece en esa figura, la función *getWeights()* es la encargada de calcular los pesos *w1*y *w2* que estiman el coste de cada una de las llamadas paralelas de la línea 12. Esta función puede basarse en cualquier heurística que sea capaz de suministrar información acerca de la cantidad de trabajo que conlleva cada uno de los subproblemas. La función *getWeights()* ejecuta el propio algoritmo Quickhull en una pequeña muestra convenientemente elegida del conjunto de puntos a tratar, y mide el tiempo consumido en resolver el problema para esa muestra. La función devuelve esta cantidad de tiempo como medida del coste del subproblema. Este método para evaluar los pesos de una llamada paralela tiene la ventaja de su generalidad, siendo posible su aplicación a cualquier algoritmo divide y vencerás.

La necesidad de redondear las proporciones de acuerdo con los pesos del árbol introduce una posibilidad de desequilibrio. Consideremos como ejemplo una división binaria con pesos w_1 y w_2. Sea $W = w1+w2$. Para realizar el reparto proporcional de los P procesadores efectuamos las divisiones enteras:

$$P * w1 = n1 * W + r1 \qquad 0 \le r2 < W$$
$$P * w2 = n2 * W + r2 \qquad \text{con } 0 \le r2 < W$$

sumando las dos ecuaciones obtenemos:

$$P * W = (n1 + n2) * W + r1 + r2 \qquad \text{con } 0 \le r1+r2 < 2W$$

y dividiendo (división flotante) por W obtenemos:

$$\text{(Ec. 4.3)} \qquad P = (n1 + n2) + (r1 + r2) / W$$

Si suponemos que los pesos suministrados *w1*y *w2* por el programador son enteros, resulta que:

$$(r1 + r2) / W = P - (n1 + n2) \text{ es entero.}$$

por tanto se deduce que $r1 + r2 = W$ ó $r1 + r2 = 0$.

La situación en la cual la técnica de hipercubos dinámicos ponderados está sujeta a desequilibrios ocurre en ciertos casos en que $r1+r2 = W$. En ese caso, es fácil deducir de la (Ec. 4.3) que $P = n1+n2+1$ y el procesador restante deberá asignarse a la tarea que tenga más carga remanente (aquella en que se alcanza $max\{r1, r2\}$). El desequilibrio que pueda aparecer será precisamente fruto de la asignación de este procesador según la carga remanente.

Por ejemplo, si $r1 = W/2$ y $r2= (W/2)-1$, tendremos que el procesador remanente se incorporará al primer grupo mientras que la carga de trabajo es aproximadamente la misma en los dos grupos. En este caso, una solución a este problema consiste en ejecutar dos threads en el procesador remanente. En estas circunstancias ($r1+r2 = W$) la situación óptima para esta técnica se alcanza cuando $r1 = W-1$ y $r2 =1$.

4.7. Implementación de algoritmos divide y vencerás en computación colectiva (CC) mediante Hipercubos Dinámicos

El Modelo de Computación Colectiva permite en su forma más general la implementación eficiente de algoritmos para diferentes tipos de problemas y no sólo los pertenecientes a la clase Común-Común.

En esta sección presentaremos algoritmos colectivos para problemas de tipo Privado-Privado en los que los datos de entrada del problema no se encuentran

inicialmente en la memoria de todos los procesadores de la máquina y la solución tampoco ha de quedar en todos los procesadores.

Asumimos que se trata de problemas P cuyos datos de entrada p_1 .... p_N y cuya solución $R= r_1 ... r_N$ se pueden representar de forma vectorial. Asumiremos también que al comienzo de la ejecución, el vector que representa el problema P está dividido en tantos segmentos (no necesariamente del mismo tamaño) como procesadores q tiene la máquina $[p_{10}...p_{1r}]$, $[p_{20}...p_{2s}]$,..., $[p_{q0}...p_{qm}]$. Análogamente suponemos que la solución queda almacenada de forma segmentada en los procesadores, $[r_{10}...r_{1h}]$, $[r_{20}...r_{2j}]$,..., $[r_{q0}...r_{qk}]$.. La lectura del vector solución en el orden dado por los procesadores nos da la que es la solución global al problema.

En esta situación es posible paralelizar internamente el proceso de división. Cada procesador divide su segmento de problema en los dos (o más) segmentos resultantes. Es posible que para llevar a cabo la división sea necesario el intercambio de cierta información global para utilizar un criterio común de división. El conjunto actual de procesadores puede ser dividido en dos subconjuntos después de conseguir que los subsegmentos correspondientes a cada tarea hayan sido desplazados a los correspondientes procesadores. Para poder realizar con libertad este movimiento de datos, es necesario que la solución de los subproblemas no dependa del orden en que figuren los datos. Se asume por tanto una *propiedad de conmutatividad* del divide y vencerás: la solución al problema no debe depender del orden de los elementos del vector. Esto es cierto por ejemplo para un algoritmo de ordenación, dado que el vector ordenado resultante no depende del orden inicial de los elementos. También es cierto para el problema de la envoltura convexa: los puntos que constituyen la envolvente son los mismos independientemente del orden en que se suministren los puntos de la nube.

Al proceder recursivamente o bien los problemas resultarán triviales o bien agotaremos los procesadores de los subconjuntos. En ambos casos resolveremos secuencialmente el subproblema resultante, obteniendo una solución r_i en el procesador q_i. Si, como ocurre en los problemas de la ordenación y de la envoltura convexa, la operación de combinación del divide y vencerás es la operación de concatenación de vectores, el problema ha sido resuelto y la solución está formada por la concatenación de los resultados obtenidos en cada procesador en el orden indicado por los identificadores de aquellos. En caso contrario, es necesario proceder a una fase de intercambio de resultados entre procesadores socios y a su combinación.

4.7.1. El Quicksort Distribuido

La función *dqs()* que aparece en la Figura 4.40 es un ejemplo de algoritmo de tipo Privado-Privado. Se trata de un algoritmo de ordenación al que hemos denominado Quicksort Distribuido en el que el vector a ordenar, *v* está inicialmente distribuido entre todos los procesadores de la máquina paralela (cada procesador almacena un segmento del vector original) y a la finalización del algoritmo, el vector ordenado quedará distribuido también entre todos los procesadores.

Si denominamos S_p a la secuencia ordenada de elementos del vector que quedan almacenadas en el procesador *p*, se verifica que

$$Si\ i < j,\ v[i] \pounds v[j]\ \ "\ v[i], v[j]\ \hat{I}\ S_p,\ "\ p\ \hat{I}\ M$$
$$Si\ NAMEA < NAMEB\ v[i] \pounds v[j]\ \ "\ v[i]\ \hat{I}\ S_{NAMEA}\ v[j]\ \hat{I}\ S_{NAMEB}$$

es decir, las secuencias ordenadas quedan a su vez ordenadas con respecto a los identificadores lógicos de los procesadores.

```
1 void dqs (int *v, int *size) {
 2   int min_size, s, i, j, pivot;
 3   double temp, sum;
 4
 5   if (NUMPROCESSORS > 1) {
 6     REDUCEBYMIN(min_size, *size);
 7     if (min_size > MIN_SEQ_SIZE) {
 8       temp = (double)v[0];
 9       REDUCEBYADDD(sum, temp);
10       pivot = (int)(sum / NUMPROCESSORS);
11       PAR(part(v, *size-1, pivot, &i, &j, &s), v+i, s,
12        revPart(v, *size-1, pivot, &i, &j, &s), v+i, s);
13       *size = j + 1 + (s / sizeof(int));
14       SPLIT(dqs(v, size), dqs(v, size));
15     }
16     else
17          qsSeq(v, 0, (*size-1));
18   }
19   else
20     qsSeq (v, 0, (*size-1));
21 }
```

Figura 4.40 El Quicksort Distribuido

Si partimos de la situación en la que cada procesador de la máquina ha obtenido ya el segmento de vector original que le corresponde, la llamada en el programa principal tendría la forma

qs(v, &size);

siendo *size* la variable paralela que almacena el tamaño del subvector en cada procesador.

Si el conjunto de procesadores hoja que ejecuta la función *qs()* sólo contiene un procesador, la ordenación se resuelve secuencialmente mediante la llamada *qsSeq()* de la línea 20. Lo mismo ocurre si el tamaño de alguno de los subvectores a ordenar por algún conjunto hoja es menor que un cierto umbral, MIN_SEQ_SIZE. Mediante la llamada a la operación colectiva REDUCEBYMIN() de la línea 6, los procesadores obtienen en la variable paralela privada *min_size* el tamaño de la menor secuencia a ordenar por alguno de los conjuntos hoja. La condición de la línea 7 es evaluada al mismo valor por todos los procesadores del conjunto hoja. Puesto que la sentencia condicional no divide al conjunto en dos subconjuntos (como ocurre si se hubiese utilizado la función de división PARBOOL), esta característica de evaluación común es necesaria para garantizar que todos los procesadores del conjunto acuden a las operaciones colectivas y de división invocadas en las líneas 8-14.

En el caso general el Quicksort Distribuido opera eligiendo un valor como pivote común en todos los conjuntos hojas. El valor del pivote se almacena en la variable *pivot*. Inicialmente cada procesador elige como pivote el primer elemento (*v[0]*) del subvector que le corresponde (línea 8) y mediante una operación de reducción (línea 9) todos los procesadores eligen un valor común como pivote (línea 10).

Cada conjunto hoja Q de procesadores se divide en dos subconjuntos a los que llamaremos Q_i y Q_d. Los procesadores en el subconjunto Q_i se encargan de particionar su subvector de modo que en la parte izquierda del vector queden los elementos menores que el pivote. Recíprocamente, los procesadores de Q_d particionan su subvector colocando a la izquierda los elementos mayores que el pivote común. La división del conjunto hoja se realiza mediante la llamada a la función de división PAR() de la línea 11 en la que se activan las tareas paralelas *part()* y *revPart()*. El segmento de

vector que ha correspondido a Q es una variable común hasta la división de Q en Q_i y Q_d. La tarea *part()* se corresponde con el conocido algoritmo de partición debido a Hoare que ya hemos utilizado anteriormente en este mismo Capítulo. *revPart()* es una versión modificada de este mismo algoritmo. La Figura 4.41 presenta el código de *revPart()*.

```
 1 void revPart (int *v,int last,int pivot,int *i,int *j,int *size1) {
 2   int temp;
 3   register int ii, jj;
 4
 5   ii = 0;
 6   jj = last;
 7   do {
 8     while ((v[ii] > pivot) && (ii <= jj)) ii++;
 9     while ((pivot >= v [jj]) && (ii <= jj)) jj--;
10     if (ii <= jj) {
11          temp = v [ii];
12          v [ii] = v [jj];
13          v [jj] = temp;
14          ii++;
15          jj--;
16     }
17   } while (ii <= jj);
18   *size1 = ((last - ii + 1) * sizeof (int));
19   *i = ii;
20   *j = jj;
21 }
```

Figura 4.41 La función *revPart()*

Como consecuencia de la ejecución de las tareas *part()* y *revPart()* el subvector que inicialmente era común, deja de serlo. A la finalización de las tareas paralelas los procesadores del conjunto Q_i retienen los elementos menores que el *pivot* común, envían a su socio los elementos mayores que *pivot* y reciben de sus procesadores socios en Q_d los elementos menores que el *pivot*. Simétricamente, los procesadores de Q_d retienen los elementos mayores que *pivot*, envían los menores y reciben de sus socios en Q_i elementos mayores que el pivote. De este modo, al finalizar la función de división PAR, los procesadores de Q_i sólo almacenan elementos menores que *pivot* y los de Q_d elementos mayores.

El algoritmo procede recursivamente mediante la activación de dos llamadas recursivas y paralelas a *dqs()* que tienen lugar mediante la llamada a la función de división SPLIT() de la línea 14. Para el buen funcionamiento del algoritmo se ha de verificar que las políticas de división del conjunto de procesadores que siguen las funciones PAR() y SPLIT() sea exactamente la misma.

La Figura 4.42 presenta una traza del algoritmo ejecutado con cuatro procesadores, *P0, P1, P2* y *P3* con un vector de 16 enteros. Inicialmente cada procesador almacena en su variable privada *v* un segmento del vector a ordenar, de tamaño 4. La realización de la reducción de la línea 9 (ver Figura 4.40) conlleva una comunicación de tipo all-to-all mediante la cual, todos los procesadores aportan un valor como elemento pivote. Los procesadores almacenan en la variable común *pivot* el valor de la suma de los valores aportados, dividido por el número de procesadores del conjunto hoja: esto produce el primer pivote de valor 61 en la Figura 4.42.

A continuación el conjunto raíz con 4 procesadores se divide en dos subconjuntos Q_0*={P0, P1}* y Q_1*={P1, P2}* el primero de los cuales ejecuta *part()* y el segundo *revPart()*. Como consecuencia de esta división, cada procesador divide su subvector en

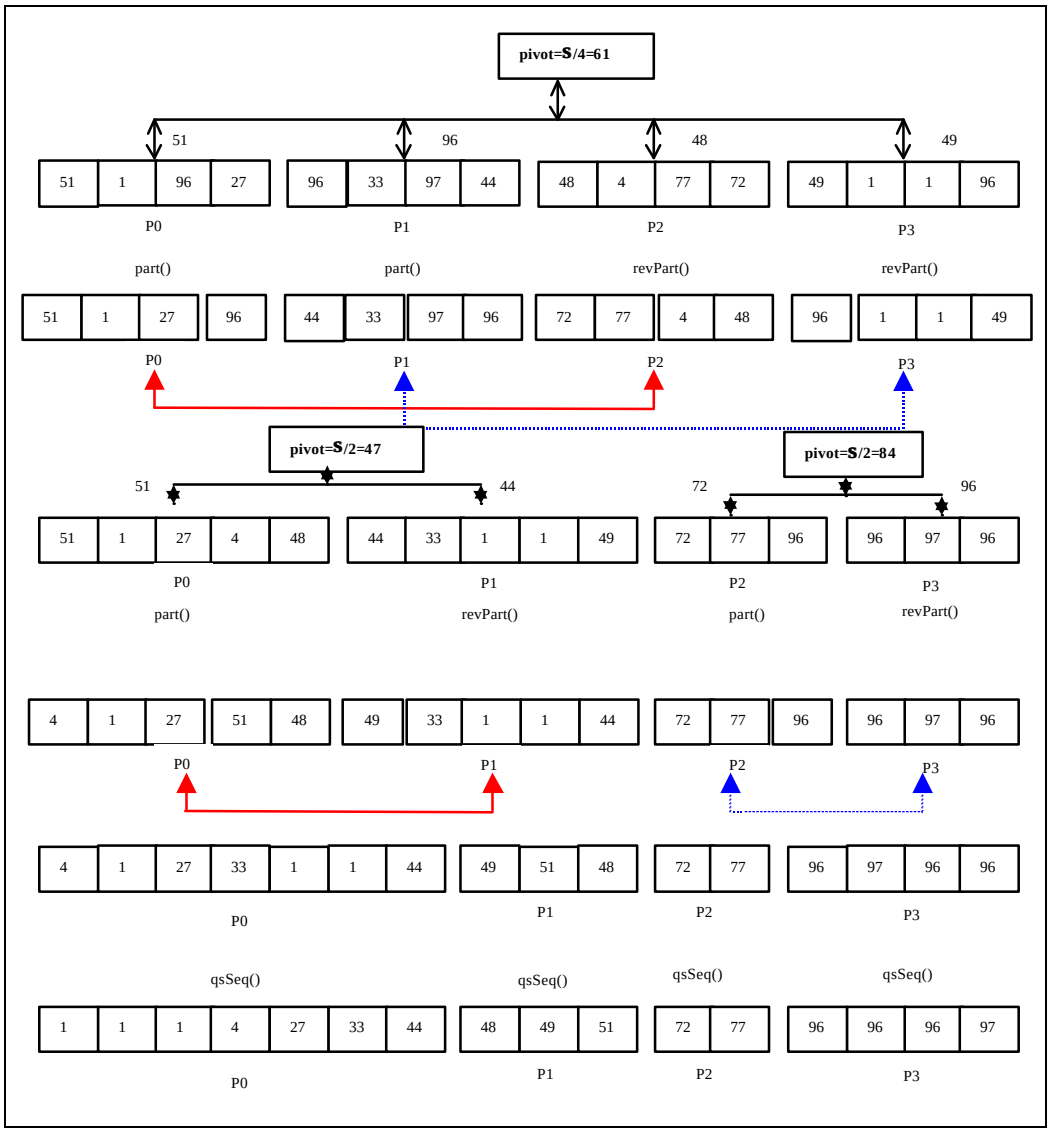

Figura 4.42 Una traza del Quicksort Distribuido

dos partes, y mediante una comunicación con su socio en el subconjunto complementario retiene una de las partes de la partición. Los procesadores de Q_0 retienen los elementos menores que el pivote global y envían a su socio en Q_1 los elementos mayores que el pivote. Los procesadores de Q_1 hacen lo contrario: retienen los elementos mayores que el pivote y envían a su socio en Q_0 los menores. Esta comunicación que se produce al finalizar la ejecución de la función de división PAR() (línea 11 de la Figura 4.40) produce la configuración que muestra el tercer vector de la Figura 4.42. En ese instante, la función de división SPLIT() (línea 14 de la Figura 4.40) divide el conjunto raíz de 4 procesadores (que se había reunificado al término de la ejecución de la función PAR()) de nuevo en los subconjuntos Q_0 y Q_1 que recursivamente ejecutan una nueva llamada a *dqs()*. Mediante sendas reducciones (línea 9 del código) los procesadores de Q_0 obtienen el valor 47 como pivote común y los de Q_1 obtienen el valor 84. La nueva llamada a la función PAR() divide los conjuntos hojas actuales en los nuevos conjuntos hojas Q_{00}={P0}, Q_{01}={P1}, Q_{10}={P2} y Q_{11}={P3}. Los procesadores de Q_{00} y Q_{10} ejecutan *part()* y los de Q_{01} y Q_{11} ejecutan *revPart()* dando lugar al vector de la cuarta fila de la Figura 4.42. Nuevamente cada procesador intercambia con su socio en el conjunto complementario una parte de su subvector antes de recuperar los conjuntos Q_0 y Q_1.

Cuando una segunda llamada a la función SPLIT() genera dos nuevas tareas paralelas *dqs()* que son ejecutadas por los conjuntos hojas Q_{00}, Q_{01}, Q_{10} y Q_{11} que vuelven a ser creados, la condición de tener más de un procesador en el conjunto hoja (línea 5 del código de la Figura 4.40) no se cumple y cada conjunto hoja (conteniendo un procesador) invoca a *qsSeq()* para ordenar secuencialmente el segmento de vector que le ha correspondido. Esta ordenación secuencial produce el resultado que aparece en la última fila de la Figura 4.42.

4.7.2. Quickhull Distribuida

```
 1 void dqh (points h, int N) {
 2   int min_size, maxidx, na, nb;
 3   points a, b;
 4
 5   if (NUMPROCESSORS > 1) {
 6     REDUCEBYMIN(min_size, N);
 7     if (min_size > MIN_SEQ_SIZE) {
 8       maxidx = maxdist(h[0], h[N-1]);
 9       REDUCE(pmax, h[maxidx]);
10       PAR(divide(h, N, pmax, a, &na, b, &nb), b, nb,
11           divide(h, N, pmax, b, &nb, a, &na), b, nb);
12       concat(a, b);
13       SPLIT(dqh(a, na), dqh(a, na));
14     }
15     else
16         seqqhull (h, N);
17   }
18   else
19     seqqhull (h, N);
20 }
```

Figura 4.43 Quickhull Distribuida

El algoritmo de la Quickhull para el cálculo de la envolvente convexa de una nube de puntos que presentamos en la sección 4.6.1.2 de este mismo Capítulo resolvía el correspondiente problema Común-Común. Si los datos de la nube de puntos se distribuyen entre los procesadores, y se admite que la solución venga dada por la concatenación de las soluciones parciales, en el orden determinado por los índices de los procesadores, tenemos un planteamiento de tipo Privado-Privado.

El código de la Figura 4.43 representa la implementación en *llc* de un algoritmo que resuelve el problema. El tipo *points* corresponde con un vector de elementos de tipo *point*¸ que es una estructura conteniendo abcisa y ordenada de cada punto. A diferencia del código de la Figura 4.30 de la sección 4.6.1.2 en el que *h* representaba un vector de índices de puntos, en este caso *h* representa directamente la nube de puntos. Al ser un problema de tipo Privado-Privado no podemos utilizar un vector de índices común a todos los procesadores para minimizar los movimientos de datos como hicimos en el caso Común-Común.

El número de puntos de la nube viene dado por el segundo parámetro, *N* de la función *dqh()*. El algoritmo de la Quickhull Distribuida procede de forma similar a la del Quicksort Distribuido que ya hemos explicado (ver Figura 4.40). De hecho, el código de las líneas 5-7 es idéntico. En el caso general, mediante la función *maxdist()* de la línea 8, el algoritmo calcula el índice (*maxidx*) del punto más alejado a la recta definida por los puntos extremos *h[0]* y *h[N-1]*.

La llamada a la función colectiva *REDUCE(pmax, h[maxidx])* de la línea 9 almacena en la variable común *pmax* el punto más alejado a la recta *h[0], h[N-1]*. Para

ello, cada procesador proporciona como entrada su punto más alejado, *h[maxidx]*. El código *llc* que implementa esta función colectiva se reduce a las dos líneas siguientes:

```
#define TYPE point
#define OPERATION(y,x) {dist(x,h[0],h[N-1])>=dist(y,h[0],h[N-1])?x:y;}
```

La macro *OPERATION(y, x)* devuelve en su primer argumento el punto más alejado a la recta utilizando para su cálculo la función *dist()* que calcula la distancia euclídea de un punto a una recta.

La rutina *divide(h, N, pmax, left, &nl, right, &nr)* se encarga de discriminar la nube *h* de *N* puntos local al procesador en tres grupos: (1) puntos interiores al triángulo formado por *h[0], h[N-1], pmax* (véase Figura 4.31), (2) los *nl* puntos *left* a la izquierda de la recta definida por *h[0], pmax* y (3) los *nr* puntos *right* a la derecha de la recta *pmax, h[N-1]*. La función garantiza que a su terminación se cumplen las siguientes igualdades:

left[0]=h[0]; *left[nl-1]=pmax;*
right[0]=pmax; *right[nr-1]=h[N-1];*

estas igualdades garantizan inductivamente la propiedad de que los dos puntos más alejados de la nube están situados en las posiciones primera y última del vector de entrada.

La llamada a la función de división PAR() de la línea 10 activará dos instancias paralelas de la función *divide()*. La primera de ellas, *divide(h, N, pmax, a, &na, b, &nb)* colocará en el vector *a* los puntos de tipo (2) (según la clasificación anterior) y en *b* los de tipo (3) mientras que (por el orden en que se le pasan los parámetros) la segunda instancia de la función *divide(h, N, pmax, b, &nb, a, &na), b, nb)* colocará en *b* los puntos de tipo (2) y en *a* los de tipo (3).

Como consecuencia del intercambio entre socios que tiene lugar a la finalización de la función de división PAR(), el subconjunto de procesadores que ejecutó la primera instancia de *divide()* envía a sus socios los puntos a la derecha de la recta *pmax, h[N-1]* almacenados en *b*. Después del envío, estos procesadores reciben de sus socios en su variable *b* los puntos a la izquierda de la recta *h[0], pmax*.

Después de la llamada a la función *concat(a, b)* de la línea 12, todos los procesadores del antiguo primer subconjunto almacenarán en *a* puntos de tipo (2) mientras que los del conjunto complementario almacenarán en *a* solamente puntos de tipo (3).

Estos dos conjuntos se dividen nuevamente con la llamada a la función de división SPLIT(*dqh(a, na), dqh(a, na)*) de la línea 13 para atacar recursivamente los dos subproblemas generados.

4.7.3. Equilibrado de la carga en Computación Colectiva

Como podemos observar en la traza de la Figura 4.42, el tamaño de vector que finalmente acaba ordenando cada procesador en el algoritmo Quicksort Distribuido es diferente para cada uno. El cuello de botella en el rendimiento del algoritmo será el procesador al que le corresponda una mayor cantidad de trabajo. El algoritmo presenta dos fuentes de desequilibrio. Por una parte, la división en subconjuntos mediante la llamada a SPLIT() de la línea 14 (Figura 4.40) particiona el conjunto de procesadores en dos mitades del mismo tamaño, cuando lo deseable sería que los tamaños de los subconjuntos fueran proporcionales a los tamaños de los subvectores correspondientes. Llamaremos "externo" a esta forma de desequilibrio. La segunda fuente de desequilibrio es "interna" a cada uno de los subconjuntos creados. Como consecuencia de los intercambios producidos por la llamada a la función de división PAR() en la línea 11, los

segmentos de subvector en cada procesador pueden ser sensiblemente diferentes. Se puede mejorar el rendimiento del algoritmo equilibrando la carga y en el caso concreto de este algoritmo, hemos considerado varias formas de equilibrio.

Una aproximación simple al equilibrado de la carga consiste en aumentar la bondad del elemento que se elige como pivote para realizar las particiones. Si la distribución de los valores que se ordenan es uniforme (como ocurre en los experimentos realizados), la mediana de los datos será la mejor elección posible como elemento pivote, puesto que la mitad de los datos sean menores que ella y la otra mitad mayores, con lo cual dividiríamos el vector en partes iguales. Una mejora del algoritmo consiste en elegir como elemento pivote la media de una muestra de cierto tamaño de los elementos a ordenar. Para ello, en el código de la Figura 4.40 basta sustituir la línea 8 en la que se elige el pivote como el primer elemento del subvector por un bucle que calcule la media de una muestra de los elementos a ordenar. Esta aproximación al equilibrado de la carga puede resolver satisfactoriamente el desequilibrio "externo" del que hablamos en el párrafo anterior.

Para resolver el desequilibrio interno utilizamos un equilibrado de carga por intercambio de datos en dimensiones. El código de la Figura 4.44 corresponde con la implementación en *llc* de esta idea.

```
 1 void qsBalanced (int *v, int *size) {
 2   int i, j, pivot, s, min_size, k, len;
 3   double sum, my_pivot;
 4
 5   if (NUMPROCESSORS > 1) {
 6     HBALANCE((*size), v, Ponderate, GetWeigth);
 7     REDUCEBYMIN(min_size, *size);
 8     if (min_size > MIN_SEQ_SIZE) {
 9       len = (*size) / SAMPLES;
10       my_pivot = 0.0;
11       for (k = 0; k < SAMPLES; k++)
12         my_pivot += (double)v[k * len];
13       my_pivot /= SAMPLES;
14       REDUCEBYADDD(sum, my_pivot);
15       pivot = (int)(sum / NUMPROCESSORS);
16       PAR(part(v, *size-1, pivot, &i, &j, &s), v+i, s,
17           revPart(v, *size-1, pivot, &i, &j, &s), v+i, s);
18       *size = j + 1 + (s / sizeof(int));
19       SPLIT(qsBalanced(v, size), qsBalanced(v, size));
20     }
21    else
22      qsSeq(v, 0, (*size - 1));
23   }
24   else
25     qsSeq (v, 0, (*size - 1));
26 }
```

Figura 4.44 Equilibrado de carga por intercambio de datos en el Quicksort distribuido

La idea del algoritmo es la misma que la del Quicksort Distribuido sin equilibrado de carga que presentamos en el epígrafe 4.7.1, pero en este caso se incluye la llamada HBALANCE de la línea 6. Esta macro considera una configuración hipercúbica del conjunto de procesadores hoja. La macro produce un equilibrado de carga por intercambio de datos en cada una de las dimensiones. En primer lugar, los procesadores equilibran su carga con su vecino en dimensión 0. La nueva carga resultante es equilibrada con el vecino en dimensión 1 y así sucesivamente. Supongamos que un conjunto hoja constituido por 8 procesadores ejecuta el algoritmo. En primer lugar cada

procesador equilibra la carga con su vecino en dimensión 0. De este modo, las parejas de procesadores (0, 1), (2, 3), (4, 5) y (6, 7) acaban teniendo la misma carga. A continuación, el equilibrado se produce en dimensión 1 y ahora las parejas de procesadores que equilibran su carga son (0, 2), (1, 3), (4, 5) y (6, 7). Por último, el intercambio de datos en dimensión 2 hará que las parejas (0, 4), (1, 5), (2, 6) y (3, 7) alcancen el equilibrio.

La función *GetWeight()* que aparece como último parámetro en la llamada a HBALANCE suministra información acerca de la carga de trabajo de cada procesador. Como hemos indicado ya anteriormente, en el caso de un algoritmo de ordenación, una buena medida de este valor es la cantidad de datos a ordenar. La función *Ponderate()* que aparece como segundo parámetro de HBALANCE calcula la cantidad de datos que cada procesador ha de enviar a su vecino para conseguir un equilibrado de carga perfecto entre ambos procesadores. La macro *HBALANCE* envía el número de datos decidido por *Ponderate()*, tomándolos del final del vector. Obsérvese que la solución obtenida es independiente de los elementos que se envían, dada la propiedad de conmutatividad del problema.

El código entre las líneas 9 y 15 se encarga de elegir un pivote común. Siguiendo el primer esquema de equilibrado de carga que hemos propuesto, cada procesador aporta a la suma de la línea 14 la media de una muestra de tamaño *SAMPLES* de entre sus elementos.

El equilibrado en dimensión adolece del inconveniente que el equilibrio que se introduce es interno y no tiene porqué traducirse en un equilibrio global de la carga de trabajo.

Una aproximación para la resolución del desequilibrio externo se basa al igual que el equilibrado introducido en el epígrafe 4.6.2 para el Quicksort de tipo Común-Común en no dividir los conjuntos hojas de procesadores de forma simétrica, sino hacerlo en proporción a la carga de trabajo que ha de asumir cada nuevo conjunto hoja. Llamaremos a esta alternativa equilibrado por asignación de procesadores: el número de procesadores que se asigna a un problema depende del tamaño del mismo.

```
 1 void qsBalWeig (int *v, int *s1, int *v2, int *s2) {
 2   if (NUMPROCESSORS > 1) {
 3     part(v, *s1, v, s1, v2, s2);
 4     WEIGPAR(v,(*s1),qsBalWeig(v,s1,v2,s2),
 5             v2,(*s2), qsBalWeig(v2,s2,v,s1));
 6   }
 7   else {
 8     if (*s1 > 0) {
 9       qsSeq (v, 0, (*s1-1));
10     }
11     else
12       if (*s2 > 0) {
13         qsSeq (v2, 0, (*s2-1));
14       }
15     }
16 }
```

Figura 4.45 Equilibrado de carga por asignación de procesadores

No es posible en el código de la Figura 4.40 sustituir directamente la llamada SPLIT() por una llamada de tipo WEIGHTEDSPLIT() análoga a la función de división WEIGHTEDPAR() explicada en el epígrafe 4.6.2, porque como hemos comentado, es fundamental que las funciones de división invocadas en las líneas 16 y 19 de la Figura 4.44 PAR() y SPLIT() produzcan la misma división del conjunto hoja.

La Figura 4.45 muestra la implementación de la nueva idea en el Quicksort Distribuido. Utilizando la función *part()* de la línea 3, el segmento de vector considerado se particiona en dos subsegmentos de tamaños *s1* y *s2*. El primer parámetro de *part()* es el vector original, y el segundo el número de elementos del vector. La función *part()* elige un pivote común de forma análoga a la que ya hemos explicado y coloca los elementos menores que el pivote en el tercer parámetro, *v* y los mayores en el vector *v2*. Los parámetros *s1* y *s2* indican los tamaños de estos dos subvectores.

La función de división WEIGPAR de la línea 4 asigna procesadores a cada uno de los dos subproblemas de forma que el número de procesadores asignados a un problema sea proporcional a la dificultad del mismo. De modo análogo a la función de división WEIGHTEDPAR que ya hemos explicado, el segundo y quinto parámetros de WEIGPAR (**s1* y **s2*) indican la proporción de procesadores que se asignará a cada subproblema.

Los procesadores del conjunto hoja que ejecutan WEIGPAR se comunican entre sí la cantidad de problemas de cada tipo que cada uno posee (en el caso que nos ocupa, elementos menores y mayores que el pivote). Supuesto que la complejidad en el número de procesadores se comporta de forma aditiva, es posible con esa información conocer el peso global correspondiente a cada una de las tareas. Esto permite construir los conjuntos de procesadores asignados a cada tarea de forma proporcional al peso global que le corresponde.

Además, todos los procesadores de cada uno de los dos subconjuntos hoja intercambian datos para equilibrar la carga a nivel del subconjunto, tratando de conseguir que el tamaño de problema a resolver sea el mismo para todos los procesadores de un conjunto hoja. De nuevo la forma de trabajo del algoritmo se apoya en la propiedad de conmutatividad del problema.

4.8. El Modelo Colectivo como Modelo de Predicción del tiempo de cómputo

En este epígrafe, introducimos una extensión al modelo de Computación Colectiva para utilizarlo como modelo de predicción del rendimiento de algoritmos paralelos. Para ilustrar la utilización del modelo propuesto, utilizaremos tres ejemplos: el Quicksort y la FFT que ya hemos presentado en las secciones 4.6.1.4 y 4.6.1.1 y el algoritmo de Ordenación por Muestreo Regular (PSRS) que presentaremos por primera vez en esta sección. El algoritmo PSRS utiliza un amplio rango de funciones de comunicación. El modelo de computación colectiva es independiente de la plataforma de implementación y predice con precisión los tiempos de ejecución de los algoritmos considerados.

Una consecuencia de la definición dada del Modelo de Computación Colectiva es que en este modelo el cómputo transcurre en pasos. Siguiendo la terminología BSP [Val90], [Hil97], los denominaremos superpasos. Esto es así porque las funciones colectivas implican la participación de todos los procesadores del conjunto. Obsérvese que, a diferencia de lo que ocurre en el modelo BSP, los superpasos son locales al conjunto.

En el modelo Colectivo existen dos tipos de superpasos. El primer tipo (denominado *normal*) esta constituido por dos fases:

1. Cómputo local
2. Ejecución de una de las funciones de comunicaciones colectivas *f* de *Col*

Donde la segunda fase de comunicación es opcional y puede ser omitida.

Asociada con cada función $f \hat{I} Col$ existe una función de coste T_f que nos da una predicción del tiempo invertido para el patrón de comunicaciones f en términos del

número *P* de procesadores de la submáquina actual y de las longitudes de los mensajes implicados. Análogamente, el modelo colectivo supone la existencia de una función de coste T_g con cada patrón de división $g \hat{I} Div$. El coste F de un paso *s* del tipo *normal* constituido por un cómputo y la ejecución de una función colectiva *f* viene dado por:

$$F = W + T_f = max\{W_i / i = 0,\ldots,P-1\} + T_f(P, h_0, \ldots, h_{P-1})$$

Donde W_i es el tiempo invertido en cómputo por el procesador *i* en ese superpaso y h_j es la cantidad de datos (dada por la suma o el máximo de los paquetes enviados y recibidos) comunicada por el procesador $j = 0, \ldots, P-1$ bajo el patrón *f*.

El otro tipo de paso en el modelo Colectivo es el de *división* o *partición*. Como ya hemos explicado, en un momento dado la máquina puede también subdividirse en un cierto número *r* de submáquinas de tamaños $P_0, \ldots, P_{r-1}$ como consecuencia de una llamada a una función de partición colectiva $g \hat{I} Div$. En su forma más general el proceso *g* de división conlleva una fase de distribución de los datos de entrada $in_0, \ldots, in_{P-1}$, una fase de ejecución de tareas $Task_0, .., Task_{r-1}$ sobre esos datos de entrada y una fase de reunificación y devolución de los resultados $out_0, \ldots, out_{r-1}$ generados por la ejecución de las tareas. Algunos de estos parámetros pueden no aparecer en algunas funciones de división (por ejemplo, la función *MPI_Comm_split* no lleva asociada una distribución de datos de entrada ni una fase devolución de resultados, y las tareas $Task_0, .., Task_{r-1}$ se reducen a una). Por tanto, un proceso *g* de división está parametrizado como sigue:

$$g(P, in_0, \ldots, in_{P-1}, r, Task_0, .., Task_{r-1}, out_0, \ldots, out_{r-1})$$

Al finalizar la ejecución de dichos códigos $Task_k$ los procesadores intercambian los resultados de su computación out_k y se reunifican en la máquina inicial. La manera en la que se ejecutan los intercambios (el "patrón de comunicaciones") viene determinada por la función de partición *g*.

El tiempo o coste *F* en este caso viene dado por una función $T_g(P, in_0, \ldots, in_{P-1}, r, out_0, \ldots, out_{r-1})$ que determina el tiempo invertido en los procesos de división y distribución de datos de entrada y la reunificación y el intercambio de los resultados de salida más el mayor de los tiempos recursivamente calculados según el modelo colectivo para los *r* grupos formados sobre las tareas $Task_0, .., Task_{r-1}$:

$$F = T_g(P, in_0, \ldots, in_{P-1}, r, out_0, \ldots, out_{r-1}) + max\{F(Task_0), \ldots, F(Task_{r-1})\}$$

Concluyendo, el modelo colectivo, como modelo de evaluación de rendimientos se extiende con dos nuevos elementos T_{Col} y T_{Div}, quedando caracterizado por la quíntupla:

$$(M, Col, T_{Col}, Div, T_{Div})$$

donde

Los elementos *M*, *Col* y *Div* tienen el significado que se dio en la Definición 4.1 de la sección 4.3.

- T_{Col} es el conjunto de funciones que determinan las “leyes” que rigen los tiempos invertidos por las funciones colectivas de *Col.*

Para determinar T_{Col} pueden usarse diferentes propuestas. Por ejemplo sería válido tomar como T_{Col} el conjunto empírico de funciones lineales a trozos que se deducen de los estudios de Abandah [Aba96] y Arruabarrena [Arr96]. Otra alternativa es usar un modelo de cómputo existente. Por ejemplo, podemos determinar T_{Col} a partir del modelo LogP de Culler [Cul93], [Cul95], [Cul96a], [Cul96b] o del modelo C^3 propuesto por Hamsbrush [Ham96a], [Ham96b],

[Ham96c]. Podríamos también considerar válida la hipótesis de la *h*-relación y calcular las leyes en T_{Col} a partir de los valores de *g* y *L* propuestos en [Rod98a].

- T_{Div} es el conjunto de funciones que determinan las "leyes" que rigen los tiempos invertidos por las funciones partición de *Div*. Se aplican aquí los mismos comentarios que se hicieron para el conjunto T_{Col}.

El modelo de Computación Colectiva se puede extender de manera natural con funciones de acceso a memoria remota como las que se proponen en el Modelo BSP (véase la Oxford BSPLib como ejemplo [Hil97]). Como en BSP, los accesos se hacen efectivos a la finalización de un superpaso *normal*. El coste de estos accesos a memoria remota puede suponerse proporcional al tamaño de la *h*-relación que ocurre en el superpaso, esto es, al número máximo *h* de paquetes comunicados por un procesador durante el superpaso. La constante de proporcionalidad está asociada a la plataforma paralela utilizada.

4.8.1. Análisis del Quicksort de tipo Común-Común

Consideremos de nuevo el código de la Figura 4.33 explicado en la sección 4.6.1.4. El proceso de división tiene un coste lineal en el tamaño *n* del subvector, *B*n*. Cuando no hay más procesadores disponibles se llama en la línea 14 al algoritmo de ordenación secuencial *seqquicksort(first, last)*. Si la elección de los pivotes es correcta, el equilibrado será casi perfecto y el tiempo de la ordenación secuencial se aproximará a *C*(n/P*log(n/P))* para alguna constante *C*. Al finalizar las llamadas los procesadores *socios* intercambian los resultados de sus segmentos ordenados dando lugar a una comunicación con tamaño *h = s1+s2 = (last-first+1)* enteros. Después del intercambio, los dos grupos se reúnen en el grupo que formaban anteriormente.

El tiempo invertido *F* según el modelo colectivo viene dado por la fórmula recursiva:

$$F = B*n/2 + max\{F(qs(first,j)),F(qs(i,last))\} + T_{PAR}(P,0,...,0,2,n,...,n)$$

El primer sumando corresponde al proceso de partición. El segundo y tercer sumando corresponden al superpaso de división. El tiempo invertido es el máximo de los tiempos invertidos en cada una de las ordenaciones más el tiempo

$$T_{PAR}(P,0,...,0,2,n,...,n)$$

que se invierte en el intercambio de resultados al que da lugar la función de división PAR. Este intercambio tiene lugar según el patrón *EXCHANGE* en una máquina con *P* procesadores (primer argumento), sin distribución inicial de datos (*P* parámetros *0*) y con un intercambio en el que cada procesador recibe datos y envía un total de *n* datos. Podemos utilizar la función T_{PAR} que consideremos más conveniente. Por ejemplo, si suponemos válida la hipótesis de *h*-relación de Valiant, existirán para cada arquitectura valores de *g* y *L* tales que:

$$T_{PAR}(P,0,...,0,2,n,...,n) = 2*n*g+L$$

El factor 2 en la fórmula anterior surge de considerar que el tiempo invertido en el patrón *EXCHANGE* es proporcional a la suma de los datos de entrada y de salida (cada procesador emite y recibe *n* datos). Aunque la existencia de puertos paralelos para la entrada y la salida sugieren utilizar el operador de máximo, nuestros experimentos [Rod98a] demuestran que en la mayoría de las máquinas actuales se obtiene una mayor precisión utilizando la suma. Si además hacemos la hipótesis de un equilibrio perfecto podemos, razonando recursivamente, obtener el tiempo total:

$$F = \sum_{i=0,\log(P)-1} B*n/2^i + C*(n/P)*\log(n/P) + \sum_{s=1,\log(P)} g*(2^s*n/P)+L$$

4.8.2. Análisis de la FFT

Consideremos el algoritmo de la Transformada Rápida de Fourier que ya presentamos en el epígrafe 3.3.9.5 del Capítulo 3 de esta memoria y su implementación en *llc* en la Figura 4.27.

El tiempo invertido en la división del vector original consume tiempo *O(n)*, siendo *n* el tamaño del vector, y la fase de combinación tiene la misma complejidad, *O(n)*.

Cuando no hay más procesadores disponibles en el código de la FFT de la Figura 4.27, se invoca en la línea 20 al algoritmo secuencial *seqFFT()*. El tiempo para el algoritmo secuencial es *O(n/P*log(n/P))*. Al final de las llamadas recursivas a *parFFT()*, los procesadores socios intercambian los resultados de sus cómputos y tiene lugar una comunicación de *n/2* complejos entre pares de procesadores socios. Después de este intercambio, ambos subconjuntos de procesadores se reunifican en el conjunto anterior.

El tiempo consumido por el algoritmo siguiendo el Modelo de Computación Colectiva viene dado por la expresión recursiva:

$$F = D*n/2 + F*n/2 + max\{F(parFFT(a2, A2, m)), F(parFFT(a1, A1, m))\} + T_{PAR}(P,0,...,0,2, m, ...,m)$$

El primer término corresponde al proceso de división de la señal original en sus componentes pares e impares. El segundo término corresponde al tiempo de la combinación de las señales transformadas. *D* y *F* son las constantes de complejidad asociadas con los procesos de división y combinación respectivamente. El tercer y cuarto término corresponden al superpaso de división. El tiempo consumido por el superpaso de división es el máximo de los tiempos consumidos en cada una de las transformadas paralelas más el tiempo T_{PAR} invertido en el intercambio de resultados que tiene lugar según el patrón EXCHANGE en una máquina con *P* procesadores (primer argumento), sin distribución inicial de los datos (*P* parámetros iguales a 0) con un intercambio en el que cada procesador envía y recibe *m* datos. Del mismo modo que en el algoritmo del Quicksort tenemos que:

$$T_{PAR}(P,0,...,0,2, m, ...,m) = 2*m*g_{PAR} + L_{PAR}$$

Razonando recursivamente podemos obtener el tiempo total:

$$F = \sum_{i=0, log(P)-1} D * n/2^i + C * (n/P) * log(n/P) + \sum_{s=1, log(P)} (g_{PAR} * 2^s * n/P) + L_{PAR} + F * 2^{s-1} * n/p)$$

C es la constante de complejidad correspondiente a la etapa en la que todos los procesadores computan la FFT secuencial.

En el Capítulo 6 presentaremos los resultados computacionales correspondientes a la comparación de los tiempos predichos por el modelo y medidos experimentalmente para este algoritmo.

4.8.3. Análisis del algoritmo de ordenación por Muestreo Regular

El algoritmo de ordenación conocido como Ordenación por Muestreo Regular (PSRS, Parallel Sort by Regular Sampling) propuesto por Li [Li93] es un ejemplo sencillo de algoritmo síncrono que encaja con el estilo BSP incluso sin utilizar una librería orientada a ese modelo. Como mostramos en el código MPI de la Figura 4.46, el algoritmo utiliza varias funciones colectivas de MPI. En este ejemplo se utiliza el

superpaso *normal* para aplicar el modelo. El coste de cada función colectiva se ha obtenido experimentalmente.

El algoritmo comienza con un broadcast personalizado (línea 2) desde el procesador 0 hacia los otros *P-1* procesadores de los diferentes segmentos del vector *A* de tamaño *N/P* que van a ser ordenados. El comunicador MPI *MPI_COMM_WORLD* se ha abreviado por *MCM*. El tiempo invertido por el procesador *NAME* en el primer superpaso $F_{1,NAME}$ es:

$$F_{1,NAME} = g_S * (P-1) * N/P + L_S \qquad (S = MPI\_Scatter)$$

En el segundo superpaso, cada uno de los *P* procesadores consume un tiempo *B*N/P*log(N/P)* en ordenar su segmento (llamada a *SequentialQuicksort()* en la línea 4). Después de eso, cada procesador elige una muestra de tamaño *P* (línea 6). Las muestras se recolectan en el procesador 0 (*MPI_Gather()* en la línea 7).

$$F_{2,NAME} = B*N/P*log(N/P) + C*P + g_G*P*(P-1) + L_G + F_{1,NAME}$$
$$(G = MPI\_Gather)$$

En el tercer superpaso, el procesador 0 coloca en *dest* el vector ordenado obtenido al mezclar los *P* vectores de muestras (línea 10) que fueron recibidos en el vector *S*. El tercer argumento *P1* en la llamada a *Pmerge()* es un vector cuyas componentes contienen los tamaños de los *P* vectores a ordenar. A continuación, el procesador 0 elige los *P-1* pivotes (línea 11) y los envía a los otros *P-1* procesadores (línea 13).

$$F_{3,NAME} = D*P^2+E*(P-1)+g_B*(P-1)*log(P)+L_B+F_{2,NAME}$$
$$(B = MPI\_Bcast)$$

El factor *log(P)* en el sumando correspondiente a las comunicaciones se debe a que asumimos una implementación eficiente de la rutina *MPI_Bcast()*. La función *ComputeFragments()* (línea 15) calcula el vector *MySzs* que contiene los tamaños de los

```
1 /* M_step = 1 */
 2 MPI_Scatter(A, N/P,MPI_INT,A,N/P,MPI_INT,0,MCW);
 3 /* M_step ++ */
 4 SequentialQuickSort(A,N/P);
 5 Rate = N/(P*P);
 6 for(i=0;i<P;i++) Sample[i] = A[i*Rate];
 7 MPI_Gather(Sample,P,MPI_INT,S,P,MPI_INT,0,MCW);
 8 /* M_step ++ */
 9 if (NAME == 0) {
10     PMerge(Dest, S,P1);
11     for(i=1;i<P;i++) Pivots[i-1] = Dest[i*P+P/2-1];
12 }
13 MPI_Bcast(Pivots,P-1,MPI_INT,0,MCW);
14 /* M_step ++ */
15 ComputeFragments(MySzs,A,Pivots);
16 MPI_Alltoallv(MySzs,Ons,Lin,MPI_INT,Szs,Ons,Lin,MPI_INT,MCW);
17 /* M_step ++ */
18 ComputeOffsets(MySize,OfIn, OfOu, Szs);
19 MPI_Alltoallv(A,MySzs,OfOu,MPI_INT,Seg,Szs,OfIn,MPI_INT,MCW);
20 /* M_step ++ */
21 PMerge(Dest,Seg,Szs);
22 MPI_Gather(&MySize,1,MPI_INT,Cnt,1,MPI_INT,0,MCW);
23 /* M_step ++ */
24 if (NAME == 0) ComputeDisplacements(OfIn, Cnt);
25 MPI_Gatherv(Dest, MySize,MPI_INT,A,Cnt,OfIn,MPI_INT,0,MCW);
```

Figura 4.46 Código MPI para el algoritmo PSRS

segmentos compuestos por los elementos que están entre dos pivots sucesivos. Este vector se calcula mediante una búsqueda binaria de cada uno de los *P-1* pivots en los *N/P* elementos del correspondiente segmento ordenado del vector *A.* Los tamaños de los fragmentos se intercambian en la comunicación *AlltoAll* personalizada de la línea 16. De acuerdo a los requisitos de la función *MPI_Alltoallv(),* los *P* elementos en el segundo argumento, *Ons*, contienen los tamaños de los elementos a enviar, esto es, uno. El tercer argumento, *Lin*, contiene los offsets, de forma que el i-ésimo elemento en el vector *Lin* es i.

$$F_{4,NAME} = F*(P-1)*log(N/P) + g_{AA}*2*(P-1) + L_{AA} + F_{3,NAME}$$
$$(AA = MPI\_Alltoallv)$$

Cuando los procesadores han obtenido sus correspondientes tamaños en el vector *Szs*, computan los offsets, *OfIn* en los que los segmentos recibidos han de almacenarse y los offsets *OfOu* donde comienzan los segmentos a enviar. También se calcula el número final de elementos que acaban en el procesador, *MySize* (llamada a *ComputeOffsets()* en la línea 18). Los resultados teóricos y prácticos presentados en [Li93] demuestran que se puede esperar un tamaño promedio de N/P^2 elementos para cada segmento. Al final del quinto superpaso, cada procesador envía los *P-1* segmentos de N/P^2 elementos y recibe *P-1* segmentos de tamaño N/P^2 de los otros procesadores (línea 19)

$$F_{5,NAME} = H*(P-1) + g_{AA}*2*(P-1)*N/P^2 + L_{AA} + F_{4,NAME}$$
$$(AA = MPI\_Alltoallv)$$

Los *P* segmentos ordenados se mezclan en el código de la línea 21. Para preparar la recolección de los segmentos por el procesador raíz, los *P-1* procesadores envían los tamaños *MySize* al procesador 0 en la línea 22.

$$F_{6,NAME} = I*N/P + g_G*(P-1) + L_G + F_{5,NAME}$$
$$(G = MPI\_Gather)$$

El procesador 0 calcula los offsets *OfIn* para los segmentos ordenados en la línea 24. En la línea 25 los *P-1* intervalos recolectados se colocan directamente en su posición final en el vector *A.* La fórmula para el tiempo total del algoritmo se puede obtener sustituyendo iterativamente:

$$F_{7,NAME} = J*(P-1) + g_G*(P-1)*N/P + L_G + F_{6,NAME}$$
$$(G = MPI\_Gather)$$

En el Capítulo 6 presentamos resultados computacionales de la predicción del modelo para este algoritmo.

4.9. La Laguna C

Las ideas del modelo de computación colectiva pueden implementarse de diferentes modos. La mayoría de los algoritmos que se han presentado en este Capítulo han sido realizados utilizando *La Laguna C* (*llc)* [Rod97a], [Rod98e]. También los resultados computacionales que se presentan en el Capítulo 6 se han obtenido utilizando esta herramienta. Hemos desarrollado *llc* utilizando C como lenguaje base. La herramienta extiende el modelo de librería de paso de mensajes con las ideas de la computación colectiva. *Llc* extiende las librerías de paso de mensajes (una de las formas más eficiente de programación paralela) con una serie de constructos que permiten al programador expresar de forma cómoda y sencilla algoritmos con paralelismo anidado e

incluso algoritmos PRAM. Dado que el programador no está obligado a abandonar el entorno de librería de paso de mensajes, la herramienta preserva toda la eficiencia de este modelo. Los resultados computacionales muestran que su utilización produce resultados similares a los obtenidos programando explícitamente con paso de mensajes, a pesar de que la herramienta facilita el desarrollo de algoritmos paralelos. El hecho de estar diseñada en base a librerías de paso de mensajes dota también a la herramienta de una gran portabilidad, en tanto en cuanto puede ser implantada en cualquier sistema con soporte para este tipo de librerías. Actualmente se dispone de versiones de *llc* para redes de transputers soportada en Inmos C [Inm90] y para multicomputadores tanto de memoria compartida como de memoria distribuida que soporten MPI [Mpi94] o PVM [Gei94]. *Llc* exige por parte del programador pocos conocimientos respecto a detalles habitualmente fundamentales en la programación paralela.

Es posible desarrollar expresamente un nuevo lenguaje de programación y un compilador que produzca código adecuado a la filosofía de la computación colectiva. Con la intención de demostrar rápidamente las ventajas del modelo, *llc* se ha diseñado soportado por macros escritas en C. Las macros de *llc* esconden al usuario no especializado lo que podríamos denominar 'detalles de bajo nivel' del paralelismo, pero el código resultante, perfectamente podría ser generado por un compilador cuyo lenguaje fuente fuera cualquier lenguaje procedural en el que se hubieran introducido las construcciones adecuadas para soportar las ideas de la computación colectiva. Los experimentos que hemos realizado en este sentido y los resultados obtenidos confirman la bondad de esta otra aproximación [Lun98].

Las macros de *llc* están distribuidas en diferentes ficheros. Un programador que desee utilizar llamadas a las macros de la librería, debe incluir en su programa en C el fichero de definiciones de estas macros:

```
#include <llcsync.h>
```

Antes de utilizar cualquier función de la librería se debe realizar una llamada a la macro INITIALIZE, que como su nombre indica es la encargada en *llc* de llevar a cabo las inicializaciones pertinentes. La macro EXIT es la encargada de finalizar la ejecución de un programa *llc*, y la llamada a esta macro ha de colocarse al final del programa del usuario. Entre ambas llamadas, el programador puede usar libremente cualquiera de las funcionalidades suministradas por la herramienta. Todas las macros y funciones del sistema se escriben en mayúsculas. El contenido de los ficheros de macros del sistema es por una parte transparente al usuario, pero por otra parte está disponible para usuarios especializados que desearan realizar cambios o introducir nuevas funcionalidades en el sistema.

```
int main(int argc, char *argv[]) {
  int first, last;
  double start, finish;

  INITIALIZE;
  initialize(array);
  INITCRONO(start);
  qs(0, SIZE-1);
  ENDCRONO(finish);
  test(array);
  GPRINTF("%lf\n", finish - start);
  EXIT;
}
```

Figura 4.47 El programa principal correspondiente al Quicksort

El programador escribe un único programa, obviando así la diferenciación que a veces es necesaria en programación mediante paso de mensajes entre código que ha de ser ejecutado en el procesador raíz (root) y código que se ejecutará en el resto de procesadores (nodes). La Figura 4.47 muestra por ejemplo el programa principal correspondiente al Quicksort que hemos presentado en el epígrafe 4.6.1.4 de este mismo Capítulo

En el código de esa figura apreciamos la llamada a la función GPRINTF de *llc* que permite que todos los procesadores de la arquitectura impriman en pantalla (con un formato análogo al de *printf()*). Las macros INITCRONO y ENDCRONO permiten al programador evaluar el tiempo transcurrido hasta el instante en que el control alcanza la llamada. Las llamadas a GPRINTF, EXIT e INITIALIZE son ejemplos de lo que en *llc* se llaman *operaciones de subconjuntos*. La siguiente

> **Regla:**
> **Todos los procesadores de un subconjunto han de participar al ejecutar una *operación de subconjunto*.**

debe satisfacerse siempre al utilizar cualquier *operación de subconjunto.*

Es un error semántico si alguno de los miembros de un conjunto de procesadores ejecuta una *operación de subconjuntos* y otros no. Una operación de subconjuntos conlleva una sincronización entre los miembros del conjunto que la llevan a cabo. Por otra parte, éstas son las únicas sincronizaciones del sistema *llc*, que carece de sincronizaciones explícitas. Las sincronizaciones tienen lugar de forma implícita cada vez que los procesadores de un conjunto se comunican como consecuencia de la realización de una operación de subconjunto. *Llc* cuenta con un gran número de este tipo de operaciones. Las funciones de división (la familia de macros PAR) son asimismo *operaciones de subconjuntos*.

Al igual que el programador tiene acceso al número de procesadores disponibles en el conjunto actual, a través de la variable NUMPROCESSORS, en *llc* la variable NAME permite identificar de forma unívoca cada procesador de un conjunto de procesadores hoja. Ambas variables (NAME y NUMPROCESSORS) son accesibles al usuario sólo para lectura. El código del algoritmo de búsqueda que presentamos en la Figura 4.35 es un ejemplo de utilización de estas variables.

```
1 #define TYPE int
 2 #define SIZEOF(x) x[0]*sizeof(int)
 3 #define OPERATION(y, x) {merge(y, x); y[0] += x[0];}
 4
 5 int *arraytosort, *arraysorted;
 6 . . .
 7 arraytosort[0] = SIZE;   /* Size of the array */
 8 quicksortseq (arraytosort, 1, SIZE);
 9 REDUCEDYN(arraysorted, arraytosort);
10  . . .
```

Figura 4.48 Merge sort utilizando una reducción dinámica

El sistema *llc* dispone de un rico y flexible conjunto de operaciones colectivas de reducción, entre las que se encuentran REDUCEBYMAX, REDUCEBYADD, REDUCEBYMULT, REDUCE (reducción utilizando una función suministrada por el usuario), etc. La implementación de estas operaciones de reducción siguiendo patrones hipercúbicos de comunicaciones se puede lograr de forma muy eficiente. Las reducciones son otro ejemplo de *operaciones de subconjuntos*.

Aparte de este tipo de reducciones estáticas, *llc* dispone también de operaciones de reducción dinámicas. Las reducciones dinámicas se utilizan en caso de que se manipulen datos cuyo tamaño en memoria no se conoce a priori. El código de la Figura 4.48 es un ejemplo de utilización de una reducción dinámica en un algoritmo de ordenación.

En este ejemplo el vector a ordenar está inicialmente distribuido entre los diferentes procesadores (cada procesador posee un segmento del vector a ordenar). *arraytosort* es el segmento correspondiente a cada procesador. Para utilizar una reducción dinámica, el programador ha de especificar el tipo de los datos involucrados (Figura 4.48, línea 1), la macro SIZEOF (línea 2) que devolverá el tamaño en memoria de los operandos implicados en la reducción y la operación (línea 3) mediante la cual se lleva a cabo la reducción. Esta operación tendrá el formato OPERATION(*x*, *y*), y ha de retornar en su primer argumento, *x* el resultado de operar los dos argumentos.

En el código de la Figura 4.48 se utiliza la primera componente de los vectores para almacenar el número de componentes de éstos, y la macro OPERATION, definida por el programador, se encarga de actualizar el tamaño del resultado de la mezcla de dos vectores, que se lleva a cabo utilizando la función *merge()*, también suministrada por el programador. El programa comienza por colocar en la primera componente del vector a ordenar el número de componentes de éste. Cada procesador ordenará su segmento utilizando para ello un Quicksort secuencial (*quicksortseq*) y la llamada a la macro de reducción dinámica REDUCEDYN de la línea 9 provocará que todos los procesadores obtengan en *arraysorted* el array completo ordenado.

Capítulo V
OTROS LENGUAJES Y HERRAMIENTAS

Capítulo V

Otros Lenguajes y Herramientas

5.1. NESL

NESL [Ble96], [Ble95] es un lenguaje paralelo que ha sido desarrollado en el Carnegie Mellon dentro del proyecto Scandal. El interés principal de este proyecto consiste en el desarrollo de un entorno de programación portable e interactivo para diversos tipos de supercomputadores. Los dos objetivos principales del proyecto son el desarrollo de un lenguaje paralelo: NESL y la implementación eficiente de algoritmos paralelos. El lenguaje integra ideas de la algorítmica paralela, de los lenguajes funcionales y del mundo del diseño de sistemas. Los dos conceptos más importantes en que se basa NESL son el paralelismo anidado y un modelo de medida de eficiencia basado en el lenguaje.

Cuando se diseñó el lenguaje, los objetivos fueron los siguientes:

1.- Soportar paralelismo a través de un conjunto de constructos de paralelismo de datos (el paralelismo se obtiene mediante operaciones sobre datos) basados en secuencias. Estos constructos proporcionan paralelismo mediante (a) la capacidad de aplicar cualquier función, concurrentemente sobre cada uno de los elementos de una secuencia, y (b) un conjunto de funciones paralelas que operan sobre secuencias, como por ejemplo la función *permute()*, que permuta el orden de los elementos de una secuencia.

2.- Soportar paralelismo anidado. NESL soporta secuencias anidadas y la capacidad de aplicar cualquier función definida por el usuario sobre los elementos de una secuencia, incluso en el caso que la función en sí misma sea paralela y los elementos de la secuencia sean asimismo secuencias.

3.- Generar código eficiente para diversas arquitecturas, incluyendo máquinas tanto SIMD como MIMD, con memoria compartida y distribuida.

4.- Que fuera un lenguaje adecuado para describir algoritmos paralelos, y suministrar un mecanismo para calcular el tiempo de ejecución directamente a partir del código. Cada función en NESL lleva asociadas dos medidas de complejidad: las complejidades de cómputo y profundidad. Una ecuación simple asocia a estas complejidades el correspondiente tiempo de ejecución teórico para el modelo PRAM.

NESL es un lenguaje funcional de primer orden (las funciones no pueden ser pasadas como datos), fuertemente tipificado y libre de efectos laterales, con una sintaxis similar a la del lenguaje ML [Mil90], que se ejecuta en un entorno interactivo. El lenguaje utiliza la secuencia como tipo de datos paralelo primitivo, y el paralelismo se obtiene exclusivamente a través de operaciones sobre estas secuencias. En este sentido, el programador ha de idear operaciones paralelas sobre conjuntos de valores más que pensar en cómo se han de asignar datos a los procesadores y el control de estos procesadores. NESL soporta secuencias anidadas (secuencias de secuencias) y el paralelismo se suministra a través de un conjunto de constructos de paralelismo de datos basados en secuencias, incluyendo un mecanismo para aplicar cualquier función sobre los elementos de una secuencia en paralelo, así como un conjunto de funciones paralelas que manipulan secuencias. El conjunto de funciones suministradas por el lenguaje fueron escogidas en base a su utilidad en el diseño de algoritmos paralelos así como a la eficiencia de su implementación en máquinas paralelas. Sus autores explican que el lenguaje no proporciona bucles secuenciales (que pueden simularse usando recursión) para promover la utilización del paralelismo.

La versión actual del compilador traduce NESL a VCODE [Ble90a], un lenguaje intermedio que corre en multiprocesadores vectoriales (los Cray C90 y J90) así como en máquinas de memoria distribuida (la IBM SP2, Intel Paragon y CM-5). El compilador utiliza la técnica del "aplanamiento del paralelismo anidado" [Ble90b] para traducir NESL al modelo de paralelismo de datos plano suministrado por VCODE. Para las diferentes arquitecturas en que es posible ejecutar NESL, los autores suministran intérpretes de VCODE. Asimismo se suministra una versión de VCODE basada en MPI, lo cual permite correr NESL en máquinas con soporte para esta librería. Los autores del lenguaje afirman que las funciones sobre secuencias en este intérprete han sido fuertemente optimizadas, de modo que la ineficiencia introducida por la interpretación es pequeña. El entorno interactivo de NESL se ejecuta sobre Common Lisp. Las llamadas interactivas son compiladas a VCODE y a continuación ejecutadas.

5.1.1. Operaciones paralelas sobre secuencias

NESL soporta paralelismo a través de las operaciones sobre secuencias, que se especifican usando corchetes. Por ejemplo:

```
[1, 7, -5, 3, -2]
```

es una secuencia de cinco enteros. Todos los elementos de una secuencia deben ser del mismo tipo y las secuencias han de ser de longitud finita. El paralelismo sobre las secuencias se puede obtener de dos modos: mediante la capacidad de aplicar una función concurrentemente sobre cada elemento de una secuencia y mediante un conjunto de funciones predefinidas que operan sobre secuencias. El constructo que llamaremos *aplicar-a-cada-uno* consiste en la aplicación de una función a una secuencia y se denota mediante una notación de conjuntos. Por ejemplo, la expresión:

```
{negate(a) : a in [2, 9, -3, 6]};
➔ [-2, -9, 3, -6] : [int]
```

cambia el signo de cada uno de los elementos de la secuencia original. Este constructo puede leerse como: "en paralelo, para cada *a* de la secuencia `[2, 9, -3, 6]` negar

a". El símbolo ➔ indica el resultado de la expresión, y la expresión `[int]` especifica el tipo del resultado: una secuencia de enteros. El constructo *aplicar-a-cada-uno* también posibilita la selección de elementos de la secuencia: la expresión

```
{negate(a) : a in [2, 9, -3, 6] | a < 4};
➔ [-2, 3] : [int]
```

puede leerse como: "en paralelo, para cada *a* de la secuencia `[2, 9, -3, 6]` tal que *a* sea menor que `4`, complementar *a*". Los elementos resultantes mantienen su orden relativo con respecto a la secuencia original. También es posible operar sobre múltiples secuencias. La expresión

```
{a + b : a in [5, -4, 3, 2]; b in [1, 2, -7, 0]};
➔ [6, -2, -4, 2] : [int]
```

suma las dos secuencias elemento a elemento.

En NESL, cualquier función, ya sea predefinida o definida por el usuario puede ser aplicada a cada uno de los elementos de una secuencia. Por ejemplo, se puede definir una función factorial:

```
function factorial(i) =
  if (i == 1) then 1
  else i*factorial(i-1);
➔ factorial : int -> int
```

y aplicarla sobre los elementos de una secuencia:

```
{factorial(x) : x in [1, 3, 7]};
➔ [1, 6, 5040] : [int]
```

el tipo de la función (`int -> int`) es deducido por el sistema de inferencia de tipos del compilador.

Un constructo *aplicar-a-cada-uno* aplica un cuerpo a cada uno de los elementos de una secuencia. Llamaremos a una de estas aplicaciones una *instancia*. Dado que NESL no tiene efectos laterales (estrictamente hablando), no hay forma de comunicación entre diferentes instancias de un constructo *aplicar-a-cada-uno*, por lo tanto, una implementación puede ejecutar las instancias en cualquier orden arbitrario sin cambiar el resultado. En particular, las instancias pueden ser implementadas en paralelo dando al constructo *aplicar-a-cada-uno* su semántica paralela.

La otra forma de obtener paralelismo en NESL es mediante un conjunto de funciones de secuencia. Las funciones de secuencia operan sobre secuencias completas y todas se caracterizan por tener implementaciones paralelas relativamente simples. Por ejemplo, la función *sum()*, suma los elementos de una secuencia:

```
sum([1, 3, 5, -2, 4]);
➔ 11 : int
```

Otra función de secuencia común es la operación permute, que permuta una secuencia basándose en una segunda secuencia de índices. Por ejemplo:

```
permute("sande", [2, 1, 3, 0, 4]);
➔ "nadse" : [char]
```

en este caso, los cinco caracteres de la cadena "sande" se permutan según los índices de la segunda secuencia.

La Tabla 5.1 muestra algunas de las funciones de secuencia disponibles en NESL. Aquellas funciones que se marcan con un asterisco constituyen un conjunto de primitivas: estas primitivas junto con los operadores escalares y el constructo *aplicar-a-cada-uno* son suficientes para implementar el resto de funciones.

Operación	*Descripción*
`* dist(a, n)`	Distribuye el valor `a` en una secuencia de longitud `n`.
`* #a`	Devuelve la longitud de la secuencia `a`
`a[i]`	Devuelve el elemento de `a` que ocupa la posición `i`
`rep(d, v, i)`	Reemplaza el elemento en posición `i` de `d` con `v`.
`[s:e]`	Retorna la secuencia con los enteros en el intervalo [`s`, `e`)
`[s:e:d]`	Igual que la anterior, con salto (stride) `d`
`Sum(a)`	Devuelve la suma de los valores de la secuencia `a`.
`* ©_scan(a)`	Devuelve 'scan' basado en el operador ©.
`count(a)`	Cuenta el número de flags true en `a`.
`permute(s, i)`	Permuta los elementos de `s` a las posiciones `i`.
`* d <- a`	Escribe los elementos `a` en `d`.
`* a -> i`	Lee de la secuencia `a` basándose en los índices `i`.
`max_index(a)`	Devuelve el índice del valor máximo.
`min_index(a)`	Devuelve el índice del valor mínimo.
`a++b`	Concatena las secuencias `a` y `b`.
`drop(a, n)`	Elimina los primeros `n` elementos de la secuencia `a`.
`take(a, n)`	Toma los primeros `n` elementos de la secuencia `a`.
`rotate(a, n)`	Rota `n` posiciones la secuencia `a`.
`* flatten(a)`	Aplana la secuencia anidada `a`.
`* partition(a, 1)`	Particiona la secuencia `a` en una secuencia anidada.
`split(a, f)`	Divide `a` en una secuencia anidada en base a las flags `f`.
`bottop(a)`	Divide `a` en una secuencia anidada.

Tabla 5.1 Algunas de las funciones de secuencia de NESL

5.1.2. Paralelismo anidado

En NESL los elementos de una secuencia pueden ser cualquier tipo de datos, incluso otras secuencias. Esta regla permite el anidamiento de secuencias a una profundidad arbitraria. Una secuencia anidada puede escribirse como:

```
[[2, 3, 5], [7, 4], [9]]
```

esta secuencia tiene tipo *[[int]]* (es decir, una secuencia de secuencias de enteros). Consideradas las secuencias anidadas y la regla que indica que cualquier función puede ser aplicada en paralelo a los elementos de una secuencia, NESL suministra la posibilidad de aplicar una función paralela varias veces en paralelo, o lo que es lo mismo, paralelismo anidado. Por ejemplo, podemos aplicar la función paralela *sum()* a una secuencia anidada:

```
{sum(v) : v in [[2, 3, 5], [7, 4], [9]]};
➔ [10, 11, 9] : [int]
```

en esta expresión aparece paralelismo tanto dentro de cada *sum*, puesto que las funciones de secuencias tienen implementaciones paralelas como por el hecho de que se invocan tres instancias de sum, dado que el constructo *aplicar-a-cada-uno* se define de tal modo que todas las instancias pueden correr en paralelo.

NESL suministra varias funciones que permiten moverse entre niveles de anidamiento. Entre estas funciones se encuentra *flatten()*, que toma una secuencia anidada y 'la aplana' (desanida) un nivel. Por ejemplo,

```
flatten([[2, 3, 5], [7, 4], [9]]);
➔[2, 3, 5, 7, 4, 9] : [int]
```

Otra función de este tipo es *bottop()*, que toma una secuencia de valores y crea una secuencia anidada de longitud dos con todos los elementos de una mitad en una subsecuencia y los de la otra mitad en la otra subsecuencia (si el número de elementos en la secuencia es impar, la primera incluirá al elemento extra). Por ejemplo,

```
bottop("La Laguna");
➔["La La", "guna"] : [[char]]
```

5.1.3. Pares.

En NESL, un par es una estructura con dos elementos, pudiendo ser cada uno de ellos de cualquier tipo. Los pares se utilizan para construir estructuras simples o para hacer que una función retorne varios valores. El operador binario *coma* se utiliza para crear pares. Por ejemplo:

```
3.14, "pi"
➔ (3.14, "pi") : (float, [char])
```

El operador binario *coma* es asociativo a derechas (por ejemplo, *(1, 2, 3, 4)* es equivalente a *(1, (2, (3, 4)))*). El resto de operadores binarios de NESL son asociativos a izquierdas. La precedencia del operador coma es menor que la de cualquier otro operador binario, por lo que es frecuentemente necesario colocar los pares entre paréntesis.

El emparejamiento de los patrones dentro de un constructo *let* puede utilizarse para descomponer la estructura de los pares. Por ejemplo:

```
let(x, y, z) = (3-2, 5, 3*4)
in x-y+z
➔8 : int
```

en el ejemplo, *x* toma el valor *3-2=1*, *y* vale *5* y *z* valdrá *3*4=12*.

Los pares anidados se diferencian de las secuencias en varios aspectos importantes. En primer lugar no hay forma de operar en paralelo sobre los elementos de un par anidado. En segundo lugar, los elementos de un par no tienen porqué ser del

mismo tipo, mientras que los elementos de una secuencia han de ser siempre del mismo tipo.

5.1.4. Tipos

NESL es un lenguaje fuertemente tipificado y polimórfico con un sistema de inferencia de tipos. Su sistema de tipos es similar al de lenguajes funcionales como ML, pero dado que es de primer orden, los tipos de las funciones sólo aparecen en el nivel superior.

El tipo de una función polimórfica en NESL se especifica mediante variables de tipos, que se declaran en lo que llamaremos un contexto de tipos. Por ejemplo, el tipo de la función *permute()* es:

```
([A], [int]) -> [A] :: A in any
```

Esto especifica que para *A* de cualquier tipo, la función *permute()* asigna a una secuencia de tipo `[A]` y a una secuencia de tipo `[int]` una secuencia de tipo `[A]`. La variable *A* es una variable de tipo, y `A in any` es el contexto. Un contexto puede tener múltiples enlaces a tipos separados por punto y coma. Por ejemplo, la función *zip()* que une dos secuencias de igual longitud para formar una secuencia de pares tiene tipo:

```
([A], [B]) -> [(A, B)] :: A in any; B in any;
```

Las funciones definidas por el usuario también pueden ser polimórficas. Por ejemplo, podríamos definir

```
function append3(s1, s2, s3) = s1 ++ s2 ++ s3;
➔ append(s1, s2, s3) : ([A], [A], [A]) :: A in any
```

El sistema de inferencia de tipos tratará siempre de determinar el tipo más general posible.

Además de polimorfismo paramétrico, NESL soporta una forma de sobrecarga a través de la noción de clases de tipos. Una clase de tipos es un conjunto de tipos junto con un conjunto de funciones asociado. Las funciones de una clase pueden aplicarse solamente a los tipos de esa clase. Por ejemplo, los tipos base `int` y `float` son ambos miembros de la clase de tipos `number`, y las funciones numéricas como `+` y `*` están definidas para operar con todo tipo de números. El tipo de una función sobrecargada en este sentido se especifica limitando el contexto de una variable de tipo a una clase de tipos concreta. Por ejemplo, el tipo de `+` es:

```
(A, A) -> A :: A in number
```

El contexto "`A in number`" especifica que `A` puede estar enlazada a cualquier miembro de la clase de tipos `number`. La especificación completamente polimórfica `any` puede verse como una clase de tipos que contiene a todos los tipos de datos. Las clases de tipos se organizan según la jerarquía que aparece en la Figura 5.1. Funciones como `=` y `<` están definidas sobre los tipos de la clase `ordinal`, funciones como `+` y `*` sobre los de `number` y funciones como `or` y `not` sobre los tipos pertenecientes a la clase `logical`.

Las funciones definidas por el usuario también pueden estar sobrecargadas. Por ejemplo:

```
function doble(a) = a + a;
➔doble(a) : A -> A :: A in number
```

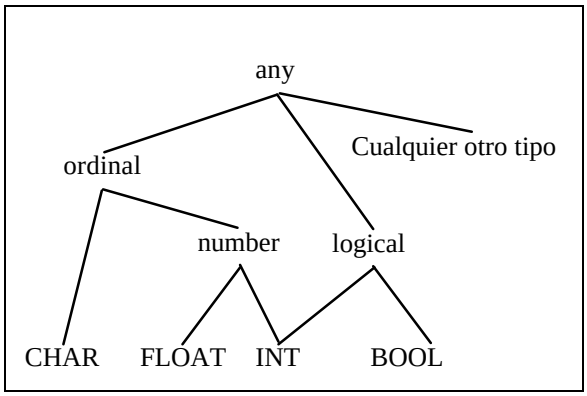

Figura 5.1 La jerarquía de las clases de tipos en NESL

También es posible restringir el tipo de una función definida por el usuario tipificándola explícitamente:

```
function doble(a) : int -> int = a + a;
➔doble(a) : int -> int
```

Aquí el tipo de la función *doble()* está limitado a `int -> int`. Los dos puntos especifican que lo que sigue es un especificador de tipos.

En algunas situaciones el sitema de inferencia de tipos no puede determinar el tipo, incluso aunque haya uno. Por ejemplo, la función:

```
function erronea(a, b) = a or (a + b);
```

no tiene un tipo adecuado porque la función *or* está definida sobre la clase de tipos `logical` y `+` está definida para los tipos de `number`. Ocurre que `int` es un tipo lógico y entero, pero el sistema de inferencia de tipos de NESL no puede hacer intersecciones de clases de tipos. En esta situación será necesario explicitar el tipo de la función:

```
function correcta(a, b) : (int, int) -> int = a or
(a + b);
➔ correcta(a, b) : (int, int) -> int
```

Por otra parte, la especificación del tipo de las funciones usando los dos puntos ‘:’ contribuye a la buena documentación de las funciones incluso en el caso de que el sistema de inferencia de tipos sea capaz de determinarlo. La noción de clases de tipos de NESL es similar a la del lenguaje Haskell [Hud90], con la diferencia de que NESL no permite al usuario añadir nuevas clases de tipos.

5.1.5. Definición del lenguaje

NESL tiene los siguientes tipos de datos:

- 4 tipos de datos atómicos: boolean (`bool`), enteros (`int`), caracteres (`char`) y reales (`float`);
- El tipo primitivo secuencia.
- El tipo primitivo par.
- Tipos de datos compuestos definibles por el programador.

Y las siguientes operaciones:

- Un conjunto de funciones predefinidas que operan sobre los tipos primitivos.

- Tres constructos primitivos: condicional *if*, un constructo de asignación *let* y el constructo *aplicar-a-cada-uno*.
- Un constructor de funciones, *function*, para definir nuevas funciones.

Describimos a continuación cada uno de estos elementos.

5.1.5.1. Datos

5.1.5.1.1. Tipos de datos atómicos

Hay cuatro tipos de datos atómicos básicos: booleanos, enteros, caracteres y reales.

El tipo booleano *bool* puede tener los valores *t* o *f*. Están predefinidas las operaciones lógicas habituales: *not, and, or, xor, nor, nand*). Las operaciones lógicas binarias utilizan notación infija. Por ejemplo:

```
not(not(t));
➔ t : bool
```

El tipo entero *int* es el conjunto de los enteros (positivos y negativos) que pueden ser representados con una precisión que viene fijada por el tamaño de palabra de la máquina. La precisión será al menos de 32 bits. Están predefinidos los operadores habituales para enteros (*+, -, *, /, ==, <, >, negate, ...*) y utilizan notación infija. Por ejemplo:

```
5*15;
➔ 45 : int

1 == 0;
➔ f : bool
```

El tipo carácter, *char* es el conjunto de caracteres ASCII. Los caracteres tienen un orden fijo y se puede usar con ellos los operadores de comparación. Los caracteres se escriben colocando una comilla simple delante. Por ejemplo:

```
‘8;
➔ ‘8 : char

‘a == ‘z
➔ f : bool

‘a < ‘z
➔ t : bool
```

Las variables globales predefinidas *space*, *newline* y *tab* están asociadas con los correspondientes caracteres ASCII.

El tipo *float* se utiliza para especificar números representados en punto flotante. NESL trata de utilizar el formato IEEE de 64 bits siempre que sea posible. Los reales soportan las mismas operaciones que los enteros y además algunas funciones adicionales (*round(), truncate(), sqrt(), log(), ...*). Los reales deben escribirse colocando un punto decimal en ellos para distinguirlos de los enteros.

```
3.0 * 2.0;
➔ 6.0 : float

round(3.1416);
```

```
➔ 3 : int
```

No hay coerción implícita entre tipos escalares. Para sumar 2 y 5.0, por ejemplo es necesario coercionar explícitamente uno de los dos operandos:

```
float(2) + 5.0;
➔ 7.0 : float
```

5.1.5.1.2. Secuencias ([])

Una secuencia puede contener cualquier tipo de datos, incluso otras secuencias, pero todos los elementos de una secuencia han de ser del mismo tipo. El tipo de una secuencia cuyos elementos son de un tipo α, se especifica como [α]. Por ejemplo:

```
[1, 2, 3, 4];
[1, 2, 3, 4] : int
```

es una secuencia de enteros, mientras que

```
[[1,2], [3, 4], [5, 6, 7]];
[[1,2], [3, 4], [5, 6, 7]] : [[int]]
```

es una secuencia de secuencias de enteros.

Las secuencias de caracteres pueden escribirse entre comillas dobles:

```
“La Laguna”
➔ “La Laguna” : [char]
```

aunque también pueden escribirse como una secuencia de caracteres:

```
[‘L, ’a, space, ‘L, ’a, ’g, ’u, ’n, ’a];
➔ “La Laguna” : [char]
```

Las secuencias vacías han de ser explícitamente tipificadas, dado que el tipo no puede ser determinado a partir de sus elementos. El tipo de una secuencia vacía se especifica utilizando paréntesis seguidos del tipo de los elementos:

```
[] int;
➔ [] : [int]

[] (int, bool);
➔ [] : [(int, bool)]
```

5.1.5.1.3. Tipos registros (`datatype`)

Los registros con un número fijo de campos pueden definirse con el constructo `datatype.` Por ejemplo,

```
datatype complejo(float, float);
➔complejo(a1, a2) : float, float -> complejo
```

define un registro con dos campos de tipo real. Al definir un registro también se define una función que se utiliza para construir el registro. Así

```
complejo(3.14, 5.8);
➔complejo(3.14, 5.8) : complejo
```

crea un registro de tipo `complejo` con `3.14` y `5.8` como componentes.

Los elementos de un registro pueden accederse utilizando asociación de patrones con el constructo *let*. Por ejemplo,

```
let complejo(real, imaginaria) = a
in real;
```

eliminará la parte real de la variable *a* (asumiendo que se mantiene en el primer campo).

Al igual que las funciones, los registros pueden parametrizarse basándose en las variables de tipo. Por ejemplo *complejo* podría haberse definido como:

```
datatype complejo(alfa, alfa) :: alfa in numero;
➔ complejo(a1, a2) : alfa, alfa -> complejo(alfa)
:: alfa in number
```

Esta definición especifica que estando *alfa* ligada a cualquier tipo de la clase de tipos *numero* (tanto *int* como *float*), ambos campos del registro han de ser de tipo alfa. Esto permitiría,

```
complejo(1.0, 3.1);
➔ complejo(1.0, 3.1) : complejo(float)

complejo(1, 3);
➔ complejo(1, 3) : complejo(int)
```

pero no permitiría

```
complejo(7, ‘a);
```

ni tampoco

```
complejo(2, 1.9);
```

El tipo de un registro se especifica mediante el nombre del registro seguido por el enlace de todas sus variables de tipo. En este ejemplo, el enlace de las variables de tipo es *int* o *float*.

5.1.5.2. Funciones y constructos

5.1.5.2.1. Condicionales (`if`)

La única forma primitiva de condicional en NESL es el constructo *if*. Su sintaxis es:

```
IF expr THEN expr ELSE expr
```

Si la primera expresión es cierta, se evalúa la segunda expresión y se retorna su valor. En caso contrario se retorna el resultado de la evaluación de la tercera expresión. La primera expresión ha de ser de tipo *bool*, y las otras dos expresiones han de tener el mismo tipo. Por ejemplo,

```
if (t or f) then 2+5 else (3+4)*5
```

es una expresión válida, pero

```
if (t or f) then 2 else 1.5
```

no lo es, porque las alternativas retornan valores de tipos diferentes.

5.1.5.2.2. Asignación a variables locales (`let`)

Las variables locales pueden asignarse con el constructo *`let`*. Su sintaxis en EBNF es:

```
LET exprenlaces in expr
exprenlaces ::= exprenlace [; exprenlaces]
exprenlace ::= patrón = expr
patrón ::= id |
     id (patrón) |
     patrón, patrón |
     (patrón)
```

Cada patrón es o bien un identificador de variable o un patrón basado en un identificador de registro. Cada expresión de enlace liga las variables del patrón a la izquierda del signo = con el resultado de la expresión de la derecha. Veamos un ejemplo:

```
let x = 5
  (w, z) = (3, 4)
in x+w*z;
➔ 17 : int
```

Aquí *x* es asignada con el valor *5* y el patrón *`(w, z)`* es emparejado con el resultado de la expresión de la derecha, de forma que a *w* se le asigna el valor *`3`* y a *z* *`4`*. Los patrones pueden anidarse y se emparejan recursivamente.

Las variables que aparecen en cada expresión de enlace pueden utilizarse en la expresión (*`expr`*) de cualquier expresión de enlace posterior (los enlaces se hacen en serie). Por ejemplo, en la expresión

```
let x = 3
   z = x + 2
in x*z;
➔ 15 : int
```

A la variable *x* se le asigna el valor *3* y a *z* se le asigna el valor de *x* más *2*, que es *5*. Cuando *x* y *z* se multiplican, el resultado es *15*.

5.1.5.2.3. El constructor *aplicar-a-cada-uno*

El constructor *aplicar-a-cada-uno* se utiliza para aplicar una función a los elementos de una secuencia. Tiene la siguiente sintaxis:

```
{[expr :] r_enlaces [ | expr]}
r_enlaces ::= r_enlace [; r_enlaces]
r_enlace ::= patrón IN expr |        Enlace completo
             id                      Enlace abreviado
```

Un constructor *aplicar-a-cada-uno* consta de tres partes: la expresión antes de los dos puntos, a la que llamaremos el cuerpo, los enlaces que siguen al cuerpo, y la expresión que sigue al símbolo |, a la que llamaremos la criba. Tanto el cuerpo como la criba son opcionales; pueden omitirse, como en

```
{x in [5, 6, 7]}
➔ [5, 6, 7] : [int]
```

Los r_enlaces pueden contener varios enlaces, separados por punto y coma. Consideremos en primer lugar el caso en el que hay un único enlace. Un enlace puede consistir tanto en un patrón seguido por la palabra reservada IN y una expresión (enlace completo) como en un identificador de variable (enlace abreviado). En un enlace completo, la expresión se evalúa (su valor ha de ser una secuencia) y las variables del patrón se asignan a cada elemento de la secuencia. El cuerpo y la criba se aplican a cada una de estas asignaciones. Por ejemplo:

```
{x+2: x in [5, 6, 7]};
➔ [7, 8, 9] : [int]

{x+z: (x, z) in [(5, 6), (7, 8), (9, 10)]};
➔ [11, 15, 19] : [int]
```

En el caso de múltiples enlaces, cada una de las secuencias (tanto el resultado de la expresión en un enlace completo como el valor de la variable en un enlace abreviado) deben ser de igual longitud. Los enlaces se entrelazan de forma que el cuerpo se evalúa para enlaces de elementos de igual índice dentro de cada secuencia. Por ejemplo:

```
{x+z: x in [5, 6, 7]; z in [8, 9, 10]};
➔ [13, 15, 17] : [int]

{dist(b, a): a in [1, 2, 3]; b in [8, 9, 10]};
[[8], [9, 9], [10, 10, 10]] : [int]
```

En este último ejemplo, la función *dist()* crea secuencias de longitud **a** (1, 2, 3) con **b** (8, 9, 10) como elementos.

Un constructo *aplicar-a-cada-uno* con un cuerpo y dos enlaces,

```
{cuerpo: patrón1 in expr1; patrón2 in expr2 | criba}
```

es equivalente al siguiente constructo con enlace único:

```
{cuerpo: (patrón1, patrón2) in zip(expr1, expr2) | criba}
```

donde la función *zip()* une elemento a elemento las dos secuencias argumento formando una secuencia de pares.

Un constructo *aplicar-a-cada-uno* sin cuerpo devuelve el valor del primer enlace. Por ejemplo:

```
{x in [3, 4, 5]; z in [5, 7, 8]};
➔ [3, 4, 5] : [int]

{x in [3, 4, 5]; z in [6, 7, 10] | z == 3+x};
➔ [3, 4] : [int]

{z in [6, 7, 10]; x in [3, 4, 5] | z == 3+x};
➔ [6, 7] : [int]
```

Si hay un cuerpo y una criba, ambos se evalúan para todos los enlaces y a continuación se aplica la subselección. Un constructo *aplicar-a-cada-uno* con una criba de la forma:

```
{cuerpo: enlaces | criba}
```

es equivalente al constructo

```
pack({(cuerpo, criba) : enlaces})
```

donde *pack()* toma una secuencia de tipo `[(alpha, bool)]` y devuelve una secuencia que contiene el primer elemento de cada par si el segundo elemento es true. *pack()* mantiene el orden de los elementos resultantes.

Por ejemplo:

```
{x+2: x in [5, 6, 7]| x>=6};
➔ [8, 9] : [int]

pack({(x+2, x>=6) : x in [5, 6, 7]});
➔ [8, 9] : [int]
```

5.1.5.2.4. Definición de nuevas funciones (`function`)

Se puede definir funciones en el nivel más externo, utilizando el constructo function. Su sintaxis es:

```
FUNCTION id patrón [: funtype] = expr;
```

Una función tiene un único argumento, pero el argumento puede ser cualquier patrón. El cuerpo de una función (la `expr` del final) se puede referir exclusivamente a variables enlazadas en el patrón o declaradas en el nivel más externo. Cualquier referencia a otra función en el cuerpo ha de serlo a funciones previamente definidas o a sí misma. No se puede utilizar recursión mutua. Como en cualquier lenguaje funcional, definir una función con el mismo nombre que otra anterior esconde aquella para usos futuros: todas las referencias a una función antes de la nueva definición se referirán a la original.

5.1.5.2.5. Asignaciones globales

Se puede enlazar con memoria una variable en el nivel más externo utilizando el operador =. Su sintaxis es:

```
id = expr;
```

Por ejemplo, *x = 256;* enlazará la variable *x* con el valor *256*. De ahora en adelante, la variable puede ser referenciada en el nivel más externo o dentro de una función. Por ejemplo,

```
function test(z) = z + x;
```

es la definición de una función que suma el valor *256* a su argumento. Los enlaces del nivel más externo se utilizan fundamentalmente para grabar resultados temporales y para definir constantes. La variable *pi* está ligada en el nivel más externo al valor de π.

5.1.6. Ejemplos

En esta sección describiremos en detalle algunos ejemplos de algoritmos diseñados en NESL. Antes de entrar en los ejemplos en sí, describiremos tres operaciones frecuentes. En primer lugar, el operador binario `->` (llamado read) se utiliza para leer varios elementos de una secuencia. Su argumento izquierdo es la secuencia de la que se van a leer los elementos y el argumento derecho es una secuencia de índices enteros que especifican de qué posiciones de la secuencia se han de leer los elementos. Por ejemplo, la expresión

```
“un ejemplo”->[3, 7, 0, 8, 9];
 ➔ “epulo” : [char]
```

lee los caracteres *e*, *p*, *u*, *l*, *o* de las posiciones *3*, *7*, *0*, *8* y *9* respectivamente de la secuencia de la izquierda. La función *read()* puede expresarse también como `read(a, i)` en lugar de `a->i`.

```
function subseq(a, inicio, fin) = a->[inicio:fin];
function take(a, n) = a->[0:n];
function drop(a, n) =a->[n:#a];
function rotate(a, n) = a->{mod(i-n, #a) : i in [n:n + #a]};
function even_elts(a) =a->[0:#a:2];
function odd_elts(a) =a->[1:#a:2];
function bottop(a) =[a->[0:#a/2], [#a/2:#a]];
```

Figura 5.2 Implementación de algunas funciones sobre secuencias

El operador binario `<-` (llamado write) se utiliza para escribir elementos en una secuencia. Su argumento izquierdo es la secuencia en la que se ha de escribir (la secuencia destino) y su argumento derecho es una secuencia de pares entero-valor. Para cada par (e, v) en la secuencia de pares, el valor v se escribe en la posición i de la secuencia de destino. Por ejemplo la expresión

```
function siguiente_char(candidatos, p, s, i) =
if (i == #p) then candidatos
else
  let letra = p[i];
    sig_l = s->{c + i : c in candidatos};
    candidatos = {c in candidatos; n in sig_l | n == letra}
  in siguiente_char(candidatos, p, s, i+1);


function buscar_cadena(p, s) = siguiente_char([0:#s-#p], p, s, 0);
```

Figura 5.3 Búsqueda de todas las instancias de la palabra *p* en la cadena *s*

```
"un ejemplo"<-[(6, space), (8, 'r), (9, 'e)];
➔ "un eje pre" : [char]
```

escribe los caracteres *space*, *r* y *e* en la cadena `"un ejemplo"` en las posiciones *6*, *8* y *9*. La función *write()* puede escribirse también como `write(d, ev)` en lugar `de d<-ev`.

Se puede crear rangos de enteros utilizando corchetes y dos puntos. La notación `[inicio:fin]` crea una secuencia de enteros que comienza en el número *inicio* y termina en el número *fin-1*. Por ejemplo:

```
[7:15];
➔[7, 8, 9, 10, 11, 12, 13, 14] : [int]
```

Se puede usar un tercer valor para indicar un paso (stride). [inicio:fin:paso] devuelve todos los enteros entre inicio y fin (excluído éste) de paso en paso números:

```
[7:15:2];
➔[7,  9, 11, 13] : [int]
```

Tal como muestra la Figura 5.2, mediante estas tres funciones es muy fácil definir algunas otras de las funciones predefinidas de NESL.

5.1.6.1. Búsqueda de una cadena en otra

El primer ejemplo que presentamos es una función que localiza todas las ocurrencias de una palabra dentro de una cadena (secuencia de caracteres). La función

buscar_cadena(w, s) (ver Figura 5.3) toma una palabra a buscar, *p* y una cadena de

```
function primos(n) =
if n == 2 then [2]
else
  let primos_raiz = primos(ceil(sqrt(float(n))));
    cribas = {[2*p:n:p] : p in primos_raiz};
    cribas_flat = flatten(sieves);
    flags = dist(t, n) <- {(i, f) : i in cribas_flat}
  in drop({i in [0:n]; flag in flags | flag}, 2);
```

Figura 5.4 Hallar todos los primos menores que `n`

caracteres, *s*, y devuelve la posición de comienzo de todas las subcadenas de *s* iguales a *p*. por ejemplo,

```
buscar_cadena("asi", "asasiararoasise");
➔[2, 10] : [int]
```

El algoritmo comienza por considerar todas las posiciones entre `0` y `#s-#p` como candidatos para un emparejamiento (ninguna otra posición de *s* debiera ser candidata a explorarse porque ello implicaría búsquedas más allá del final de la cadena *s*). Los candidatos se almacenan como índices que apuntan a las posiciones de *s* en las que comienza un emparejamiento. El algoritmo progresa a través de la cadena a explorar, utilizando llamadas recursivas a la función *siguiente_char()*, disminuyendo el conjunto de candidatos a emparejamientos válidos en cada paso.

Basándose en los candidatos en cada momento, *siguiente_char()* disminuye el conjunto de candidatos manteniendo como tales sólo aquellos que concuerden con el siguiente carácter de *p*. Para conseguir esto, cada candidato comprueba si el i-ésimo carácter de *p* coincide con la i-ésima posición a partir del índice del candidato. Todos los candidatos válidos se empaquetan y pasan a la siguiente llamada recursiva de *siguiente_char()*. La recursividad finaliza cuando el algoritmo alcanza el final de la *p*.

La evolución del conjunto de candidatos en nuestro ejemplo sería:

i	candidatos
0	[0, 2, 5, 7, 10]
1	[0, 2, 10]
2	[2, 10]

5.1.6.2. Cálculo de números primos

Nuestro segundo ejemplo en NESL es un programa que calcula todos los números primos menores que un cierto *n*. El algoritmo se basa en la criba de Eratóstenes. La idea de la criba es hallar todos los primos menores que *sqrt(n)* y utilizar los múltiplos de estos números para "cribar" todos los números compuestos menores que *n*. Puesto que todos los números compuestos menores que `n` han de tener un divisor menor que *sqrt(n),* los únicos elementos no cribados serán los primos.

El algoritmo que presentamos en la Figura 5.4 trabaja hallando recursivamente todos los primos hasta *sqrt(n)* (*primos_raiz*). Entonces, para cada *p* de *primos_raiz* el algoritmo genera mediante el constructo de la forma `[s:e:d]` la secuencia (*cribas*), que contiene todos los múltiplos de *p* menores que *n*. La secuencia *cribas* es una secuencia anidada siendo cada subsecuencia la criba para uno de los primos de *primos_raiz.* La función *flatten()* se utiliza para aplanar un nivel la secuencia anidada, devolviendo una secuencia que contiene todas las cribas. Por ejemplo,

```
flatten([[4, 6, 8, 10, 12, 14], [6, 9, 12, 15]]);
➔[4, 6, 8, 10, 12, 14, 6, 9, 12, 15] : [int]
```

Esta secuencia de cribas se utiliza en la función `<-` para colocar un flag a false en todas las posiciones en las que corresponden a un múltiplo de uno de los primos de *primos_raiz*. Esto devolverá una secuencia de booleanos, *flags*, que contiene *t* en todas las posiciones que no fueron eliminadas por una criba, es decir, las posiciones correspondientes a los primos. No obstante queremos que el programa devuelva los índices de los primos en lugar de estos flags. Para ello, el algoritmo crea una secuencia con todos los índices entre `0` y `n` (`[0:n]`) y utiliza subselección para eliminar los no primos. La función *drop()* se utiliza para eliminar los dos primeros elementos (0 y 1), que no se consideran primos.

5.1.6.3. El Quicksort

La Figura 5.5 presenta la elegante expresión que tiene en NESL el conocido algoritmo de ordenación Quicksort.

En este caso el algoritmo procede recursivamente, finalizando la recursión cuando la secuencia a ordenar contiene un único elemento. En caso que la secuencia tenga más de un elemento, toma como elemento pivote el elemento central de la secuencia a ordenar, y divide la secuencia de entrada en tres subsecuencias: una conteniendo los elementos menores que el pivote, otra conteniendo aquellos que son iguales al pivote y una tercera con los mayores que el pivote (líneas 5, 6 y 7 en la figura) y procede recursivamente generando una secuencia (*result*) que contiene las dos subsecuencias correspondiente a la ordenación de los elementos menores y de los mayores. El resultado de la ordenación es la concatenación de las secuencias de los menores, seguida

```
1 function quicksort(a) =
2 if (#a < 2) then a
3 else
4   let pivot   = a[#a/2];
5       menores = {e in a| e < pivot};
6       iguales = {e in a| e == pivot};
7       mayores = {e in a| e > pivot};
8       result  = {quicksort(v): v in menores,mayores]};
9  in result[0] ++ equal ++ result[1];
```

Figura 5.5 El Quicksort expresado en NESL

por los iguales y los mayores.

5.2. V

El lenguaje V [Cha95], [Cha94a], [Cha94b] está siendo desarrollado en un proyecto conjunto entre el Instituto de Arquitectura de Computadores y Tecnología del Software (FIRST) [Fir] del Centro Nacional Alemán para la Tecnología de la Información y el Grupo de Investigación en Ingeniería del Software de la Universidad Politécnica de Berlín [Swt]. El lenguaje sigue muy de cerca las ideas de NESL. De hecho podríamos considerarlo como la versión imperativa de NESL. Se trata de un lenguaje imperativo para paralelismo de datos.

El mayor inconveniente de NESL para convertirse en un lenguaje paralelo de amplia difusión es su naturaleza de lenguaje funcional. Los lenguajes funcionales no son fácilmente aceptados por programadores acostumbrados a lenguajes como C o Fortran que por otra parte son actualmente los predominantes en la programación de máquinas paralelas. La propiedad de ser un lenguaje funcional ayuda a evitar algunos

problemas en el lenguaje, pero dificulta mucho una implementación eficiente del mismo. Estos motivos fueron los que condujeron a sus autores al desarrollo de V.

V extiende C con un nuevo constructor de tipos para crear vectores (diferentes de los arrays clásicos de C) e introduce paralelismo a través de la utilización del constructo *aplicar-a-cada-uno* (igual al de NESL) y de algunas primitivas predefinidas. Al igual que en NESL, el constructo *aplicar-a-cada-uno* permite la aplicación simultánea de cualquier función predefinida o definida por el programador sobre todos o algunos de los elementos de un vector.

V trata de aproximar el modelo de computación paralelo subyacente a NESL al mundo de la programación imperativa de forma que los cómputos paralelos puedan ser expresados de forma procedural siempre que no tengan efectos laterales. De este modo, el programador de C puede seguir utilizando el entorno con el que está familiarizado, el software, las herramientas e interfaces existentes. Sus autores eligieron C como lenguaje base para soportar V por su amplio uso y disponibilidad en máquinas paralelas y pusieron especial énfasis en integrar las nuevas características ortogonalmente en el lenguaje de forma que se fomentara la reutilización de código existente y se facilitara el cambio a V a programadores acostumbrados a C

5.2.1. Elementos de V

Dado que las características fundamentales del lenguaje V son totalmente equivalentes a las de NESL, nos centraremos en este epígrafe en describir brevemente cómo han sido trasladadas a un lenguaje imperativo.

V amplía C con un único constructor de tipos: el vector. Los vectores son conjuntos ordenados y homogéneos de cualquier tipo de datos primitivos o definidos por el usuario, excepto punteros o uniones. Los vectores pueden a su vez ser elementos de otros vectores, pudiendo ser los vectores componentes de diferentes tamaños pero siempre del mismo tipo de elementos. Aunque no se permiten vectores de punteros, sí es lícito declarar punteros a vectores. Una variable *v* de tipo vector de enteros se declara como:

```
int v[*];
```

Es decir, una declaración de array en la que la especificación del tamaño se sustituye por un asterisco, para indicar que no es fijo. Los vectores son entidades dinámicas. La variable *v* puede ser asignada con un vector de cualquier longitud siempre que sus elementos sean de tipo entero. El número de elementos de un vector se obtiene mediante el operador `$`: `$v` indica el número de elementos de *v*. Un elemento del vector se obtiene igual que el de un array: `v[i]`. Un vector constante se construye por enumeración de sus elementos entre corchetes: `[1, 2, 3]`.

Otros ejemplos de declaraciones relacionadas con vectores serían:

```
int (*ptr_a_vector)[*];      /* Puntero a un vector */
int matriz[*][*];            /* Matriz de enteros */
int funcion () [*] {...}     /* Función que devuelve un vector
de enteros */
```

Para aplicar cualquier cómputo (por ejemplo una función *f()*) sobre todos los elementos de un vector se utiliza el constructo *aplicar-a-cada-uno* de V:

```
w = [f(x) : x in v];
```

El resultado de esta expresión es el vector de los resultados de aplicar *f* a cada uno de los elementos *x* de *v*. A la expresión `f(x)` se le llama el cuerpo y a `x in v` el

generador del constructo *aplicar-a-cada-uno*. Si *v* es un vector de vectores, *x* es un vector y por tanto *f* tomaría un vector como argumento. Los vectores pueden pasarse como argumentos a las funciones y pueden ser devueltos por ellas. Una variante del constructo *aplicar-a-cada-uno* permite un predicado llamado filtro que restringe los elementos del vector a considerar:

```
w = [f(x) : x in v : x < k];
```

```
pure int qsort (int v[*]) [*] {
  int pivot;
  int menores[*], iguales[*], mayores[*];
  int ordenado[*][*];

  if ($v <= 1) return v;
  pivot   = v[$v / 2];
  less    = [x : x in v : x < pivot];
  iguales = [x : x in v : x == pivot];
  mayores = [x : x in v : x > pivot];
  ordenado  = [qsort (subseq) : subseq in [menores, mayores]];
  return ordenado[0] >< iguales >< ordenado[1];
}
```

Figura 5.6 El Quicksort en V

en este caso se restringe la aplicación de *f* a aquellos elementos de *v* que verifiquen el predicado.

Consideremos a modo de ejemplo la implementación en V del Quicksort que presentamos en la Figura 5.6 observando en primer lugar que es casi idéntico a su expresión en NESL que aparecía en la Figura 5.5.

Utilizando el constructo *aplicar-a-cada-uno*, el vector inicial, *v* es dividido en paralelo en tres vectores que contienen los elementos *menores*, *iguales* y *mayores* que el *pivot*. A continuación la función es invocada recursivamente sobre los subvectores *menores* y *mayores* para ordenarlos. Las llamadas recursivas finalizan cuando el vector a ordenar tiene longitud uno. Dado que ambas llamadas recursivas aparecen dentro del constructo *aplicar-a-cada-uno*, las llamadas se ejecutarán también en paralelo. El resultado de las llamadas recursivas (*ordenado*) y los elementos *iguales* al pivote se concatenan mediante el operador >< para formar el resultado final.

Dos de las grandes ventajas del modelo de paralelismo de datos consiste en que los programas tienen un único flujo de control y son deterministas. Para conservar estas ventajas, los autores de V necesitan restringir los efectos laterales que se pueden provocar en el lenguaje, dado que los efectos laterales amenazan el determinismo con que se ejecutan los programas paralelos. Para entender la necesidad de estas restricciones, consideremos una función *f(x)* que asigna el valor *x* a una determinada variable global *y*. Si utilizamos esa función dentro de un constructo *aplicar-a-cada-uno* del siguiente modo:

```
[f(x) : x in v];
```

Tenemos que, dado que no existe un orden fijo de aplicación de la función *f()* a los diferentes elementos *x* de *v*, el valor final de la variable global *y* podría ser diferente en diferentes ejecuciones. Como consecuencia de este no determinismo, V restringe la clase de funciones que pueden aparecer dentro de un constructo *aplicar-a-cada-uno*. Las funciones permitidas se denominan funciones “puras”, y su prototipo se declara utilizando la palabra reservada `pure`. Veamos las restricciones de este tipo de funciones:

- Las funciones puras pueden invocar exclusivamente a otras funciones puras.
- No se permite el uso de uniones en las funciones puras.
- Las funciones puras no pueden utilizar la dirección o el valor de lado izquierdo (l-value) de variables globales (para evitar efectos laterales sobre aquellas).
- Una función pura no puede acceder a una variable global que sea de tipo puntero o bien contenga un puntero.
- Las funciones puras pueden utilizar arrays de tamaño fijado estáticamente. (Por ejemplo, no se permite un parámetro de tipo `int v[]` pero sí `int v[10]`).

La mayoría de estas restricciones tratan de evitar que las funciones puras produzcan efectos laterales, y alguna de ellas se impone debido al proceso de compilación que se requiere para este tipo de funciones.

Otra restricción que impone V para evitar efectos laterales consiste en prohibir expresiones de tipo puntero en el cuerpo y filtro de un constructo *aplicar-a-cada-uno*. Si no se impusiera este requisito, podrían provocarse efectos laterales en una expresión como:

```
[f(&x) : test in [1, 2, 5]]
```

dado que las tres instancias de *f* que se activan como consecuencia de esta llamada podrían cambiar todas ellas el valor de la variable *x*.

Como extensión de esta restricción, V prohibe también la utilización de vectores de punteros. Se permiten vectores de estructuras pero éstas no han de contener punteros ni uniones.

Todas las restricciones que impone el lenguaje tanto sobre las funciones puras como sobre el constructo *aplicar-a-cada-uno* pueden ser comprobadas estáticamente por el compilador de V.

5.3. El aplanamiento del paralelismo

La principal idea subyacente a la técnica de aplanamiento del paralelismo que utilizan tanto NESL como V consiste en que para todo programa con paralelismo anidado de datos que utilice un conjunto restringido de operadores, existe un programa con paralelismo de datos plano (es decir, con un único nivel de paralelismo) que utiliza vectores segmentados. Este programa con paralelismo de datos plano puede implementarse mediante un conjunto restringido de operaciones primitivas definidas en el denominado *scan vector model (SVM)*.

Un vector segmentado es la estructura de datos que se utiliza para representar un vector anidado de V o una secuencia de NESL. Por claridad, utilizaremos la terminología de V en lo que sigue de este epígrafe, aunque las mismas ideas pueden exponerse con respecto a NESL. Un vector segmentado consiste en:

- Un vector de valores que contiene los mismos valores escalares que el vector anidado.
- Un descriptor de segmento para cada nivel de anidamiento del vector anidado. Los descriptores de segmento indican el número de elementos de cada subvector.

por ejemplo, al vector anidado: [[3, 4], [], [5, 9, 2]] le corresponde el vector de valores [3, 4, 5, 9, 2] y el descriptor de segmentos [2, 0, 3]. Hay un único descriptor de segmento porque el nivel de anidamiento del vector es sólo uno. Parte de las operaciones primitivas suministradas por el modelo SVM son operaciones sobre segmentos. Por ejemplo, las reducciones paralelas y las operaciones de prefijos están

disponibles en versiones segmentadas y no segmentadas. Por ejemplo, si se efectúa una operación de suma de prefijos sobre el vector segmentado del ejemplo anterior daría como resultado el vector de valores [0, 3, 0, 5, 14] con el mismo descriptor de segmento: [2, 0, 3]. Esto significa que la secuencia anidada que representa es: [[0, 3], [], [0, 5, 14]], es decir, el resultado de aplicar suma de prefijos a cada una de los subvectores anidados.[1]

Un proceso necesario en la compilación de NESL y V es el de elevación del código (lifting). Para traducir un constructo *aplicar-a-cada-uno*, se eleva el código del cuerpo del constructo. El proceso de elevación consiste en que las operaciones escalares se convierten en operaciones sobre vectores y las operaciones sobre vectores se convierten en operaciones sobre vectores segmentados.

Si consideramos a modo de ejemplo el constructo:

```
[x + y : x in v1; y in v2]
```

en el que x + y es una operación escalar sobre enteros (por ejemplo). Durante el proceso de elevación del código, que se produce como consecuencia de que la expresión está en el cuerpo de un constructo *aplicar-a-cada-uno*, esta expresión da lugar a una suma de vectores elemento a elemento, *suma_vec(v1, v2)*, de forma que en el código transformado la operación elevada (el código de *suma_vec*) sustituye al constructo *aplicar-a-cada-uno*.

De forma análoga, el cuerpo de

```
[prefix_sum(xs) : xs in xss]
```

es elevado dando lugar a la llamada *seg_prefix_sum(xss)* donde *seg_prefix_sum(xss)* es una suma de prefijos en versión segmentada. Este tipo de traducciones se lleva a cabo con todas las operaciones primitivas escalares y vectoriales del lenguaje.

Sin embargo, en un código de NESL o V pueden aparecer funciones definidas por el usuario. Se necesita por tanto elevar el código de estas funciones. En realidad se mantienen dos versiones: la ordinaria y la elevada. La versión ordinaria de la función se utiliza para traducir llamadas 'normales', mientras que la elevada se utiliza cuando se traducen constructos *aplicar-a-cada-uno* en los que la función definida por el usuario aparece en el cuerpo del constructo o bien en la traducción de versiones elevadas de otras funciones que invocan a la de usuario.

Elevar una función de usuario significa simplemente vectorizar todos sus argumentos y su resultado. Por ejemplo una función `f: int -> int` de un argumento entero y resultado entero se transformaría al elevarse en una función `felev: [int] -> [int]` es decir, que tendría como argumento y resultado un vector de enteros.

5.4. Aperitif

Aperitif [Ape] es un lenguaje de programación que fue diseñado en 1995 por Thomas Erlebach de la Universidad Politécnica de Münich (Alemania) para la paralelización automática de algoritmos divide y vencerás. El lenguaje ha cambiado su nombre original (April) por el de Aperitif, aunque los desarrollos disponibles públicamente siguen usando el nombre original. El compilador, llamado April traduce programas escritos en Aperitif en programas en C apoyados con llamadas a PVM ejecutables en máquinas paralelas o redes de ordenadores ejecutando PVM. Según

[1] Para entender esta explicación hemos de tener en cuenta que en NESL y en V, el primer valor de una suma de prefijos es 0, el segundo el valor del primer elemento, el tercero es la suma del primer y segundo elementos, y así sucesivamente.

indica su autor, el objetivo en el diseño de Aperitif fue suministrar un modelo de programación fácil de utilizar que permitiera la explotación eficiente de máquinas paralelas.

Aperitif es muy similar a Pascal, aunque introduce algunas diferencias con respecto a éste. La más importante es la presencia de las palabras reservadas **parbegin** y **parend** que permiten indicar al programador que ciertas llamadas recursivas (típicas de los divide y vencerás) pueden ser ejecutadas en paralelo en diferentes procesadores. Otras diferencias con Pascal son que Aperitif carece de tipos registro, conjunto y punteros (y por tanto de variables dinámicas), no dispone de sentencias *case* ni *goto* y permite dos tipos reales con diferente precisión (real y double).

```
program divide_and_conquer;
{ Declaraciones globales }
{ El procedimiento paralelo }
procedure solve({ parámetros por valor, variable o resultado});
{ Declaraciones locales }
begin
  if { Condición de finalización } then
    { Calcular la solución secuencialmente }
  else
    { Dividir el problema }
    parbegin
      call solve({ Parámetros para el primer subproblema });
      call solve({ Parámetros para el segundo subproblema });
    parend
    { Combinar las soluciones de los subproblemas }
  fi
end;

{ Programa principal }
begin
{ Leer datos de entrada }
call solve({ Parámetros });
{ Salida de resultados }
end.
```

Figura 5.7 Un algoritmo divide y vencerás genérico en Aperitif

La documentación disponible públicamente acerca de Aperitif [Ape] consiste en el manual de usuario de la versión 1.0 así como los ficheros de instalación de la aplicación, que incluyen el fichero fuente del compilador April, el código fuente de las librerías de soporte y algunos programas de ejemplos escritos en Aperitif. Para instalar el sistema, lo único que se necesita es PVM (versión 3.2 o superior) y un compilador de C (preferentemente gcc). El autor indica que ha probado la herramienta en redes de estaciones de trabajo HP 9000/720 corriendo HP-UX y de Sun SparcStations corriendo SunOS 4.1. Nosotros hemos ejecutado los códigos Aperitif en un cluster de estaciones Sun y en la IBM-SP2.

La implementación de Aperitif está hecha sobre PVM: el programa que escribe el usuario se traduce con el compilador April a un programa en C apoyado por rutinas de paso de mensajes de PVM. Este fichero en C contiene dos funciones: *ROOTMAIN()* y *NODEMAIN()* que son ejecutadas respectivamente por los procesadores raíz y nodos de la red de procesadores que forman la arquitectura.

April ha sido escrito utilizando las herramientas flex y bison para diseñar los analizadores léxico y sintáctico. La generación de código está apoyada en una librería de funciones (denominada *pronto*) que forman parte del entorno Aperitif y que contienen llamadas a las funciones de PVM.

En la documentación que acompaña a la herramienta no se encuentran disponibles resultados computacionales para ningún programa concreto. Parece más bien que el esfuerzo de su autor fue encaminado hacia el diseño de una herramienta para la implementación paralela de algoritmos divide y vencerás, no importándole demasiado la eficiencia que se alcanzaba con la aproximación.

El esquema general que sigue la codificación de un algoritmo divide y vencerás en Aperitif es el que presentamos en la Figura 5.7. Observamos la similaridad de Apertifi con Pascal en su sintaxis salvo pequeños detalles (sentencia if o llamadas a procedimiento, por ejemplo).

Repasemos brevemente los elementos de Aperitif que soportan paralelismo. Básicamente son:

- La sentencia de composición paralela **parbegin-parend**.

En Aperitif, para indicar que varias sentencias pueden ejecutarse en paralelo (si ello es posible) se escribe:

```
parbegin <sentencias> parend
```

El lenguaje está limitado en cuanto a las sentencias que se pueden escribir dentro del cuerpo de la sentencia de composición paralela: han de ser exactamente dos sentencias de llamada a procedimiento.

La sentencia de composición paralela puede aparecer solamente dentro del cuerpo de un sólo procedimiento en un programa Aperitif. A este procedimiento se le llama el procedimiento paralelo. Por otra parte, las dos llamadas a procedimiento que han de aparecer en el cuerpo de la sentencia de composición paralela han de ser llamadas recursivas al propio procedimiento paralelo.

El procedimiento paralelo ha de ser invocado exclusivamente desde dentro del programa principal o bien recursivamente desde dentro de su propio cuerpo.

La sentencia de composición paralela está pensada para utilizarse exclusivamente dentro de un procedimiento divide y vencerás binario: después de dividir el problema en dos subproblemas de menor tamaño se lanzan dos llamadas recursivas paralelas para resolverlos.

Si los parámetros por variable o resultado se solapan en memoria, el comportamiento de la llamada paralela resulta indefinido.

- El paso de parámetros de tipo **result**.

Los cambios realizados por la rutina sobre los parámetros pasados *por resultado* son visibles en la rutina llamadora (igual que en el caso de parámetros pasados por variable). La diferencia consiste en que sus valores no están definidos al comienzo de la ejecución de la rutina llamada, por lo que leer el valor de un parámetro por resultado antes de haberle asignado un valor es un error.

Los parámetros por resultado se utilizan en la implementación de Aperitif para evitar comunicaciones innecesarias.

- La sentencia **replicate**.

Esta sentencia tiene la forma:

```
replicate (<lvalues>)
```

donde <lvalues> es una lista de valores izquierdos (expresiones que podrían aparecer en el lado izquierdo de una asignación) separados por comas. El efecto de esta sentencia es que los valores se replican desde el procesador raíz hacia el resto de procesadores. La sentencia se utiliza para realizar un ‘broadcast’ de

valores que son necesarios en todos los procesadores y que no son pasados a los procedimientos paralelos a través de sus parámetros.

- La función **child_available**.

Esta función permite decidir en tiempo de ejecución si es posible seguir dividiendo un problema o ha de ser resuelto secuencialmente, dependiendo de si hay más procesadores disponibles para ejecutar tareas en paralelo.

- La utilización de arrays con rango de índices dinámicos como parámetros de un procedimiento.

La utilización de rangos dinámicos permite pasar partes de un array a un procedimiento, disminuyendo así el número de comunicaciones necesarias.

Estudiaremos con más detalle algunas de estas características de Aperitif a través del código correspondiente al Quicksort (Figura 5.8) en Aperitif que ya hemos presentado en otros lenguajes.

```
 1 procedure quicksort(var v: array[dynamic first..last] of integer;
 2                     first, last: integer);
 3 var i, j: integer;
 4 begin
 5 if child_available
 6   if last > first then
 7     call partition(v[first..last], first, last, i, j);
 8     parbegin
 9       call quicksort(v[first..j], first, j);
10       call quicksort(v[i..last], i, last);
11     parend
12   fi
13 else
14   call seq_quicksort(v[first..last], first, last);
15 fi
16 end;
```

Figura 5.8 El Quicksort en Aperitif

Observamos en primer lugar en la línea 1 de la Figura 5.8 un parámetro array con índices dinámicos. En Aperitif los arrays pueden ser dinámicos en sus rangos (no en su tamaño, como cabría esperar de esta denominación). Un array dinámico sólo se puede declarar como parámetro formal de un procedimiento, usándose para ello la palabra reservada **dynamic**. Si no se utilizara un array dinámico, el código de la Figura 5.8 no funcionaría correctamente en paralelo, puesto que en las llamadas recursivas y paralelas al procedimiento Quicksort, se devolvería todo el array (puesto que es un parámetro por variable) y esto provocaría que sólo una parte del mismo quedara ordenada en el procesador raíz porque cada uno de los procesadores ordena sólo una parte del vector, y esa parte es la que han de devolver.

El programa utiliza la función *child_available()* en la línea 5 del código para comprobar que existen procesadores libres antes de lanzar una nueva llamada paralela. En la línea 7 se produce una llamada al procedimiento *partition()*, al que sólo se le pasa el segmento de vector que se quiere dividir (no el vector completo). Para esto, el parámetro formal de *partition()* también es un vector dinámico, como vemos en la cabecera de su declaración:

```
procedure partition(var v: array[dynamic first..last] of integer;
first, last: integer; result i, j: integer);
```

obsérvese también en esta declaración que los parámetros *i* y *j* de *partition()* que indican los límites de las particiones generadas están pasados como parámetros por resultado, lo

cual indica que al comienzo de la rutina, su valor no está inicializado; son parámetros 'sólo de salida'. El impacto de la utilización de este tipo de parámetros sobre la eficiencia del código generado es mucho mayor cuando se utilizan en llamadas a procedimientos que se ejecutan en paralelo.

# PROCS.	TAMAÑO	SEC	PAR	ACEL
2	1M	18.3205	19.1341	1.0185
2	2M	38.2926	38.0264	1.0712
2	3M	59.8958	66.5181	0.9606
2	4M	80.6461	76.2440	1.1084
4	1M	18.3974	20.7238	1.0283
4	2M	38.3293	44.6435	0.9279
4	3M	59.5897	80.4975	0.9250
4	4M	82.5526	107.4361	1.0215
8	1M	18.4477	21.3029	0.9600
8	2M	38.4845	49.2150	0.8786
8	3M	59.8099	79.0964	0.9043
8	4M	80.8939	83.9207	1.0341
16	1M	18.3824	23.1623	0.8852
16	2M	38.7317	53.6797	0.8518
16	3M	60.4770	84.3715	0.8180
16	4M	82.2724	92.9802	0.9476

Tabla 5.2 Resultados del Quicksort en una red de estaciones de trabajo utilizando Aperitif

En las llamadas paralelas a Quicksort de las líneas 9 y 10 así como en la llamada al código secuencial en la línea 14 apreciamos la utilización de los arrays dinámicos como parámetros actuales: sólo se pasa a las rutinas aquella parte del vector de la cual se han de ocupar.

La Tabla 5.2 presenta los resultados computacionales obtenidos en la implementación del Quicksort de la Figura 5.8 con 2, 4, 8 y 16 procesadores en una red de estaciones de trabajo. Se presentan los tiempos y las aceleraciones medias obtenidos con 10 ejecuciones del algoritmo sobre vectores de tamaños 1M-4M generados aleatoriamente. En la tabla, la columna etiquetada SEC corresponde al tiempo secuencial, PAR al tiempo paralelo y ACEL es la aceleración correspondiente al número de procesadores y tamaño que figura en las dos primeras columnas.

	2	4	8	16
1M	1.0185	1.0283	0.9600	0.8852
2M	1.0712	0.9279	0.8786	0.8518
3M	0.9606	0.9250	0.9043	0.8180
4M	1.1084	1.0215	1.0341	0.9476

Tabla 5.3 Aceleraciones para el Quicksort con Aperitif

En la tabla observamos que los tiempos secuenciales varían dependiendo del número de procesadores y tamaño considerado. Ello es debido a la diferente carga de trabajo de las máquinas en el momento en que se llevaron a cabo las diferentes ejecuciones. La Tabla 5.3 resume las aceleraciones alcanzadas en función del número de máquinas utilizado en la ejecución. La red de estaciones de trabajo utilizadas fueron Sun Ultra-1 a 143 MHz con diferentes tamaños de memoria (de 64 a 448 Mb) interconectadas por una red conmutada a 10Mb/sec. Para las ejecuciones se eligieron

días en los que el número de usuarios era muy bajo, con lo cual también lo era la carga de las máquinas. De hecho, en las ejecuciones con 2, 4 y 8 estaciones, las CPUs estaban siendo usadas casi en exclusiva por nuestros procesos.

Capítulo VI
RESULTADOS COMPUTACIONALES

Capítulo VI

Resultados Computacionales

6.1. Introducción

En este Capítulo presentamos los resultados computacionales obtenidos para los algoritmos de la Transformada rápida de Fourier, el Quicksort de tipo Común-Común, el algoritmo Quickhull, el Quicksort Distribuido (Privado-Privado), y el algoritmo de búsqueda de una clave en un vector que fueron presentados en el Capítulo 4. Se presentan también los resultados obtenidos con el Modelo de Computación Colectiva como Modelo de predicción del tiempo de cómputo que también se introdujo en el Capítulo 4. Utilizamos para ello dos algoritmos, la FFT y el algoritmo de ordenación PSRS. El Capítulo finaliza con los resultados correspondientes a la evaluación del coste de las funciones de división propuestas en esta memoria en comparación con las funciones de división de MPI. En todos los casos los algoritmos se han codificado utilizando *La Laguna C*, herramienta que fue descrita también en el Capítulo 4, salvo el caso del algoritmo PSRS que se codificó en MPI. Hemos utilizado la versión de *llc* implementada sobre MPI (ver Capítulo 1) sobre las máquinas: Cray T3E, Cray T3D, IBM SP2, Digital Alphaserver 8400, SGI Origin 2000 e Hitachi SR2201. En el Capítulo 1 se pueden encontrar descripciones de las características técnicas de cada una de estas plataformas así como referencias bibliográficas para una descripción más extensa de las mismas.

Como ya se comentó en el Capítulo 1 de esta memoria las máquinas IBM SP2 y SGI Origin 2000 del C4 sufrieron cambios de hardware en el periodo de realización de los experimentos de esta memoria. Dichos cambios se explicaron en el Capítulo 1. Es por este motivo que para estas dos máquinas y en algunos casos presentamos resultados para las configuraciones antigua y actual de las máquinas. A la configuración antigua de la SGI Origin 2000 la denominaremos karnak2, mientras que a su configuración actual la llamaremos karnak3. En cuanto a la IBM SP2, le añadiremos la etiqueta ‘switch

antiguo' a la antigua configuración de la máquina, puesto que el cambio del switch fue el cambio más importante introducido en la máquina.

Todos los resultados que presentamos corresponden a experimentos realizados utilizando los procesadores de las máquinas de forma exclusiva (los procesadores ejecutaban sólo nuestros procesos) mientras que la red de comunicaciones sí podía ser compartida con otros usuarios de la máquina.

Para todos los algoritmos presentamos tablas con los tiempos y las aceleraciones obtenidas y representaciones gráficas de los casos que muestran particular interés. En los resultados que se han obtenido realizando más de un experimento, las medidas presentadas son tiempos y aceleraciones promedio, y la aceleración promedio se ha calculado como el promedio de los cocientes entre el tiempo secuencial y el paralelo de cada una de las ejecuciones realizadas.

A la hora de interpretar los resultados de las gráficas, hemos de tener en cuenta que la escala de las mismas no es uniforme sino que hemos optado por elegir para cada gráfica la escala que mejor permitía comparar el comportamiento de los diferentes algoritmos en la misma figura más que la comparación de diferentes gráficas entre sí.

En las secciones que siguen se describen concretamente los experimentos realizados para cada uno de los algoritmos.

6.2. La Transformada rápida de Fourier

Las tablas que presentamos en este epígrafe muestran los tiempos correspondientes a los algoritmos secuencial y paralelo (columnas etiquetadas T SEC y T PAR) así como la aceleración (ACEL) obtenida para diferentes tamaños de vector de señal y diferente número de procesadores.

Las máquinas utilizadas para la FFT fueron los Cray T3E y T3D, Digital Alphaserver, Hitachi SR2201, IBM SP2 (versión actualizada) y las dos configuraciones (karnak2 y karnak3) de la SGI Origin 2000.

Para cada número de procesadores se utilizaba un vector de complejos (la señal de entrada) con una señal cuya transformada se conoce para comprobar que el resultado final es correcto. Los tamaños de señal de entrada que se utilizaron dependieron de la máquina considerada, y en la mayoría de ellas fueron 64K, 128K, 256K, 512K, 1M y 2M (salvo en los Cray T3E y T3D del EPCC).

Las Figuras 6.1 a la 6.8 presentan gráficamente los resultados para cada una de las máquinas para diferentes tamaños y variando el número de procesadores. Se ha tratado de representar en estas gráficas el comportamiento del algoritmo para un problema de tamaño pequeño, de tamaño intermedio y grande. Las Figuras 6.9, 6.10 y 6.11 presentan conjuntamente los resultados para las distintas plataformas para tamaños 2M, 64K y 256K.

En los resultados observamos que las aceleraciones se comportan de forma lineal y creciente para todos los tamaños de problema.

6.2.1. Cray T3E (Ciemat)

# de Procs.	Tamaño	T SEC	T PAR	ACEL
2	65536	0.5183	0.2666	1.9440
2	131072	1.0768	0.5588	1.9271
2	262144	2.2512	1.1704	1.9234
2	524288	4.7025	2.4460	1.9225
2	1048576	9.8231	5.1113	1.9219
2	2097152	20.4654	10.6507	1.9215
4	65536	0.5181	0.1550	3.3427
4	131072	1.0762	0.3219	3.3431
4	262144	2.2527	0.6669	3.3780
4	524288	4.7037	1.3891	3.3862
4	1048576	9.7938	2.8783	3.4026
4	2097152	20.4713	5.9845	3.4207
8	65536	0.5181	0.1012	5.1179
8	131072	1.0770	0.2086	5.1622
8	262144	2.2520	0.4292	5.2465
8	524288	4.7020	0.8849	5.3134
8	1048576	9.8240	1.8236	5.3872
8	2097152	20.4664	3.7564	5.4484
16	65536	0.5198	0.0777	6.6920
16	131072	1.0765	0.1591	6.7663
16	262144	2.2557	0.3235	6.9718
16	524288	4.7032	0.6570	7.1589
16	1048576	9.8002	1.3424	7.3005
16	2097152	20.4117	2.7394	7.4512

Tabla 6.1 La FFT en el Cray T3E del Ciemat

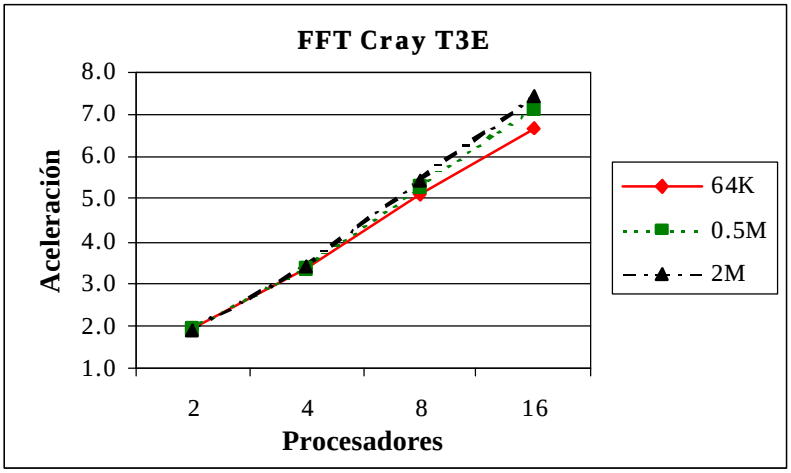

Figura 6.1 Resultados de la FFT en el Cray T3E del Ciemat

6.2.2. Cray T3E (EPCC)

# de Procs.	Tamaño	T SEC	T PAR	ACEL
2	65536	0.5191	0.2663	1.9495
2	131072	1.0766	0.5638	1.9095
2	262144	2.2519	1.1849	1.9005
2	524288	4.7301	2.4942	1.8964
4	65536	0.5187	0.1547	3.3535
4	131072	1.0730	0.3230	3.3222
4	262144	2.2598	0.6761	3.3425
4	524288	4.7135	1.4047	3.3555
8	65536	0.5197	0.1021	5.0898
8	131072	1.0759	0.2108	5.1050
8	262144	2.2516	0.4349	5.1770
8	524288	4.7302	0.9011	5.2493
16	65536	0.5214	0.0780	6.6872
16	131072	1.0739	0.1592	6.7438
16	262144	2.2611	0.3253	6.9517
16	524288	4.7117	0.6651	7.0837
32	65536	0.5245	0.0674	7.7859
32	131072	1.0798	0.1360	7.9419
32	262144	2.2571	0.2743	8.2288
32	524288	4.7302	0.5554	8.5168
64	65536	0.5390	0.0615	8.7702
64	131072	1.0854	0.1227	8.8434
64	262144	2.2690	0.2468	9.1941
64	524288	4.7179	0.4953	9.5248

Tabla 6.2 La FFT en el Cray T3E del EPCC

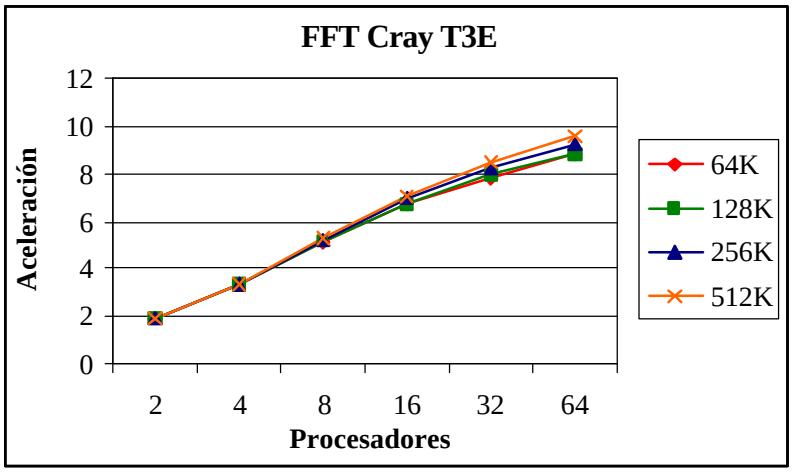

Figura 6.2 Resultados de la FFT en el Cray T3E del EPCC

6.2.3. Cray T3D

# de Procs.	Tamaño	T SEC	T PAR	ACEL
2	65536	2.4721	1.3736	1.7997
2	131072	5.3373	2.9536	1.8070
2	262144	11.5061	6.2294	1.8471
2	524288	24.8138	13.2503	1.8727
2	1048576	51.8310	28.1015	1.8444
4	65536	2.4719	0.8348	2.9611
4	131072	5.2802	1.7566	3.0059
4	262144	11.5867	3.6660	3.1606
4	524288	24.3855	7.7365	3.1520
4	1048576	52.2531	16.3927	3.1876
8	65536	2.4718	0.5823	4.2447
8	131072	5.3375	1.2092	4.4140
8	262144	11.5065	2.4981	4.6062
8	524288	24.8139	5.2423	4.7334
8	1048576	51.8308	10.7499	4.8215
16	65536	2.4719	0.4658	5.3068
16	131072	5.2804	0.9508	5.5536
16	262144	11.5869	1.9583	5.9168
16	524288	24.4267	4.0236	6.0709
16	1048576	51.8405	8.1853	6.3334
32	65536	2.4719	0.4119	6.0019
32	131072	5.3374	0.8372	6.3752
32	262144	11.5210	1.6932	6.8041
32	524288	24.8129	3.4454	7.2018
32	1048576	51.8176	6.9344	7.4725
64	65536	2.4710	0.3892	6.3497
64	131072	5.2788	0.7868	6.7090
64	262144	11.5829	1.5848	7.3089
64	524288	24.4164	3.1874	7.6603
64	1048576	51.8244	6.3811	8.1216

Tabla 6.3 La FFT en el Cray T3D

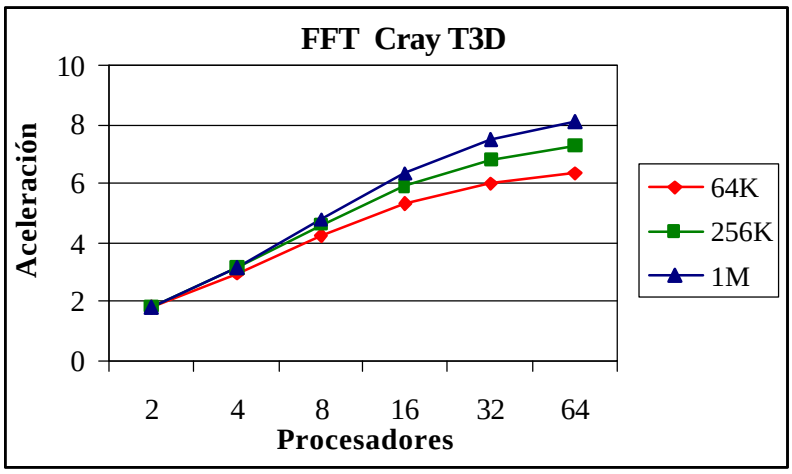

Figura 6.3 Resultados de la FFT en el Cray T3D

6.2.4. Digital Alphaserver 8400

# de Procs.	Tamaño	T SEC	T PAR	ACEL
2	65536	0.8916	0.4541	1.9634
2	131072	1.7998	0.9385	1.9178
2	262144	3.8232	1.9512	1.9595
2	524288	7.7607	4.0352	1.9233
2	1048576	16.0518	8.3408	1.9245
2	2097152	34.8652	18.0166	1.9352
4	65536	0.8896	0.2783	3.1965
4	131072	1.7979	0.5596	3.2129
4	262144	3.8350	1.1465	3.3450
4	524288	7.7959	2.3467	3.3221
4	1048576	16.2031	4.8408	3.3472
4	2097152	33.8135	9.9844	3.3866
8	65536	0.9121	0.2197	4.1511
8	131072	1.8438	0.4072	4.5276
8	262144	3.8047	0.8760	4.3434
8	524288	7.9893	1.7891	4.4656
8	1048576	16.6006	3.7754	4.3971
8	2097152	33.2438	7.2469	4.5883

Tabla 6.4 La FFT en la Digital Alphaserver 8400

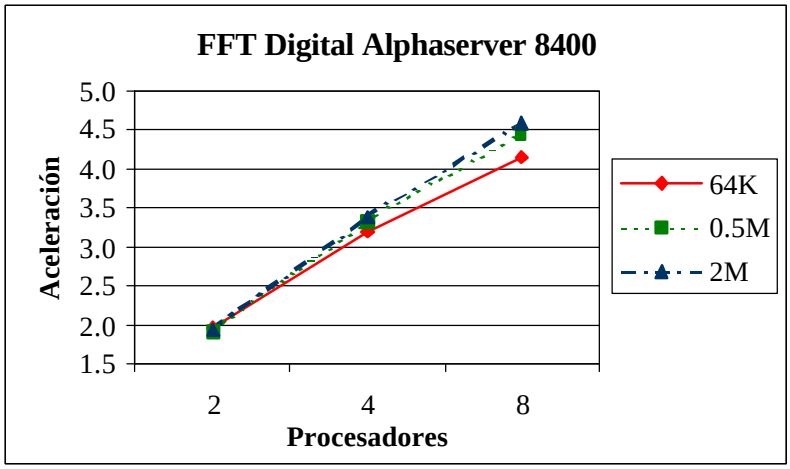

Figura 6.4 Resultados de la FFT en la Digital Alphaserver 8400

6.2.5. Hitachi SR2201

# de Procs.	Tamaño	T SEC	T PAR	ACEL
2	65536	12.2004	6.4789	1.8831
2	131072	26.0616	13.8508	1.8816
2	262144	55.6849	29.5267	1.8859
2	524288	118.7096	62.7743	1.8911
2	1048576	252.0190	132.7708	1.8982
2	2097152	538.4981	282.4782	1.9063
4	65536	12.2024	3.8773	3.1471
4	131072	26.0796	8.2199	3.1727
4	262144	55.6742	17.4066	3.1984
4	524288	118.8154	36.9047	3.2195
4	1048576	252.1960	77.3287	3.2613
4	2097152	537.2926	164.0616	3.2749
8	65536	12.2004	6.4789	1.8831
8	131072	26.0616	13.8508	1.8816
8	262144	55.6849	29.5267	1.8859
8	524288	118.7096	62.7743	1.8911
8	1048576	252.0190	132.7708	1.8982
8	2097152	538.4981	282.4782	1.9063

Tabla 6.5 La FFT en la Hitachi SR2201

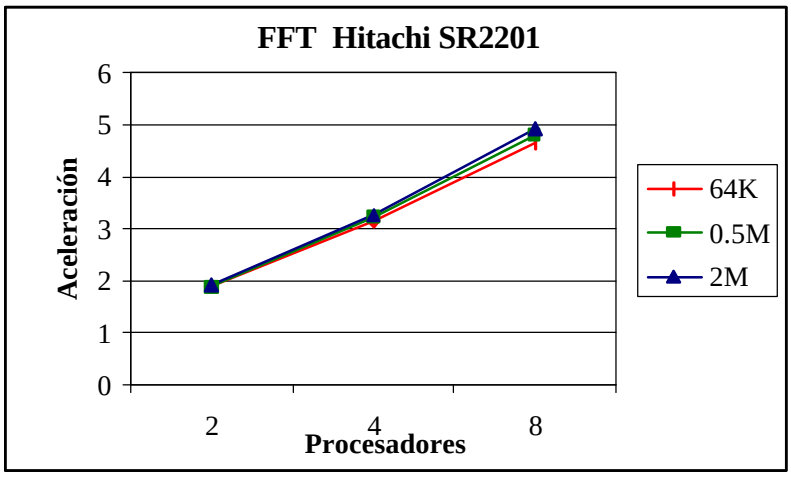

Figura 6.5 Resultados de la FFT en la Hitachi SR2201

6.2.6. IBM SP2

# de Procs.	Tamaño	T SEC	T PAR	ACEL
2	65536	0.8989	0.1605	5.6018
2	131072	1.9078	0.3034	6.2886
2	262144	4.1667	0.6291	6.6235
2	524288	8.7669	1.3030	6.7284
2	1048576	18.5416	2.7672	6.7004
2	2097152	39.0625	5.5749	7.0068
4	65536	0.8933	0.2002	4.4613
4	131072	1.9554	0.3973	4.9213
4	262144	4.0622	0.8554	4.7487
4	524288	8.7438	1.7085	5.1178
4	1048576	18.5014	3.5904	5.1530
4	2097152	39.0948	7.4712	5.2328
8	65536	0.9106	0.2925	3.1131
8	131072	1.9053	0.5975	3.1890
8	262144	4.0927	1.2428	3.2932
8	524288	8.6271	2.6254	3.2860
8	1048576	18.2659	5.5188	3.3098
8	2097152	39.2937	11.6258	3.3799
16	65536	0.9012	0.4831	1.8655
16	131072	1.9243	1.0057	1.9134
16	262144	4.1107	2.1340	1.9263
16	524288	8.6658	4.5164	1.9187
16	1048576	18.3787	9.5424	1.9260
16	2097152	38.7331	20.2468	1.9130

Tabla 6.6 La FFT en la IBM SP2

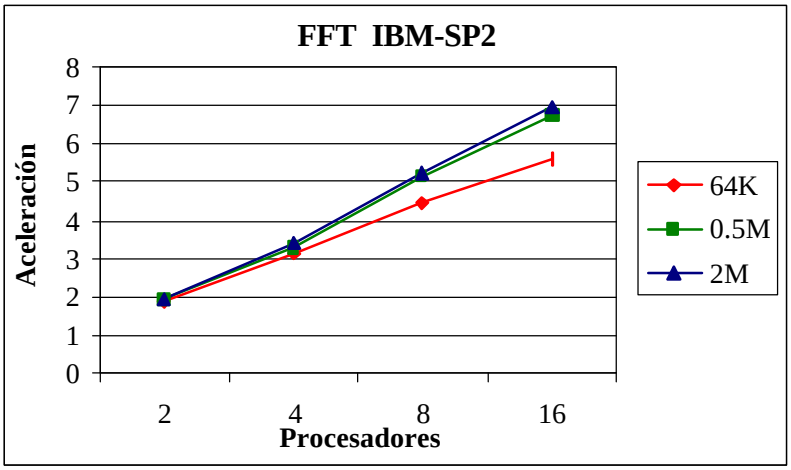

Figura 6.6 Resultados de la FFT en la IBM-SP2

6.2.7. Silicon Graphics Origin 2000 (karnak3)

# de Procs.	Tamaño	T SEC	T PAR	ACEL
2	65536	0.8621	0.4497	1.9169
2	131072	1.7827	0.9424	1.8917
2	262144	3.9149	2.0854	1.8773
2	524288	8.1490	4.2857	1.9015
2	1048576	20.8010	11.2461	1.8496
2	2097152	40.1922	20.9847	1.9153
4	65536	0.8705	0.2733	3.1852
4	131072	1.8025	0.5725	3.1487
4	262144	3.8824	1.2696	3.0580
4	524288	8.3242	2.6058	3.1945
4	1048576	17.6302	5.5045	3.2029
4	2097152	40.4553	14.6289	2.7654
8	65536	0.8924	0.1984	4.4971
8	131072	1.8265	0.4185	4.3643
8	262144	3.9721	0.9067	4.3810
8	524288	8.6430	1.9356	4.4654
8	1048576	21.1938	6.2662	3.3822
8	2097152	36.7129	8.7043	4.2178
16	65536	0.9334	0.1792	5.2075
16	131072	1.8722	0.3792	4.9371
16	262144	4.1923	0.9014	4.6508
16	524288	10.0621	2.4704	4.0731
16	1048576	18.5026	3.8125	4.8532
16	2097152	42.8448	13.0672	3.2788
32	65536	1.0283	0.2113	4.8669
32	131072	2.0792	0.4797	4.3345
32	262144	4.9162	0.8239	5.9671
32	524288	10.5351	1.7492	6.0230
32	1048576	21.1407	2.8144	7.5116
32	2097152	51.5039	6.7622	7.6164

Tabla 6.7 La FFT en la SGI Origin 2000 (karnak3)

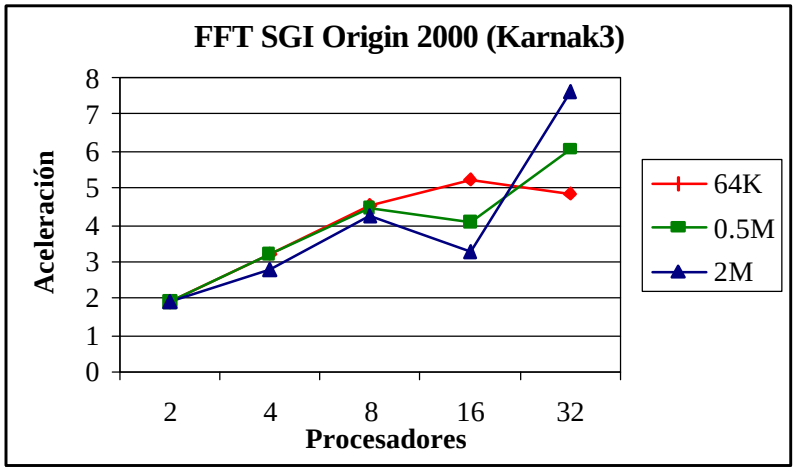

Figura 6.7 Resultados de la FFT en la SGI Origin 2000 (karnak3)

6.2.8. Silicon Graphics Origin 2000 (karnak2)

# de Procs.	Tamaño	T SEC	T PAR	ACEL
2	65536	1.3868	0.7306	1.8981
2	131072	2.9240	1.5441	1.8937
2	262144	6.2859	3.3116	1.8981
2	524288	14.2321	7.4289	1.9158
2	1048576	33.3403	17.3419	1.9225
2	2097152	61.5950	32.1090	1.9183
4	65536	1.4033	0.4402	3.1879
4	131072	2.9414	0.9253	3.1788
4	262144	6.3367	1.9897	3.1848
4	524288	15.0031	4.9045	3.0591
4	1048576	36.9891	12.4911	2.9612
4	2097152	75.4075	24.7816	3.0429
8	65536	1.4289	0.3138	4.5536
8	131072	2.9744	0.6655	4.4692
8	262144	6.4500	1.3854	4.6556
8	524288	14.5747	3.4339	4.2444
8	1048576	33.6287	7.9734	4.2176
8	2097152	62.7217	21.9453	2.8581
16	65536	1.4397	0.2720	5.2935
16	131072	3.0524	0.5753	5.3058
16	262144	6.6285	1.2278	5.3987
16	524288	13.6858	2.7054	5.0587
16	1048576	31.9971	5.2152	6.1353
16	2097152	69.6507	15.0071	4.6412

Tabla 6.8 La FFT en la SGI Origin 2000 (karnak2)

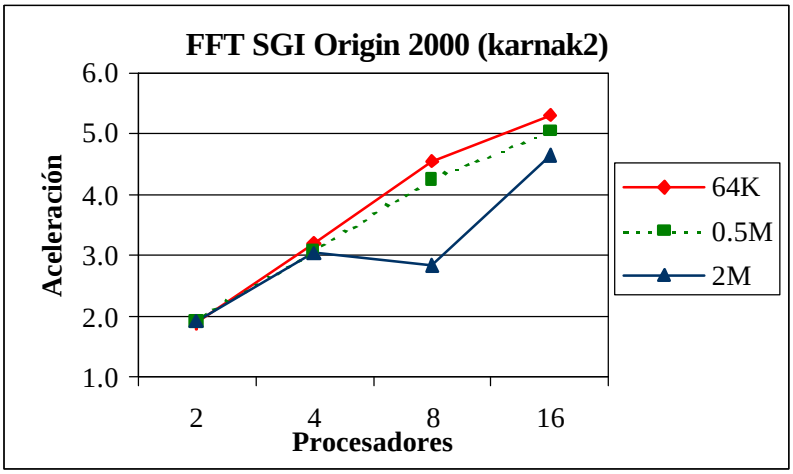

Figura 6.8 Resultados de la FFT en la SGI Origin 2000 (karnak2)

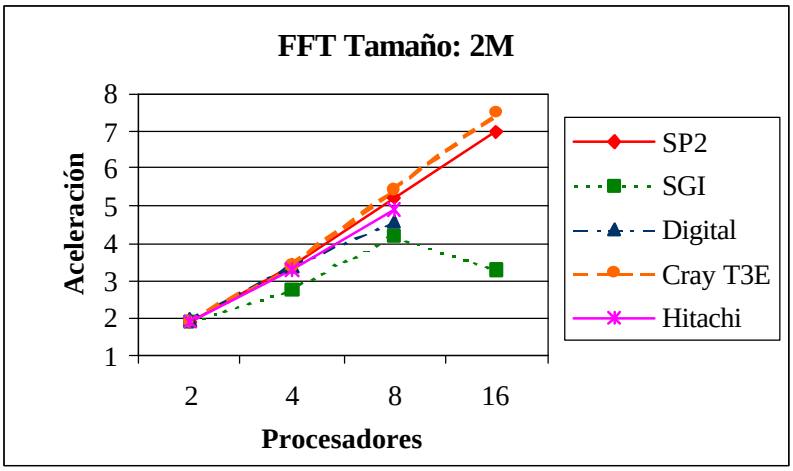

Figura 6.9 Resultados de la FFT para tamaño 2M

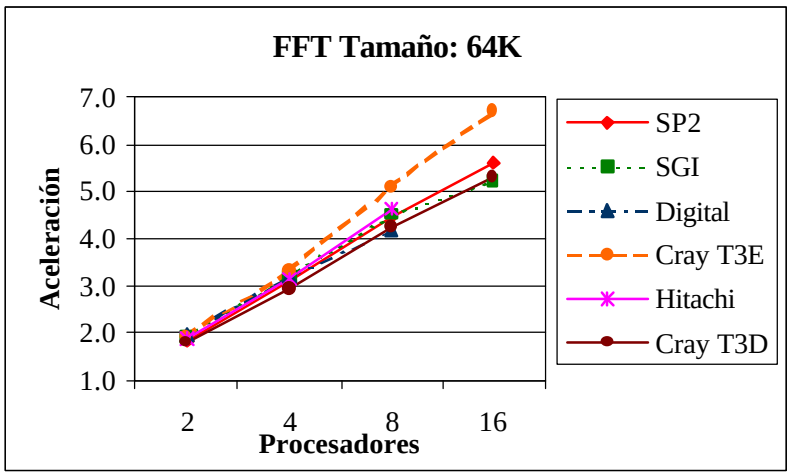

Figura 6.10 Resultados de la FFT para tamaño 64K

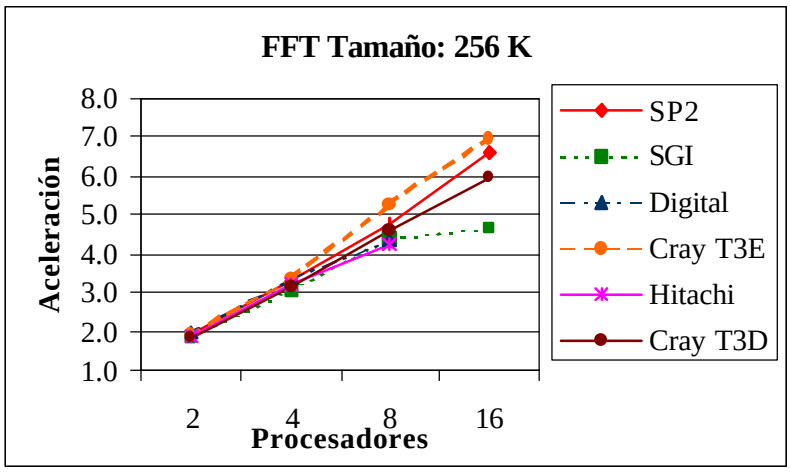

Figura 6.11 Resultados de la FFT para tamaño 256K

6.3. El Quicksort

En esta sección presentamos los resultados obtenidos para seis versiones diferentes del Quicksort implementadas en *llc*.

La etiqueta BH (por Brinch Hansen) corresponde con una implementación del Quicksort realizada utilizando MPI para una topología en hipercubo debida al profesor P. B. Hansen [Han94]. El código de esta implementación se muestra en el Apéndice A-1. Presentamos este algoritmo como un ejemplo del tipo de algoritmo que un programador experimentado produciría programando explícitamente mediante paso de mensajes, para comparar la eficiencia obtenida con la alcanzada programando con *llc*.

La etiqueta NO BAL (por no balanceado) corresponde a la versión del Quicksort que se presentó en la Figura 4.33 del Capítulo 4, es decir, la versión que utiliza la función de división PAR y el procedimiento *partition* y que por tanto no equilibra la carga de trabajo entre los procesadores. La carga de trabajo dependerá del pivote que se elija para realizar la partición del vector. Los resultados de la columna etiquetada VIRTUAL corresponden a la misma versión del algoritmo, pero utilizando la macro PARVIRTUAL en lugar de PAR, es decir, utilizando virtualización de procesadores. Recordemos que el constructo PARVIRTUAL secuencializa las llamadas paralelas cuando se agotan los procesadores disponibles en los conjuntos hoja mientras que el constructo PAR utiliza un algoritmo secuencial suministrado por el programador, y será por tanto más eficiente.

Los resultados que se presentan bajo la etiqueta MANUAL corresponden a un algoritmo similar al que se presentó en la Figura 4.37 del Capítulo 4, y que utiliza la función de división PAR y la función *find()* (ver Figura 4.36) para producir particiones del vector de igual tamaño, tratando de equilibrar la carga de trabajo entre los conjuntos de procesadores creados.

Por último, los resultados etiquetados BALAN y BALVIRT corresponden al algoritmo que utiliza el equilibrado automático de carga de *llc* a través de las funciones de división WEIGHTEDPAR y WEIGHTEDPARVIRTUAL respectivamente. El primero de ellos no virtualiza procesadores, mientras que BALVIRT corresponde con el que realiza virtualización de procesadores (Figura 4.38 del Capítulo 4).

Para cada algoritmo y cada máquina se ordenaron vectores de números enteros que fueron generados aleatoriamente mediante la función *random()* de C en el rango [0, $2^{(L-1)}$-1] siendo L la longitud de palabra (en bits) de la máquina considerada. Se eligieron tamaños de vector entre 1 y 7 M. Para cada una de las máquinas y cada uno de los algoritmos, fijado un número de procesadores y para cada tamaño de vector se realizaron 10 experimentos y presentamos la media de los resultados. Para cada máquina presentamos dos tablas. La primera de ella muestra los tiempos medios secuencial y paralelo de cada uno de los algoritmos (etiquetas T SEC y T PAR respectivamente) mientras que en la segunda aparece la aceleración promedio y la desviación estándar (cuasidesviación) de la aceleración (etiquetas ACEL y DESV).

La desviación estándar de la aceleración es una medida de cuan dispares son las aceleraciones obtenidas en cada uno de los experimentos con respecto a la media. Un algoritmo con poca desviación en su aceleración indica que es poco sensible a la muestra de datos que se ordene. Esto es lo que le ocurre por ejemplo al algoritmo de equilibrado de carga manual (etiqueta MANUAL) en el que se garantiza que las dos mitades en que se particiona el vector son exactamente del mismo tamaño. La desviación de la aceleración de este algoritmo será pequeña porque las divisiones del vector se hacen siempre del mismo modo independientemente de cuáles sean los valores a ordenar.

En todos los algoritmos en que había que elegir un elemento como pivote (incluido el algoritmo secuencial) el pivote se eligió como la media de tres elementos del subvector considerado: los dos extremos y el elemento central del subvector.

Para cada plataforma presentamos una primera gráfica que muestra el comportamiento de todos los algoritmos, fijado un número de procesadores (generalmente el número máximo de procesadores para la arquitectura considerada) variando el tamaño del vector ordenado. Las tres gráficas siguientes de cada máquina comparan tres situaciones diferentes de equilibrado de la carga de trabajo representadas por los algoritmos NO BAL, MANUAL, BAL y BALVIRT que corresponden respectivamente con: no equilibrar la carga (NO BAL), equilibrarla (MANUAL) a través de la función *find()* que particiona el vector en segmentos de igual tamaño, o equilibrarla automáticamente mediante las funciones de división WEIGHTEDPAR y WEIGHTEDPARVIRTUAL (etiquetas BAL y BALVIRT). La diferencia entre estas dos últimas alternativas es la utilización o no de virtualización de procesadores. Las subsecciones correspondientes a cada una de las plataformas terminan con una gráfica que muestra el comportamiento de la desviación de la aceleración fijado un tamaño de vector y variando el número de procesadores. Al final de esta sección, las Figuras 6.47-6.49 muestran el comportamiento del algoritmo BALVIRT en diferentes plataformas.

Para las diferentes versiones del Quicksort las plataformas que hemos utilizado son: Cray T3E, Digital Alphaserver y las dos configuraciones tanto de la IBM SP2 como de la SGI Origin 2000 (karnak2 y karnak3).

Es importante resaltar que el problema que estamos considerando es de tipo Común-Común, es decir, el vector a ordenar está disponible inicialmente en todos los procesadores de la máquina y el vector ordenado también queda almacenado en un vector común y por tanto todos los procesadores almacenan finalmente la solución del problema.

El ancho de banda de entrada/salida del Quicksort es lineal en el tamaño del vector a ordenar. Este ancho de banda de entrada salida constituye un límite inferior al tiempo de cualquier algoritmo de ordenación. Dado que la aceleración está limitada por el logaritmo del tamaño del vector ordenado, la única forma de conseguir mayores aceleraciones es utilizar vectores de tamaño mucho mayor que los empleados en estos experimentos. Esta es la justificación de los valores relativamente bajos de la aceleración alcanzados por el algoritmo.

Si se analizan los resultados obtenidos por el algoritmo debido al profesor Hansen (etiqueta BH) en comparación con las otras implementaciones, observamos que los resultados son comparables, e incluso mejores para las versiones basadas en el modelo colectivo. Solamente el algoritmo con virtualización (etiqueta VIRT) y sin equilibrado de carga presenta en líneas generales peores resultados.

En cuanto a la comparación entre los algoritmos que utilizan equilibrado de carga y los que no, observamos que utilizar equilibrado de carga mejora siempre los resultados, con un coste mínimo a nivel de implementación. Los algoritmos con equilibrado automático de la carga de trabajo (BALAN y BALVIRT) son los que consiguen los mejores resultados en casi todos los casos. Son incluso mejores que el algoritmo que consigue equilibrado de carga mediante la utilización de código explícito para ello (etiqueta MANUAL).

En cuanto al comportamiento de la desviación estándar de la aceleración, observamos que a pesar de que el algoritmo que no equilibra la carga de trabajo (NO BAL), consigue una aceleración comparable a los otros, su comportamiento es irregular puesto que depende de la bondad del pivot que se elige para particionar el vector. En el otro extremo, el algoritmo con equilibrado de carga explícito (MANUAL) presenta un

comportamiento muy regular. Los algoritmos con equilibrado automático (BAL y BALVIRT) se mantienen generalmente en un término medio respecto a lo predecible de su comportamiento.

6.3.1. Cray T3E

P	SIZE	BH		NO BAL		VIRTUAL		MANUAL		BALAN		BALVIRT	
		T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR
2	1	1.0375	0.6856	1.0384	0.7224	1.0391	1.1445	1.0388	0.6485	1.0395	0.7204	1.0386	0.8226
2	2	2.2008	1.4101	2.2025	1.4725	2.2033	2.3657	2.2037	1.3233	2.2035	1.4689	2.2021	1.6636
2	3	3.4222	2.1827	3.4243	2.3559	3.4260	3.7461	3.4260	2.0711	3.4273	2.3528	3.4247	2.6561
2	4	4.6410	2.9212	4.6436	3.0366	4.6465	5.0111	4.6465	2.8013	4.6464	3.0343	4.6441	3.4137
2	5	5.9111	3.7557	5.9141	4.1524	5.9168	6.4166	5.9173	3.5755	5.9163	4.1481	5.9146	4.6606
2	6	7.2288	4.5118	7.2332	4.7841	7.2367	7.8153	7.2367	4.3155	7.2396	4.7795	7.2337	5.3559
2	7	8.4741	5.3616	8.4786	5.6007	8.4827	9.1587	8.4825	5.0851	8.4828	5.6050	8.4789	6.2619
4	1	1.0375	0.5274	1.0386	0.5297	1.0392	0.7165	1.0389	0.4759	1.0397	0.4842	1.0387	0.5444
4	2	2.2003	1.0608	2.2018	1.1020	2.2034	1.4494	2.2032	0.9467	2.2034	1.0025	2.2022	1.1123
4	3	3.4219	1.6334	3.4241	1.8152	3.4262	2.2966	3.4260	1.4893	3.4275	1.6250	3.4248	1.8008
4	4	4.6404	2.1444	4.6436	2.2461	4.6459	3.0707	4.6461	2.0018	4.6465	2.0459	4.6440	2.2609
4	5	5.9099	2.7448	5.9146	2.9967	5.9173	3.9078	5.9170	2.5324	5.9162	2.6025	5.9147	2.8661
4	6	7.2274	3.3171	7.2334	3.5678	7.2365	4.7329	7.2368	3.0226	7.2399	3.1201	7.2339	3.4423
4	7	8.4730	3.9231	8.4778	4.1342	8.4821	5.5453	8.4822	3.5793	8.4831	3.6766	8.4792	4.0717
8	1	1.0376	0.4544	1.0383	0.4222	1.0388	0.5208	1.0389	0.4000	1.0394	0.3627	1.0384	0.3919
8	2	2.2003	0.8986	2.2016	0.8432	2.2033	1.0298	2.2030	0.7745	2.2030	0.7799	2.2022	0.8370
8	3	3.4221	1.3800	3.4237	1.4287	3.4252	1.6282	3.4253	1.2175	3.4266	1.1356	3.4241	1.2073
8	4	4.6405	1.7965	4.6431	1.6972	4.6455	2.1719	4.6457	1.6300	4.6457	1.4082	4.6438	1.5054
8	5	5.9096	2.2913	5.9131	2.2618	5.9159	2.7387	5.9160	2.0452	5.9152	1.9678	5.9138	2.0939
8	6	7.2281	2.7649	7.2325	2.7353	7.2356	3.3068	7.2356	2.4401	7.2381	2.2809	7.2318	2.4211
8	7	8.4732	3.2579	8.4763	3.1103	8.4808	3.8810	8.4812	2.8885	8.4815	2.6962	8.4774	2.8421
16	1	1.0377	0.4229	1.0383	0.3658	1.0390	0.4290	1.0389	0.3661	1.0395	0.3119	1.0384	0.3269
16	2	2.2004	0.8261	2.2019	0.7280	2.2032	0.8321	2.2032	0.7005	2.2032	0.6106	2.2020	0.6263
16	3	3.4219	1.2654	3.4239	1.1619	3.4259	1.3114	3.4256	1.0972	3.4264	0.9268	3.4239	0.9516
16	4	4.6409	1.6370	4.6436	1.3367	4.6457	1.7450	4.6461	1.4635	4.6459	1.2000	4.6435	1.2403
16	5	5.9099	2.0840	5.9141	1.9179	5.9171	2.1878	5.9170	1.8288	5.9154	1.5980	5.9134	1.6301
16	6	7.2281	2.5111	7.2328	2.1580	7.2365	2.6293	7.2365	2.1813	7.2380	1.8715	7.2325	1.9151
16	7	8.4738	2.9599	8.4778	2.5901	8.4817	3.0846	8.4824	2.5747	8.4815	2.1156	8.4780	2.1489

Tabla 6.9 Tiempos medios secuencial y paralelo para el Quicksort en el Cray T3E

P	SIZE	BH		NO BAL		VIRTUAL		MANUAL		BALAN		BALVIRT	
		ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV
2	1	1.5155	0.0570	1.4757	0.2362	0.9081	0.0170	1.6037	0.0544	1.4797	0.2327	1.2960	0.2070
2	2	1.5625	0.0552	1.5390	0.2469	0.9315	0.0147	1.6668	0.0517	1.5429	0.2459	1.3628	0.2212
2	3	1.5701	0.0539	1.4993	0.2483	0.9147	0.0136	1.6568	0.0631	1.5019	0.2472	1.3299	0.2208
2	4	1.5898	0.0401	1.5528	0.1870	0.9274	0.0144	1.6598	0.0415	1.5547	0.1868	1.3815	0.1671
2	5	1.5744	0.0334	1.4632	0.2392	0.9222	0.0196	1.6563	0.0511	1.4651	0.2393	1.3040	0.2144
2	6	1.6034	0.0518	1.5388	0.2141	0.9262	0.0170	1.6779	0.0461	1.5416	0.2139	1.3746	0.1911
2	7	1.5816	0.0426	1.5328	0.1671	0.9262	0.0135	1.6688	0.0323	1.5322	0.1663	1.3709	0.1495
4	1	1.9737	0.1088	2.0490	0.3967	1.4518	0.0463	2.1891	0.1114	2.1897	0.2994	1.9442	0.2596
4	2	2.0784	0.0967	2.1005	0.4462	1.5210	0.0343	2.3303	0.0822	2.2396	0.3030	2.0245	0.2954
4	3	2.1012	0.1100	1.9501	0.3634	1.4935	0.0450	2.3076	0.1243	2.1431	0.2867	1.9351	0.2706
4	4	2.1669	0.0773	2.1365	0.3887	1.5140	0.0381	2.3251	0.0956	2.2965	0.2382	2.0783	0.2179
4	5	2.1560	0.0772	2.1199	0.5291	1.5153	0.0430	2.3410	0.0991	2.3471	0.3921	2.1341	0.3665
4	6	2.1828	0.0996	2.0920	0.4003	1.5299	0.0415	2.3981	0.1013	2.3675	0.3241	2.1458	0.3029
4	7	2.1647	0.1064	2.1347	0.4174	1.5300	0.0310	2.3720	0.0738	2.3366	0.2639	2.1089	0.2369
8	1	2.2938	0.1482	2.6346	0.6532	1.9995	0.0930	2.6087	0.1648	2.9177	0.3879	2.6979	0.3535
8	2	2.4575	0.1525	2.7039	0.4884	2.1420	0.0683	2.8501	0.1235	2.8770	0.3602	2.6789	0.3300
8	3	2.4926	0.1776	2.4840	0.4916	2.1090	0.1013	2.8283	0.2015	3.0532	0.3043	2.8701	0.2847
8	4	2.5879	0.1068	2.8434	0.5672	2.1417	0.0756	2.8569	0.1375	3.3214	0.2626	3.1046	0.2415
8	5	2.5853	0.1210	2.8045	0.6578	2.1639	0.0901	2.9026	0.1660	3.1115	0.4672	2.9113	0.4095
8	6	2.6206	0.1379	2.6958	0.3904	2.1913	0.0889	2.9744	0.1723	3.1915	0.2458	3.0038	0.2327
8	7	2.6095	0.1545	2.8126	0.4860	2.1870	0.0662	2.9412	0.1215	3.1976	0.3882	3.0304	0.3653
16	1	2.4673	0.1774	3.0058	0.7246	2.4315	0.1439	2.8545	0.2088	3.4092	0.4989	3.2692	0.5296
16	2	2.6749	0.1778	3.1332	0.5589	2.6526	0.1094	3.1530	0.1496	3.6726	0.4642	3.5788	0.4524
16	3	2.7218	0.2190	3.0624	0.6232	2.6234	0.1660	3.1418	0.2472	3.7395	0.3699	3.6324	0.3239
16	4	2.8412	0.1273	3.5514	0.5463	2.6679	0.1215	3.1846	0.1769	3.8991	0.3246	3.7727	0.3227
16	5	2.8441	0.1474	3.2715	0.6954	2.7126	0.1449	3.2504	0.2163	3.8524	0.6335	3.7693	0.6055
16	6	2.8869	0.1664	3.4153	0.4937	2.7585	0.1384	3.3294	0.2085	3.9010	0.3793	3.8046	0.3418
16	7	2.8753	0.1940	3.4068	0.6740	2.7539	0.1113	3.3028	0.1684	4.0707	0.4846	3.9896	0.4073

Tabla 6.10 Aceleración media y Desviación estándar de la aceleración para el Quicksort en el Cray T3E

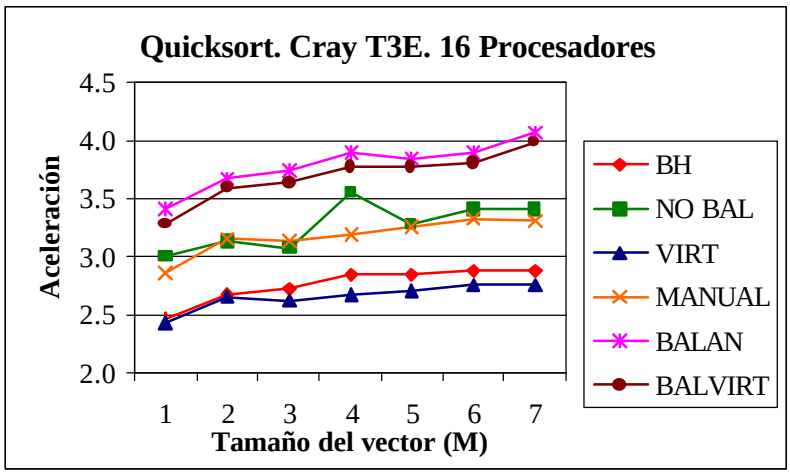

Figura 6.12 Quicksort. Cray T3E. Resultados computacionales para 16 procesadores

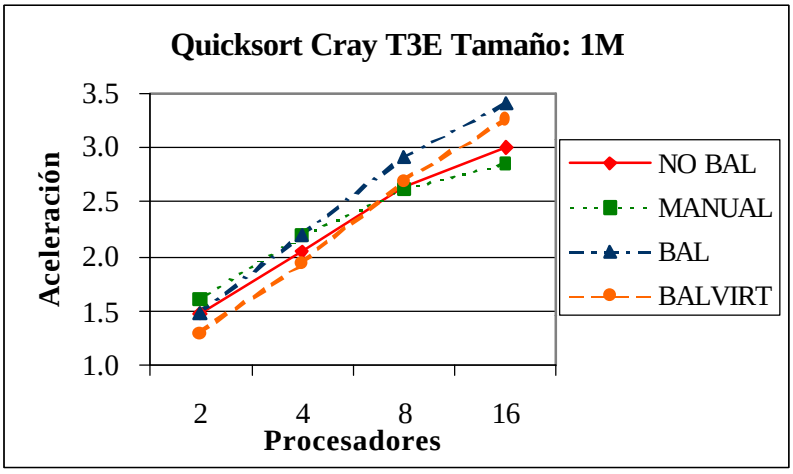

Figura 6.13 Quicksort. Equilibrado de carga. Tamaño del vector: 1M

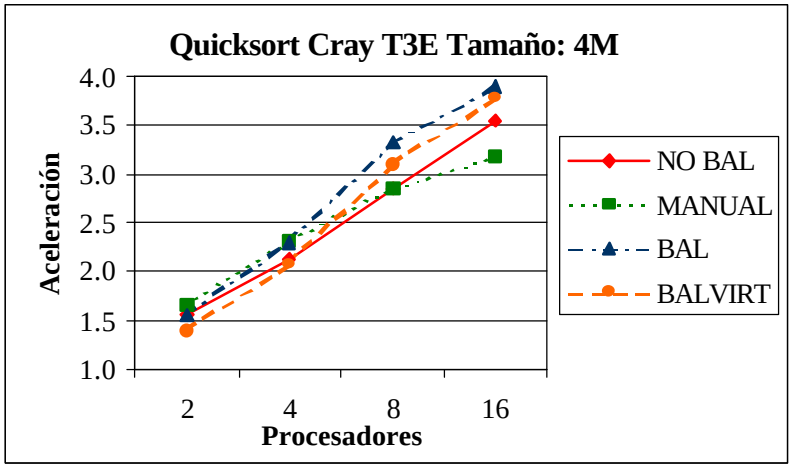

Figura 6.14 Quicksort. Equilibrado de carga. Tamaño del vector: 4M

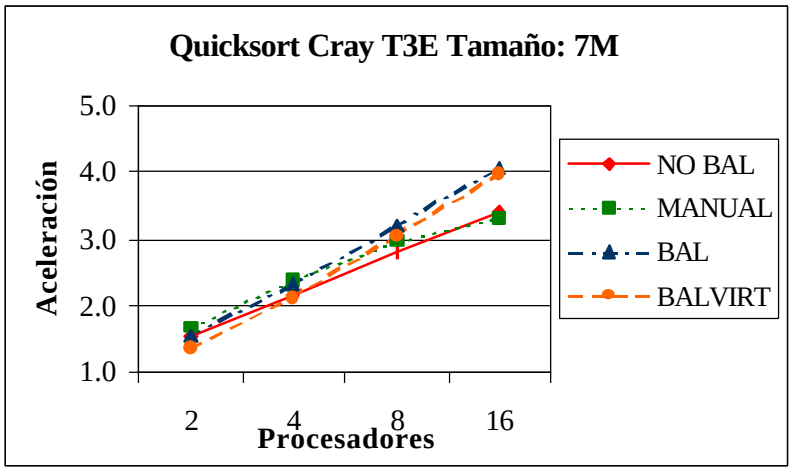

Figura 6.15 Quicksort. Equilibrado de carga. Tamaño del vector: 7M

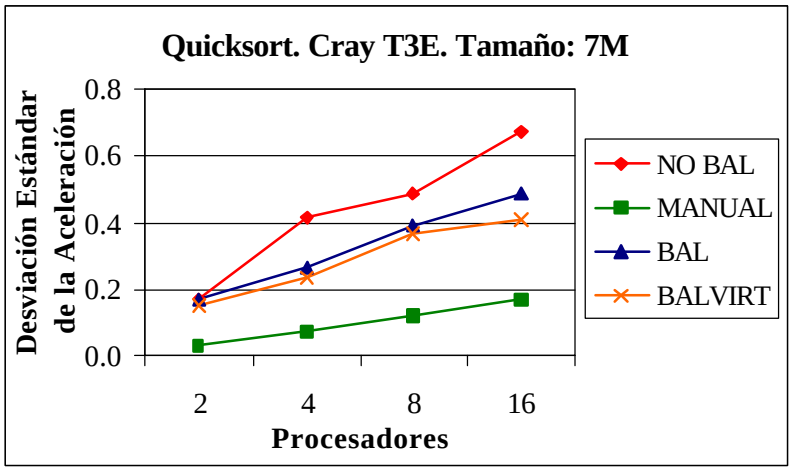

Figura 6.16 Quicksort. Cray T3E. Desviación Estándar de la Aceleración para un vector de 7M enteros

6.3.2. Cray T3D

P	SIZE	BH		NO BAL		VIRTUAL		MANUAL		BALAN		BALVIRT	
		T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR
2	1	3.2024	2.1619	3.2027	2.2983	3.2040	3.9563	3.2026	2.0375	3.2029	2.2912	3.2222	2.8338
2	2	6.7178	4.4761	6.7184	4.6266	6.7211	8.2219	6.7182	4.1658	6.7189	4.6186	6.7470	5.6596
2	3	10.3599	6.8376	10.3608	7.4316	10.3652	12.8358	10.3606	6.4211	10.3616	7.3403	10.3967	9.0532
2	4	14.0171	9.1305	14.0184	9.5492	14.0242	17.0881	14.0181	8.5997	14.0195	9.5960	14.0613	11.5962
2	5	17.8312	10.6760	17.8325	12.3153	17.8260	21.3795	17.8326	10.4654	17.8260	12.3019	17.8698	15.0398
2	6	21.5758	12.8113	21.5776	14.0289	21.5696	26.0118	21.5776	12.5569	21.5698	14.0155	21.6231	17.0973
2	7	25.4004	15.1758	25.4033	16.5054	25.3937	30.4474	25.4032	14.8839	25.3938	16.4916	25.4469	20.0829
4	1	3.2025	1.6917	3.2026	1.7160	3.2027	2.4528	3.2027	1.4751	3.2029	1.5756	3.2222	1.8903
4	2	6.7178	3.4578	6.7183	3.5144	6.7182	5.0193	6.7182	2.9747	6.7189	3.1872	6.7470	3.7831
4	3	10.3596	5.2531	10.3608	5.8601	10.3606	7.8243	10.3606	4.5845	10.3615	5.1951	10.3967	6.3422
4	4	14.0169	6.9069	14.0185	7.2090	14.0184	10.4802	14.0184	6.1931	14.0194	6.6620	14.0615	7.7685
4	5	17.8314	7.2684	17.8327	8.7684	17.8259	12.5552	17.8339	6.9939	17.8258	7.3515	17.8699	8.8135
4	6	21.5761	8.7394	21.5773	10.3349	21.5694	15.2416	21.5792	8.4118	21.5697	8.8476	21.6232	10.5971
4	7	25.4008	10.3544	25.4034	11.9686	25.3937	17.8692	25.4053	9.9852	25.3937	10.4476	25.4471	12.5176
8	1	3.2024	1.4742	3.2026	1.3832	3.2026	1.7432	3.2027	1.2218	3.2028	1.1848	3.2222	1.3537
8	2	6.7177	2.9757	6.7183	2.8234	6.7182	3.5171	6.7182	2.4201	6.7189	2.5158	6.7471	2.9552
8	3	10.3600	4.5027	10.3608	4.7284	10.3605	5.5232	10.3607	3.7929	10.3615	3.6822	10.3966	4.2572
8	4	14.0173	5.8899	14.0187	5.7063	14.0185	7.3184	14.0185	5.0380	14.0195	5.0201	14.0615	5.3582
8	5	17.8246	5.6855	17.8261	6.4330	17.8340	8.3642	17.8260	5.3953	17.8323	5.2014	17.8764	6.1118
8	6	21.5679	6.8293	21.5707	7.7235	21.5791	10.1226	21.5694	6.4711	21.5777	5.9897	21.6314	6.9348
8	7	25.3911	8.0975	25.4064	8.7564	25.4049	11.9044	25.3936	7.6953	25.4032	7.2008	25.4568	8.2197
16	1	3.2025	1.3742	3.2026	1.2087	3.2027	1.4122	3.2027	1.1042	3.2027	1.0219	3.2220	1.1209
16	2	6.7179	2.7581	6.7183	2.5634	6.7182	2.8127	6.7182	2.1678	6.7185	1.9940	6.7466	2.2523
16	3	10.3598	4.1613	10.3610	3.9311	10.3605	4.4037	10.3606	3.3975	10.3608	3.0085	10.3962	3.4606
16	4	14.0169	5.4154	14.0187	4.5010	14.0185	5.8293	14.0185	4.5008	14.0185	4.2834	14.0607	4.3674
16	5	17.8247	5.1661	17.8288	5.3640	17.8260	6.4895	17.8260	4.7756	17.8345	3.8833	17.8756	4.5181
16	6	21.5676	5.9100	21.5800	5.7749	21.5694	7.6691	21.5695	5.5356	21.5804	4.5032	21.6334	4.9632
16	7	25.3907	7.0275	25.4069	7.0810	25.3938	9.0625	25.3938	6.6076	25.4037	5.3953	25.4602	5.7534

Tabla 6.11 Tiempos medios secuencial y paralelo para el Quicksort en el Cray T3D

P	SIZE	BH		NO BAL		VIRTUAL		MANUAL		BALAN		BALVIRT	
		ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV
2	1	1.4823	0.0406	1.4264	0.2187	0.8099	0.0065	1.5726	0.0340	1.4311	0.2202	1.1643	0.1802
2	2	1.5015	0.0327	1.4933	0.2385	0.8176	0.0088	1.6136	0.0393	1.4962	0.2394	1.2273	0.2003
2	3	1.5162	0.0405	1.4440	0.2527	0.8076	0.0097	1.6149	0.0489	1.4532	0.2377	1.1884	0.2072
2	4	1.5356	0.0256	1.4967	0.1996	0.8208	0.0077	1.6308	0.0368	1.4898	0.2017	1.2355	0.1635
2	5	1.6710	0.0359	1.4878	0.2504	0.8338	0.0077	1.7049	0.0380	1.4888	0.2506	1.2210	0.2064
2	6	1.6848	0.0375	1.5658	0.2204	0.8293	0.0088	1.7192	0.0411	1.5668	0.2207	1.2878	0.1825
2	7	1.6743	0.0304	1.5570	0.1718	0.8341	0.0088	1.7074	0.0339	1.5578	0.1721	1.2820	0.1424
4	1	1.8953	0.0666	1.9372	0.3532	1.3061	0.0207	2.1738	0.0742	2.0712	0.2821	1.7325	0.2203
4	2	1.9448	0.0649	2.0022	0.4140	1.3390	0.0251	2.2610	0.0755	2.1465	0.2888	1.8180	0.2512
4	3	1.9742	0.0668	1.8406	0.3689	1.3247	0.0284	2.2640	0.1005	2.0275	0.2783	1.6653	0.2339
4	4	2.0306	0.0506	2.0158	0.3764	1.3386	0.0364	2.2695	0.1178	2.1337	0.2494	1.8355	0.2123
4	5	2.4565	0.0861	2.1811	0.5519	1.4202	0.0219	2.5537	0.0946	2.5020	0.4201	2.0928	0.3545
4	6	2.4715	0.0863	2.1489	0.4039	1.4155	0.0255	2.5686	0.0982	2.4870	0.3480	2.0845	0.3027
4	7	2.4559	0.0874	2.1993	0.4156	1.4216	0.0300	2.5475	0.0950	2.4578	0.2607	2.0559	0.2200
8	1	2.1756	0.0857	2.4560	0.5779	1.8384	0.0464	2.6264	0.1144	2.7505	0.3682	2.4164	0.3026
8	2	2.2609	0.0910	2.4949	0.5049	1.9116	0.0536	2.7814	0.1229	2.7144	0.3296	2.3258	0.3112
8	3	2.3049	0.1021	2.2909	0.4874	1.8808	0.1018	2.7469	0.2150	2.8453	0.2860	2.4722	0.2684
8	4	2.3819	0.0725	2.5502	0.4882	1.9187	0.0789	2.7944	0.1838	2.8719	0.4816	2.6685	0.3325
8	5	3.1422	0.1455	2.9634	0.6943	2.1337	0.0553	3.3125	0.1627	3.4885	0.4129	2.9794	0.3792
8	6	3.1643	0.1497	2.8470	0.4308	2.1334	0.0637	3.3419	0.1841	3.6251	0.2988	3.1405	0.2696
8	7	3.1414	0.1403	2.9740	0.4610	2.1362	0.0702	3.3067	0.1568	3.5837	0.4558	3.1375	0.3663
16	1	2.3345	0.1010	2.7759	0.6173	2.2702	0.0743	2.9077	0.1442	3.2095	0.4963	2.9441	0.4571
16	2	2.4401	0.1064	2.7543	0.5977	2.3915	0.0849	3.1062	0.1502	3.4283	0.4480	3.0727	0.4895
16	3	2.4949	0.1208	2.7044	0.4474	2.3623	0.1570	3.0708	0.2676	3.4805	0.3468	3.0752	0.4323
16	4	2.5911	0.0882	3.1655	0.4269	2.4112	0.1254	3.1315	0.2318	3.3860	0.5956	3.2638	0.3675
16	5	3.4678	0.2473	3.4719	0.6558	2.7542	0.1426	3.7545	0.2867	4.6944	0.6395	4.0403	0.5754
16	6	3.6589	0.2003	3.8017	0.5366	2.8161	0.1084	3.9101	0.2490	4.8261	0.4280	4.3904	0.3889
16	7	3.6225	0.1927	3.7260	0.7542	2.8071	0.1247	3.8544	0.2168	4.7653	0.5332	4.4445	0.3068

Tabla 6.12 Aceleración media y Desviación estándar de la aceleración para el Quicksort en el Cray T3D

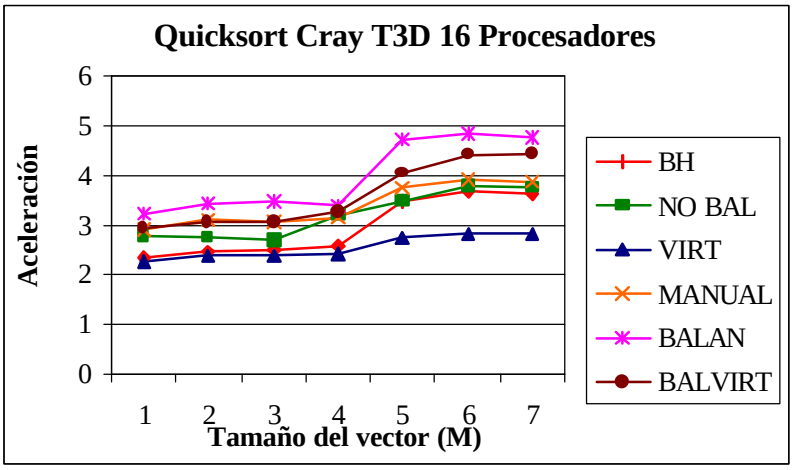

Figura 6.17 Quicksort. Cray T3D. Resultados computacionales para 16 procesadores

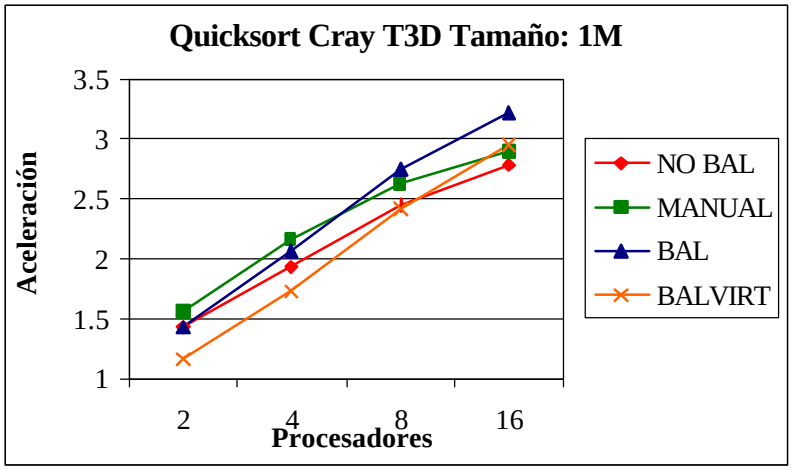

Figura 6.18 Quicksort. Equilibrado de carga. Tamaño del vector: 1M

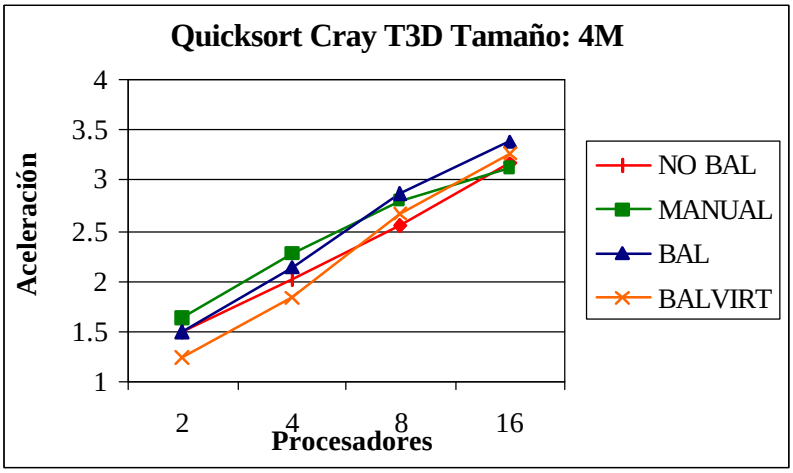

Figura 6.19 Quicksort. Equilibrado de carga. Tamaño del vector: 4M

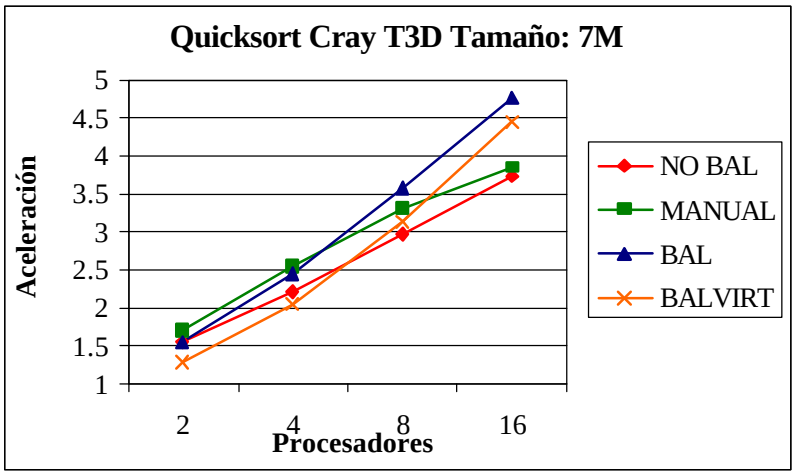

Figura 6.20 Quicksort. Equilibrado de carga. Tamaño del vector: 7M

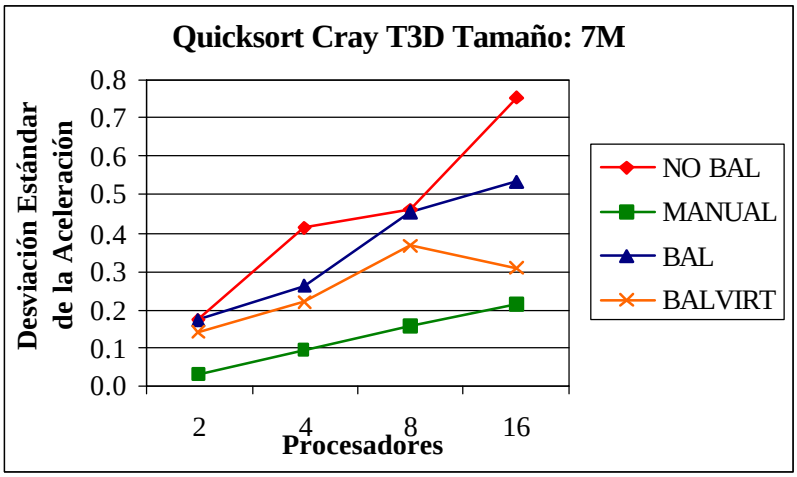

Figura 6.21 Quicksort. Cray T3D. Desviación Estándar de la Aceleración para un vector de 7M enteros

6.3.3. Digital Alphaserver 8400

P	SIZE	BH		NO BAL		VIRTUAL		MANUAL		BALAN		BALVIRT	
		T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR
2	1	0.7223	0.5068	0.7227	0.4930	0.7226	0.9926	0.7298	0.4616	0.7229	0.4933	0.7227	0.5328
2	2	1.5563	1.1011	1.5617	1.0385	1.5652	2.1174	1.5748	1.0178	1.5653	1.0401	1.5615	1.1141
2	3	2.4342	1.7274	2.4464	1.7122	2.4558	3.3573	2.4661	1.6046	2.4605	1.7162	2.4458	1.8301
2	4	3.3066	2.3590	3.3252	2.2926	3.3390	4.4570	3.3516	2.1947	3.3388	2.2981	3.3249	2.4475
2	5	4.2062	2.9823	4.2336	2.8987	4.2550	5.7485	4.2658	2.7773	4.2520	2.9055	4.2323	3.0896
2	6	5.1122	3.5627	5.1470	3.4399	5.1729	6.9562	5.1865	3.3264	5.1729	3.4499	5.1480	3.6684
2	7	6.0738	4.2867	6.1115	4.2352	6.1442	8.2018	6.1554	4.0170	6.1431	4.2466	6.1109	4.5006
4	1	0.7224	0.4111	0.7229	0.3590	0.7227	0.6063	0.7226	0.3473	0.7228	0.3328	0.7228	0.3554
4	2	1.5558	0.8904	1.5726	0.8235	1.5706	1.3313	1.5730	0.7957	1.5702	0.7529	1.5814	0.7950
4	3	2.4325	1.3993	2.4852	1.3972	2.4736	2.1153	2.5007	1.2609	2.4749	1.3166	2.4775	1.3902
4	4	3.3077	1.9092	3.4075	1.7274	3.3768	2.8533	3.4101	1.7651	3.3778	1.6078	3.3801	1.6970
4	5	4.2039	2.4164	4.3512	2.1788	4.2954	3.6353	4.3449	2.2172	4.2988	2.0420	4.2983	2.1461
4	6	5.1119	2.8925	5.2916	2.7060	5.2300	4.3617	5.2930	2.6268	5.2269	2.5834	5.2291	2.7314
4	7	6.0624	3.4737	6.2924	3.3984	6.2149	5.1823	6.2958	3.1681	6.2048	3.1521	6.2082	3.2920
8	1	0.7221	0.3817	0.7229	0.3304	0.7228	0.4989	0.7231	0.3359	0.7226	0.3263	0.7265	0.3148
8	2	1.5553	0.8156	1.6376	1.0416	1.6329	1.3557	1.6104	0.8327	1.6347	0.7528	1.6292	0.9005
8	3	2.4337	1.2588	2.5869	1.3622	2.6054	1.7663	2.6063	1.2972	2.6297	1.3181	2.6354	1.3623
8	4	3.3037	1.7122	3.6053	1.6453	3.5713	2.3794	3.5764	1.8727	3.5723	1.6748	3.5831	1.8022
8	5	4.2018	2.1633	4.5945	2.1649	4.5940	3.0944	4.5957	2.3175	4.5934	1.9478	4.5104	2.0511
8	6	5.1055	2.6036	5.6033	2.6369	5.5354	3.7287	5.5345	2.7980	5.6276	2.4767	5.6151	2.5338
8	7	6.0577	3.1790	6.6978	3.2838	6.6987	4.3373	6.6464	3.3257	6.7256	3.0244	6.7068	3.1073

Tabla 6.13 Tiempos medios secuencial y paralelo para el Quicksort en la Digital Alphaserver 8400

P	SIZE	BH		NO BAL		VIRTUAL		MANUAL		BALAN		BALVIRT	
		ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV
2	1	1.4256	0.0307	1.4879	0.1900	0.7280	0.0088	1.5817	0.0389	1.4869	0.1872	1.3767	0.1763
2	2	1.4137	0.0239	1.5299	0.1962	0.7393	0.0067	1.5476	0.0223	1.5308	0.1960	1.4264	0.1849
2	3	1.4098	0.0293	1.4504	0.1777	0.7315	0.0087	1.5378	0.0343	1.4548	0.1749	1.3565	0.1658
2	4	1.4020	0.0173	1.4677	0.1620	0.7492	0.0043	1.5275	0.0248	1.4700	0.1619	1.3750	0.1532
2	5	1.4108	0.0229	1.4819	0.1786	0.7402	0.0044	1.5366	0.0309	1.4848	0.1786	1.3904	0.1691
2	6	1.4354	0.0280	1.5090	0.1434	0.7437	0.0065	1.5598	0.0311	1.5121	0.1428	1.4154	0.1351
2	7	1.4172	0.0218	1.4791	0.2215	0.7492	0.0057	1.5328	0.0270	1.4823	0.2207	1.3917	0.2087
4	1	1.7579	0.0444	2.0479	0.2713	1.1925	0.0259	2.0837	0.0833	2.2018	0.2619	2.0614	0.2419
4	2	1.7489	0.0565	1.9513	0.2822	1.1805	0.0305	1.9798	0.0805	2.0932	0.1377	1.9975	0.1383
4	3	1.7398	0.0460	1.8188	0.2610	1.1697	0.0174	1.9846	0.0522	1.9290	0.2888	1.8392	0.3136
4	4	1.7329	0.0264	1.9833	0.1520	1.1839	0.0226	1.9343	0.0681	2.1185	0.2033	2.0061	0.1776
4	5	1.7405	0.0344	2.0200	0.2142	1.1820	0.0225	1.9628	0.0790	2.1104	0.1067	2.0092	0.1170
4	6	1.7680	0.0388	1.9640	0.1363	1.1993	0.0172	2.0164	0.0557	2.0370	0.1765	1.9279	0.1688
4	7	1.7461	0.0377	1.8921	0.2826	1.1995	0.0166	1.9886	0.0480	2.0119	0.3140	1.9259	0.2925
8	1	1.8939	0.0714	2.2093	0.2258	1.4964	0.2241	2.1808	0.2546	2.3102	0.4127	2.3181	0.1735
8	2	1.9100	0.0810	1.8069	0.5607	1.3659	0.3946	1.9579	0.2156	2.2105	0.3005	2.0589	0.5419
8	3	1.9368	0.0732	1.9296	0.2160	1.4806	0.0932	2.0106	0.0612	2.0893	0.3456	2.0530	0.3854
8	4	1.9300	0.0300	2.2032	0.1615	1.5025	0.0535	1.9263	0.1911	2.1513	0.2034	2.0545	0.3061
8	5	1.9434	0.0445	2.1329	0.1619	1.4903	0.0939	1.9868	0.0849	2.3629	0.0983	2.2062	0.1320
8	6	1.9617	0.0420	2.1485	0.2323	1.4884	0.0865	1.9811	0.0966	2.2814	0.1573	2.2224	0.1278
8	7	1.9081	0.0710	2.0741	0.2681	1.5465	0.0555	2.0001	0.0691	2.2366	0.1657	2.1714	0.1735

Tabla 6.14 Aceleración media y Desviación estándar de la aceleración para el Quicksort en la Digital Alphaserver 8400

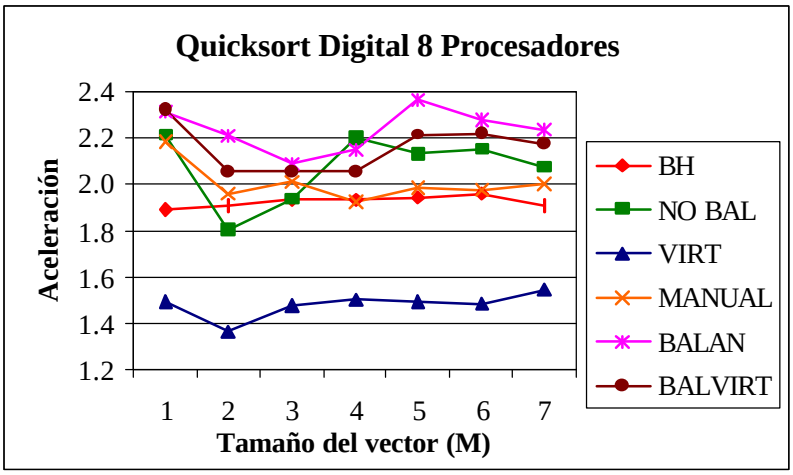

Figura 6.22 Quicksort. Digital Alphaserver 8400. Resultados computacionales para 8 procesadores

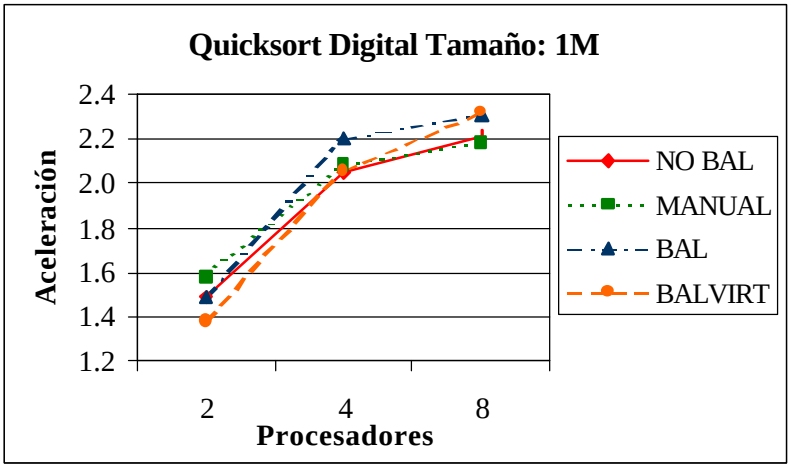

Figura 6.23 Quicksort. Digital Alphaserver 8400
Equilibrado de carga. Tamaño del vector: 1M

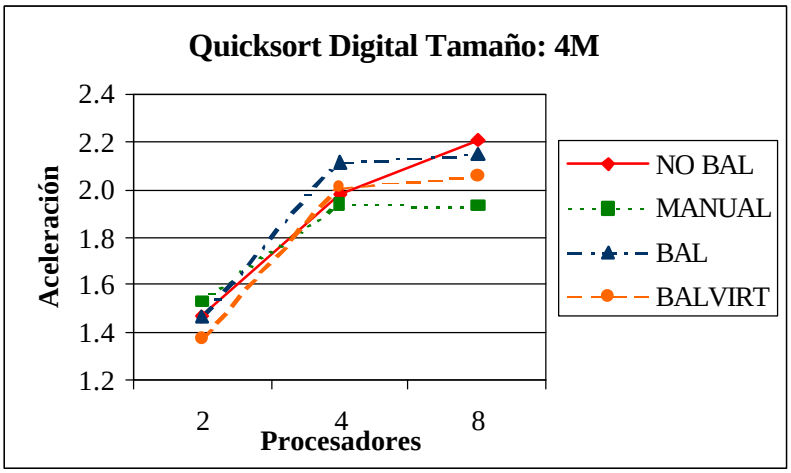

Figura 6.24 Quicksort. Digital Alphaserver 8400
Equilibrado de carga. Tamaño del vector: 4M

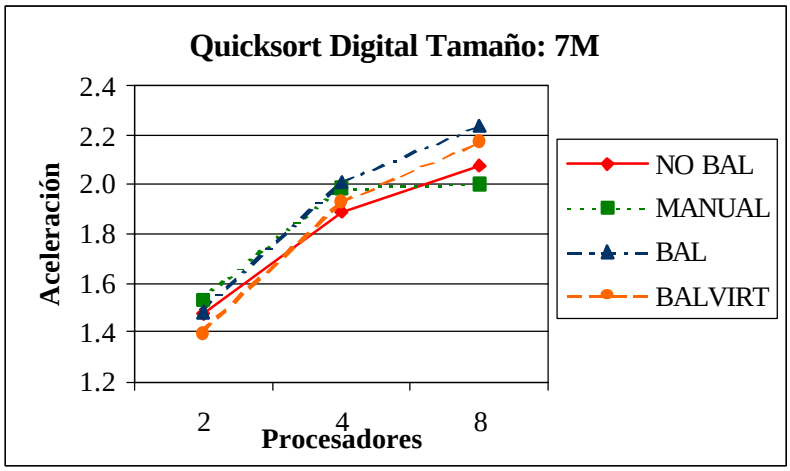

Figura 6.25 Quicksort. Digital Alphaserver 8400
Equilibrado de carga. Tamaño del vector: 7M

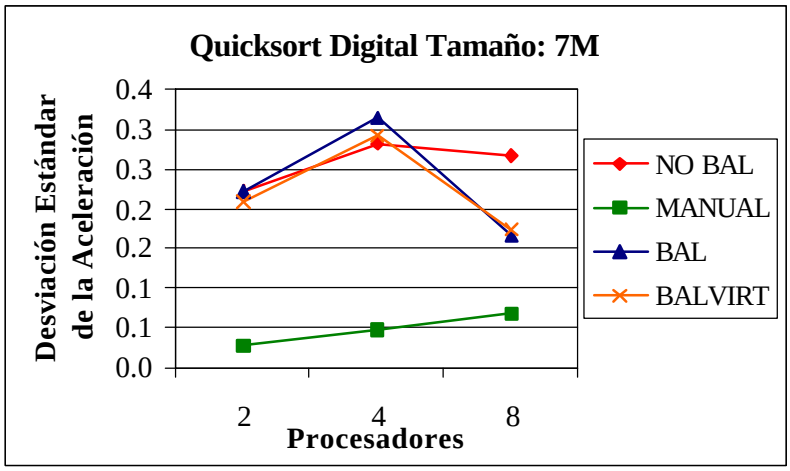

Figura 6.26 Quicksort. Digital Alphaserver 8400.
Desviación Estándar de la Aceleración para un vector de 7M enteros

6.3.4. IBM SP2

P	SIZE	BH		NO BAL		VIRTUAL		MANUAL		BALAN		BALVIRT	
		T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR
2	1	1.2563	0.7834	1.2569	0.8171	1.2702	1.8138	1.2574	0.7580	1.2561	0.8543	1.2578	0.9531
2	2	2.5886	1.5847	2.5893	1.6257	2.5889	3.7040	2.5914	1.5449	2.5974	1.8423	2.6108	2.0476
2	3	3.9599	2.4260	3.9584	2.6193	3.9734	5.7896	3.9600	2.3642	3.9537	2.4819	3.9575	2.8666
2	4	5.3367	3.2986	5.3331	3.4866	5.3312	7.6017	5.3396	3.2092	5.3362	3.5475	5.3625	3.9834
2	5	6.7300	4.1296	6.7292	4.3711	6.7334	9.7610	6.7319	4.0261	6.7354	4.4203	6.7402	4.8882
2	6	8.1711	5.0032	8.1321	5.1299	8.1379	11.8062	8.1406	4.8100	8.1439	5.7455	8.1423	6.3590
2	7	9.5594	5.8839	9.6477	6.2912	9.5681	13.7596	9.5741	5.7502	9.5649	6.4969	9.5549	7.1810
4	1	1.2716	0.5518	1.2663	0.5497	1.2664	1.0382	1.2751	0.5204	1.2723	0.5230	1.2718	0.5743
4	2	2.6085	1.1227	2.6080	1.1312	2.6149	2.1055	2.6171	1.0575	2.6211	1.1524	2.6109	1.2685
4	3	4.6789	1.9617	3.9883	1.8995	3.9964	3.2746	4.4723	1.6819	4.0016	1.7025	3.9914	1.8771
4	4	5.3863	2.2970	5.3856	2.2581	5.3956	4.3650	5.7384	2.3404	5.3923	2.2319	5.8636	2.4535
4	5	6.7885	2.8865	6.7838	2.8513	6.7959	5.5426	6.7866	2.7354	6.7896	2.6877	6.7945	2.9438
4	6	8.2744	3.4310	8.2029	3.4896	8.2722	6.9099	8.2762	3.2486	8.2130	3.4955	8.2353	3.9624
4	7	9.6502	4.0896	9.6661	4.4757	9.7470	7.8423	9.6535	3.8797	9.6462	3.7955	9.6523	4.1605
8	1	1.2646	0.4398	1.2747	0.3997	1.2811	0.6532	1.2683	0.4051	1.2675	0.3357	1.6930	0.4053
8	2	2.6143	0.8830	2.8432	0.7915	2.6129	1.3393	2.6033	0.8336	2.8470	0.7602	2.7477	0.8327
8	3	4.0116	1.3307	4.0234	1.3508	3.9959	2.0496	3.9996	1.2329	3.9923	1.0974	4.0278	1.1886
8	4	5.4022	1.8178	5.3701	1.5945	5.3827	2.7392	5.4685	1.7290	5.3991	1.4582	5.3743	1.5804
8	5	6.7880	2.2768	6.8134	2.0374	6.8143	3.4557	6.8231	2.0921	6.9978	1.7865	6.8866	1.9454
8	6	8.2189	2.6842	9.1881	2.6266	8.2136	4.1286	8.2816	2.5919	8.2122	2.1575	8.2352	2.3444
8	7	9.6457	3.2421	9.6611	3.2173	9.6710	4.8846	9.6580	2.9647	9.6781	2.5962	9.6629	2.8085
16	1	1.2856	0.3894	1.2633	0.3052	1.2664	0.4707	1.2890	0.3481	1.2661	0.2503	1.2641	0.2650
16	2	2.6019	0.7723	2.6020	0.6240	2.6086	0.9458	2.6104	0.6983	2.6144	0.5493	2.6118	0.5822
16	3	4.0117	1.1662	4.0033	1.0158	4.0104	1.4563	4.0315	1.0482	4.0100	0.7514	3.9995	0.7996
16	4	5.3672	1.5853	5.3660	1.2388	5.4153	1.9546	5.3872	1.7005	5.4068	1.0754	5.4164	1.1394
16	5	6.8414	1.9796	6.7974	1.5472	6.8492	2.4427	6.7358	1.7725	6.8282	1.2860	6.8520	1.3669
16	6	8.2287	2.3533	8.2323	1.9350	8.2455	2.8964	8.2294	2.0883	8.2677	1.5792	8.2333	1.6597
16	7	9.6765	2.8021	9.7268	2.5372	9.6425	3.4583	9.6975	2.5245	9.5814	1.8375	9.5633	2.0202

Tabla 6.15 Tiempos medios secuencial y paralelo para el Quicksort en la IBM SP2

P	SIZE	BH		NO BAL		VIRTUAL		MANUAL		BALAN		BALVIRT	
		ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV
2	1	1.6049	0.0514	1.5625	0.2063	0.7004	0.0262	1.6598	0.0452	1.5005	0.2165	1.3463	0.1911
2	2	1.6338	0.0236	1.6192	0.2054	0.6990	0.0041	1.6777	0.0235	1.4404	0.2217	1.3008	0.1907
2	3	1.6329	0.0310	1.5384	0.2096	0.6863	0.0097	1.6756	0.0321	1.6140	0.1873	1.4057	0.1968
2	4	1.6186	0.0349	1.5527	0.1946	0.7013	0.0027	1.6641	0.0205	1.5285	0.1923	1.3660	0.1687
2	5	1.6301	0.0277	1.5695	0.2153	0.6898	0.0029	1.6724	0.0258	1.5483	0.1967	1.4006	0.1756
2	6	1.6364	0.0809	1.6011	0.1643	0.6893	0.0055	1.6929	0.0306	1.4292	0.1393	1.2905	0.1215
2	7	1.6249	0.0199	1.5730	0.2407	0.6954	0.0038	1.6653	0.0210	1.4965	0.2036	1.3520	0.1809
4	1	2.3069	0.0804	2.3584	0.3682	1.2200	0.0185	2.4525	0.0868	2.4528	0.2312	2.2297	0.1934
4	2	2.3257	0.0791	2.3645	0.3708	1.2421	0.0189	2.4763	0.0655	2.3116	0.3132	2.0944	0.2941
4	3	2.4611	0.5903	2.1694	0.3864	1.2206	0.0164	2.4890	0.0733	2.3746	0.2477	2.1477	0.2185
4	4	2.3455	0.0416	2.4090	0.2574	1.2364	0.0195	2.4719	0.2619	2.4364	0.2280	2.4028	0.5963
4	5	2.3530	0.0565	2.4219	0.3172	1.2263	0.0149	2.4824	0.0627	2.5470	0.2417	2.3274	0.2211
4	6	2.4131	0.0826	2.3604	0.1558	1.2007	0.0654	2.5486	0.0624	2.4059	0.3434	2.1511	0.3815
4	7	2.3607	0.0495	2.2306	0.4218	1.2431	0.0171	2.4893	0.0546	2.5743	0.2941	2.3512	0.2754
8	1	2.8779	0.0842	3.2847	0.5256	1.9627	0.0725	3.1333	0.1082	3.8130	0.3988	4.2250	0.9575
8	2	2.9637	0.1008	3.6626	0.8253	1.9519	0.0487	3.1364	0.2053	3.7741	0.8024	3.3550	0.4867
8	3	3.0168	0.0949	3.1762	0.6477	1.9503	0.0421	3.2480	0.1254	3.6790	0.3859	3.4228	0.3324
8	4	2.9725	0.0481	3.4167	0.4451	1.9655	0.0344	3.1801	0.2427	3.7782	0.5333	3.4712	0.5010
8	5	2.9844	0.0982	3.3776	0.3205	1.9726	0.0392	3.2644	0.1175	4.0175	0.7263	3.6164	0.4925
8	6	3.0647	0.1003	3.5097	0.5723	1.9902	0.0423	3.2553	0.4035	3.8305	0.3244	3.5316	0.2796
8	7	2.9779	0.0907	3.0997	0.5546	1.9806	0.0437	3.2604	0.0997	3.7617	0.3754	3.4719	0.3414
16	1	3.3093	0.1795	4.2005	0.5255	2.6921	0.0707	3.7140	0.3282	5.0798	0.3434	4.7843	0.2774
16	2	3.3728	0.1198	4.2995	0.7565	2.7606	0.0925	3.7433	0.1439	4.8104	0.5035	4.5294	0.4540
16	3	3.4437	0.1336	4.1869	0.8328	2.7564	0.1060	3.8523	0.1847	5.3592	0.3496	5.0161	0.2664
16	4	3.3873	0.0766	4.4032	0.6022	2.7719	0.0928	3.4019	0.7663	5.1170	0.6925	4.8330	0.6300
16	5	3.4581	0.1079	4.4443	0.5187	2.8052	0.1124	3.8042	0.1323	5.3611	0.5801	5.0609	0.5321
16	6	3.5096	0.2216	4.3239	0.5691	2.8490	0.1009	3.9490	0.2183	5.3007	0.6143	5.0189	0.5570
16	7	3.4568	0.1177	3.8917	0.4806	2.7896	0.0634	3.8456	0.1506	5.1110	0.5969	4.8046	0.5809

Tabla 6.16 Aceleración media y Desviación estándar de la aceleración para el Quicksort en la IBM SP2

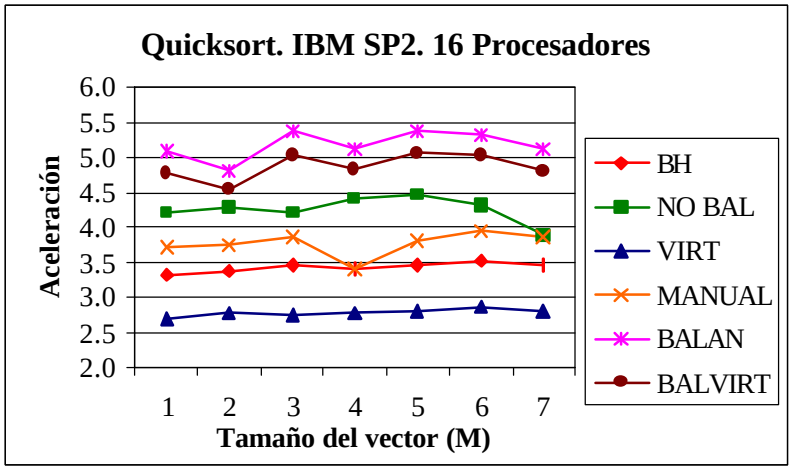

Figura 6.27 Quicksort. IBM SP2. Resultados computacionales para 16 procesadores

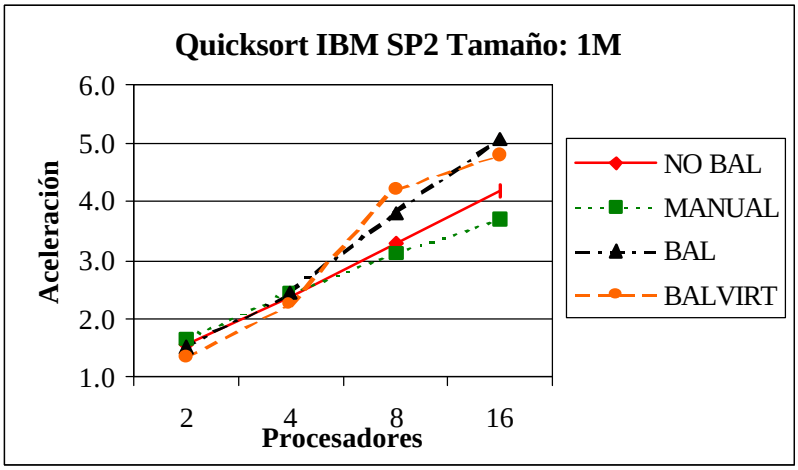

Figura 6.28 Quicksort. IBM SP2
Equilibrado de carga. Tamaño del vector: 1M

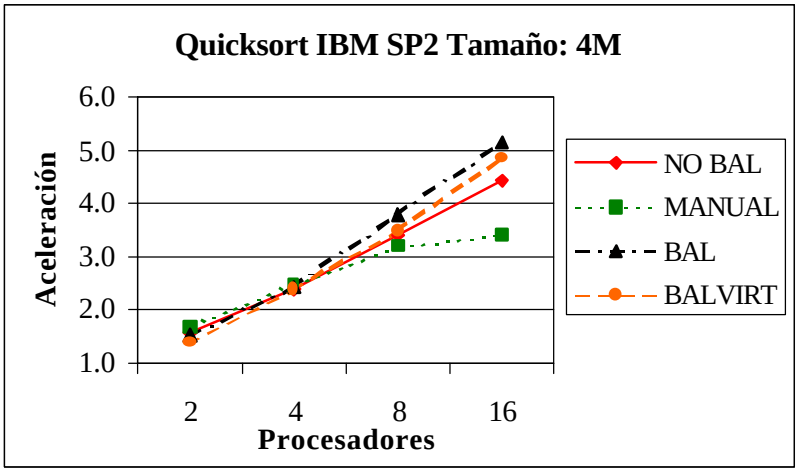

Figura 6.29 Quicksort. IBM SP2
Equilibrado de carga. Tamaño del vector: 4M

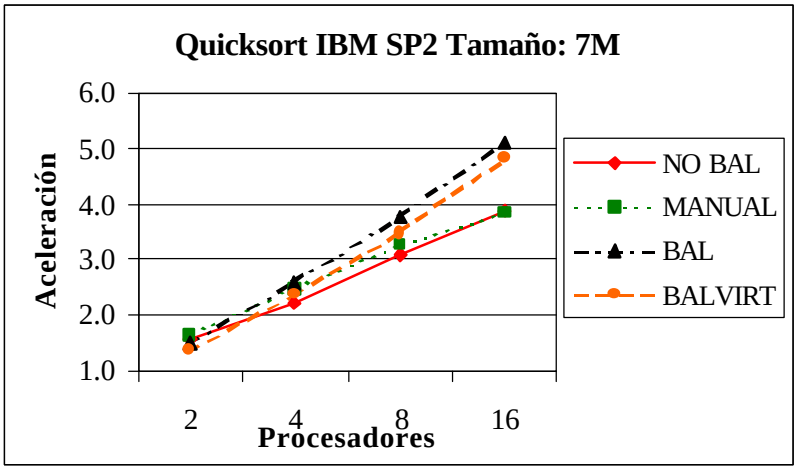

Figura 6.30 Quicksort. IBM SP2
Equilibrado de carga. Tamaño del vector: 7M

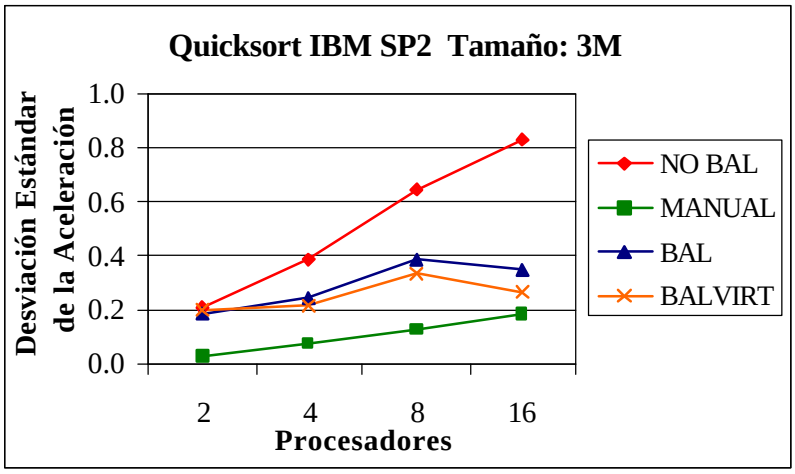

Figura 6.31 Quicksort. IBM SP2. Desviación Estándar de la Aceleración para un vector de 3M enteros

6.3.5. IBM SP2 (switch antiguo)

P	SIZE	BH		NO BAL		VIRTUAL		MANUAL		BALAN		BALVIRT	
		T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR
2	1	3.1796	2.2920	3.1782	2.0732	3.1930	4.6861	3.1767	1.9384	3.2100	2.0857	3.1689	2.3204
2	2	6.5619	4.1585	6.5438	4.1352	6.5486	9.6079	6.5550	3.9436	6.5776	4.4337	6.5423	4.5563
2	3	10.0115	6.4212	10.0111	7.1607	10.0340	15.1645	10.0239	6.0449	10.0802	6.6918	10.0254	7.6512
2	4	13.5177	8.6013	13.6298	8.9119	13.5180	19.7328	13.5171	8.1745	13.5794	8.9340	13.5324	10.0810
2	5	17.0265	10.5996	17.0373	11.0609	17.0268	25.3498	17.0397	10.5911	17.1660	11.1192	17.0547	12.4248
2	6	20.5985	12.6374	20.5747	13.0521	20.6010	30.7518	20.6012	12.2519	20.7258	13.1284	20.7760	14.6524
2	7	24.2140	15.4785	24.2209	16.0042	24.2179	35.8836	24.2323	14.9797	24.3835	16.2313	24.2089	17.6966
4	1	3.1809	1.4311	3.1732	1.4063	3.1777	2.6685	3.1760	1.3429	3.2060	1.2924	3.1819	1.4204
4	2	6.5754	3.2085	6.5783	2.8791	6.5648	5.4701	6.5455	2.7064	6.5977	2.9346	6.5479	2.9193
4	3	10.0260	4.4232	11.3760	5.4669	10.0204	8.5047	10.0240	4.4370	10.0812	4.4982	10.0223	4.9151
4	4	13.6263	6.2559	16.8817	7.4956	13.5115	11.5545	13.5072	5.6064	13.7795	5.4621	13.5166	5.9612
4	5	17.0165	7.5511	17.0352	7.6248	17.0355	14.4082	17.0263	7.2586	19.4042	7.6695	17.0546	7.3047
4	6	20.5792	9.2408	20.6019	9.1170	20.5886	17.6138	20.8575	8.4412	23.3178	9.9026	20.6086	9.4554
4	7	24.2349	10.6280	24.2337	11.7629	24.2543	20.2560	24.2090	9.9866	24.4075	10.6593	24.2349	11.3568
8	1	3.1885	1.2376	3.1610	1.0212	3.1593	1.7131	3.1641	1.0503	3.1752	0.8537	3.1859	0.9400
8	2	6.5606	2.3395	6.5136	2.2187	6.5169	3.4483	6.5076	2.1015	6.5665	1.6804	6.5364	2.1415
8	3	10.0133	3.8264	9.9653	3.7245	9.9806	5.3324	9.9910	3.2810	10.0285	2.5973	10.0158	2.8270
8	4	13.7118	4.8465	13.4344	4.0817	13.4409	7.3213	13.4414	4.5470	13.5118	3.5584	13.4957	3.8367
8	5	17.0236	5.9952	16.9280	5.2276	16.9528	9.1468	16.9789	5.7267	17.0718	4.5001	17.0248	4.8866
8	6	20.5874	7.0981	20.4865	6.8188	20.5088	10.7319	20.5239	6.3720	20.6306	5.6220	20.5826	6.2422
8	7	24.2149	8.5505	24.1173	8.2321	24.2265	13.0610	24.1523	7.8335	24.5269	6.7842	24.2537	7.3137

Tabla 6.17 Tiempos medios secuencial y paralelo para el Quicksort en la IBM SP2 (switch antiguo)

P	SIZE	BH		NO BAL		VIRTUAL		MANUAL		BALAN		BALVIRT	
		ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV
2	1	1.4907	0.2944	1.5584	0.2100	0.6814	0.0120	1.6397	0.0417	1.5643	0.2161	1.3875	0.1845
2	2	1.5835	0.0948	1.6102	0.2108	0.6817	0.0077	1.6625	0.0267	1.5632	0.3228	1.4624	0.1962
2	3	1.5686	0.1180	1.4225	0.1949	0.6622	0.0174	1.6587	0.0282	1.5342	0.2118	1.3378	0.2010
2	4	1.5764	0.0867	1.5513	0.1903	0.6851	0.0036	1.6538	0.0164	1.5432	0.1941	1.3613	0.1679
2	5	1.6068	0.0289	1.5679	0.2089	0.6717	0.0037	1.6195	0.1248	1.5707	0.2061	1.3930	0.1682
2	6	1.6305	0.0315	1.5918	0.1620	0.6700	0.0064	1.6819	0.0291	1.5941	0.1611	1.4400	0.1962
2	7	1.5680	0.0775	1.5531	0.2425	0.6750	0.0061	1.6229	0.0918	1.5419	0.2437	1.4048	0.2225
4	1	2.2246	0.0710	2.3046	0.3348	1.1911	0.0199	2.3676	0.0867	2.5118	0.2826	2.2676	0.2508
4	2	2.1383	0.3548	2.3498	0.3866	1.2004	0.0192	2.4199	0.0638	2.4014	0.4758	2.2667	0.2341
4	3	2.2681	0.0588	2.1288	0.4528	1.1784	0.0170	2.3178	0.3217	2.3291	0.4469	2.1190	0.4002
4	4	2.1937	0.1578	2.2699	0.2957	1.1752	0.0798	2.4100	0.0436	2.5421	0.2286	2.2879	0.2295
4	5	2.2542	0.0411	2.2798	0.3329	1.1825	0.0144	2.3635	0.1962	2.5690	0.3994	2.3504	0.2011
4	6	2.2455	0.1940	2.2862	0.2384	1.1717	0.0568	2.4763	0.1395	2.3636	0.2505	2.1941	0.1816
4	7	2.2814	0.0521	2.1374	0.4380	1.1976	0.0145	2.4254	0.0565	2.3520	0.4120	2.1850	0.3572
8	1	2.6282	0.3337	3.1787	0.4877	1.8466	0.0707	3.0176	0.1358	3.7396	0.2902	3.4152	0.3256
8	2	2.8064	0.0813	3.1284	0.7558	1.8915	0.0576	3.1010	0.1214	3.9465	0.3799	3.3884	0.8129
8	3	2.7088	0.4194	2.9025	0.7220	1.8725	0.0409	3.0601	0.2095	3.8786	0.2812	3.5605	0.2741
8	4	2.8305	0.1121	3.3333	0.4033	1.8499	0.1510	2.9854	0.2790	3.8181	0.2903	3.5363	0.2631
8	5	2.8420	0.0876	3.2710	0.3251	1.8617	0.1214	3.0225	0.3739	3.8069	0.2369	3.4977	0.2313
8	6	2.9022	0.0755	3.0905	0.5304	1.9114	0.0305	3.2241	0.1086	3.6787	0.1950	3.3534	0.4038
8	7	2.8342	0.0809	3.0216	0.5313	1.8597	0.0999	3.0874	0.1156	3.6327	0.2791	3.3309	0.2280

Tabla 6.18 Aceleración media y Desviación estándar de la aceleración para el Quicksort en la IBM SP2 (switch antiguo)

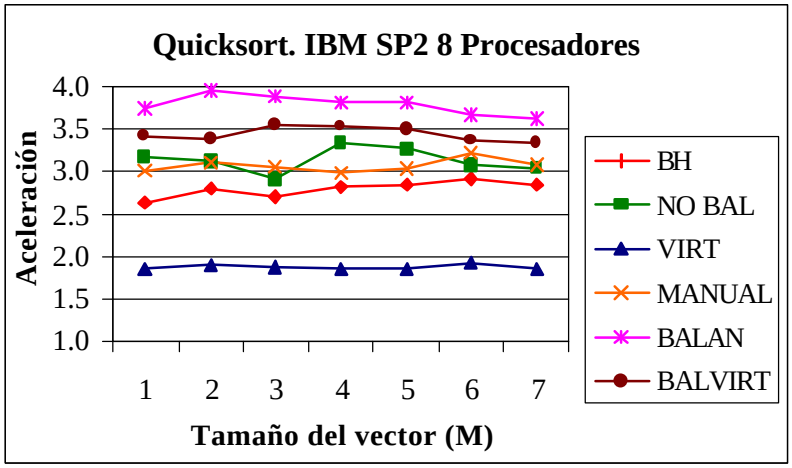

Figura 6.32 Quicksort. IBM SP2 (switch antiguo). Resultados computacionales para 8 procesadores

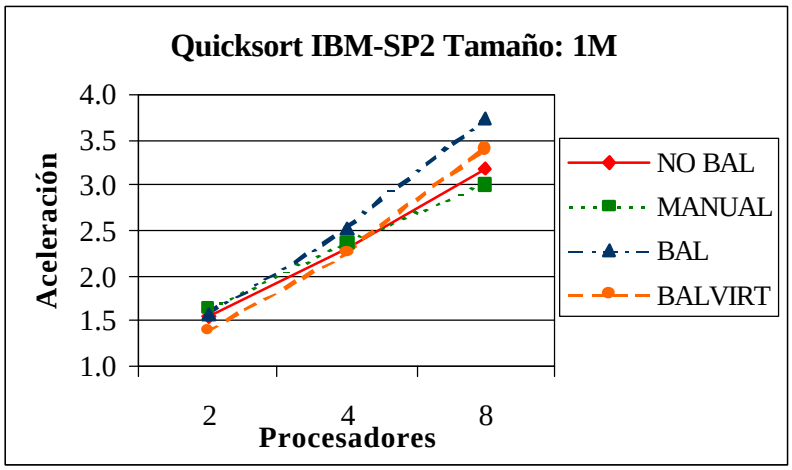

Figura 6.33 Quicksort. IBM SP2 (switch antiguo)
Equilibrado de carga. Tamaño del vector: 1M

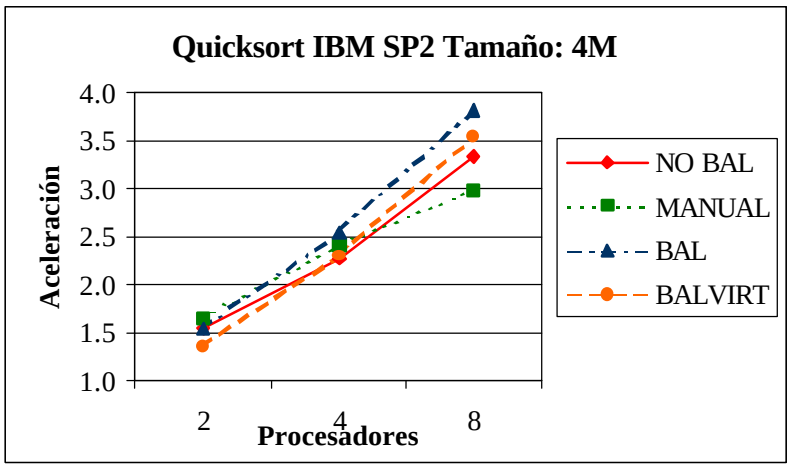

Figura 6.34 Quicksort. IBM SP2 (switch antiguo)
Equilibrado de carga. Tamaño del vector: 4M

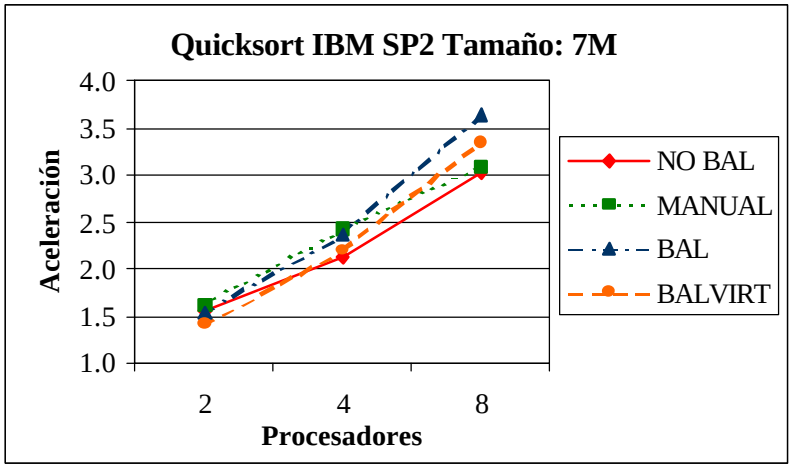

Figura 6.35 Quicksort. IBM SP2 (switch antiguo)
Equilibrado de carga. Tamaño del vector: 7M

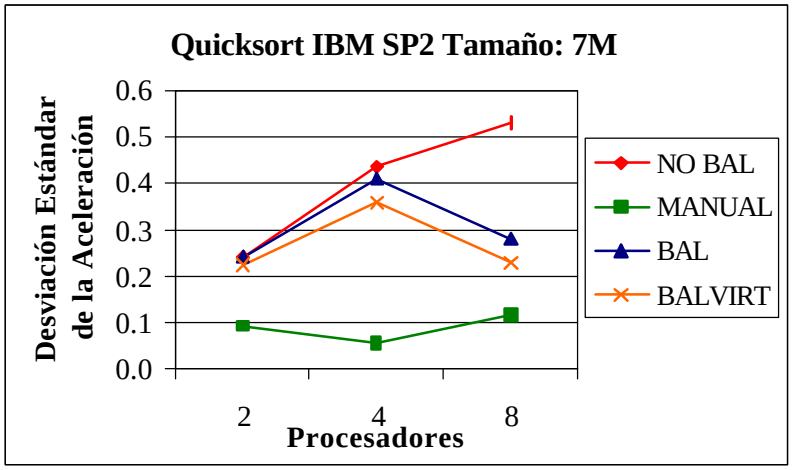

Figura 6.36 Quicksort. IBM SP2 (switch antiguo)
Desviación Estándar de la Aceleración para un vector de 7M enteros

6.3.6. Silicon Graphics Origin 2000 (karnak3)

P	SIZE	BH		NO BAL		VIRTUAL		MANUAL		BALAN		BALVIRT	
		T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR
2	1	1.1290	0.6743	1.1188	0.7555	1.1235	1.2528	1.1160	0.6551	1.1123	0.7565	1.1178	0.8298
2	2	2.3748	1.4542	2.3525	1.6927	2.3515	2.6400	2.3580	1.4113	2.3591	1.6989	2.3582	1.8545
2	3	3.6478	2.2594	3.6205	2.3145	3.6189	4.1265	3.6476	2.1900	3.6317	2.3219	3.6293	2.5276
2	4	4.9513	3.0703	4.9159	3.3278	4.9143	5.4764	4.9753	3.0000	4.9300	3.3355	4.9245	3.6248
2	5	6.2764	3.9011	6.2426	4.1657	6.2352	7.0233	6.3354	3.8366	6.2498	4.1754	6.2411	4.5240
2	6	7.6193	4.7073	7.6047	5.4331	7.5753	8.5538	7.7218	4.6552	7.5908	5.4331	7.5742	5.8799
2	7	8.9746	5.6087	8.9869	6.1773	8.9293	10.0414	9.1198	5.5756	8.9345	6.1479	8.9255	6.6645
4	1	1.1272	0.4630	1.1190	0.5071	1.1116	0.7400	1.1186	0.4472	1.1089	0.4649	1.1133	0.5046
4	2	2.3747	1.0064	2.3515	1.3700	2.3516	1.5740	2.3517	0.9740	2.3529	1.0934	2.3508	1.1769
4	3	3.6448	1.5723	3.6189	1.7539	3.6189	2.4529	3.6199	1.4979	3.6207	1.6381	3.6194	1.7660
4	4	4.9494	2.1524	4.9126	2.3308	4.9121	3.2815	4.9145	2.0510	4.9168	2.1761	4.9125	2.3393
4	5	6.2735	2.7466	6.2351	2.9210	6.2318	4.1985	6.2359	2.6128	6.2375	2.6470	6.2323	2.8386
4	6	7.6152	3.3248	7.5734	3.7725	7.5711	5.1058	7.5755	3.1574	7.5767	3.4114	7.5716	3.6608
4	7	8.9691	3.9647	8.9272	4.4119	8.9215	6.0280	8.9289	3.7832	8.9359	3.7503	8.9236	4.0190
8	1	1.1327	0.3664	1.1116	0.3872	1.1126	0.4942	1.1097	0.3596	1.1097	0.3167	1.1112	0.3379
8	2	2.3949	0.8206	2.3520	1.0200	2.3509	1.0800	2.3523	0.7931	2.3550	0.7636	2.3607	0.8201
8	3	3.7090	1.3075	3.6195	1.3781	3.6181	1.7165	3.6201	1.2427	3.6473	1.1239	3.6405	1.2576
8	4	5.0624	1.8268	4.9132	1.8644	4.9126	2.3173	4.9151	1.7447	4.9893	1.5266	4.9572	1.7048
8	5	6.4475	2.3715	6.2350	2.3350	6.2319	2.9765	6.2360	2.2210	6.3654	1.8865	6.2940	2.0791
8	6	7.8533	2.8893	7.5736	3.0762	7.5709	3.6657	7.5752	2.7550	7.7657	2.3221	7.6324	2.5250
8	7	9.2750	3.4656	8.9289	3.3161	8.9214	4.4031	8.9291	3.3487	9.1861	2.7927	9.0391	3.1679
16	1	1.1285	0.3127	1.1131	0.3083	1.1113	0.3905	1.1104	0.3048	1.1203	0.2503	1.1109	0.2554
16	2	2.3700	0.6935	2.3532	0.8423	2.3703	0.8727	2.3515	0.7080	2.4063	0.6149	2.3510	0.6429
16	3	3.6407	1.0916	3.6217	1.1449	3.6371	1.3708	3.6256	1.1051	3.7303	0.8757	3.6189	0.9250
16	4	4.9418	1.4962	4.9279	1.5551	4.9574	1.8783	4.9283	1.5231	5.1209	1.3041	4.9128	1.3432
16	5	6.2643	1.9233	6.2799	1.9546	6.3332	2.4407	6.2757	1.9841	6.5258	1.6097	6.2325	1.6288
16	6	7.6038	2.3286	7.6339	2.5947	7.7655	2.9923	7.6184	2.4380	7.9604	2.0567	7.5913	2.0076
16	7	8.9559	2.7934	9.0116	2.7309	9.2676	3.6561	8.9804	2.9759	9.4334	2.3960	8.9720	2.3944

Tabla 6.19 Tiempos medios secuencial y paralelo para el Quicksort en la SGI Origin 2000 (karnak3)

P	SIZE	BH		NO BAL		VIRTUAL		MANUAL		BALAN		BALVIRT	
		ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV
2	1	1.6744	0.0197	1.5129	0.2407	0.8968	0.0396	1.7039	0.0407	1.4991	0.2125	1.3747	0.2074
2	2	1.6342	0.0453	1.4189	0.2151	0.8909	0.0122	1.6720	0.0488	1.4178	0.2155	1.2985	0.1980
2	3	1.6151	0.0326	1.5840	0.1791	0.8771	0.0099	1.6663	0.0364	1.5834	0.1769	1.4537	0.1633
2	4	1.6127	0.0101	1.4997	0.1834	0.8974	0.0032	1.6586	0.0131	1.5007	0.1842	1.3796	0.1699
2	5	1.6094	0.0299	1.5217	0.1885	0.8879	0.0073	1.6520	0.0348	1.5200	0.1885	1.4010	0.1736
2	6	1.6189	0.0227	1.4114	0.1365	0.8857	0.0077	1.6593	0.0316	1.4087	0.1358	1.2987	0.1244
2	7	1.6003	0.0178	1.4783	0.1980	0.8893	0.0047	1.6359	0.0184	1.4761	0.1954	1.3606	0.1813
4	1	2.4351	0.0372	2.2814	0.3939	1.5023	0.0239	2.5020	0.0925	2.4029	0.2158	2.2227	0.2039
4	2	2.3625	0.0863	1.7742	0.3513	1.4948	0.0349	2.4177	0.0931	2.1815	0.2673	2.0262	0.2543
4	3	2.3204	0.0779	2.0923	0.2518	1.4759	0.0304	2.4193	0.0854	2.2373	0.2547	2.0737	0.2317
4	4	2.2996	0.0201	2.1548	0.3006	1.4970	0.0091	2.3963	0.0208	2.2785	0.2132	2.1172	0.1956
4	5	2.2853	0.0527	2.1861	0.3439	1.4846	0.0226	2.3883	0.0618	2.3727	0.2024	2.2113	0.1925
4	6	2.2913	0.0458	2.0503	0.3014	1.4832	0.0236	2.4007	0.0620	2.2723	0.3178	2.1160	0.2953
4	7	2.2626	0.0308	2.0724	0.3420	1.4801	0.0151	2.3608	0.0392	2.4102	0.2618	2.2471	0.2490
8	1	3.0930	0.0655	2.9456	0.4425	2.2516	0.0437	3.0871	0.0697	3.5395	0.3815	3.3245	0.3698
8	2	2.9241	0.1311	2.3746	0.4442	2.1809	0.0991	2.9735	0.1574	3.1211	0.3489	2.9053	0.2837
8	3	2.8401	0.1056	2.6916	0.4351	2.1095	0.0627	2.9205	0.1580	3.2775	0.3322	2.9278	0.3221
8	4	2.7718	0.0411	2.7183	0.4686	2.1209	0.0467	2.8205	0.1016	3.3268	0.4425	2.9397	0.3138
8	5	2.7220	0.0933	2.7424	0.4571	2.0960	0.0703	2.8119	0.1095	3.4428	0.4565	3.0992	0.4392
8	6	2.7201	0.0756	2.6155	0.6151	2.0673	0.0654	2.7539	0.1153	3.3714	0.3111	3.0531	0.3276
8	7	2.6774	0.0535	2.7174	0.2671	2.0281	0.0631	2.6683	0.0732	3.3226	0.3451	2.9057	0.4178
16	1	3.6119	0.1004	3.6972	0.5807	2.8468	0.0644	3.6455	0.1000	4.4944	0.2968	4.3647	0.2679
16	2	3.4277	0.1986	2.9022	0.5729	2.7217	0.1272	3.3324	0.1950	3.9635	0.4427	3.7015	0.4202
16	3	3.3412	0.1529	3.2442	0.5528	2.6592	0.1325	3.2905	0.1901	4.2777	0.2779	3.9309	0.2837
16	4	3.3039	0.0635	3.3167	0.6641	2.6410	0.0660	3.2386	0.0992	3.9896	0.4850	3.7169	0.4749
16	5	3.2612	0.1170	3.3093	0.5673	2.5975	0.0828	3.1694	0.1397	4.0803	0.3286	3.8620	0.3890
16	6	3.2672	0.0805	3.1387	0.7595	2.6001	0.1153	3.1308	0.1388	3.9219	0.4606	3.8371	0.4655
16	7	3.2079	0.0773	3.3269	0.3078	2.5364	0.0626	3.0202	0.0891	3.9751	0.3951	3.7770	0.3400

Tabla 6.20 Aceleración media y Desviación estándar de la aceleración para el Quicksort en la SGI Origin 2000 (karnak3)

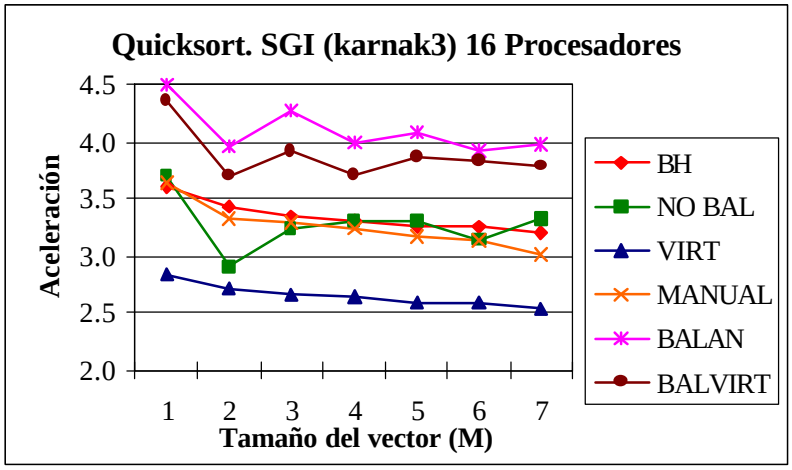

Figura 6.37 Quicksort. SGI Origin 2000 (karnak3). Resultados computacionales para 16 procesadores

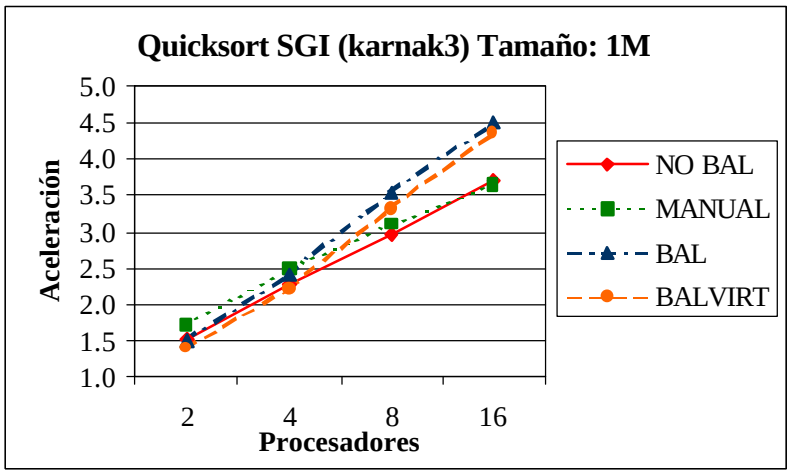

Figura 6.38 Quicksort. SGI Origin 2000 (karnak3)
Equilibrado de carga. Tamaño del vector: 1M

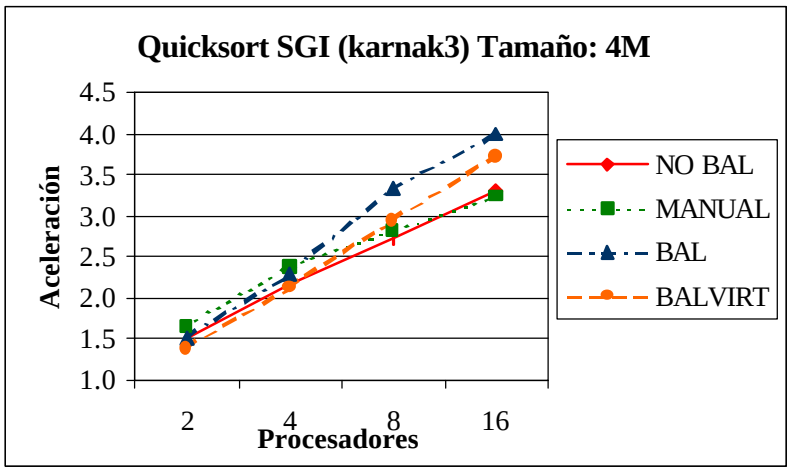

Figura 6.39 Quicksort. SGI Origin 2000 (karnak3)
Equilibrado de carga. Tamaño del vector: 4M

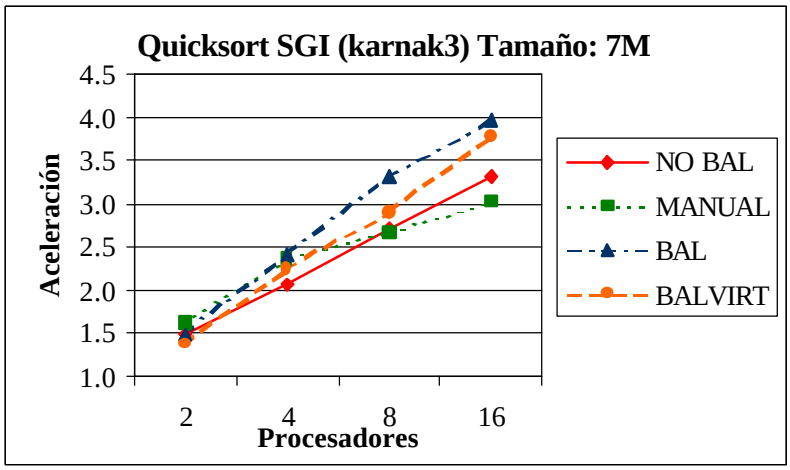

Figura 6.40 Quicksort. SGI Origin 2000 (karnak3)
Equilibrado de carga. Tamaño del vector: 7M

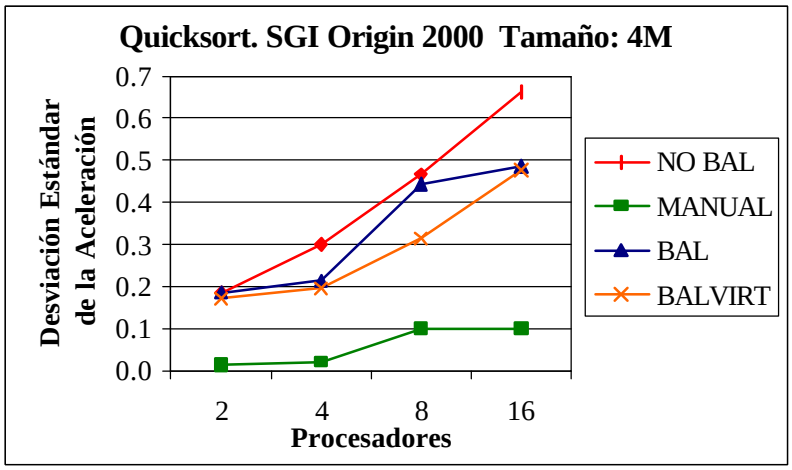

Figura 6.41 Quicksort. SGI Origin 2000 (karnak3)
Desviación Estándar de la Aceleración para un vector de 4M enteros

6.3.7. Silicon Graphics Origin 2000 (karnak2)

P	SIZE	BH		NO BAL		VIRTUAL		MANUAL		BALAN		BALVIRT	
		T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR	T SEC	T PAR
2	1	1.3828	0.8419	1.3546	0.8765	1.3654	1.5894	1.3608	0.8034	1.3685	0.8925	1.3545	0.9653
2	2	2.8647	1.7489	2.8431	1.7925	2.8502	3.2985	2.8549	1.7080	2.8614	1.8029	2.8427	1.9609
2	3	4.4216	2.7190	4.3984	2.9423	4.4092	5.1784	4.4323	2.6551	4.4207	2.9517	4.3979	3.2115
2	4	5.9909	3.7189	5.9603	3.9448	5.9812	6.8984	6.0354	3.6387	5.9918	3.9580	5.9603	4.2994
2	5	7.5922	4.7060	7.5573	4.9457	7.5816	8.8471	7.6758	4.6366	7.5911	4.9608	7.5580	5.3873
2	6	9.2042	5.6322	9.1733	5.8625	9.1979	10.6631	9.3308	5.5769	9.2074	5.8737	9.1704	6.3731
2	7	10.8605	6.7598	10.8311	7.1796	10.8650	12.5187	11.0498	6.7286	10.8911	7.2026	10.8299	7.7984
4	1	1.3607	0.5632	1.3532	0.5986	1.3626	0.9304	1.3520	0.5451	1.3690	0.5521	1.3556	0.5976
4	2	2.8514	1.2096	2.8480	1.2880	2.8561	1.9541	2.8428	1.1778	2.8560	1.1626	2.8490	1.2561
4	3	4.4040	1.9101	4.4261	2.1911	4.4169	3.0630	4.3990	1.8252	4.4183	2.0440	4.4072	2.1757
4	4	5.9705	2.6220	5.9968	2.6336	5.9902	4.1125	5.9616	2.5219	5.9921	2.4927	5.9764	2.6635
4	5	7.5679	3.3202	7.5971	3.3503	7.5992	5.2538	7.5599	3.1873	7.5922	3.1105	7.5809	3.3029
4	6	9.1788	3.9591	9.2281	4.1315	9.2096	6.2949	9.1729	3.8103	9.2133	4.0354	9.1967	4.2775
4	7	10.8338	4.7652	10.9077	5.2936	10.8733	7.4548	10.8321	4.5665	10.8805	4.7935	10.8583	5.1269
8	1	1.3625	0.4457	1.3597	0.4549	1.3639	0.6155	1.3519	0.4252	1.3537	0.3710	1.3525	0.3997
8	2	2.8574	0.9726	2.8526	0.9580	2.8875	1.3437	2.8425	0.9219	2.8396	0.7809	2.8802	0.8379
8	3	4.4148	1.5211	4.4138	1.6536	4.4852	2.1295	4.3979	1.4436	4.3967	1.2388	4.4788	1.3391
8	4	5.9827	2.0922	5.9857	1.9994	6.0885	2.8733	5.9599	1.9815	5.9454	1.6967	6.0763	1.8315
8	5	7.5829	2.6439	7.5889	2.5838	7.7529	3.6507	7.5570	2.4936	7.5435	2.1850	7.7176	2.3312
8	6	9.2008	3.1553	9.2040	3.1702	9.3825	4.3171	9.1697	2.9642	9.1473	2.6485	9.3705	2.8584
8	7	10.8538	3.8186	10.8682	3.9776	11.1003	5.1379	10.8291	3.5874	10.7942	3.2700	11.0891	3.5087
16	1	1.3573	0.3863	1.3572	0.3560	1.3610	0.4777	1.3532	0.3770	1.3542	0.2779	1.3596	0.2856
16	2	2.8514	0.8366	2.8558	0.7887	2.8527	1.0390	2.8424	0.8122	2.8598	0.6414	2.8604	0.6592
16	3	4.4032	1.3356	4.4146	1.2956	4.4154	1.6666	4.3980	1.2679	5.2323	1.1782	4.4205	1.0712
16	4	5.9689	1.8350	5.9889	1.6097	5.9895	2.2616	5.9597	1.7321	6.0818	1.8828	6.0105	1.4167
16	5	7.5656	2.3215	7.5959	2.0291	7.5973	2.9066	7.5569	2.1829	7.6047	1.7150	7.6005	1.7622
16	6	9.1767	2.7595	9.2129	2.5845	9.2167	3.4483	9.1691	2.5824	9.7073	3.1242	9.2713	4.3216
16	7	10.8312	3.3554	10.9298	3.3568	10.9303	4.0283	10.8298	3.1496	10.8848	2.4926	10.8782	2.5741

Tabla 6.21 Tiempos medios secuencial y paralelo para el Quicksort en la SGI Origin 2000 (karnak2)

P	SIZE	BH		NO BAL		VIRTUAL		MANUAL		BALAN		BALVIRT	
		ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV	ACEL	DESV
2	1	1.6430	0.0288	1.5691	0.2007	0.8591	0.0161	1.6941	0.0356	1.5569	0.1994	1.4249	0.1838
2	2	1.6382	0.0206	1.6149	0.2137	0.8641	0.0057	1.6718	0.0240	1.6164	0.2150	1.4763	0.1966
2	3	1.6267	0.0306	1.5208	0.2024	0.8515	0.0075	1.6699	0.0321	1.5239	0.2039	1.3933	0.1858
2	4	1.6111	0.0184	1.5327	0.1870	0.8671	0.0042	1.6589	0.0189	1.5359	0.1883	1.4065	0.1725
2	5	1.6136	0.0229	1.5543	0.2025	0.8570	0.0052	1.6559	0.0275	1.5566	0.2031	1.4271	0.1863
2	6	1.6347	0.0300	1.5798	0.1588	0.8627	0.0080	1.6737	0.0349	1.5824	0.1586	1.4527	0.1463
2	7	1.6068	0.0176	1.5473	0.2392	0.8679	0.0061	1.6425	0.0197	1.5509	0.2394	1.4246	0.2209
4	1	2.4176	0.0655	2.3116	0.3425	1.4647	0.0348	2.4821	0.0771	2.5106	0.2941	2.2991	0.2904
4	2	2.3586	0.0563	2.2668	0.3532	1.4618	0.0187	2.4150	0.0576	2.4672	0.1702	2.2768	0.1456
4	3	2.3070	0.0582	2.0824	0.3601	1.4424	0.0236	2.4121	0.0725	2.2385	0.4086	2.1038	0.4003
4	4	2.2776	0.0354	2.2944	0.2069	1.4568	0.0169	2.3650	0.0500	2.4215	0.2184	2.2614	0.2075
4	5	2.2802	0.0451	2.3037	0.2876	1.4467	0.0207	2.3731	0.0532	2.4583	0.2194	2.3108	0.1982
4	6	2.3195	0.0529	2.2422	0.1441	1.4633	0.0221	2.4089	0.0644	2.3012	0.2055	2.1646	0.1808
4	7	2.2742	0.0415	2.1227	0.3861	1.4588	0.0175	2.3735	0.0587	2.3128	0.3297	2.1668	0.3449
8	1	3.0613	0.1139	3.0640	0.4566	2.2190	0.1188	3.1825	0.1157	3.6651	0.2738	3.4013	0.2680
8	2	2.9402	0.0893	3.0575	0.5074	2.1507	0.0657	3.0858	0.0954	3.6766	0.3832	3.4671	0.3178
8	3	2.9056	0.0998	2.8159	0.5467	2.1079	0.0550	3.0496	0.1043	3.5664	0.2580	3.3614	0.2453
8	4	2.8607	0.0591	3.0219	0.3023	2.1197	0.0380	3.0091	0.0643	3.5145	0.2006	3.3279	0.1931
8	5	2.8696	0.0674	2.9621	0.2668	2.1260	0.0729	3.0330	0.0905	3.4678	0.2433	3.3247	0.2290
8	6	2.9182	0.0844	2.9454	0.3746	2.1742	0.0461	3.0965	0.1035	3.4624	0.1848	3.2867	0.1762
8	7	2.8438	0.0646	2.8044	0.4477	2.1615	0.0465	3.0205	0.0753	3.3155	0.2263	3.1745	0.2175
16	1	3.5235	0.1898	3.8664	0.4864	2.8516	0.1045	3.5960	0.1817	4.8931	0.3236	4.7812	0.3673
16	2	3.4125	0.1229	3.7293	0.6221	2.7484	0.0916	3.5034	0.1233	4.4878	0.3671	4.3743	0.3936
16	3	3.3016	0.1317	3.5568	0.6083	2.6536	0.1089	3.4731	0.1281	4.4409	0.4062	4.1606	0.3628
16	4	3.2546	0.0776	3.7639	0.4300	2.6489	0.0364	3.4428	0.0849	4.0134	0.8429	4.2426	0.3966
16	5	3.2609	0.0825	3.7719	0.3501	2.6166	0.0875	3.4655	0.1152	4.4757	0.4454	4.3533	0.4304
16	6	3.3284	0.1044	3.6065	0.4077	2.6760	0.0975	3.5551	0.1355	3.6932	1.3304	3.6759	1.5664
16	7	3.2299	0.0784	3.3014	0.3923	2.7159	0.0843	3.4415	0.1033	4.4414	0.5300	4.2966	0.5075

Tabla 6.22 Aceleración media y Desviación estándar de la aceleración para el Quicksort en la SGI Origin 2000 (karnak2)

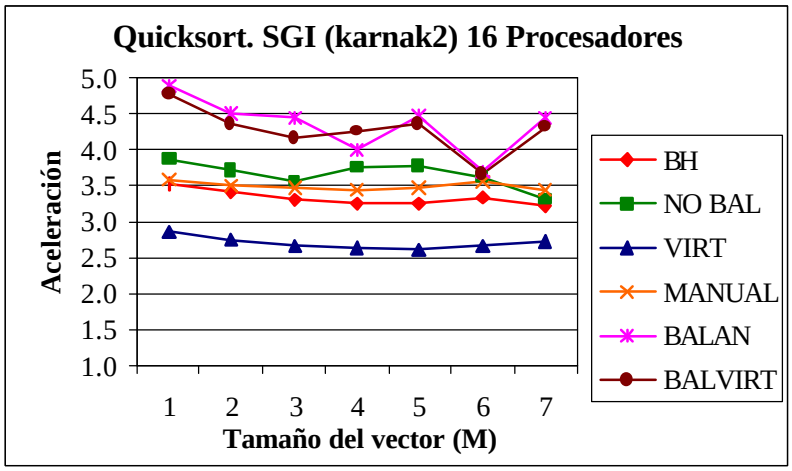

Figura 6.42 Quicksort. SGI Origin 2000 (karnak2). Resultados computacionales para 16 procesadores

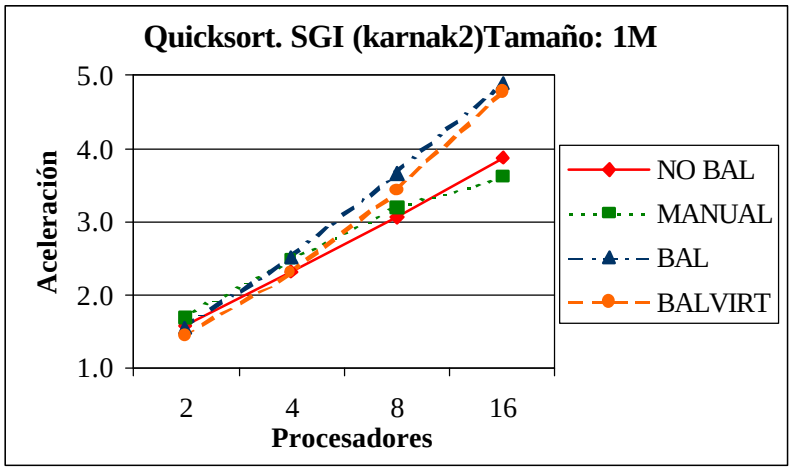

Figura 6.43 Quicksort. SGI Origin 2000 (karnak2)
Equilibrado de carga. Tamaño del vector: 1M

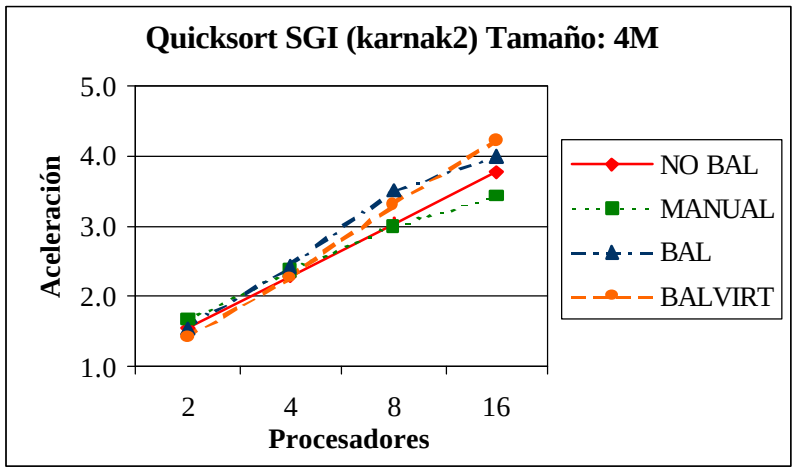

Figura 6.44 Quicksort. SGI Origin 2000 (karnak2)
Equilibrado de carga. Tamaño del vector: 4M

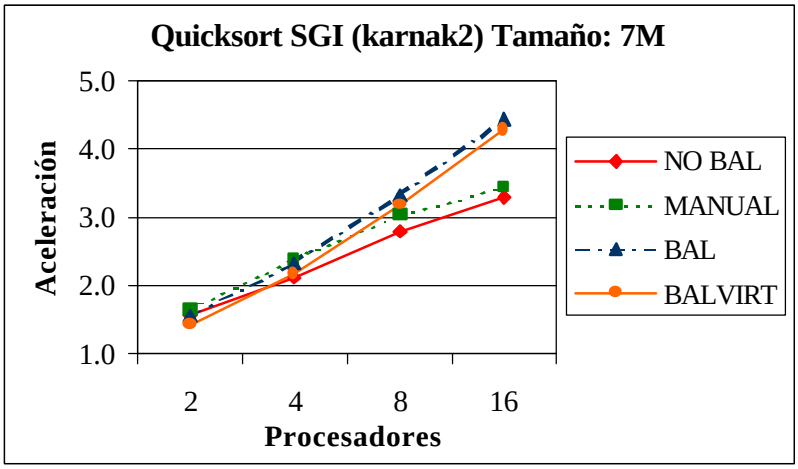

Figura 6.45 Quicksort. SGI Origin 2000 (karnak2)
Equilibrado de carga. Tamaño del vector: 7M

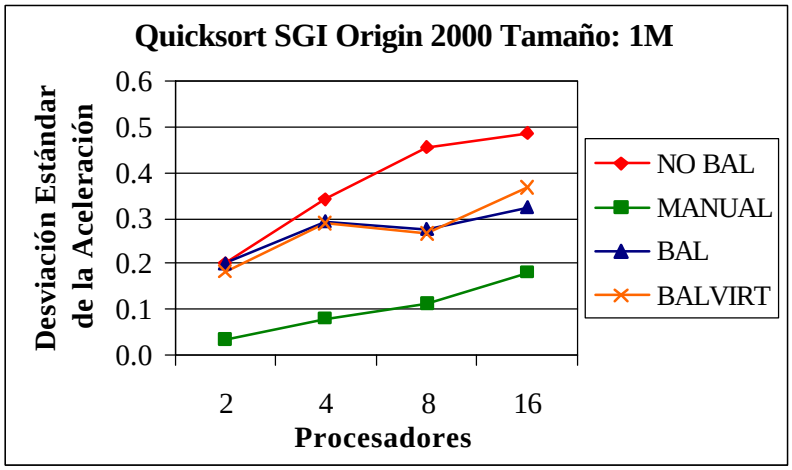

Figura 6.46 Quicksort. SGI Origin 2000 (karnak2)
Desviación Estándar de la Aceleración para un vector de 1M enteros

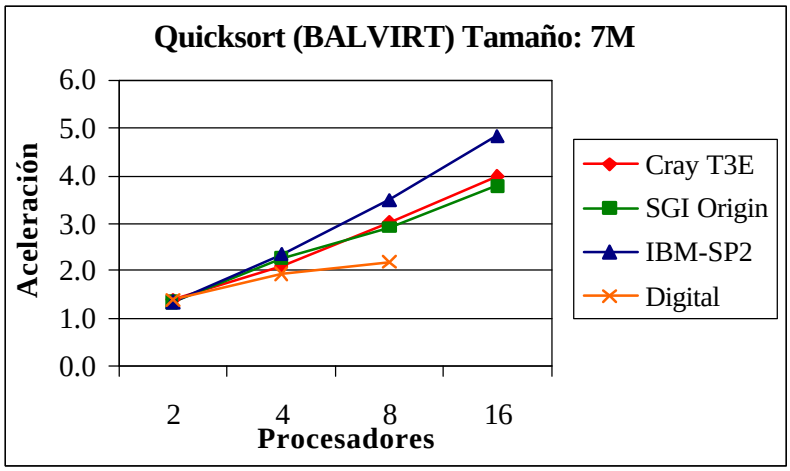

Figura 6.47 El algoritmo Quicksort con BALVIRT en las diferentes plataformas. Tamaño 7M.

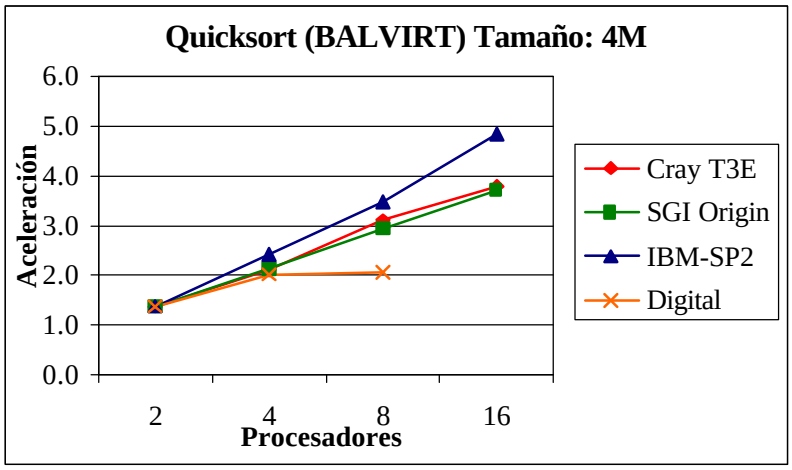

Figura 6.48 El algoritmo Quicksort con BALVIRT en las diferentes plataformas. Tamaño 4M.

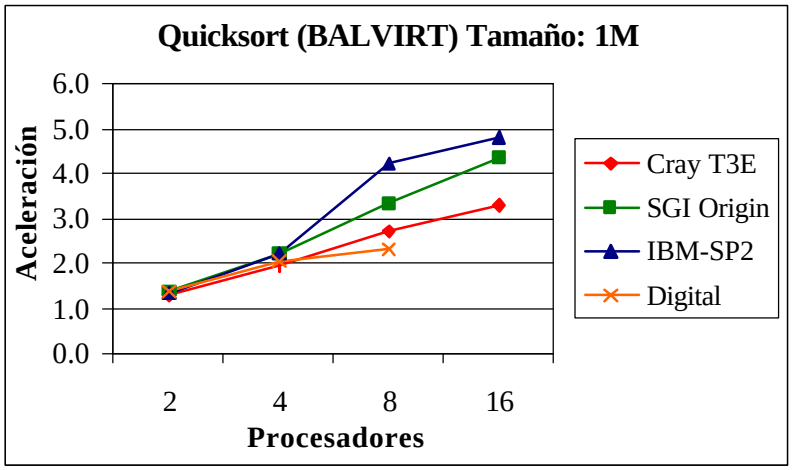

Figura 6.49 El algoritmo Quicksort con BALVIRT en las diferentes plataformas. Tamaño 1M.

6.4. La Quickhull

En esta sección exponemos los resultados computacionales obtenidos con la implementación en *llc* del algoritmo Quickhull que calcula la envolvente convexa de una nube de puntos (ver Figura 4.30 del Capítulo 4).

Las tablas presentan los tiempos secuencial y paralelo (T SEC y T PAR) así como la aceleración obtenida (ACEL) para diferentes tamaños de problema y diferente número de procesadores mientras las correspondientes representaciones gráficas muestran la aceleración obtenida en cada caso.

Las máquinas utilizadas para el algoritmo fueron los Cray T3E y T3D, Digital Alphaserver, IBM SP2 (versión actualizada), y la SGI Origin 2000 (versión actual).

Para cada número de procesadores se han generado nubes de puntos de tamaños 2M, 3M y 6M de forma que los puntos estuvieran distribuidos según una semicircunferencia. De este modo, todos los puntos del vector de entrada han de pertenecer a la envolvente, y el problema tiene la máxima dificultad posible.

Las figuras 6.50 a la 6.55 presentan gráficamente los resultados para cada una de las máquinas y diferente número de procesadores (dependiendo de la máquina). Las figuras 6.56-6.58 presentan una comparativa del comportamiento del algoritmo para las diferentes plataformas utilizadas.

En todas las máquinas las aceleraciones crecen de forma lineal para los tamaños de problema considerados. En las gráficas que comparan las diferentes máquinas entre sí se observa que el Cray T3D es el que obtiene mejores aceleraciones. No obstante, los tiempos absolutos de esta máquina son inferiores a los del Cray T3E.

6.4.1. Cray T3E

# de Procs.	Tamaño	T SEC	T PAR	ACEL
2	2M	23.6705	12.4214	1.9056
4	2M	23.6764	9.4098	2.5161
8	2M	23.6740	6.0659	3.9028
16	2M	23.6681	4.2117	5.6196
32	2M	23.6704	3.3128	7.1452
64	2M	23.6712	2.8778	8.2253
2	4M	49.3685	25.8540	1.9095
4	4M	49.3682	19.5127	2.5301
8	4M	49.3686	12.4894	3.9528
16	4M	49.3626	8.5881	5.7478
32	4M	49.3652	6.6969	7.3713
64	4M	49.3660	5.7665	8.5608
2	6M	77.6655	40.6732	1.9095
4	6M	77.6672	30.6134	2.5370
8	6M	77.6620	19.5395	3.9746
16	6M	77.6568	13.4267	5.7838
32	6M	77.6612	10.3817	7.4806
64	6M	77.6665	8.9308	8.6965

Tabla 6.23 Resultados de la Quickhull en el Cray T3E

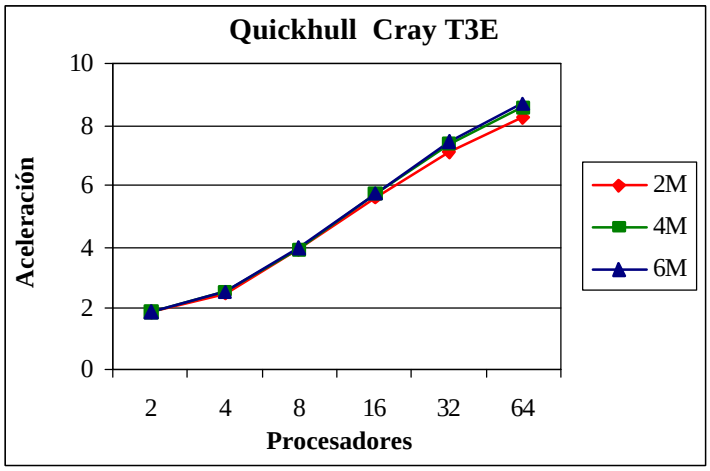

Figura 6.50 Resultados de la Quickhull en el Cray T3E

6.4.2. Cray T3D

# de Procs.	Tamaño	T SEC	T PAR	ACEL
2	2M	126.0793	66.9552	1.8830
4	2M	126.0781	49.2204	2.5615
8	2M	126.0957	31.1557	4.0473
16	2M	126.0809	21.5890	5.8400
32	2M	126.0798	16.4028	7.6865
64	2M	126.1361	14.0104	9.0030
128	2M	126.1236	12.8808	9.7916
2	4M	261.7440	138.7705	1.8862
4	4M	261.8281	101.8111	2.5717
8	4M	261.8277	64.1030	4.0845
16	4M	261.8282	44.0079	5.9496
32	4M	261.7440	33.1672	7.8917
64	4M	261.8685	28.1464	9.3038
128	4M	261.7396	25.7776	10.1538
2	6M	404.2576	214.0758	1.8884
4	6M	404.2578	156.9282	2.5761
8	6M	404.2585	98.0902	4.1213
16	6M	404.2723	67.3489	6.0027
32	6M	404.2503	50.6874	7.9754
64	6M	404.3199	42.9558	9.4125
128	6M	404.2592	39.2951	10.2878

Tabla 6.24 Resultados de la Quickhull en el Cray T3D

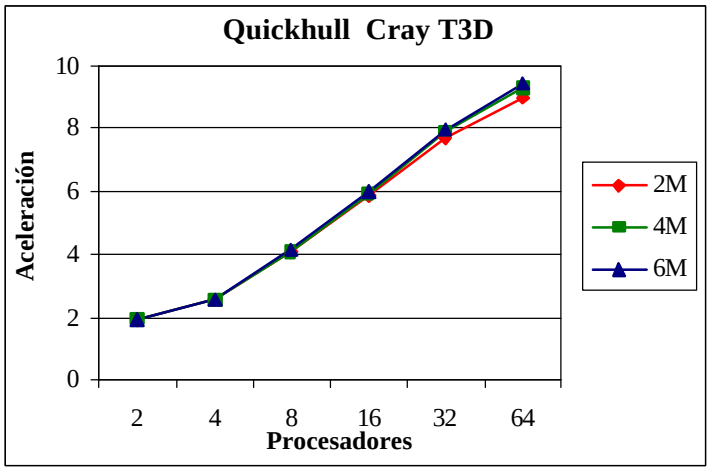

Figura 6.51 Resultados de la Quickhull en el Cray T3D

6.4.3. Digital Alphaserver 8400

# de Procs.	Tamaño	T SEC	T PAR	ACEL
2	2M	27.0088	14.5410	1.8574
4	2M	27.0029	11.0840	2.4362
8	2M	27.0635	7.7705	3.4828
2	4M	56.4229	30.4219	1.8547
4	4M	56.4365	22.8887	2.4657
8	4M	56.4785	16.1270	3.5021
2	6M	89.3545	47.6914	1.8736
4	6M	89.3887	35.3154	2.5312
8	6M	89.0205	26.1787	3.4005

Tabla 6.25 Resultados de la Quickhull en el Digital Alphaserver 8400

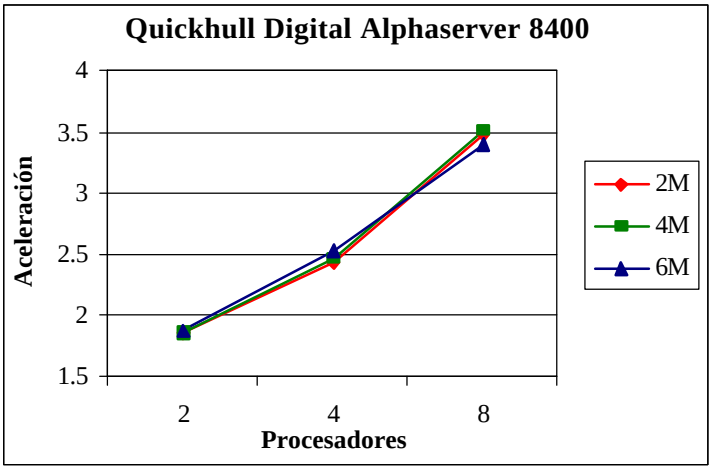

Figura 6.52 Resultados de la Quickhull en el Digital Alphaserver 8400

6.4.4. Hitachi SR2201

# de Procs.	Tamaño	T SEC	T PAR	ACEL
2	2M	103.0613	53.9973	1.9086
4	2M	103.0694	40.7472	2.5295
8	2M	104.6024	26.1133	4.0057
2	4M	215.1226	112.6054	1.9104
4	4M	215.2842	84.6780	2.5424
8	4M	218.5759	53.9450	4.0518
2	6M	330.0782	172.5762	1.9127
4	6M	330.3416	129.5972	2.5490
8	6M	334.9004	82.2620	4.0711

Tabla 6.26 Resultados de la Quickhull en el Hitachi SR2201

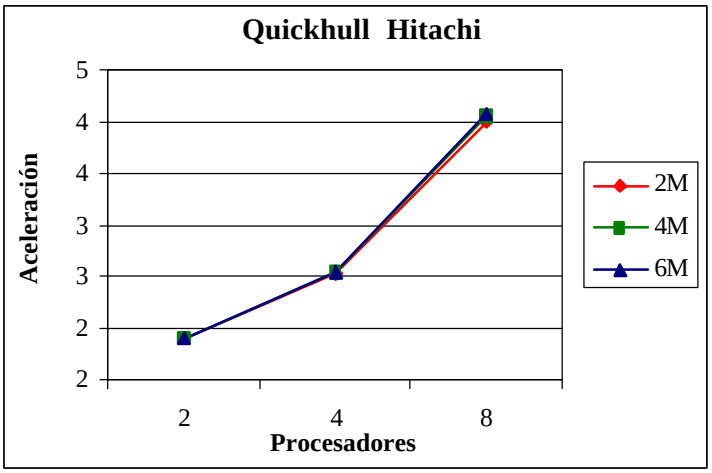

Figura 6.53 Resultados de la Quickhull en el Hitachi SR2201

6.4.5. IBM SP2

# de Procs.	Tamaño	T SEC	T PAR	ACEL
2	2M	52.7279	27.5325	1.9151
4	2M	54.1362	19.0295	2.8449
8	2M	53.8192	12.0375	4.4710
16	2M	54.1643	11.4089	4.7476
2	4M	110.0664	57.3421	1.9195
4	4M	109.9721	42.5705	2.5833
8	4M	111.5263	25.6277	4.3518
16	4M	111.8114	26.9129	4.1546
2	6M	168.5441	88.1766	1.9114
4	6M	171.2568	63.0194	2.7175
8	6M	171.0995	39.0011	4.3870
16	6M	179.4294	33.7183	5.3214

Tabla 6.27 Resultados de la Quickhull en el IBM SP2

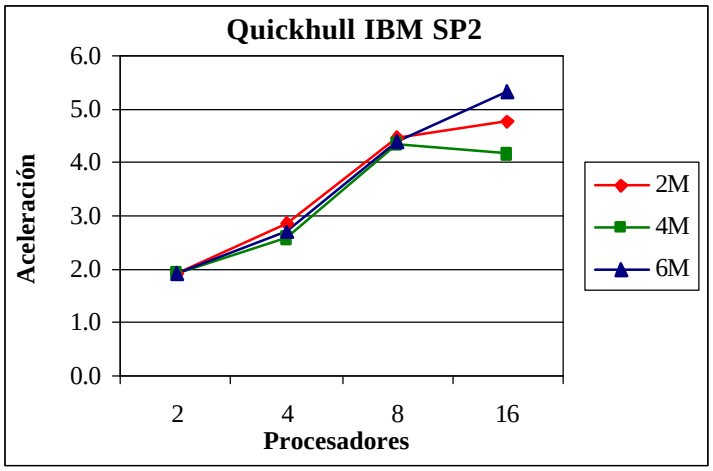

Figura 6.54 Resultados de la Quickhull en el IBM SP2

6.4.6. Silicon Graphics Origin 2000 (karnak3)

# de Procs.	Tamaño	T SEC	T PAR	ACEL
2	2M	16.4892	8.7899	1.8759
4	2M	16.4453	6.7465	2.4376
8	2M	16.3307	4.6344	3.5238
16	2M	16.4381	3.2793	5.0127
2	4M	34.2010	18.2486	1.8742
4	4M	34.1999	14.0221	2.4390
8	4M	34.3323	9.2881	3.6964
16	4M	34.3728	6.5396	5.2561
2	6M	54.1118	28.5518	1.8952
4	6M	53.8871	21.3512	2.5238
8	6M	53.9707	14.0467	3.8422
16	6M	53.6571	10.4585	5.1305

Tabla 6.28 Resultados de la Quickhull en el SGI Origin 2000 (karnak3)

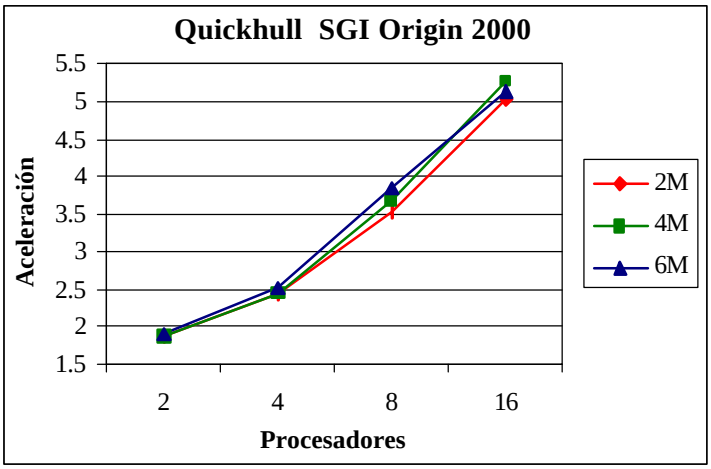

Figura 6.55 Resultados de la Quickhull en el SGI Origin 2000 (karnak3)

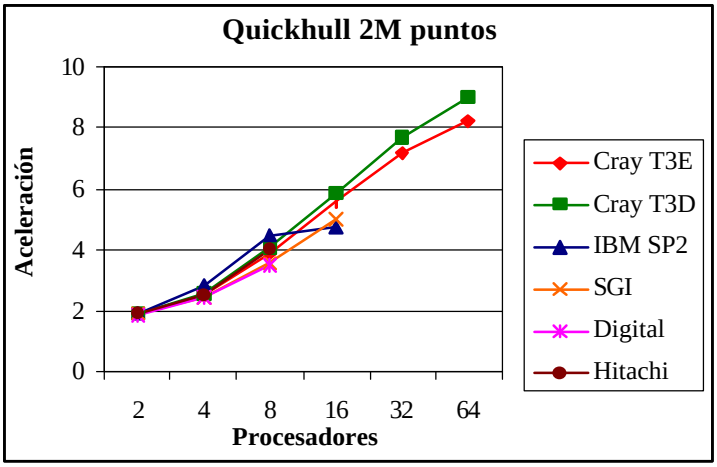

Figura 6.56 Resultados de la Quickhull en diferentes máquinas para 2M puntos

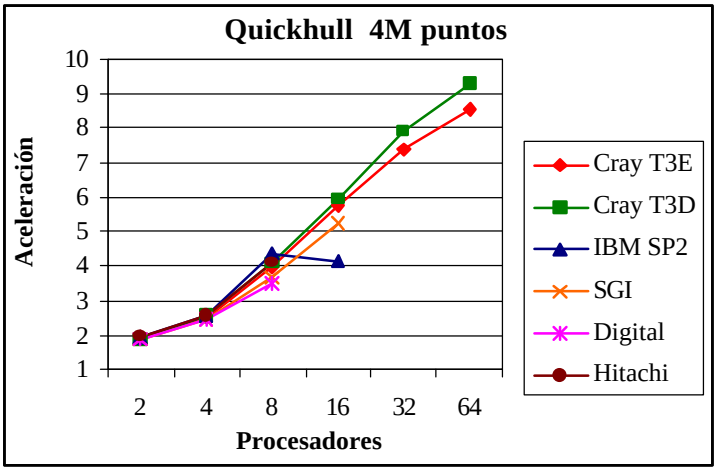

Figura 6.57 Resultados de la Quickhull en diferentes máquinas para 4M puntos

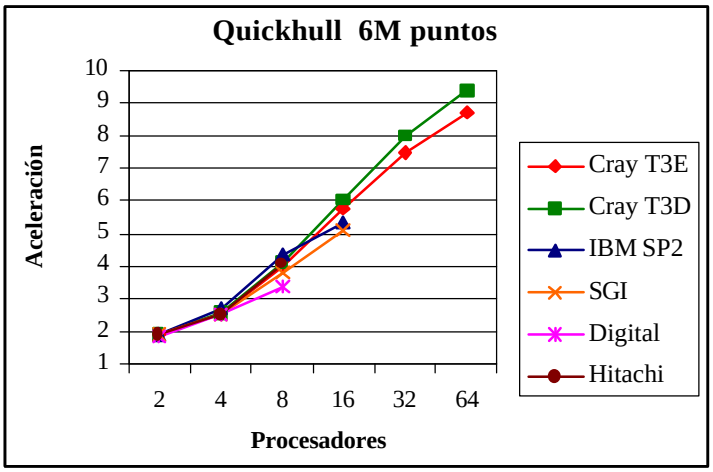

Figura 6.58 Resultados de la Quickhull en diferentes máquinas para 6M puntos

6.5. El Quicksort Distribuido

Los resultados experimentales correspondientes al algoritmo Quicksort Distribuido que se presentó en el Capítulo 4 se exponen en esta sección.

Se muestran los resultados obtenidos para las diferentes plataformas con distintos números de procesadores y variando el tamaño del vector de enteros a ordenar entre 1 y 10M. Para cada algoritmo, cada máquina, cada número de procesadores y cada tamaño de vector a ordenar, se generaron aleatoriamente vectores de números enteros en el rango $[0, 10^8]$. Se realizaron 10 experimentos para cada caso y los valores que presentamos son la media de los resultados obtenidos.

Los tiempos en la columna etiquetada SEC corresponden con el promedio del tiempo secuencial. Las columnas etiquetadas NBAL corresponden al algoritmo sin equilibrado de carga que aparece en la Figura 4.40 del Capítulo 4. Las columnas etiquetadas BSAMP presentan los valores medios obtenidos para el algoritmo en que se hace un primer equilibrado de carga eligiendo el pivote a través de un muestreo de 50 elementos en cada uno de los procesadores utilizados. Los resultados etiquetados BDIM y BWEIG corresponden con los algoritmos con equilibrio de carga en dimensión y por asignación de procesadores que se explicaron en el Capítulo 4 y que aparecen en las Figuras 4.44 y 4.45 de ese Capítulo.

Dado que en este caso se trata de un problema de tipo Privado-Privado, el vector ordenado no queda almacenado en ninguno de los procesadores. Por otra parte, los tamaños de los segmentos de vector que cada procesador acaba ordenando varían de un procesador a otro, por lo cual, el tiempo paralelo de cada procesador es diferente. Las aceleraciones se han calculado comparando el tiempo del algoritmo secuencial con el peor de los tiempos paralelos de los procesadores considerados.

Las gráficas que presentamos muestran el comportamiento de la aceleración de los cuatro algoritmos variando el tamaño del vector a ordenar considerando diferentes números de procesadores (4, 8 y 16 para Cray, IBM y SGI y 2, 4 y 8 para el Digital).

En todas las plataformas observamos que el equilibrado de la carga de trabajo mejora las prestaciones del algoritmo. Sin embargo no existe diferencia sustancial entre las tres técnicas de equilibrado de carga presentadas. Hemos de tener en cuenta que las aceleraciones que se consiguen sin equilibrado de carga son altas para cualquier número de procesadores y por tanto no es fácil conseguir un incremento sustancial de la misma. Como se explicó en el Capítulo 4, los equilibrados en dimensión y por asignación de procesadores presentan limitaciones. El primero produce un equilibrio en el grupo de procesadores local, pudiendo existir desequilibrios entre grupos. Las divisiones enteras realizadas en el equilibrado por asignación pueden también conducir en algunos casos a desequilibrios de la carga, como ya explicamos. La ventaja que presentan estas dos técnicas sobre la técnica consistente en utilizar una muestra para seleccionar el pivote de la partición es su carácter general.

En lo que sigue se presentan los resultados obtenidos para las diferentes máquinas utilizadas en este problema: Cray T3E, Digital Alphaserver, y las configuraciones actuales tanto de la IBM SP2 como de la SGI Origin 2000 (karnak3).

6.5.1. Cray T3E

		TIEMPO					ACELERACIÓN			
PROC	**SIZE**	**SEC**	**NBAL**	**BSAMP**	**BDIM**	**BWEIG**	**NBAL**	**BSAMP**	**BDIM**	**BWEIG**
2	1M	0.8930	0.6199	0.4859	0.4842	0.4855	1.5017	1.8408	1.8466	1.8425
2	2M	1.8949	1.2281	1.0283	1.0204	1.0276	1.5877	1.8450	1.8594	1.8464
2	3M	2.9729	2.0631	1.5952	1.5950	1.5947	1.4627	1.8649	1.8660	1.8654
2	4M	4.0190	2.6825	2.2103	2.1779	2.2095	1.5272	1.8216	1.8513	1.8222
2	5M	5.1172	3.7857	2.7521	2.7143	2.7516	1.3770	1.8618	1.8866	1.8620
2	6M	6.2038	4.2212	3.2822	3.2627	3.2811	1.5224	1.8909	1.9046	1.8916
2	7M	7.4640	4.8867	4.0321	3.9198	4.0309	1.5565	1.8537	1.9062	1.8542
2	8M	8.4673	5.7875	4.5552	4.5527	4.5531	1.5311	1.8602	1.8623	1.8610
2	9M	9.7960	6.2398	5.3094	5.2524	5.3080	1.5929	1.8460	1.8670	1.8465
2	10M	10.8028	7.5591	5.8723	5.8315	5.8701	1.4891	1.8421	1.8545	1.8428
4	1M	0.8932	0.3376	0.2606	0.2613	0.2621	2.7314	3.4361	3.4208	3.4197
4	2M	1.8949	0.6308	0.5498	0.5447	0.5458	3.0177	3.4518	3.4830	3.4753
4	3M	2.9728	1.0060	0.8431	0.8577	0.8421	2.9961	3.5268	3.4704	3.5353
4	4M	4.0187	1.2875	1.1530	1.1448	1.1590	3.1568	3.4886	3.5132	3.4693
4	5M	5.1170	1.8485	1.4667	1.4713	1.4653	2.8330	3.4927	3.4794	3.4946
4	6M	6.2033	2.2528	1.7749	1.7861	1.7822	2.8049	3.4981	3.4764	3.4864
4	7M	7.4633	2.4403	2.1037	2.0770	2.0880	3.0878	3.5579	3.5964	3.5799
4	8M	8.5307	2.9853	2.4821	2.4419	2.4788	2.8990	3.4426	3.4975	3.4466
4	9M	9.7966	3.3651	2.7872	2.7803	2.8219	2.9590	3.5177	3.5274	3.4792
4	10M	10.8030	3.6465	3.0340	3.0423	3.0325	2.9922	3.5649	3.5543	3.5642
8	1M	0.8933	0.1667	0.1401	0.1409	0.1402	5.4258	6.3845	6.3496	6.3747
8	2M	1.8955	0.3439	0.2901	0.2841	0.2851	5.6753	6.5372	6.6902	6.6545
8	3M	2.9731	0.5011	0.4652	0.4559	0.4504	5.9889	6.4054	6.5335	6.6117
8	4M	4.0193	0.6831	0.6098	0.6181	0.6069	5.9533	6.6058	6.5182	6.6272
8	5M	5.1174	0.8903	0.7736	0.7908	0.7773	5.8025	6.6242	6.4742	6.5894
8	6M	6.2039	1.0607	0.9501	0.9523	0.9342	5.8965	6.5393	6.5294	6.6461
8	7M	7.4645	1.3336	1.1009	1.1008	1.0980	5.6888	6.7985	6.7819	6.8034
8	8M	8.4676	1.4382	1.2973	1.2771	1.2817	5.9165	6.5381	6.6369	6.6103
8	9M	9.7964	1.8015	1.4682	1.4692	1.4271	5.4750	6.6810	6.6813	6.8692
8	10M	10.8025	1.8293	1.6981	1.5900	1.6307	5.9494	6.3830	6.8000	6.6320
16	1M	0.8935	0.0814	0.0740	0.0763	0.0753	11.0442	12.0724	11.7233	11.8839
16	2M	1.8952	0.1760	0.1539	0.1579	0.1572	10.9294	12.3210	12.0313	12.0771
16	3M	2.9735	0.2562	0.2413	0.2323	0.2436	11.6697	12.3309	12.8065	12.2212
16	4M	4.0197	0.3621	0.3304	0.3205	0.3224	11.2437	12.1764	12.5436	12.4823
16	5M	5.1175	0.4436	0.4147	0.4108	0.4132	11.5687	12.3533	12.4739	12.3951
16	6M	6.2044	0.5372	0.4980	0.5004	0.4969	11.6607	12.4680	12.4087	12.4946
16	7M	7.4649	0.6385	0.5894	0.5844	0.5789	11.7945	12.6861	12.7818	12.9013
16	8M	8.4683	0.7224	0.6789	0.6714	0.6794	11.8212	12.4958	12.6174	12.4748
16	9M	9.7975	0.8317	0.7599	0.7699	0.7608	11.8815	12.8975	12.7375	12.8904
16	10M	10.8035	0.9507	0.8709	0.9001	0.8620	11.4559	12.4114	12.1086	12.5595

Tabla 6.29 Resultados del Quicksort Distribuido para el Cray T3E

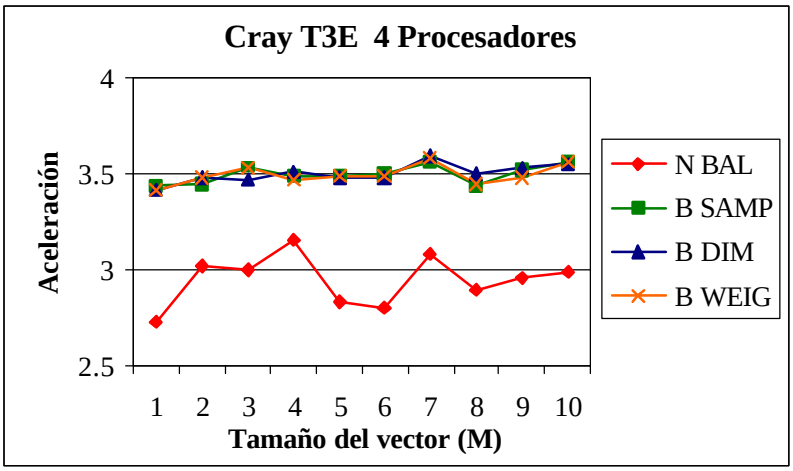

Figura 6.59 Quicksort Distribuido. Cray T3E 4 Procesadores.

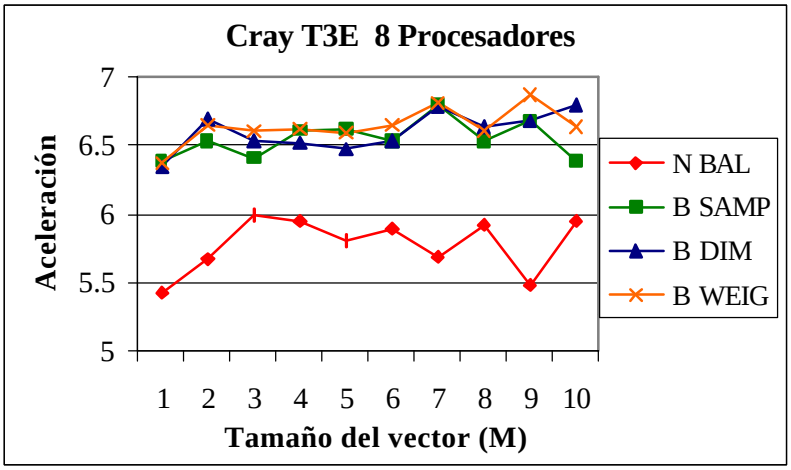

Figura 6.60 Quicksort Distribuido. Cray T3E 8 Procesadores.

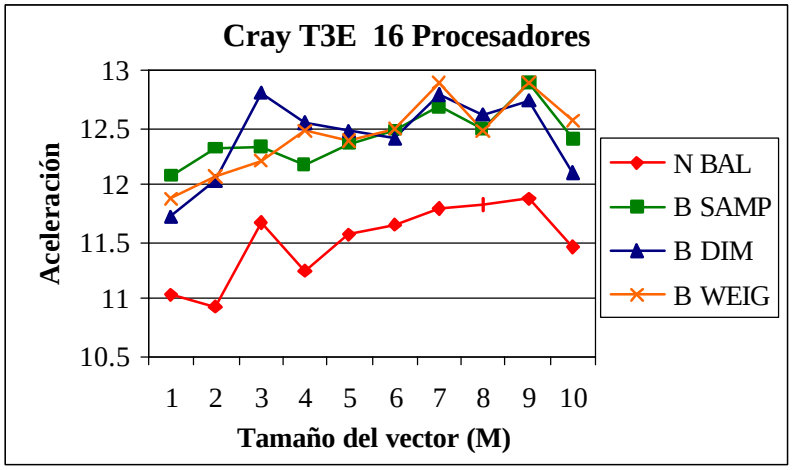

Figura 6.61 Quicksort Distribuido. Cray T3E 16 Procesadores.

6.5.2. Digital Alphaserver 8400

		TIEMPO					ACELERACIÓN			
PROC	**SIZE**	**SEC**	**NBAL**	**BSAMP**	**BDIM**	**BWEIG**	**NBAL**	**BSAMP**	**BDIM**	**BWEIG**
2	1M	0.5998	0.4843	0.3766	0.3671	0.3767	1.2842	1.5947	1.6353	1.5943
2	2M	1.3369	0.9517	0.7728	0.7566	0.7729	1.4449	1.7326	1.7691	1.7323
2	3M	2.1006	1.3508	1.2132	1.1820	1.2133	1.5699	1.7320	1.7776	1.7319
2	4M	2.9170	2.1384	1.6847	1.6425	1.6837	1.3781	1.7344	1.7786	1.7355
2	5M	3.7183	2.7110	2.1640	2.1269	2.1638	1.4014	1.7195	1.7486	1.7197
2	6M	4.6037	3.8418	2.6643	2.5846	2.6621	1.2232	1.7300	1.7832	1.7315
2	7M	5.3901	3.7195	3.1230	3.0648	3.1233	1.4780	1.7268	1.7606	1.7266
2	8M	6.2347	4.4613	3.5788	3.5011	3.5802	1.4326	1.7441	1.7821	1.7434
2	9M	7.1104	5.0660	4.0839	4.0712	4.0845	1.4318	1.7435	1.7474	1.7432
2	10M	7.9779	5.8961	4.6614	4.5074	4.6609	1.3995	1.7139	1.7724	1.7141
4	1M	0.5996	0.2626	0.2317	0.2313	0.2354	2.2983	2.5887	2.5936	2.5499
4	2M	1.3588	0.5777	0.4797	0.4796	0.4763	2.3877	2.8369	2.8340	2.8556
4	3M	2.1493	0.7963	0.7358	0.7408	0.7280	2.7274	2.9264	2.9051	2.9542
4	4M	3.0011	1.2385	0.9941	1.0147	0.9842	2.4536	3.0216	2.9616	3.0505
4	5M	3.8257	1.4718	1.3069	1.2770	1.2729	2.6264	2.9291	2.9978	3.0069
4	6M	4.7584	1.7914	1.5644	1.5947	1.5801	2.6793	3.0434	2.9888	3.0141
4	7M	5.5672	2.0768	1.9097	1.9322	1.9121	2.6956	2.9230	2.8878	2.9199
4	8M	6.4407	2.5186	2.1830	2.1377	2.1804	2.5866	2.9520	3.0137	2.9603
4	9M	7.3544	2.9446	2.4846	2.5240	2.5631	2.5154	2.9649	2.9234	2.8764
4	10M	8.2609	3.2158	2.8547	2.7694	2.8168	2.5856	2.8981	2.9864	2.9337
8	1M	0.6084	0.2059	0.1813	0.1849	0.1786	2.9886	3.3581	3.2972	3.4078
8	2M	1.4358	0.3932	0.3600	0.3666	0.3709	3.6721	3.9992	3.9236	3.8728
8	3M	2.3023	0.5822	0.5385	0.5537	0.5351	3.9683	4.2763	4.1609	4.3116
8	4M	3.2520	0.7867	0.7140	0.7323	0.7232	4.1607	4.5613	4.4435	4.5056
8	5M	4.1595	0.9670	0.8997	0.9457	0.9188	4.3218	4.6245	4.4009	4.5403
8	6M	5.2031	1.2111	1.0896	1.1379	1.0693	4.3290	4.7801	4.5757	4.8790
8	7M	6.0586	1.3824	1.2842	1.3622	1.2994	4.3982	4.7242	4.4508	4.6663
8	8M	7.0681	1.6639	1.4890	1.5301	1.4850	4.2873	4.7541	4.6199	4.7648
8	9M	8.1064	1.8485	1.7084	1.7503	1.7164	4.4079	4.7491	4.6384	4.7265
8	10M	9.0763	2.0812	1.9117	1.9672	1.9146	4.3763	4.7500	4.6144	4.7451

Tabla 6.30 Resultados del Quicksort Distribuido para la Digital Alphaserver 8400

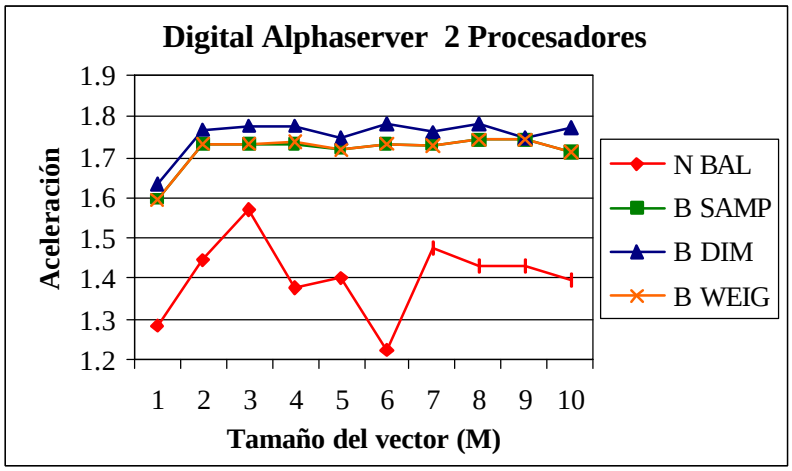

Figura 6.62 Quicksort Distribuido. Digital Alphaserver 2 Procesadores.

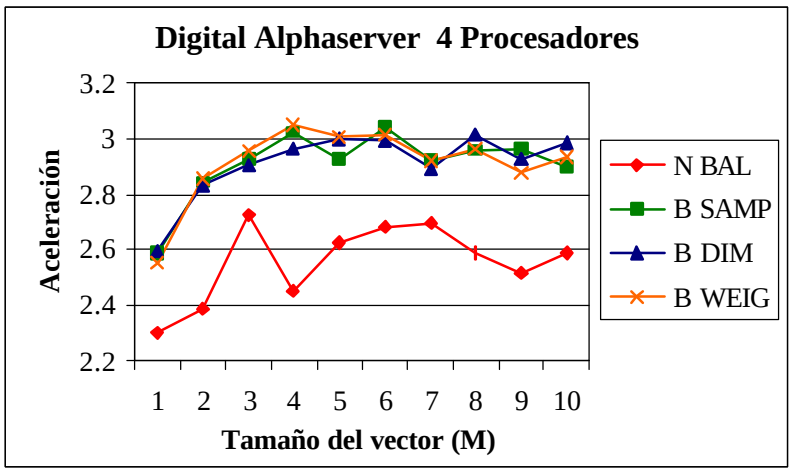

Figura 6.63 Quicksort Distribuido. Digital Alphaserver 4 Procesadores.

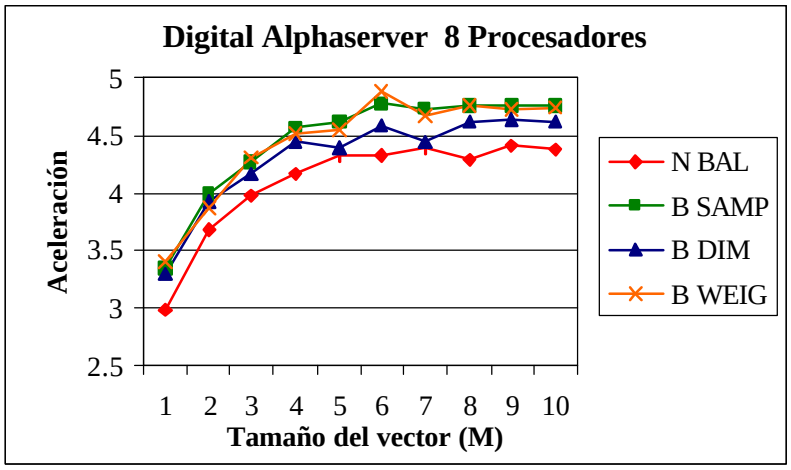

Figura 6.64 Quicksort Distribuido. Digital Alphaserver 8 Procesadores.

6.5.3. IBM SP2

		TIEMPO					ACELERACIÓN			
PROC	**SIZE**	**SEC**	**NBAL**	**BSAMP**	**BDIM**	**BWEIG**	**NBAL**	**BSAMP**	**BDIM**	**BWEIG**
2	1M	0.9624	0.7115	0.5406	0.5399	0.5392	1.4069	1.7828	1.7843	1.7869
2	2M	2.0231	1.3717	1.1019	1.1034	1.1045	1.5140	1.8398	1.8364	1.8353
2	3M	3.1044	1.8915	1.6928	1.6952	1.6880	1.6591	1.8343	1.8318	1.8395
2	4M	4.2325	2.9569	2.3039	2.3009	2.2969	1.4500	1.8385	1.8414	1.8442
2	5M	5.3409	3.6795	2.9049	2.9346	2.9050	1.4797	1.8396	1.8207	1.8394
2	6M	6.4962	5.1941	3.5550	3.5587	3.5593	1.2787	1.8304	1.8278	1.8282
2	7M	7.6252	5.0400	4.1207	4.1361	4.1148	1.5440	1.8511	1.8438	1.8536
2	8M	8.7695	5.9664	4.7575	4.7587	4.8127	1.5084	1.8453	1.8444	1.8276
2	9M	9.9220	6.7975	5.4293	5.4646	5.4263	1.4876	1.8287	1.8163	1.8297
2	10M	11.1238	7.7578	6.0932	6.0841	6.0916	1.4827	1.8271	1.8298	1.8277
4	1M	0.9665	0.3475	0.2946	0.2985	0.2937	2.8194	3.2838	3.2430	3.2950
4	2M	2.0232	0.7600	0.6116	0.6330	0.6129	2.7230	3.3130	3.2126	3.3046
4	3M	3.1151	1.0354	0.9314	0.9431	0.9258	3.0561	3.3510	3.3086	3.3696
4	4M	4.2372	1.5956	1.2772	1.2804	1.2573	2.6860	3.3204	3.3142	3.3705
4	5M	5.3511	1.8558	1.6092	1.5960	1.5726	2.9163	3.3295	3.3543	3.4036
4	6M	6.5162	2.2297	1.9077	1.9498	1.9124	2.9505	3.4173	3.3476	3.4113
4	7M	7.6475	2.5249	2.2738	2.3628	2.2709	3.0518	3.3704	3.2433	3.3760
4	8M	8.8084	3.0243	2.5691	2.6001	2.5857	2.9621	3.4320	3.3885	3.4126
4	9M	10.0657	3.6679	2.9477	3.0579	2.9873	2.7734	3.4179	3.3061	3.3788
4	10M	11.1222	3.8299	3.3346	3.3368	3.3037	2.9270	3.3380	3.3391	3.3701
8	1M	0.9797	0.1863	0.1576	0.1705	0.1585	5.3445	6.2244	5.7599	6.1827
8	2M	2.0444	0.3860	0.3389	0.3427	0.3313	5.3486	6.0825	5.9684	6.1766
8	3M	3.1259	0.5605	0.4986	0.5330	0.4976	5.6277	6.2746	5.8835	6.2893
8	4M	4.2486	0.7764	0.6820	0.7116	0.6789	5.5563	6.2377	5.9751	6.2636
8	5M	5.5101	1.0251	0.8667	0.9289	0.8887	5.4790	6.3564	5.9441	6.2139
8	6M	6.5510	1.2062	1.0428	1.0855	1.0473	5.5078	6.2933	6.0413	6.2596
8	7M	7.6758	1.3580	1.2210	1.3199	1.2335	5.6869	6.2921	5.8231	6.2322
8	8M	8.8110	1.6494	1.4013	1.4538	1.3941	5.4136	6.2941	6.0623	6.3223
8	9M	9.9923	1.7862	1.5845	1.6753	1.5831	5.6225	6.3120	5.9732	6.3174
8	10M	11.1858	1.9987	1.7696	1.8917	1.7608	5.6253	6.3244	5.9149	6.3572
16	1M	0.9829	0.0907	0.0894	0.1882	0.0874	10.8674	11.0034	8.7276	11.2666
16	2M	2.0372	0.2515	0.1773	0.2062	0.1785	10.1046	11.5095	9.8857	11.4153
16	3M	3.1266	0.2943	0.2677	0.3049	0.2676	10.6891	11.6834	10.2684	11.6927
16	4M	4.2536	0.3804	0.3660	0.4091	0.3630	11.2283	11.6277	10.4070	11.7369
16	5M	5.3775	0.4837	0.4562	0.5203	0.4609	11.2080	11.7979	10.3419	11.6833
16	6M	6.5410	0.5948	0.5443	0.6326	0.5516	11.0200	12.0286	10.3458	11.8825
16	7M	7.6851	0.6781	0.6458	0.7394	0.6517	11.3817	11.9062	10.3974	11.8075
16	8M	8.8227	0.8234	0.7540	0.8557	0.7480	10.7789	11.7139	10.3238	11.8034
16	9M	9.9683	0.8962	0.8451	0.9546	0.8404	11.1573	11.8093	10.4471	11.8728
16	10M	11.1849	1.0195	0.9973	1.0579	0.9383	11.1471	11.5710	10.5836	11.9300

Tabla 6.31 Resultados del Quicksort Distribuido para la IBM SP2

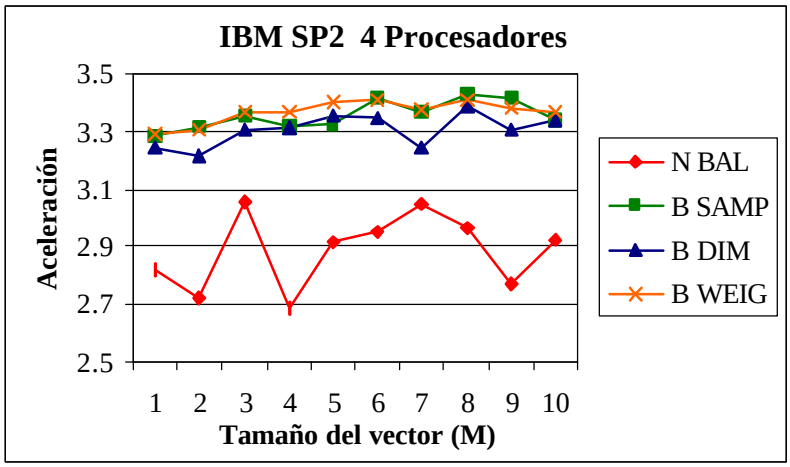

Figura 6.65 Quicksort Distribuido. IBM SP2 4 Procesadores.

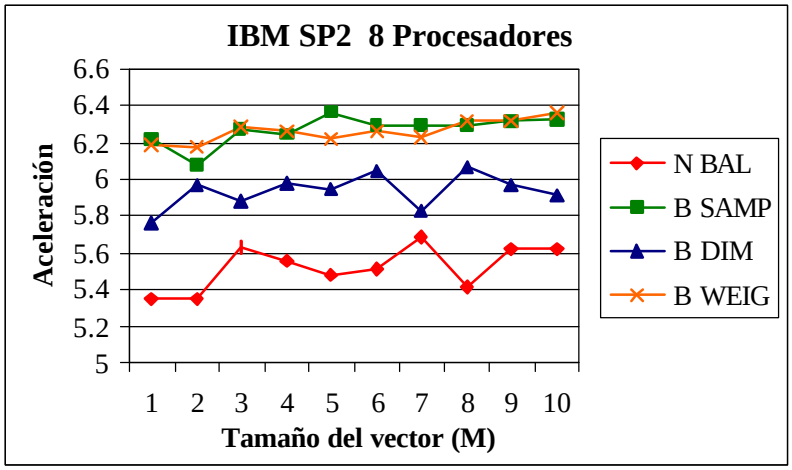

Figura 6.66 Quicksort Distribuido. IBM SP2 8 Procesadores.

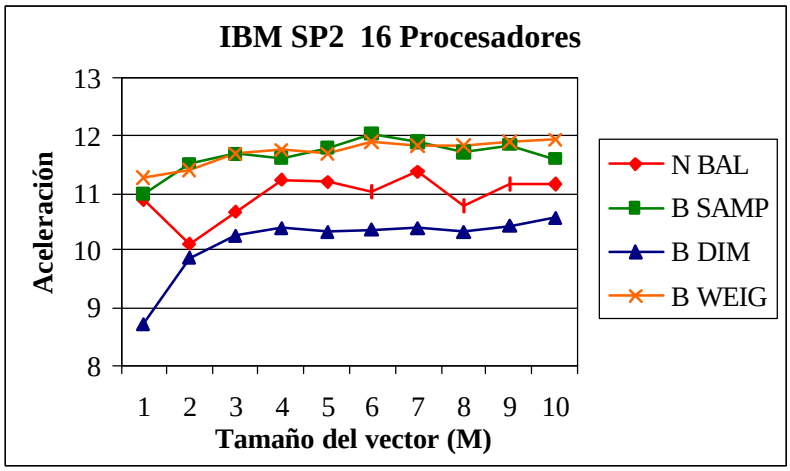

Figura 6.67 Quicksort Distribuido. IBM SP2 16 Procesadores.

6.5.4. Silicon Graphics Origin 2000 (karnak3)

		TIEMPO					ACELERACIÓN			
PROC	**SIZE**	**SEC**	**NBAL**	**BSAMP**	**BDIM**	**BWEIG**	**NBAL**	**BSAMP**	**BDIM**	**BWEIG**
2	1M	0.9836	0.7212	0.5391	0.5459	0.5393	1.4186	1.8269	1.8037	1.8263
2	2M	2.0909	1.4038	1.1166	1.1389	1.1177	1.5340	1.8753	1.8388	1.8734
2	3M	3.2457	1.9501	1.7409	1.7749	1.7417	1.6833	1.8647	1.8295	1.8638
2	4M	4.4668	3.0851	2.3835	2.4454	2.3838	1.4677	1.8760	1.8280	1.8757
2	5M	5.6948	3.8550	3.0376	3.1280	3.0378	1.5072	1.8759	1.8217	1.8758
2	6M	7.0013	5.4975	3.7352	3.7681	3.7357	1.3046	1.8770	1.8609	1.8767
2	7M	8.2598	5.3217	4.3366	4.4577	4.3370	1.5843	1.9053	1.8543	1.9052
2	8M	9.5340	6.3598	5.0246	5.1540	5.0244	1.5418	1.8993	1.8506	1.8994
2	9M	10.8802	7.2682	5.7693	5.9892	5.7702	1.5280	1.8873	1.8176	1.8869
2	10M	12.2161	8.3564	6.5228	6.5736	6.5230	1.5156	1.8744	1.8602	1.8744
4	1M	0.9858	0.3355	0.2921	0.2930	0.2926	2.9644	3.3766	3.3674	3.3739
4	2M	2.0956	0.7590	0.6098	0.6169	0.6102	2.8243	3.4416	3.3990	3.4374
4	3M	3.2509	1.0464	0.9359	0.9581	0.9282	3.1562	3.4789	3.3978	3.5063
4	4M	4.4900	1.6384	1.2918	1.3252	1.2686	2.7785	3.4776	3.3927	3.5398
4	5M	5.7441	1.9271	1.6625	1.6678	1.6227	3.0193	3.4592	3.4461	3.5407
4	6M	7.0799	2.3221	1.9855	2.0686	1.9935	3.0781	3.5679	3.4302	3.5554
4	7M	8.3640	2.6581	2.3859	2.5186	2.3862	3.1717	3.5139	3.3251	3.5155
4	8M	9.6572	3.1955	2.7020	2.7896	2.7178	3.0704	3.5781	3.4632	3.5610
4	9M	11.0391	3.8171	3.0953	3.2809	3.1704	2.9191	3.5707	3.3785	3.4911
4	10M	12.4194	4.0664	3.5502	3.6323	3.5048	3.0787	3.5016	3.4254	3.5468
8	1M	0.9955	0.1912	0.1594	0.1637	0.1579	5.2985	6.2526	6.0858	6.3077
8	2M	2.1154	0.3815	0.3214	0.3349	0.3334	5.6172	6.5842	6.3212	6.3580
8	3M	3.3432	0.5598	0.4987	0.5222	0.4941	6.0397	6.7066	6.4183	6.7720
8	4M	4.6385	0.7843	0.6771	0.7064	0.6783	6.0117	6.8563	6.5755	6.8457
8	5M	5.9170	0.9747	0.8549	0.9142	0.8798	6.1673	6.9245	6.4782	6.7391
8	6M	7.3215	1.2311	1.0551	1.1264	1.0531	6.0369	6.9484	6.5086	6.9599
8	7M	8.6314	1.3897	1.2455	1.3843	1.2513	6.2542	6.9380	6.2441	6.9053
8	8M	9.9476	1.7211	1.4442	1.5482	1.4391	5.8777	6.8951	6.4282	6.9125
8	9M	11.3670	1.8758	1.6430	1.7746	1.6457	6.0997	6.9295	6.4150	6.9129
8	10M	12.7931	2.1086	1.8579	2.0075	1.8549	6.1007	6.8913	6.3741	6.9001
16	1M	0.9885	0.1070	0.0944	0.0966	0.0877	9.9679	10.6686	10.2959	11.3042
16	2M	2.1129	0.2076	0.1819	0.1946	0.1783	10.5688	11.6753	10.8653	11.8568
16	3M	3.3248	0.3119	0.2772	0.2915	0.2708	10.8636	12.0459	11.4192	12.2905
16	4M	4.6272	0.3992	0.3693	0.3945	0.3653	11.7160	12.5503	11.7335	12.6811
16	5M	5.9115	0.5034	0.4626	0.5051	0.4659	11.9080	12.7841	11.7098	12.7083
16	6M	7.3104	0.6067	0.5538	0.6150	0.5550	12.1038	13.2100	11.8934	13.2045
16	7M	8.6111	0.6953	0.6553	0.7221	0.6599	12.4863	13.1494	11.9297	13.0697
16	8M	9.9706	0.8414	0.7622	0.8450	0.7500	11.9533	13.0966	11.8133	13.3024
16	9M	11.4124	0.9168	0.8577	0.9540	0.8543	12.5031	13.3200	11.9693	13.3701
16	10M	12.8319	1.0460	0.9488	1.0719	0.9601	12.4876	13.5310	11.9824	13.3757

Tabla 6.32 Resultados del Quicksort Distribuido para la SGI Origin 2000 (karnak3)

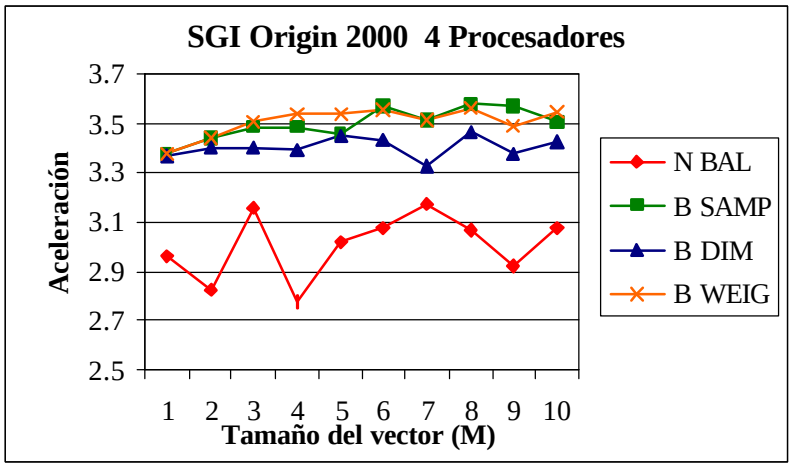

Figura 6.68 Quicksort Distribuido. SGI Origin 2000 4 Procesadores.

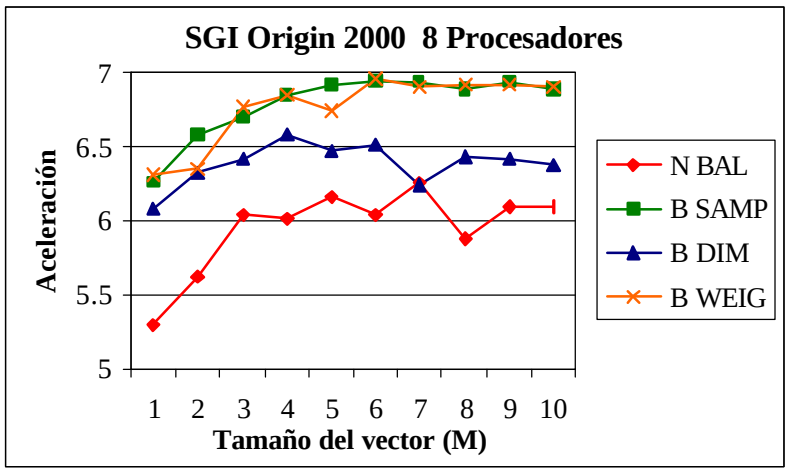

Figura 6.69 Quicksort Distribuido. SGI Origin 2000 8 Procesadores.

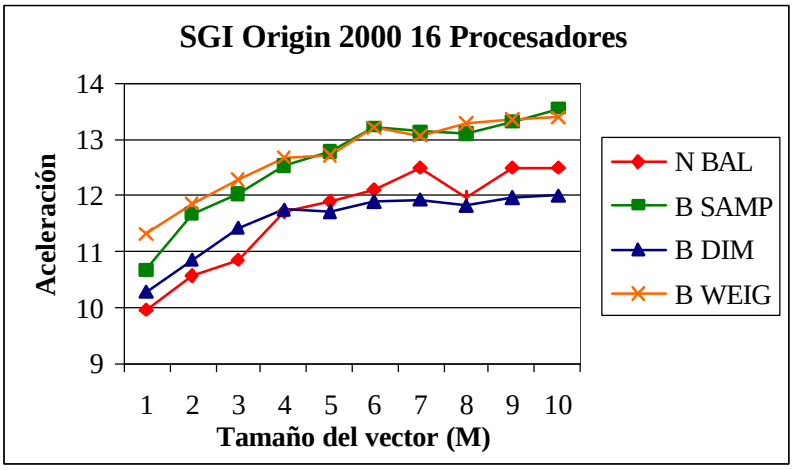

Figura 6.70 Quicksort Distribuido. SGI Origin 2000 16 Procesadores.

6.6. El algoritmo de búsqueda

En este epígrafe presentamos los resultados obtenidos en la ejecución del algoritmo de búsqueda que se presentó en el Capítulo 4 (Figura 4.35). Se presentan los resultados para las diferentes plataformas, con diferente número de procesadores y variando el tamaño del vector de enteros entre 2 y 10M.

Es importante recordar que se trata de un problema de tipo Común-Común. El vector de búsqueda es conocido por todos los procesadores, y el resultado de la misma queda almacenado también en una variable paralela común.

Se diferencian dos tipos de ejecuciones, aquellas en las que la clave buscada sí se encontraba en el vector de entrada (etiquetas FIND SI) y aparecía en éste sin repetir (sólo se encuentra una vez) y aquellas en las que la clave buscada no aparece en el vector de entrada (etiqueta FIND NO). En el caso del algoritmo en que la clave sí se encontraba en el vector, los resultados que se presentan son la media de 10 ejecuciones, en las que la posición en el vector de la clave buscada se generaba de forma aleatoria, mientras que el caso en que la clave no está presente en el vector de búsqueda, los resultados corresponden a una única ejecución. Se presentan los tiempos correspondientes a la ejecución secuencial (etiqueta T SEC), paralela (T PAR) y la aceleración obtenida (ACEL).

Las aceleraciones que aparecen en el caso del algoritmo FIND SI son las medias de las aceleraciones sobre los diez experimentos realizados (no el cociente entre el tiempo secuencial medio y el tiempo paralelo medio).

Las gráficas muestran el comportamiento de la aceleración para cada una de las máquinas variando el tamaño del vector de entrada para un número fijo de procesadores. Se han realizado experimentos para el Cray T3E, Cray T3D, Digital Alphaserver, Hitachi SR2201 y las configuraciones actuales tanto de la IBM SP2 como de la SGI Origin 2000 (karnak3).

En los resultados observamos que siempre el caso FIND NO presenta mejor aceleración que el caso FIND SI, como era de esperar. El tiempo secuencial es generalmente mayor en caso que la clave no se encuentre en el vector de búsqueda, mientras que los tiempos paralelos de ambos casos son iguales (ver las tablas) debido al sincronismo del algoritmo. Un determinado procesador puede finalizar su trabajo antes que el resto debido a que encuentre pronto la clave buscada, pero ha de esperar a la comunicación que se produce en la operación colectiva mediante la cual el resultado de la búsqueda es comunicado al resto de procesadores.

Se observa que para tamaños del vector de búsqueda suficientemente grandes, todas las arquitecturas presentan un comportamiento lineal de la aceleración.

Observamos también que como cabría esperar, la aceleración del caso FIND NO es aproximadamente el doble de la del caso FIND SI. Esto es debido a que en promedio, el algoritmo secuencial recorrerá la mitad del intervalo en caso que la clave sí esté presente en el vector de búsqueda.

6.6.1. Cray T3E

		FIND NO			FIND SI		
PROCS	**TAMAÑO**	**T SEC**	**T PAR**	**ACEL**	**T SEC**	**T PAR**	**ACEL**
2	2M	0.0475	0.0239	1.9866	0.0188	0.0194	0.9689
2	3M	0.0712	0.0357	1.9950	0.0323	0.0291	1.1111
2	4M	0.0950	0.0477	1.9894	0.0304	0.0387	0.7853
2	5M	0.1188	0.0594	1.9979	0.0497	0.0484	1.0267
2	6M	0.1428	0.0713	2.0025	0.0497	0.0581	0.8557
2	7M	0.1662	0.0834	1.9931	0.0729	0.0677	1.0764
2	8M	0.1900	0.0950	1.9997	0.0614	0.0775	0.7924
2	9M	0.2139	0.1072	1.9958	0.0905	0.0871	1.0388
2	10M	0.2375	0.1190	1.9956	0.1079	0.0969	1.1130
4	2M	0.0475	0.0121	3.9402	0.0188	0.0098	1.9146
4	3M	0.0712	0.0181	3.9470	0.0323	0.0147	2.1967
4	4M	0.0950	0.0240	3.9638	0.0304	0.0195	1.5501
4	5M	0.1188	0.0298	3.9890	0.0497	0.0244	2.0390
4	6M	0.1427	0.0360	3.9613	0.0497	0.0293	1.7001
4	7M	0.1665	0.0416	3.9988	0.0730	0.0341	2.1379
4	8M	0.1904	0.0476	4.0036	0.0614	0.0389	1.5785
4	9M	0.2138	0.0535	3.9958	0.0905	0.0438	2.0654
4	10M	0.2377	0.0602	3.9501	0.1078	0.0486	2.2175
8	2M	0.0475	0.0061	7.8339	0.0189	0.0050	3.7840
8	3M	0.0712	0.0090	7.8949	0.0324	0.0075	4.2353
8	4M	0.0950	0.0123	7.7283	0.0305	0.0100	3.0712
8	5M	0.1188	0.0149	7.9587	0.0497	0.0124	4.0083
8	6M	0.1427	0.0183	7.8098	0.0498	0.0149	3.3473
8	7M	0.1665	0.0214	7.7972	0.0730	0.0174	4.2137
8	8M	0.1902	0.0238	7.9855	0.0614	0.0197	3.1141
8	9M	0.2137	0.0269	7.9452	0.0904	0.0221	4.0880
8	10M	0.2377	0.0300	7.9128	0.1079	0.0245	4.3917
16	2M	0.0477	0.0036	13.2953	0.0188	0.0028	6.8578
16	3M	0.0712	0.0050	14.3076	0.0323	0.0039	8.1630
16	4M	0.0950	0.0066	14.4650	0.0304	0.0051	5.9713
16	5M	0.1188	0.0080	14.8461	0.0497	0.0065	7.7488
16	6M	0.1427	0.0095	15.0157	0.0497	0.0076	6.5780
16	7M	0.1664	0.0106	15.6733	0.0730	0.0088	8.2334
16	8M	0.1902	0.0122	15.6213	0.0614	0.0100	6.1179
16	9M	0.2140	0.0137	15.6381	0.0904	0.0112	8.0187
16	10M	0.2377	0.0151	15.7296	0.1079	0.0125	8.6323
32	2M	0.0475	0.0018	26.5062	0.0188	0.0017	10.6840
32	3M	0.0712	0.0028	25.7859	0.0323	0.0023	14.3821
32	4M	0.0950	0.0034	27.8622	0.0304	0.0030	10.3916
32	5M	0.1188	0.0041	28.8101	0.0497	0.0036	14.0415
32	6M	0.1425	0.0049	28.8101	0.0498	0.0043	11.5814
32	7M	0.1663	0.0057	29.3073	0.0730	0.0047	15.3861
32	8M	0.1902	0.0062	30.5010	0.0615	0.0054	11.4570
32	9M	0.2138	0.0072	29.5562	0.0905	0.0059	15.3238
32	10M	0.2377	0.0080	29.6638	0.1079	0.0065	16.7527

Tabla 6.33 Resultados del algoritmo de búsqueda para el Cray T3E

		FIND NO			FIND SI		
PROCS	TAMAÑO	T SEC	T PAR	ACEL	T SEC	T PAR	ACEL
64	2M	0.0387	0.0010	40.3061	0.0188	0.0012	17.8272
64	3M	0.0581	0.0015	39.7925	0.0323	0.0015	22.3420
64	4M	0.0774	0.0017	46.1198	0.0304	0.0018	17.0687
64	5M	0.0967	0.0023	41.9586	0.0498	0.0024	20.5862
64	6M	0.1162	0.0022	53.4486	0.0497	0.0025	19.2576
64	7M	0.1357	0.0023	59.7086	0.0729	0.0029	26.2820
64	8M	0.1551	0.0027	57.0446	0.0614	0.0031	20.0569
64	9M	0.1744	0.0029	59.3098	0.0905	0.0034	27.1552
64	10M	0.1936	0.0037	52.5505	0.1079	0.0036	31.0599

Tabla 6.34 Resultados del algoritmo de búsqueda para el Cray T3E (continuación)

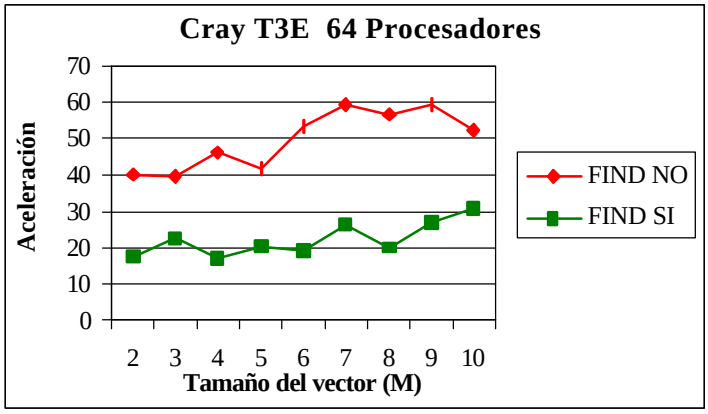

Figura 6.71 Algoritmo de Búsqueda. Resultados para el Cray T3E con 64 procesadores

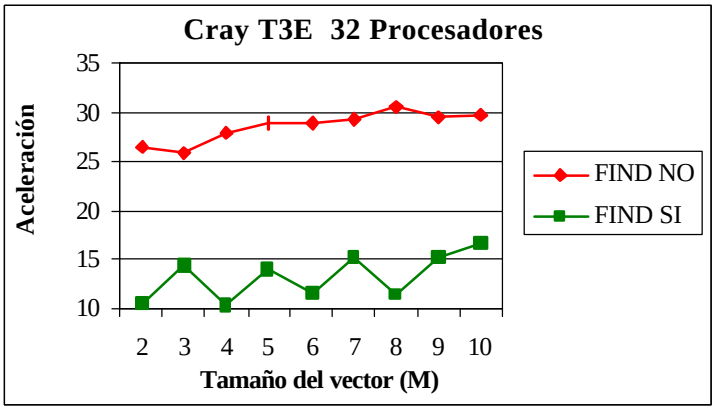

Figura 6.72 Algoritmo de Búsqueda. Resultados para el Cray T3E con 32 procesadores

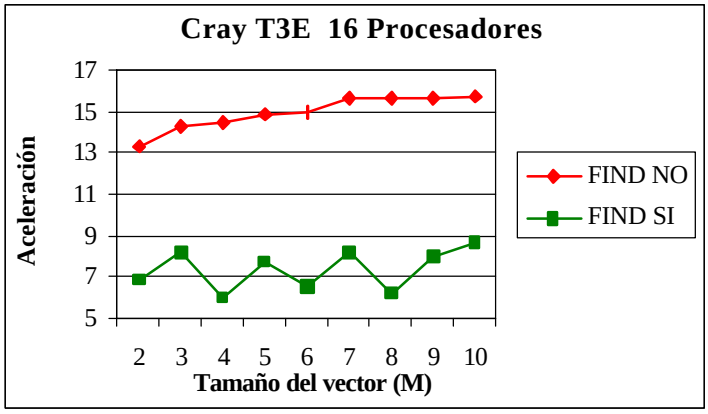

Figura 6.73 Algoritmo de Búsqueda. Resultados para el Cray T3E con 16 procesadores

6.6.2. Cray T3D

		FIND NO			FIND SI		
PROCS	**TAMAÑO**	**T SEC**	**T PAR**	**ACEL**	**T SEC**	**T PAR**	**ACEL**
2	2M	0.1811	0.0906	1.9994	0.0878	0.0906	0.9692
2	3M	0.2716	0.1360	1.9961	0.1512	0.1359	1.1129
2	4M	0.3623	0.1812	1.9995	0.1422	0.1811	0.7850
2	5M	0.4526	0.2264	1.9989	0.2327	0.2264	1.0277
2	6M	0.5432	0.2717	1.9996	0.2328	0.2716	0.8569
2	7M	0.6338	0.3169	2.0002	0.3414	0.3170	1.0769
2	8M	0.7244	0.3624	1.9989	0.2871	0.3622	0.7924
2	9M	0.8149	0.4076	1.9990	0.4229	0.4075	1.0376
2	10M	0.9054	0.4528	1.9997	0.5043	0.4528	1.1139
4	2M	0.1810	0.0454	3.9843	0.0878	0.0454	1.9342
4	3M	0.2718	0.0682	3.9864	0.1512	0.0681	2.2216
4	4M	0.3621	0.0906	3.9947	0.1422	0.0906	1.5674
4	5M	0.4526	0.1135	3.9887	0.2327	0.1133	2.0537
4	6M	0.5432	0.1360	3.9942	0.2327	0.1359	1.7113
4	7M	0.6339	0.1586	3.9959	0.3414	0.1586	2.1525
4	8M	0.7244	0.1812	3.9986	0.2871	0.1812	1.5840
4	9M	0.8147	0.2039	3.9962	0.4229	0.2039	2.0742
4	10M	0.9052	0.2265	3.9969	0.5044	0.2265	2.2266
8	2M	0.1810	0.0228	7.9251	0.0878	0.0228	3.8482
8	3M	0.2716	0.0342	7.9406	0.1512	0.0342	4.4270
8	4M	0.3621	0.0454	7.9695	0.1422	0.0455	3.1262
8	5M	0.4528	0.0568	7.9736	0.2328	0.0568	4.0993
8	6M	0.5431	0.0681	7.9775	0.2328	0.0681	3.4177
8	7M	0.6339	0.0794	7.9792	0.3413	0.0795	4.2973
8	8M	0.7243	0.0908	7.9816	0.2870	0.0908	3.1630
8	9M	0.8148	0.1021	7.9836	0.4229	0.1021	4.1425
8	10M	0.9055	0.1133	7.9908	0.5044	0.1134	4.4477
16	2M	0.1810	0.0116	15.5898	0.0878	0.0116	7.5969
16	3M	0.2717	0.0173	15.7396	0.1512	0.0173	8.7657
16	4M	0.3621	0.0229	15.8272	0.1422	0.0229	6.2020
16	5M	0.4526	0.0287	15.7516	0.2327	0.0286	8.1458
16	6M	0.5433	0.0343	15.8282	0.2327	0.0342	6.8054
16	7M	0.6338	0.0399	15.8956	0.3413	0.0399	8.5608
16	8M	0.7244	0.0456	15.8949	0.2871	0.0455	6.3054
16	9M	0.8150	0.0512	15.9151	0.4229	0.0512	8.2601
16	10M	0.9054	0.0568	15.9460	0.5044	0.0569	8.8707
32	2M	0.1813	0.0059	30.5816	0.0879	0.0060	14.7293
32	3M	0.2716	0.0089	30.5124	0.1512	0.0088	17.2037
32	4M	0.3621	0.0116	31.2097	0.1422	0.0116	12.2267
32	5M	0.4526	0.0145	31.1135	0.2327	0.0144	16.1103
32	6M	0.5433	0.0173	31.4190	0.2328	0.0173	13.4675
32	7M	0.6339	0.0202	31.4277	0.3414	0.0201	16.9513
32	8M	0.7242	0.0230	31.5399	0.2871	0.0229	12.5150
32	9M	0.8149	0.0258	31.5804	0.4229	0.0258	16.4088
32	10M	0.9055	0.0286	31.6972	0.5044	0.0286	17.6195

Tabla 6.35 Resultados del algoritmo de búsqueda para el Cray T3D

		FIND NO			FIND SI		
PROCS	TAMAÑO	T SEC	T PAR	ACEL	T SEC	T PAR	ACEL
64	2M	0.1810	0.0031	57.6299	0.0878	0.0032	27.4118
64	3M	0.2716	0.0046	58.5076	0.1512	0.0046	32.7974
64	4M	0.3624	0.0060	60.0465	0.1422	0.0060	23.6250
64	5M	0.4528	0.0074	61.0896	0.2327	0.0075	31.2004
64	6M	0.5432	0.0089	60.6961	0.2327	0.0089	26.1754
64	7M	0.6338	0.0103	61.7017	0.3413	0.0103	33.0907
64	8M	0.7243	0.0118	61.2005	0.2870	0.0117	24.5486
64	9M	0.8149	0.0133	61.3293	0.4229	0.0131	32.2803
64	10M	0.9052	0.0146	62.0396	0.5044	0.0146	34.6230

Tabla 6.36 Resultados del algoritmo de búsqueda para el Cray T3D (continuación)

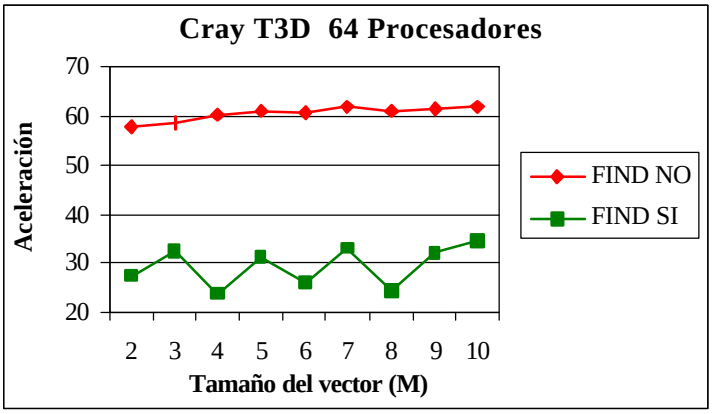

Figura 6.74 Algoritmo de Búsqueda. Resultados para el Cray T3D con 64 procesadores

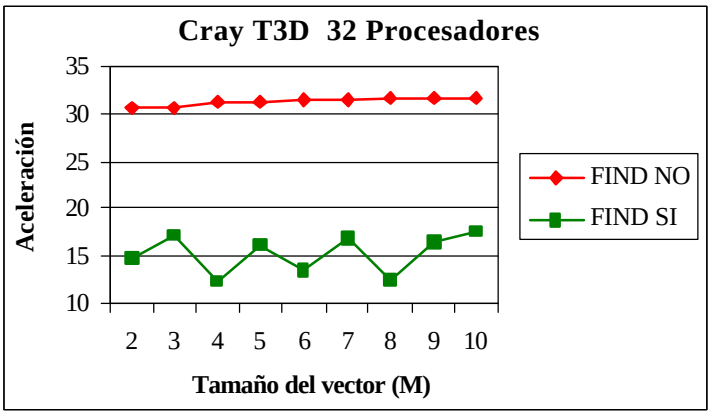

Figura 6.75 Algoritmo de Búsqueda. Resultados para el Cray T3D con 32 procesadores

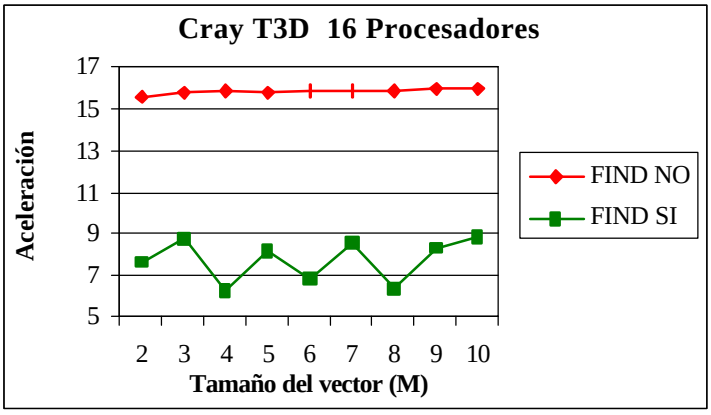

Figura 6.76 Algoritmo de Búsqueda. Resultados para el Cray T3D con 16 procesadores

6.6.3. Digital Alphaserver 8400

		FIND NO			FIND SI		
PROCS	**TAMAÑO**	**T SEC**	**T PAR**	**ACEL**	**T SEC**	**T PAR**	**ACEL**
2	2M	0.0439	0.0293	1.4978	0.0207	0.0278	0.6977
2	3M	0.0820	0.0526	1.5573	0.0396	0.0460	0.8489
2	4M	0.1201	0.0604	1.9864	0.0602	0.0674	0.9237
2	5M	0.1494	0.0751	1.9891	0.0795	0.0794	1.0247
2	6M	0.1796	0.0898	1.9994	0.0737	0.0949	0.7888
2	7M	0.2109	0.1045	2.0181	0.0991	0.1114	0.9248
2	8M	0.2402	0.1200	2.0013	0.1112	0.1248	0.9020
2	9M	0.2705	0.1347	2.0084	0.1299	0.1383	0.9453
2	10M	0.2997	0.1503	1.9946	0.1731	0.1550	1.1173
4	2M	0.0439	0.0116	3.7882	0.0277	0.0176	1.5042
4	3M	0.0790	0.0302	2.6212	0.0391	0.0330	1.1340
4	4M	0.1044	0.0409	2.5537	0.0502	0.0474	1.0487
4	5M	0.1367	0.0536	2.5495	0.0712	0.0536	1.3771
4	6M	0.1718	0.0566	3.0333	0.0683	0.0573	1.2378
4	7M	0.2050	0.0604	3.3928	0.0979	0.0610	1.6116
4	8M	0.2392	0.0643	3.7177	0.1117	0.0674	1.6585
4	9M	0.2695	0.0673	4.0064	0.1311	0.0735	1.7653
4	10M	0.3134	0.0790	3.9683	0.1744	0.0811	2.1322
8	2M	0.0459	0.0098	4.6962	0.0492	0.0162	3.4228
8	3M	0.0878	0.0154	5.6996	0.0635	0.0271	2.3461
8	4M	0.1181	0.0223	5.2898	0.0739	0.0373	1.9532
8	5M	0.1484	0.0282	5.2684	0.0880	0.0495	1.7888
8	6M	0.1767	0.0340	5.1897	0.0744	0.0587	1.3225
8	7M	0.2060	0.0399	5.1644	0.0989	0.0605	1.6295
8	8M	0.2344	0.0457	5.1236	0.1032	0.0735	1.5572
8	9M	0.2695	0.0518	5.2062	0.1247	0.0695	1.7775
8	10M	0.3085	0.0546	5.6555	0.1711	0.0753	2.2612

Tabla 6.37 Resultados del algoritmo de búsqueda para la Digital Alphaserver 8400

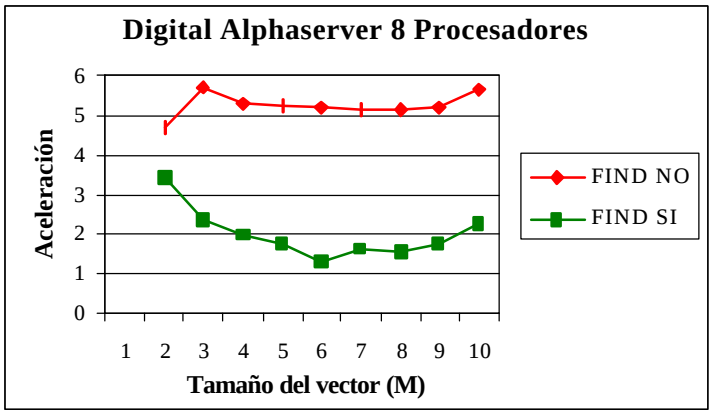

Figura 6.77 Algoritmo de Búsqueda. Resultados para el Digital Alphaserver con 8 procesadores

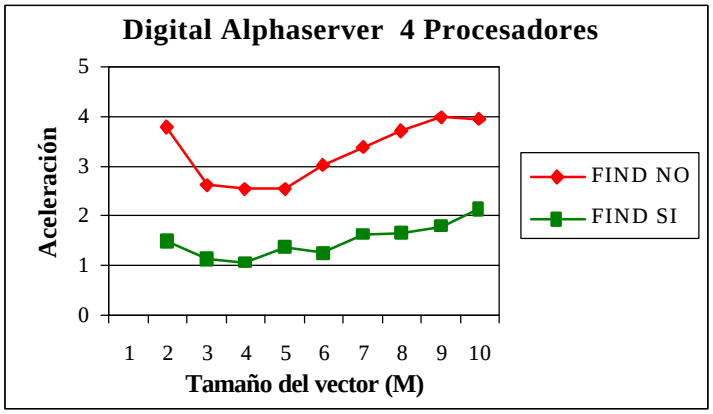

Figura 6.78 Algoritmo de Búsqueda. Resultados para el Digital Alphaserver con 4 procesadores

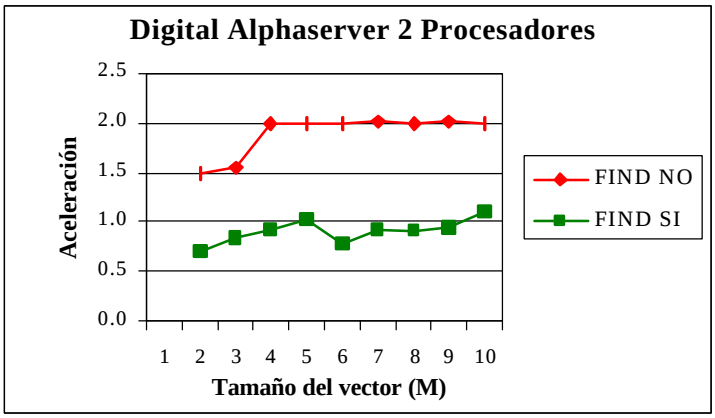

Figura 6.79 Algoritmo de Búsqueda. Resultados para el Digital Alphaserver con 2 procesadores

6.6.4. IBM SP2

		FIND NO			FIND SI		
PROCS	**TAMAÑO**	**T SEC**	**T PAR**	**ACEL**	**T SEC**	**T PAR**	**ACEL**
2	2M	0.1859	0.0920	2.0208	0.1009	0.0901	1.0954
2	3M	0.2769	0.1387	1.9956	0.1467	0.1373	1.0626
2	4M	0.3699	0.1845	2.0051	0.1922	0.1832	1.0443
2	5M	0.4757	0.2305	2.0639	0.2481	0.2298	1.0770
2	6M	0.5521	0.2782	1.9846	0.2293	0.2717	0.8434
2	7M	0.6441	0.3249	1.9821	0.3128	0.3212	0.9651
2	8M	0.7377	0.3878	1.9024	0.3398	0.3676	0.9239
2	9M	0.8363	0.4159	2.0109	0.3964	0.4120	0.9573
2	10M	0.9225	0.4608	2.0021	0.5248	0.4607	1.1388
4	2M	0.1871	0.0557	3.3621	0.1002	0.0502	2.0993
4	3M	0.2773	0.0690	4.0187	0.1464	0.0710	2.0482
4	4M	0.3696	0.0955	3.8712	0.1931	0.0933	2.0622
4	5M	0.4624	0.1168	3.9589	0.2491	0.1186	2.1050
4	6M	0.5539	0.1391	3.9826	0.2292	0.1386	1.6520
4	7M	0.6463	0.1669	3.8716	0.3129	0.1706	1.8532
4	8M	0.7387	0.1946	3.7971	0.3395	0.1916	1.7831
4	9M	0.8308	0.2103	3.9512	0.3960	0.2153	1.8277
4	10M	0.9234	0.2460	3.7536	0.5241	0.2398	2.1803
8	2M	0.1847	0.0231	7.9932	0.1000	0.0299	3.4758
8	3M	0.2774	0.0423	6.5512	0.1461	0.0409	3.5310
8	4M	0.3689	0.0488	7.5545	0.1925	0.0536	3.6161
8	5M	0.4620	0.0584	7.9166	0.2488	0.0695	3.7063
8	6M	0.5532	0.0693	7.9772	0.2296	0.0813	2.8510
8	7M	0.6480	0.0974	6.6532	0.3133	0.0972	3.4712
8	8M	0.7396	0.0982	7.5281	0.3399	0.1091	3.1996
8	9M	0.8305	0.1228	6.7607	0.3976	0.1179	3.4110
8	10M	0.9226	0.1338	6.8976	0.5252	0.1330	3.9222
16	2M	0.1845	0.0154	12.0116	0.1004	0.0194	6.4578
16	3M	0.2787	0.0213	13.0961	0.1464	0.0242	6.0719
16	4M	0.3689	0.0294	12.5290	0.1927	0.0314	6.0433
16	5M	0.4619	0.0367	12.5860	0.2476	0.0408	6.4145
16	6M	0.5542	0.0430	12.8818	0.2291	0.0477	4.8902
16	7M	0.6471	0.0490	13.2142	0.3127	0.0560	5.8250
16	8M	0.7387	0.0636	11.6098	0.3438	0.0598	6.1305
16	9M	0.8295	0.0652	12.7248	0.3951	0.0675	6.4688
16	10M	0.9293	0.0780	11.9126	0.5255	0.0685	8.0526

Tabla 6.38 Resultados del algoritmo de búsqueda para la IBM SP2

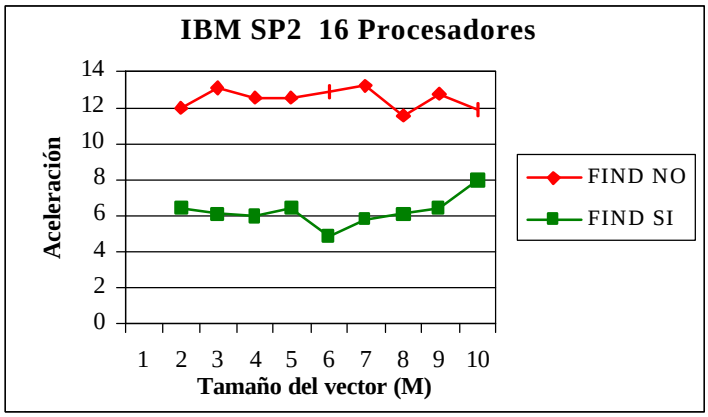

Figura 6.80 Algoritmo de Búsqueda. Resultados para la IBM SP2 con 16 procesadores

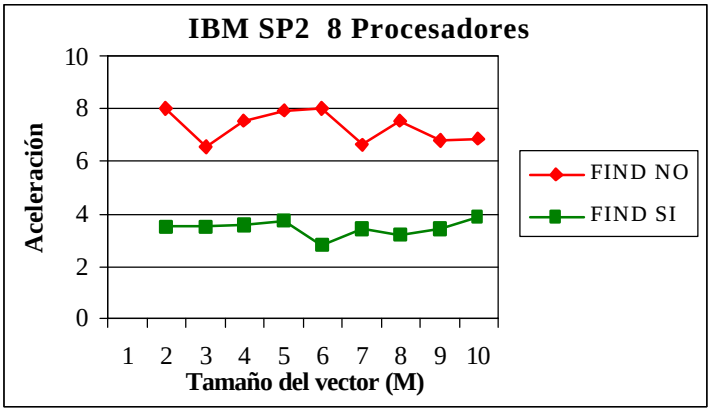

Figura 6.81 Algoritmo de Búsqueda. Resultados para la IBM SP2 con 8 procesadores

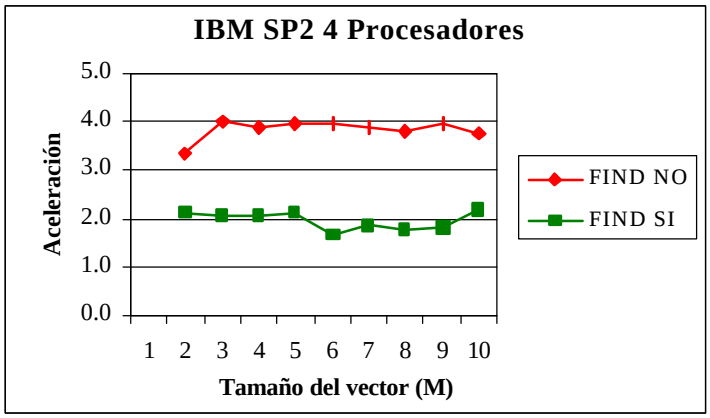

Figura 6.82 Algoritmo de Búsqueda. Resultados para la IBM SP2 con 4 procesadores

6.6.5. Silicon Graphics Origin 2000 (karnak3)

		FIND NO			FIND SI		
PROCS	**TAMAÑO**	**T SEC**	**T PAR**	**ACEL**	**T SEC**	**T PAR**	**ACEL**
2	2M	0.0546	0.0278	1.9615	0.0245	0.0256	0.9643
2	3M	0.0822	0.0401	2.0492	0.0392	0.0411	0.9715
2	4M	0.1100	0.0545	2.0167	0.0523	0.0546	0.9680
2	5M	0.1386	0.0686	2.0219	0.0695	0.0668	1.0504
2	6M	0.1666	0.0831	2.0043	0.0645	0.0807	0.8088
2	7M	0.1944	0.0973	1.9975	0.0903	0.0936	0.9549
2	8M	0.2222	0.1109	2.0035	0.0990	0.1061	0.9235
2	9M	0.2499	0.1246	2.0063	0.1160	0.1200	0.9574
2	10M	0.2769	0.1384	2.0000	0.1541	0.1326	1.1552
4	2M	0.0498	0.0171	2.9101	0.0230	0.0176	1.2810
4	3M	0.0769	0.0255	3.0197	0.0376	0.0291	1.2315
4	4M	0.1062	0.0321	3.3103	0.0530	0.0348	1.5654
4	5M	0.1372	0.0395	3.4750	0.0696	0.0451	1.5772
4	6M	0.1835	0.0617	2.9726	0.0649	0.0473	1.3829
4	7M	0.2266	0.0678	3.3441	0.0896	0.0566	1.5866
4	8M	0.2740	0.0919	2.9814	0.0998	0.0607	1.6601
4	9M	0.3174	0.0957	3.3180	0.1144	0.0718	1.6085
4	10M	0.3620	0.1135	3.1883	0.1549	0.0770	2.0463
8	2M	0.0682	0.0247	2.7612	0.0248	0.0137	1.8054
8	3M	0.0799	0.0096	8.3696	0.0376	0.0204	1.7916
8	4M	0.1061	0.0208	5.0975	0.0499	0.0240	2.0535
8	5M	0.1331	0.0186	7.1640	0.0664	0.0303	2.2324
8	6M	0.1601	0.0225	7.1026	0.0621	0.0339	1.8475
8	7M	0.1880	0.0273	6.8973	0.0889	0.0339	2.6596
8	8M	0.2176	0.0305	7.1362	0.1002	0.0325	3.1260
8	9M	0.2455	0.0326	7.5340	0.1164	0.0371	3.1815
8	10M	0.2728	0.0364	7.5021	0.1541	0.0450	3.4233
16	2M	0.0552	0.0235	2.3469	0.0280	0.0057	4.8167
16	3M	0.0818	0.0168	4.8622	0.0426	0.0085	5.1305
16	4M	0.1073	0.0195	5.5056	0.0567	0.0111	5.1469
16	5M	0.1341	0.0144	9.3041	0.0730	0.0195	4.0846
16	6M	0.1608	0.0158	10.1976	0.0668	0.0255	2.7987
16	7M	0.1878	0.0284	6.6116	0.1022	0.0291	3.2791
16	8M	0.2147	0.0234	9.1875	0.1117	0.0319	3.7471
16	9M	0.2427	0.0412	5.8857	0.1424	0.0335	4.6077
16	10M	0.2710	0.0567	4.7747	0.2044	0.0415	5.1205

Tabla 6.39 Resultados del algoritmo de búsqueda para la SGI Origin 2000 (karnak3)

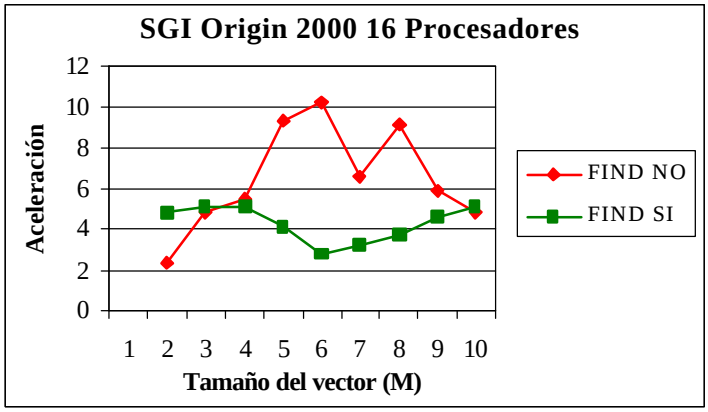

Figura 6.83 Algoritmo de Búsqueda. Resultados para la SGI Origin 2000 con 16 procesadores

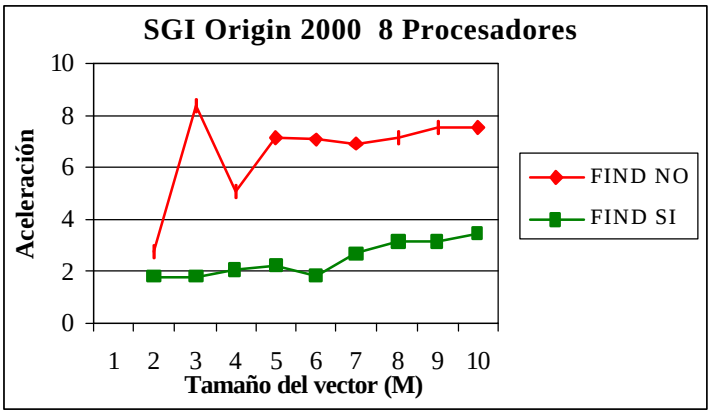

Figura 6.84 Algoritmo de Búsqueda. Resultados para la SGI Origin 2000 con 8 procesadores

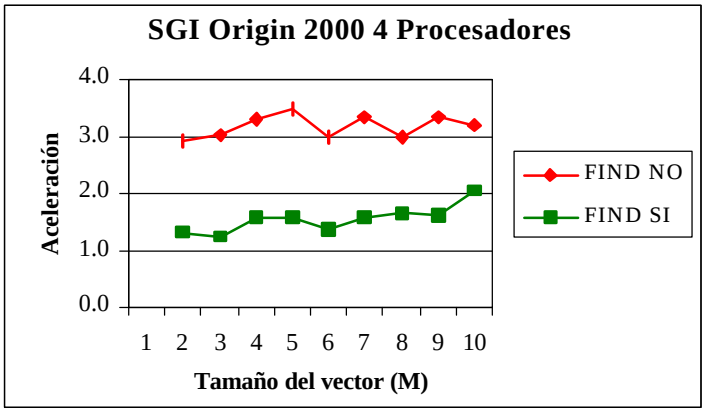

Figura 6.85 Algoritmo de Búsqueda. Resultados para la SGI Origin 2000 con 4 procesadores

6.6.6. Hitachi SR2201

		FIND NO			FIND SI		
PROCS	**TAMAÑO**	**T SEC**	**T PAR**	**ACEL**	**T SEC**	**T PAR**	**ACEL**
2	2M	0.0950	0.0478	1.9855	0.0493	0.0457	1.0803
2	3M	0.1421	0.0710	2.0006	0.0720	0.0685	1.0509
2	4M	0.1894	0.0953	1.9879	0.0949	0.0914	1.0379
2	5M	0.2364	0.1186	1.9926	0.1243	0.1158	1.0708
2	6M	0.2838	0.1424	1.9927	0.1163	0.1397	0.8317
2	7M	0.3315	0.1663	1.9926	0.1587	0.1639	0.9650
2	8M	0.3777	0.1911	1.9768	0.1735	0.1874	0.9245
2	9M	0.4246	0.2132	1.9917	0.2022	0.2114	0.9546
2	10M	0.4720	0.2384	1.9799	0.2684	0.2356	1.1371
4	2M	0.0945	0.0244	3.8687	0.0492	0.0231	2.1454
4	3M	0.1425	0.0373	3.8164	0.0720	0.0346	2.0828
4	4M	0.1902	0.0482	3.9466	0.0949	0.0459	2.0629
4	5M	0.2367	0.0614	3.8544	0.1245	0.0584	2.1311
4	6M	0.2843	0.0736	3.8629	0.1160	0.0711	1.6319
4	7M	0.3314	0.0855	3.8773	0.1590	0.0837	1.8878
4	8M	0.3799	0.0965	3.9349	0.1740	0.0955	1.8221
4	9M	0.4265	0.1072	3.9771	0.2025	0.1073	1.8834
4	10M	0.4747	0.1206	3.9356	0.2693	0.1195	2.2576
8	2M	0.0981	0.0135	7.2769	0.0495	0.0116	4.2396
8	3M	0.1440	0.0189	7.6330	0.0735	0.0173	4.2229
8	4M	0.1925	0.0242	7.9662	0.0958	0.0228	4.1532
8	5M	0.2402	0.0299	8.0209	0.1265	0.0299	4.2241
8	6M	0.2872	0.0361	7.9572	0.1182	0.0359	3.2949
8	7M	0.3374	0.0415	8.1222	0.1617	0.0427	3.8106
8	8M	0.3865	0.0484	7.9873	0.1772	0.0482	3.6563
8	9M	0.4334	0.0553	7.8446	0.2112	0.0534	3.9405
8	10M	0.4819	0.0599	8.0500	0.2737	0.0606	4.5084

Tabla 6.40 Resultados del algoritmo de búsqueda para la Hitachi SR2201

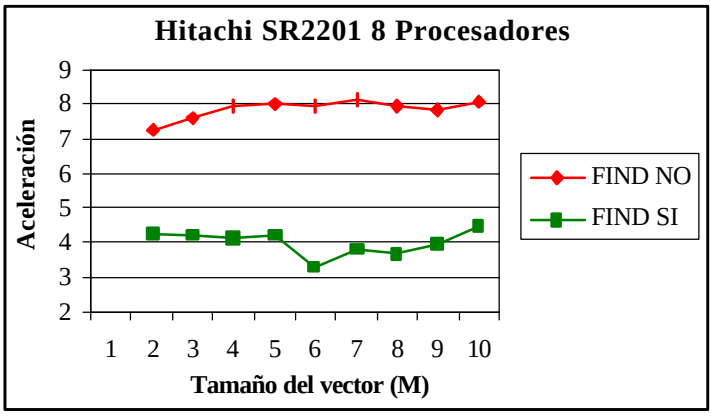

Figura 6.86 Algoritmo de Búsqueda. Resultados para la Hitachi SR2201 con 8 procesadores

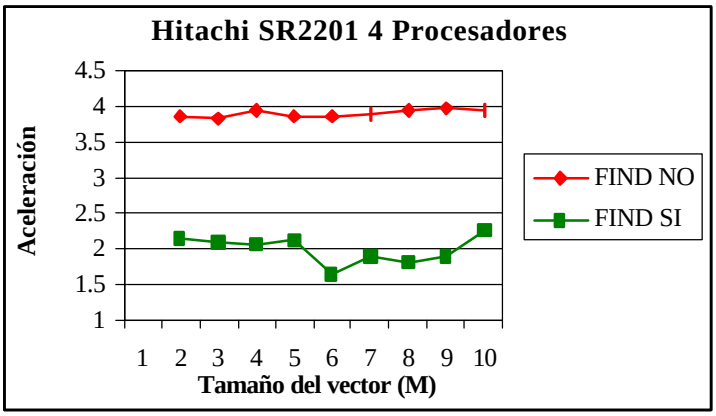

Figura 6.87 Algoritmo de Búsqueda. Resultados para la Hitachi SR2201 con 4 procesadores

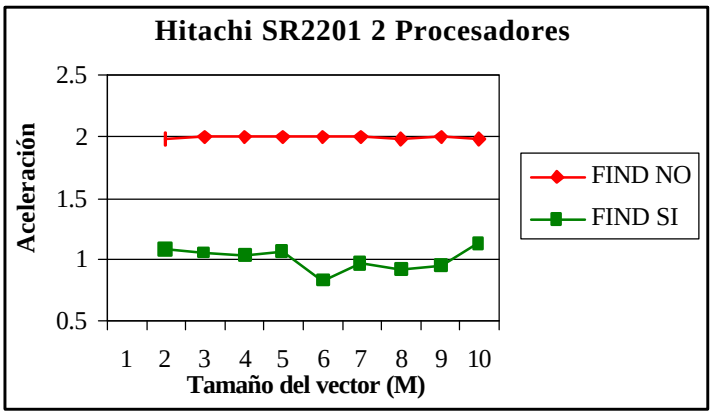

Figura 6.88 Algoritmo de Búsqueda. Resultados para la Hitachi SR2201 con 2 procesadores

6.7. Resultados del Modelo Colectivo como modelo de predicción del tiempo de cómputo

Presentamos en esta sección los resultados correspondientes a la evaluación de los algoritmos de la Transformada Rápida de Fourier y el de Ordenación por Muestreo Regular que estudiamos en el Capítulo 4 utilizando el Modelo de Computación Colectiva como Modelo de predicción del tiempo de cómputo.

Las plataformas que hemos utilizado en estos experimentos han sido el Cray T3E, SGI Origin 2000 y Digital Alphaserver.

6.7.1. La Transformada rápida de Fourier

#Procesadores	1	2	4	8	16
Cray Modelo	20.5399	10.8137	6.1497	3.9318	2.9125
Cray Real	20.5399	10.7665	6.0739	3.8730	2.9439
Origin Modelo	72.846	38.1533	27.4897	20.7049	12.2696
Origin Real	72.846	39.9944	28.9499	24.2797	12.1295
Digital Modelo	40.9167	23.2506	13.7852	10.6048	-
Digital Real	40.9167	23.4941	14.1147	12.2208	-

Tabla 6.41 Tiempos estimados y medidos para el algoritmo FFT

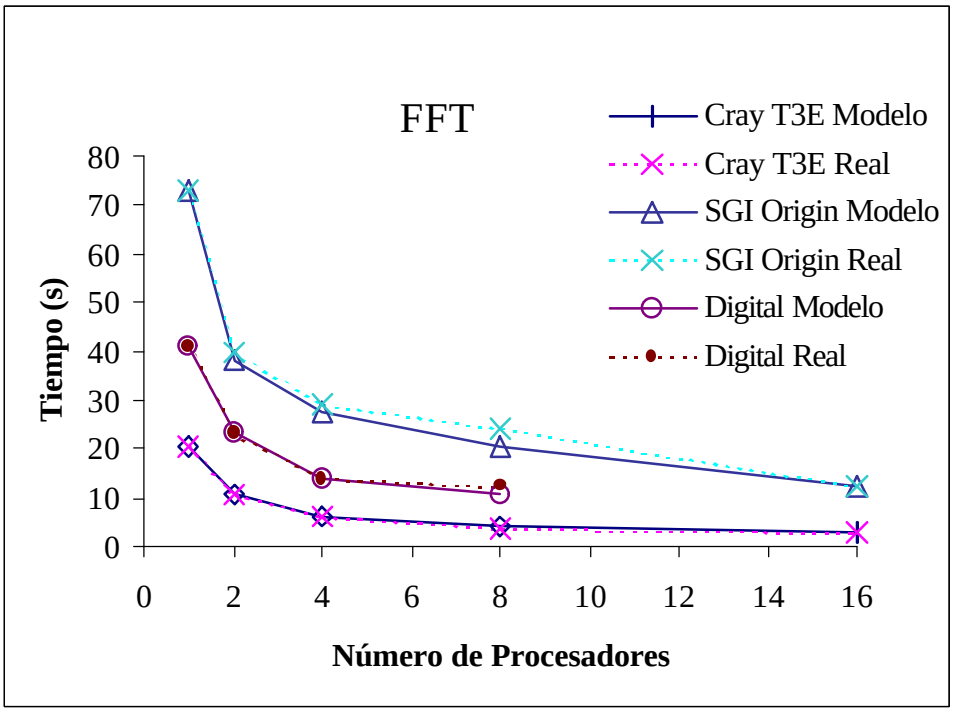

Figura 6.89 Tiempos estimados y medidos para el algoritmo de la FFT.

6.7.2. El Algoritmo de ordenación por Muestreo Regular (PSRS)

En este caso hemos utilizado un número significativo de procesadores para observar el comportamiento del modelo. La Tabla 6.42 y la Figura 6.90 presentan los tiempos calculados por el modelo de Computación Colectiva y los medidos para el Cray T3E y la SGI Origin 2000 con un máximo de 24 procesadores. Los resultados prueban la validez del modelo. Las curvas reales y previstas aparecen solapadas para el Cray en la figura. Hasta 16 procesadores, las curvas aparecen solapadas para la SGI Origin 2000. A partir de este punto, la escalabilidad más limitada de esta máquina [Rod98a] separa las curvas. La Tabla 6.43 muestra el error relativo porcentual.

Para evitar la influencia de factores secundarios que pudieran oscurecer la precisión del modelo (los tamaños de las memorias cache, por ejemplo), las constantes computacionales se han ajustado al número de procesadores utilizados. Los valores utilizados pueden consultarse en la referencia [Psr].

#Procesadores	2	4	8	16	24
CRAY REAL	0.7954	0.4645	0.2867	0.2038	0.1838
CRAY MODELO	0.7729	0.4631	0.2926	0.2085	0.1861
ORIGIN REAL	1.0251	0.7096	0.5291	0.4328	0.4640
ORIGIN MODELO	1.0059	0.6894	0.5151	0.4212	0.3945

Tabla 6.42 Tiempos estimados y medidos para el algoritmo PSRS.

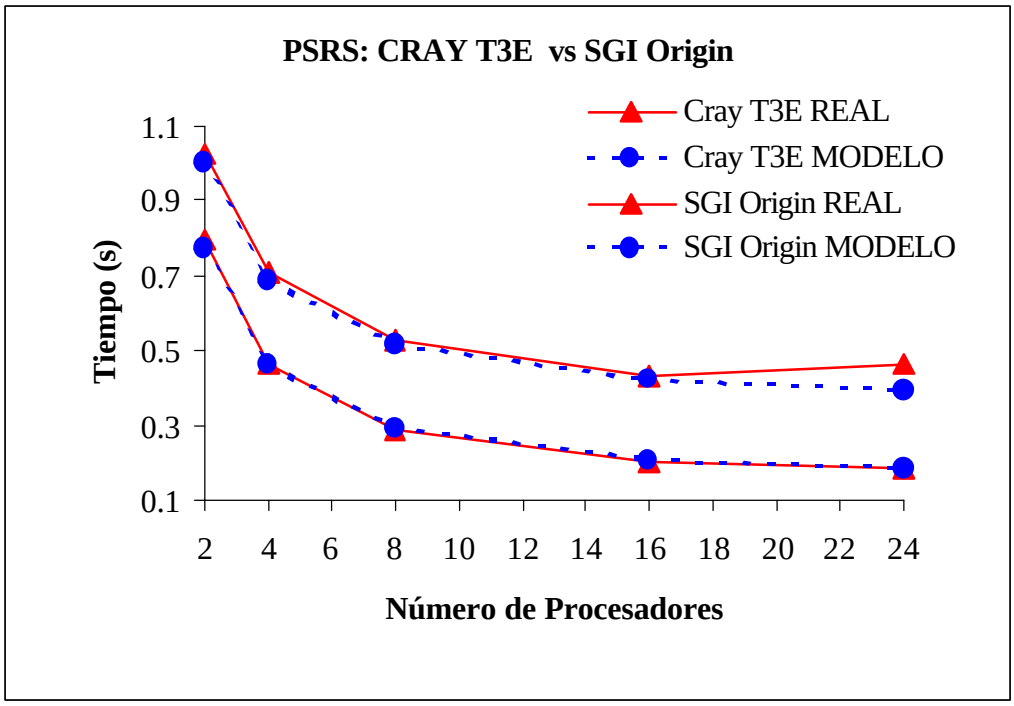

Figura 6.90 Tiempos estimados y medidos para el algoritmo PSRS

#Procesadores	2	4	8	16	24
Cray T3E	2.83	0.30	2.06	2.28	1.21
SGI Origin 2000	1.88	2.85	2.66	2.69	14.96

Tabla 6.43 Porcentaje de error para el algoritmo PSRS.

6.8. El coste de las funciones de división

Para medir el coste computacional de las funciones de división que hemos propuesto en el modelo de Computación Colectiva en comparación con la posibilidad de utilizar las funciones de división de los conjuntos de procesadores de MPI hemos realizado experimentos en los que comparamos el tiempo invertido por un cierto número de repeticiones de la ejecución de ambos tipos de funciones de división. En concreto hemos comparado el coste de la macro PAR() de *llc* con el del par de funciones *MPI_Comm_split()* y *MPI_Comm_free()* (con las que implementaríamos la funcionalidad de la función de división PAR())

Las tablas que siguen para las diferentes arquitecturas muestran el tiempo en segundos correspondiente a la ejecución de un cierto número de repeticiones de ambos tipos de funciones de división. Las curvas de la Figura 6.96 y la Figura 6.97 comparan la evolución de estos tiempos variando el número de procesadores mientras que las figuras 6.93 a la 6.95 comparan los rendimientos de *MPI_Comm_split()* y la macro PAR() en el caso del Cray T3D.

Presentamos resultados para los Cray T3E y T3D, el SGI Origin 2000 y el Hitachi SR2201. Los comportamientos observados para el resto de arquitecturas consideradas en esta memoria son completamente análogos.

Se observa que en todos los casos, el tiempo invertido por las funciones de MPI es varios órdenes de magnitud mayor que el empleado por la macro PAR(), que puede ser considerado constante. Por otra parte, el tiempo consumido por la macro PAR() no depende del número de procesadores empleados en la división, mientras que crece con el número de procesadores en el caso de las funciones de MPI.

6.8.1. Cray T3E

Repeticiones	Número de Procesadores						
	128	64	32	16	8	4	2
1000	0.9330	0.3501	0.1616	0.0822	0.0488	0.0342	0.0294
2000	1.8484	0.7067	0.3080	0.1643	0.0969	0.0677	0.0550
3000	2.7626	1.0416	0.4673	0.2472	0.1450	0.1015	0.0828
4000	3.7050	1.4189	0.6211	0.3344	0.1952	0.1353	0.1100
5000	4.6183	1.7493	0.7797	0.4063	0.2419	0.1699	0.1375
6000	5.5559	2.0815	0.9332	0.4863	0.2915	0.2030	0.1655
7000	6.4585	2.4432	1.0971	0.5667	0.3399	0.2374	0.1929
8000	7.3859	2.8227	1.2412	0.6490	0.3872	0.2708	0.2205
9000	8.3009	3.1700	1.3992	0.7285	0.4364	0.3049	0.2477
10000	9.2473	3.4847	1.5600	0.8110	0.4849	0.3383	0.2750

Tabla 6.44 Tiempo en segundos para la función MPI_Comm_split en el Cray T3E

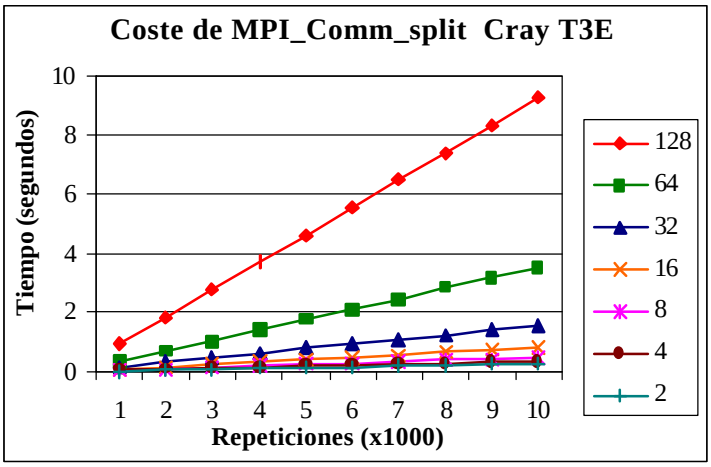

Figura 6.91 El coste de MPI_Comm_split en el Cray T3E para diferente número de procesadores

Repeticiones	Número de Procesadores						
	128	64	32	16	8	4	2
1000	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003
2000	0.0007	0.0007	0.0007	0.0007	0.0007	0.0007	0.0007
3000	0.0010	0.0010	0.0010	0.0010	0.0010	0.0010	0.0010
4000	0.0014	0.0014	0.0014	0.0014	0.0014	0.0014	0.0014
5000	0.0017	0.0017	0.0017	0.0017	0.0017	0.0017	0.0017
6000	0.0020	0.0020	0.0020	0.0020	0.0020	0.0020	0.0020
7000	0.0024	0.0024	0.0024	0.0024	0.0024	0.0024	0.0024
8000	0.0027	0.0027	0.0027	0.0027	0.0027	0.0027	0.0027
9000	0.0030	0.0031	0.0030	0.0030	0.0030	0.0031	0.0030
10000	0.0034	0.0034	0.0034	0.0034	0.0034	0.0034	0.0034

Tabla 6.45 Tiempo en segundos para la función de división PAR() en el Cray T3E

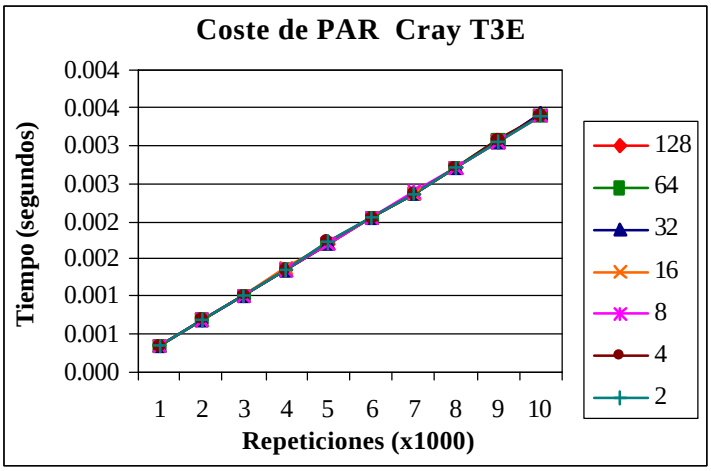

Figura 6.92 El coste de la función de división PAR en el Cray T3E

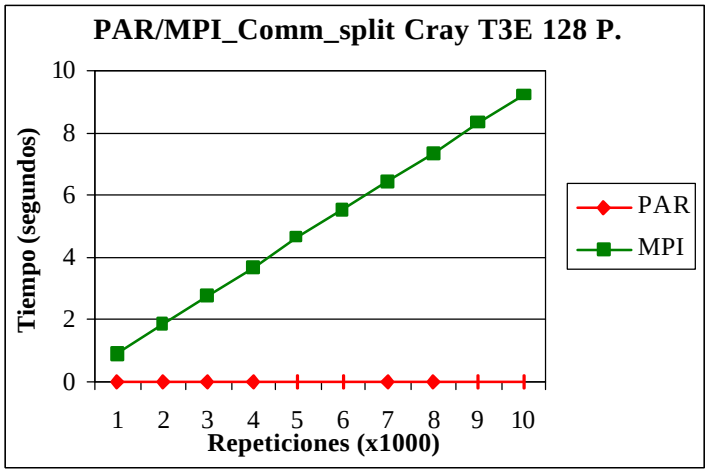

Figura 6.93 El coste de las funciones de división en el Cray T3E. 128 Procesadores

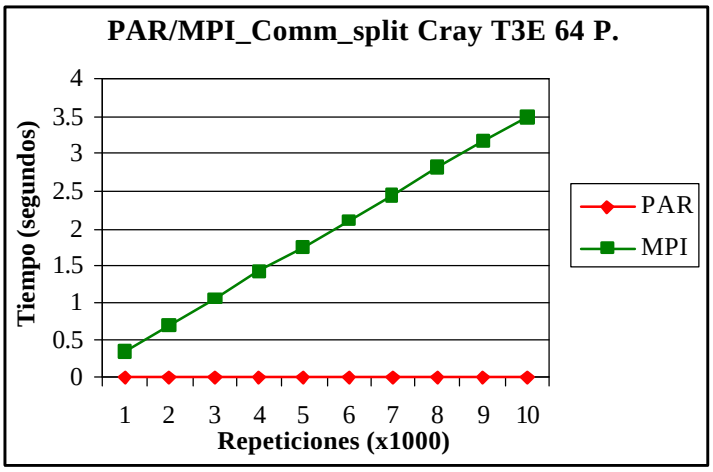

Figura 6.94 El coste de las funciones de división en el Cray T3E. 64 Procesadores

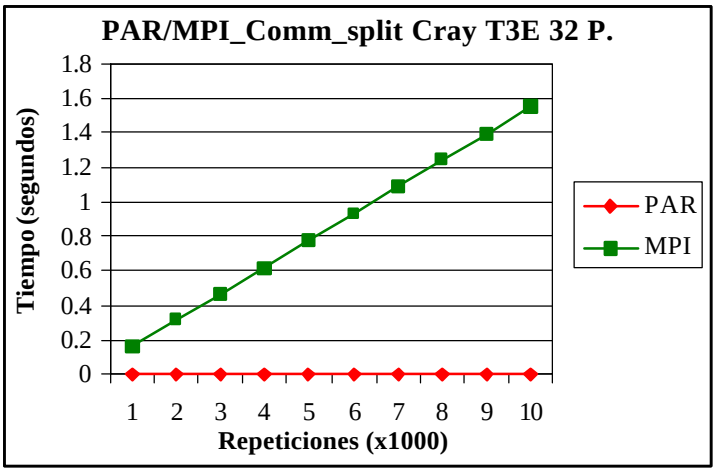

Figura 6.95 El coste de las funciones de división en el Cray T3E. 32 Procesadores

6.8.2. Cray T3D

Repeticiones	Número de procesadores							
	256	128	64	32	16	8	4	2
1000	10.7728	3.7734	1.5351	0.741775	0.4390	0.2986	0.2369	0.2119
2000	21.5530	7.5492	3.0711	1.482878	0.8785	0.5973	0.4741	0.4240
3000	32.3327	11.3236	4.6063	2.224502	1.3186	0.8962	0.7114	0.6356
4000	43.1080	15.0980	6.1405	2.965981	1.7586	1.1948	0.9483	0.8476
5000	53.8879	18.8732	7.6765	3.707514	2.1980	1.4939	1.1853	1.0595
6000	64.6641	22.6487	9.2118	4.449026	2.6376	1.7923	1.4227	1.2716
7000	75.4449	26.4229	10.7461	5.190659	3.0765	2.0911	1.6593	1.4832
8000	86.2200	30.1984	12.2828	5.932267	3.5164	2.3898	1.8969	1.6952
9000	97.0001	33.9727	13.8179	6.673722	3.9568	2.6884	2.1340	1.9069
10000	107.7799	37.7479	15.3539	7.415352	4.3939	2.9878	2.3711	2.1190

Tabla 6.46 Tiempo en segundos para la función MPI_Comm_split en el Cray T3D

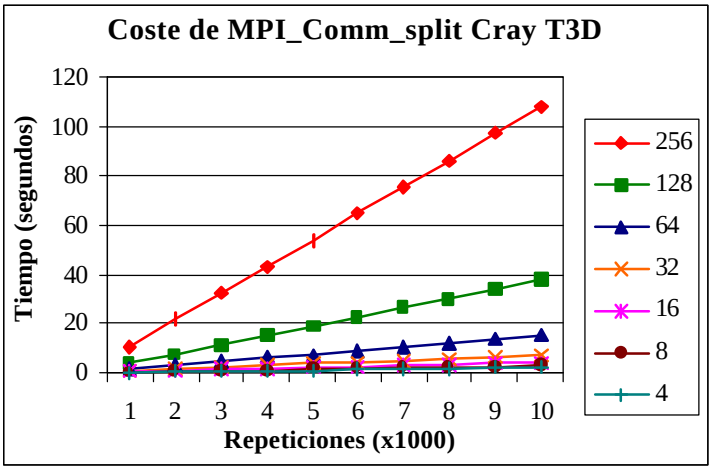

Figura 6.96 El coste de MPI_Comm_split en el Cray T3D para diferente número de procesadores

Repeticiones	Número de procesadores							
	256	128	64	32	16	8	4	2
1000	0.0014	0.0015	0.0014	0.0014	0.0014	0.0014	0.0014	0.0014
2000	0.0029	0.0028	0.0028	0.0028	0.0028	0.0029	0.0028	0.0028
3000	0.0043	0.0043	0.0043	0.0043	0.0043	0.0043	0.0043	0.0043
4000	0.0057	0.0058	0.0058	0.0057	0.0058	0.0058	0.0058	0.0058
5000	0.0072	0.0072	0.0071	0.0072	0.0072	0.0071	0.0072	0.0071
6000	0.0086	0.0086	0.0086	0.0086	0.0086	0.0086	0.0086	0.0086
7000	0.0100	0.0100	0.0100	0.0100	0.0100	0.0100	0.0100	0.0100
8000	0.0114	0.0114	0.0114	0.0114	0.0114	0.0114	0.0115	0.0114
9000	0.0129	0.0128	0.0128	0.0129	0.0129	0.0129	0.0129	0.0129
10000	0.0143	0.0143	0.0143	0.0143	0.0143	0.0143	0.0143	0.0143

Tabla 6.47 Tiempo en segundos para la función de división PAR() en el Cray T3D

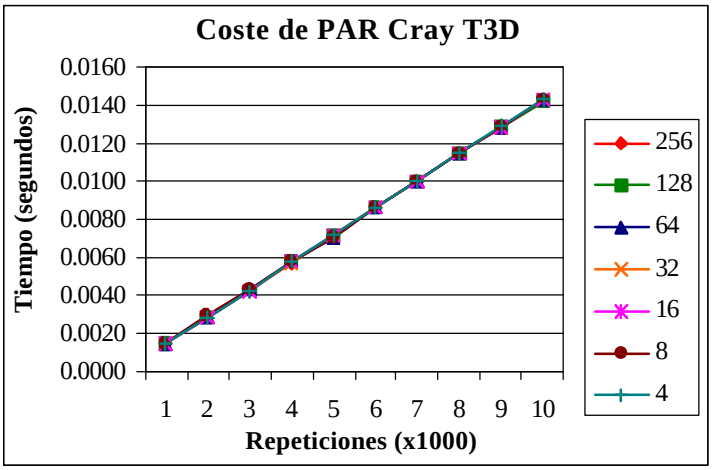

Figura 6.97 El coste de la función de división PAR en el Cray T3D

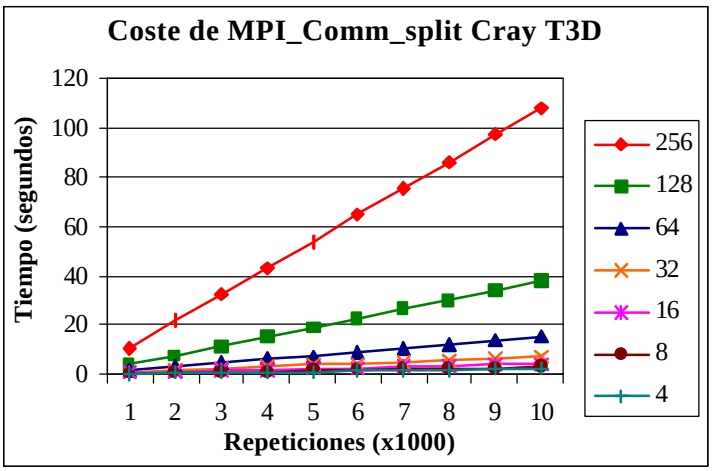

Figura 6.98 El coste de las funciones de división en el Cray T3D. 256 Procesadores

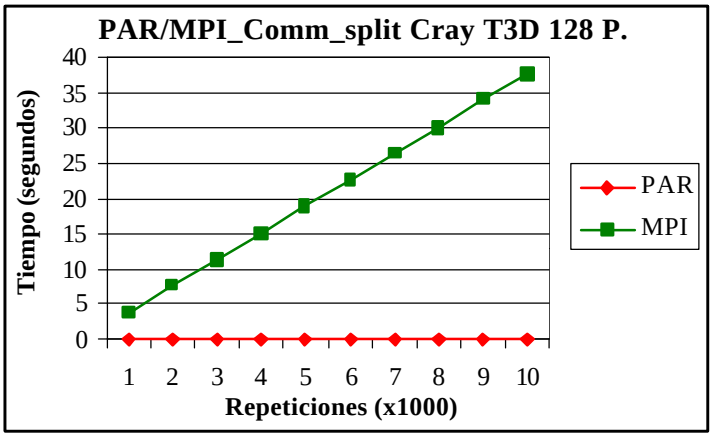

Figura 6.99 El coste de las funciones de división en el Cray T3D. 128 Procesadores

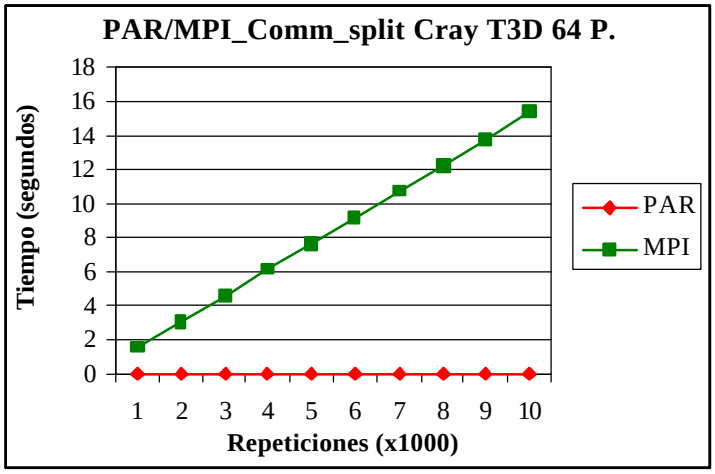

Figura 6.100 El coste de las funciones de división en el Cray T3D. 64 Procesadores

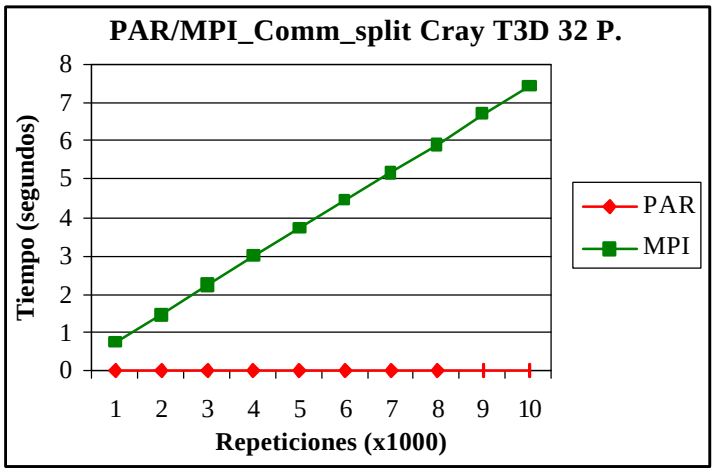

Figura 6.101 El coste de las funciones de división en el Cray T3D. 32 Procesadores

6.8.3. Silicon Graphics Origin 2000 (karnak3)

Repeticiones	Número de Procesadores				
	2	4	8	16	32
1000	0.0683	0.1145	0.2157	0.3818	0.9509
2000	0.1090	0.2070	0.4094	0.7280	1.3969
3000	0.1553	0.3127	0.6137	1.0903	2.0960
4000	0.2079	0.4139	0.8181	1.4508	2.7950
5000	0.2589	0.5308	1.0288	1.8121	3.5004
6000	0.3122	0.6206	1.2325	2.1752	4.2024
7000	0.3622	0.7249	1.4372	2.5383	4.8973
8000	0.4159	0.8295	1.6375	2.9005	5.6054
9000	0.4658	0.9315	1.8458	3.2632	6.3149
10000	0.5199	1.0362	2.0513	3.6181	7.0007

Tabla 6.48 Tiempo en segundos para la función MPI_Comm_split en el SGI Origin 2000

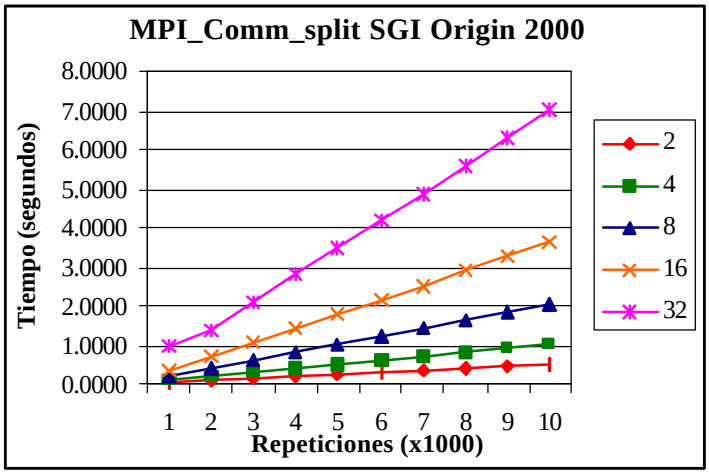

Figura 6.102 El coste de MPI_Comm_split en el SGI Origin 2000 para diferente número de procesadores

Repeticiones	Número de Procesadores				
	2	4	8	16	32
1000	0.0002	0.0002	0.0002	0.0002	0.0002
2000	0.0003	0.0004	0.0003	0.0003	0.0003
3000	0.0005	0.0005	0.0005	0.0005	0.0005
4000	0.0007	0.0007	0.0007	0.0007	0.0007
5000	0.0008	0.0008	0.0008	0.0008	0.0008
6000	0.0010	0.0010	0.0010	0.0010	0.0010
7000	0.0011	0.0012	0.0011	0.0011	0.0011
8000	0.0013	0.0013	0.0013	0.0013	0.0014
9000	0.0015	0.0015	0.0015	0.0015	0.0015
10000	0.0016	0.0016	0.0016	0.0016	0.0016

Tabla 6.49 Tiempo en segundos para la función de división PAR() en el SGI Origin 2000

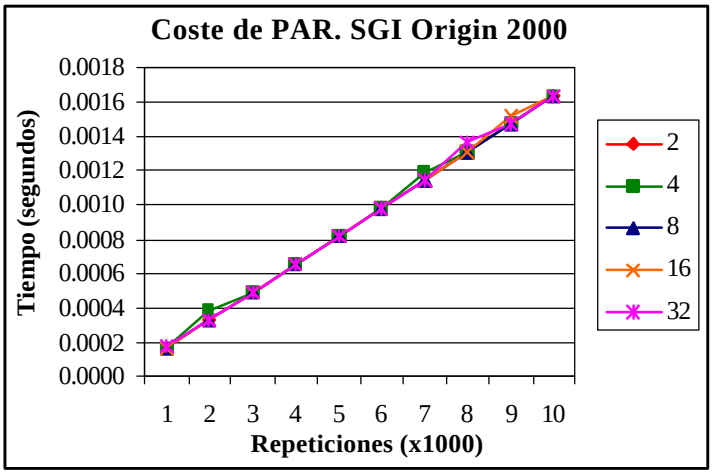

Figura 6.103 El coste de la función de división PAR en el SGI Origin 2000

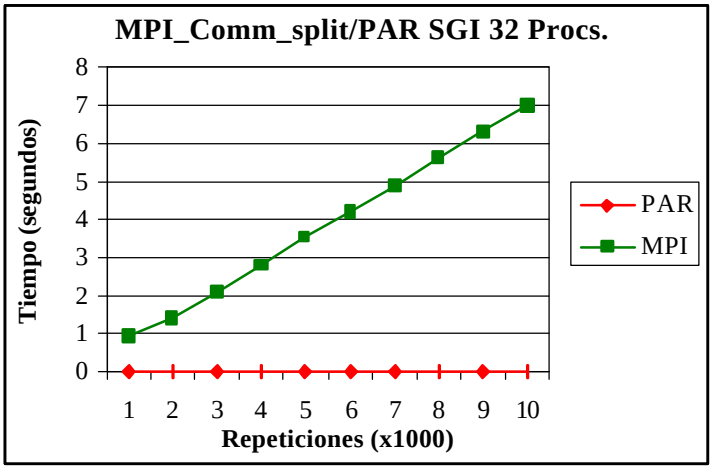

Figura 6.104 El coste de las funciones de división en el SGI Origin 2000. 32 Procesadores

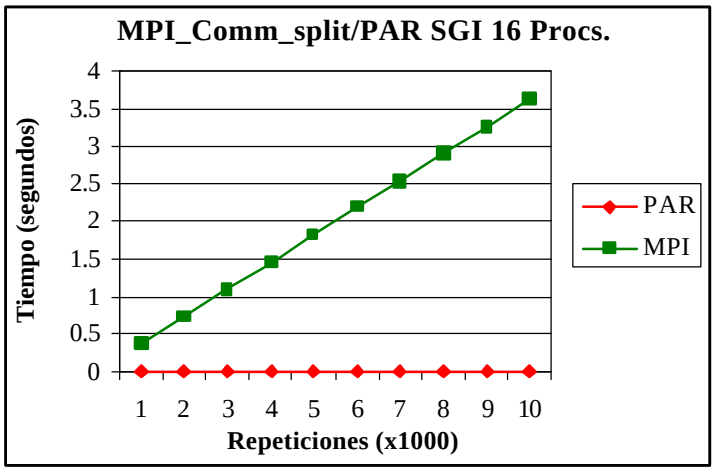

Figura 6.105 El coste de las funciones de división en el SGI Origin 2000. 16 Procesadores

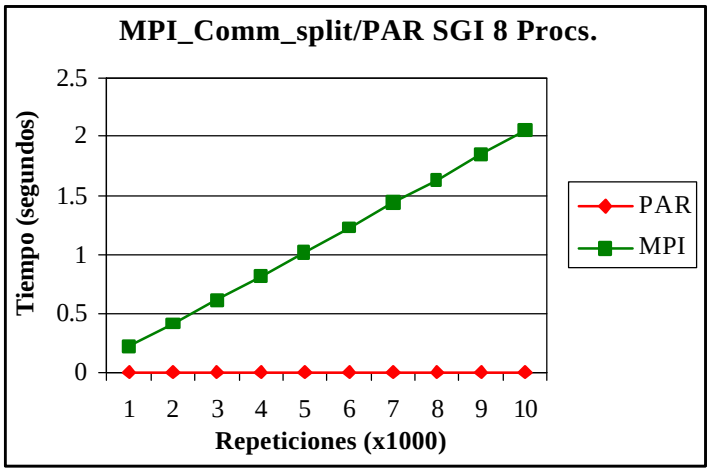

Figura 6.106 El coste de las funciones de división en el SGI Origin 2000. 8 Procesadores

6.8.4. Hitachi SR2201

Repeticiones	Número de Procesadores		
	2	4	8
1000	0.1558	0.2722	0.3755
2000	0.3107	0.5430	0.7485
3000	0.4662	0.8149	1.1208
4000	0.6212	1.0846	1.4895
5000	0.7749	1.3559	1.8671
6000	0.9298	1.6270	2.2361
7000	1.0858	1.8980	2.6164
8000	1.2392	2.1699	2.9833
9000	1.3961	2.4366	3.5508
10000	1.5492	2.7101	3.7268

Tabla 6.50 Tiempo en segundos para la función MPI_Comm_split en el Hitachi SR2201

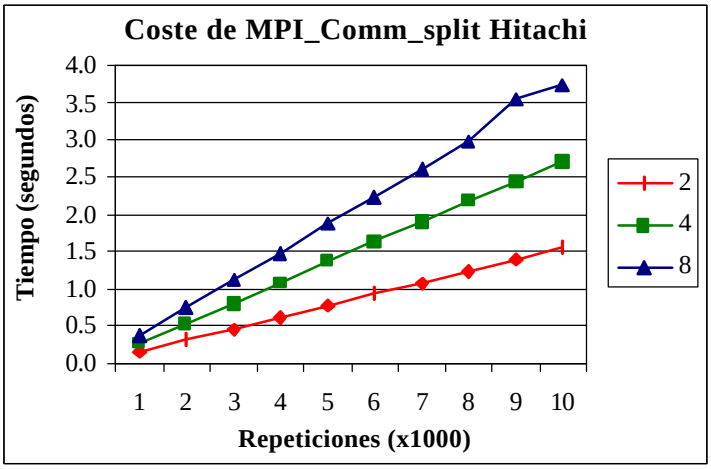

Figura 6.107 El coste de MPI_Comm_split en el Hitachi SR2201 para diferente número de procesadores

Repeticiones	Número de Procesadores		
	2	4	8
1000	0.0005	0.0005	0.0005
2000	0.0010	0.0010	0.0010
3000	0.0015	0.0015	0.0015
4000	0.0020	0.0020	0.0020
5000	0.0025	0.0024	0.0025
6000	0.0029	0.0029	0.0030
7000	0.0035	0.0034	0.0035
8000	0.0039	0.0039	0.0039
9000	0.0044	0.0044	0.0044
10000	0.0049	0.0049	0.0049

Tabla 6.51 Tiempo en segundos para la función de división PAR() en el Hitachi SR2201

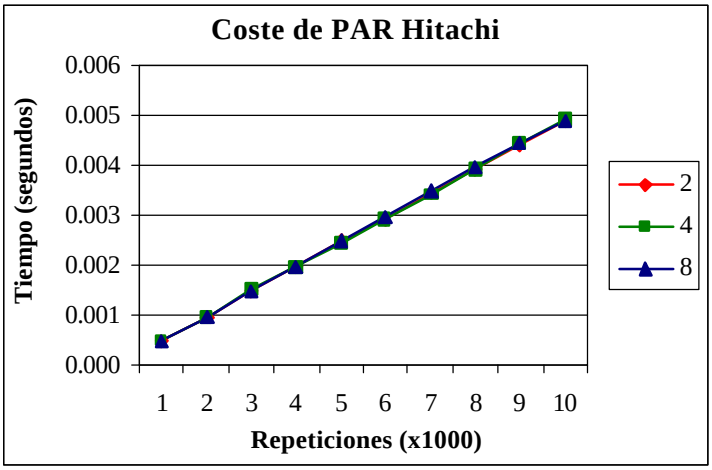

Figura 6.108 El coste de la función de división PAR en el Hitachi SR2201

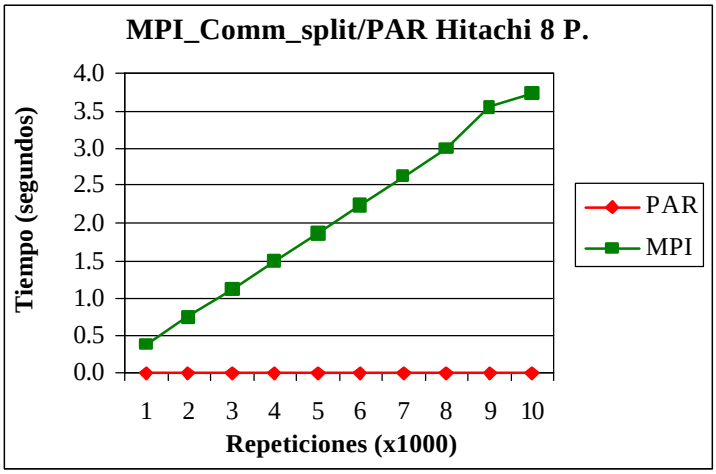

Figura 6.109 El coste de las funciones de división en el Hitachi SR2201. 8 Procesadores

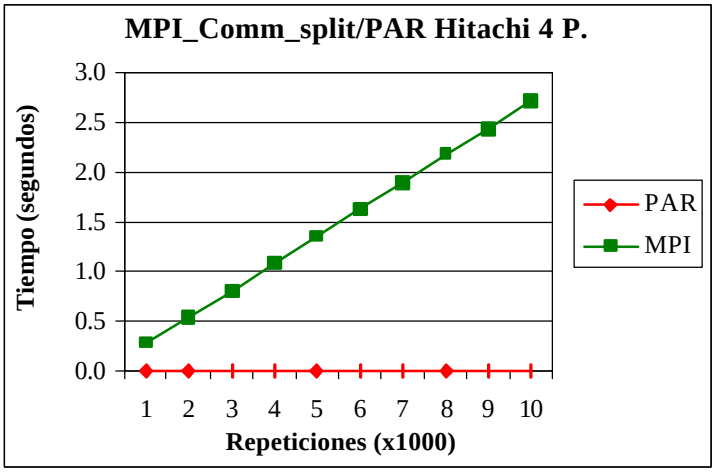

Figura 6.110 El coste de las funciones de división en el Hitachi SR2201. 4 Procesadores

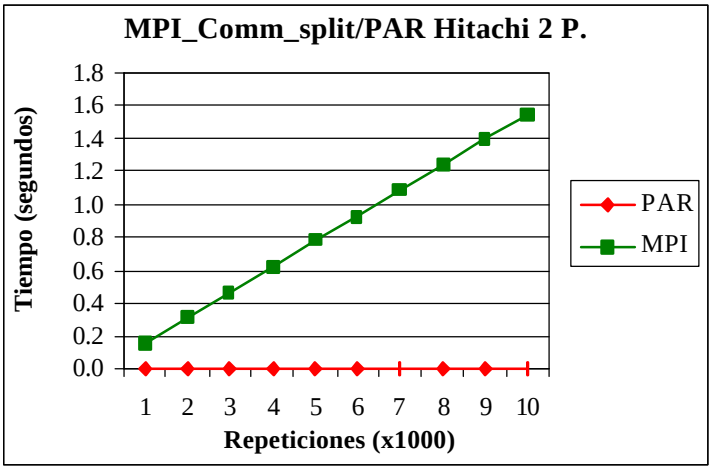

Figura 6.111 El coste de las funciones de división en el Hitachi SR2201. 2 Procesadores

Conclusiones y Trabajos Futuros

Conclusiones y Trabajos Futuros

Pretendemos en esta sección pasar revista a las que consideramos las aportaciones más relevantes del presente trabajo.

En el Capítulo 2 se puntualizó el concepto de sentencia de asignación de procesadores y se propuso una implementación eficiente de este tipo de sentencias, imprescindibles en cualquier lenguaje orientado al modelo PRAM. Se revisan en ese Capítulo las diferentes versiones del lenguaje *La Laguna*. También allí se establecieron las condiciones exactas para el cumplimiento del conocido Teorema de Brent.

En el Capítulo 3 se profundiza en el problema de la asignación de procesadores, estudiando las soluciones que aporta *fork95*, que junto con *La Laguna* es una de las pocas implementaciones de lenguajes orientados al modelo PRAM.

Las soluciones presentadas en los Capítulos 2 y 3 para sistemas altamente síncronos de memoria compartida fueron generalizados para máquinas distribuidas de sincronía gruesa en el Capítulo 4. Para ello se introdujeron los conceptos de *socio* e hipercubo dinámico. Esta aproximación ha permitido una implementación eficiente de los procesos de división. También en el Capítulo 4 se realizó una clasificación de problemas y algoritmos paralelos en términos de la distribuciones inicial y final de los datos en la máquina. Otra contribución de ese Capítulo la constituye el Modelo de Computación Colectiva. Hemos estudiando su validez para la implementación de algoritmos divide y vencerás de diferentes tipos (Común-Común y Privado-Privado). La metodología desarrollada para la paralelización de algoritmos divide y vencerás sobre problemas de tipo Común-común (aquellos en los que los datos de entrada y de salida son replicados en todos los procesadores) además de una expresión sencilla de algoritmos con paralelismo anidado proporciona una eficiencia óptima. Esta optimalidad se confirma con los resultados experimentales del Capítulo 6, que han sido realizados sobre un rango de arquitecturas paralelas que incluye la mayoría de los supercomputadores más potentes. Con el objetivo de incrementar el rendimiento de los algoritmos equilibrando la carga de trabajo, se plantean nuevas sentencias de asignación

de procesadores y se contrastan los resultados que se obtienen con las mismas. Todos los algoritmos fueron codificados en *La Laguna C*, la herramienta que se ha desarrollado siguiendo la filosofía del Modelo Colectivo.

Esta línea de investigación, lejos de quedar cerrada con el presente trabajo, abre puntos de investigación que intentaremos abordar en un futuro cercano. Entre estos destacamos:

- Acometer la implementación de más algoritmos tanto de tipo Común-Común como Privado-Privado. En concreto pensamos estudiar el algoritmo jerárquico de Barnes y Hut [Bar86] para el cálculo de fuerzas en un problema de *n* cuerpos. En el caso Privado-Privado tenemos pendiente implementar y estudiar los resultados de algunos algoritmos que fueron presentados para el caso Común-Común en el Capítulo 4, como son el algoritmo Quickhull Distribuido o la Transformada Rápida de Fourier Distribuida.
- Estudiar y proponer mecanismos para una expresión de más alto nivel de los problemas de tipo Privado-Privado. Este tipo de problemas, en su forma de expresión actual requieren una mayor implicación del programador en detalles de bajo nivel propios de la programación paralela.
- Incorporar *llc* en una herramienta de más alto nivel que posea la posibilidad de integrar paralelismo de sincronismo grueso (el de *llc*) con paralelismo pipeline (como el que proporciona *llp* [Rod98g], otra herramienta desarrollada en el seno del Grupo de Paralelismo de la La Laguna).
- Realizar más experimentos que además de corroborar la validez del Modelo Colectivo como modelo de predicción del tiempo de cómputo de los algoritmos, nos permitan conocer la precisión de estas predicciones.
- Estudiar los requisitos para establecer una metodología que permita la traducción de algoritmos PRAM al Modelo Colectivo. La Simulación de la Memoria Compartida y la Sincronización son los dos problemas más importantes que conlleva esta traducción. Existe una similitud entre el Modelo PRAM y el Modelo Colectivo: el concepto de variable común introducido en este trabajo presenta analogías con las variables compartidas del modelo PRAM, mientras que la sincronización implícita del modelo PRAM se ve relajada por la utilización de las funciones colectivas. La traducción se ve facilitada si el lenguaje orientado al modelo PRAM utiliza un esquema de sincronización como el de *fork95*, en el que una sentencia de asignación de procesadores crea grupos asíncronos.
- Para intentar paliar el principal inconveniente de *fork95* (la imposibilidad de su ejecución en una máquina paralela real), hemos puesto en marcha un proyecto conjunto entre nuestro grupo y el de la SB-PRAM. El objetivo inicial del proyecto es el desarrollo del back-end del compilador de *fork95* para el sistema operativo PRAMOS de la SB-PRAM. Nuestro interés se centra también en hacer que el compilador de *fork95* produzca código ejecutable en arquitecturas paralelas más estándar.
- Las dificultades a la hora de conseguir versiones ejecutables de NESL sobre máquinas paralelas hacen difícil la evaluación de su rendimiento. Esperamos poder realizar los experimentos necesarios para contrastar el rendimiento que alcanza nuestra aproximación en comparación con los resultados que se obtienen utilizando NESL.

Apéndices

Apéndice A-1

El algoritmo de ordenación en hipercubos de Brinch Hansen

```
/*
Algoritmo: Ordenación en Hipercubos de P. B. Hansen
Autor: F. de Sande
Bibliografía:
Notas: Los ficheros common.? contienen rutinas, prototipos y
definiciones comunes a todos los algoritmos. En particular el
quicksort secuencial que se utiliza en último término.
*/
#include "common.h"
#include "common.c"
#include "code.c"

int array[MAXSIZE];
int dim;

int main(int argc, char*argv[]) {
  int first, last, middle;  /* Range to sort in the array */
  int i, j;
  int level;    /* Level of the processor in the hypercube */
  int range;    /* Number of elements sent/received */
  int partner;
  int REP, seed;
  int my_rank;
  MPI_Status status; /* Para las recepciones de mensajes */
  int tag = 0;
  int tmp_proc;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &NAME);
  MPI_Comm_size(MPI_COMM_WORLD, &NUMPROCESSORS);
  switch (NUMPROCESSORS) {
    case 32: dim = 5; break;
    case 16: dim = 4; break;
    case 8: dim = 3; break;
    case 4: dim = 2; break;
    case 2: dim = 1; break;
  }
  if (NAME == 0) {
    for (j = 1; j <= MAX_REP_SIZE; j++) {
      SIZE = j * M;
      level = node_level(NAME);
      for (REP = 0; REP < MAX_REP; REP++) {
      seed = REP + 1;
      printf("Rep: %d Tamao: %d\n", REP, SIZE);
      /*========= EJECUCION SECUENCIAL =========*/
      initialize(array, seed);
      start = MPI_Wtime();
      quicksortseq(array, 0, SIZE - 1);
      finish = MPI_Wtime();
      SEQ[REP][j-1] = finish - start;
      test(array);
      /*========= EJECUCION PARALELA =========*/
      initialize(array, seed);
```

```
        first = 0;
        last = SIZE - 1;
        start = MPI_Wtime();
        /* ---------------- Send to others ------------------*/
        for (i = 0; i < dim; i++) {
          middle = (first + last) / 2;
          find(array, first, last, middle);
          range = middle - first + 1;
          partner = NAME ^ (1 << i);
          MPI_Send(&first,1,MPI_INT, partner, tag, MPI_COMM_WORLD);
          MPI_Send(&middle, 1, MPI_INT, partner, tag, MPI_COMM_WORLD);
          MPI_Send(array+first, range, MPI_INT, partner, tag,
        MPI_COMM_WORLD);
          first = middle + 1;
        }
        /* ------------------------------------------------ */
        quicksortseq(array, first, last);
        /* ------- Receive from others --------------------- */
        for (i = (dim - 1); i >= level; i--) {
          partner = NAME ^ (1 << i);
          MPI_Recv(&first,1,MPI_INT,partner,tag,MPI_COMM_WORLD,&status)
        ;
          MPI_Recv(&middle,1,MPI_INT,partner,tag,MPI_COMM_WORLD,&status
        );
          range = middle - first + 1;
          MPI_Recv(array+first,range,MPI_INT,partner,tag,MPI_COMM_WORLD
        , &status);
        } /* ------------------------------------------------ */
        finish = MPI_Wtime();
        PAR[REP][j-1] = finish - start;
        test(array);
        /*=====================================*/
        } /* Fin del bucle REP */
      } /* j */
      printf("\n*************************************************\n");
      for (j = 1; j <= MAX_REP_SIZE; j++) {
        SIZE = j * M;
        par_m[j-1] = seq_m[j-1] = speed_m[j-1] = 0.0;
        for (REP = 0; REP < MAX_REP; REP++) {
          SPEED[REP][j-1] = SEQ[REP][j-1] / PAR[REP][j-1];
          par_m[j-1] += PAR[REP][j-1];
          seq_m[j-1] += SEQ[REP][j-1];
          speed_m[j-1] += SPEED[REP][j-1];
          printf("#%d SIZ %d REP %d SEQ %9lf PAR %9lf SPE %9lf \n",
        NAME, SIZE, REP, SEQ[REP][j-1], PAR[REP][j-1], SPEED[REP][j-
        1]);
        } /* REP */
      } /* j */
      printf("\n************ MEDIA ***************************\n");
      printf("NAME\tSEQ\tPAR\tSPE\n");
      for (j = 1; j <= MAX_REP_SIZE; j++) {
        SIZE = j * M;
        par_m[j-1] = par_m[j-1] / (1.0 * MAX_REP);
        seq_m[j-1] = seq_m[j-1] / (1.0 * MAX_REP);
        speed_m[j-1] = speed_m[j-1] / (1.0 * MAX_REP);
        printf("%d\t%d\t%9lf\t%9lf\t%9lf\n", NAME, SIZE, seq_m[j-1],
        par_m[j-1], speed_m[j-1]);
      } /* j */
    } /* root */
else
    {
```

```
  for (j = 1; j <= MAX_REP_SIZE; j++) {
  SIZE = j * M;
  level = node_level(NAME);
  for (REP = 0; REP < MAX_REP; REP++) {
    seed = REP + 1;
    /* ---------------- Reception ------------------*/
    partner = NAME ^ (1 << (level - 1));
    MPI_Recv(&first, 1, MPI_INT, partner, tag, MPI_COMM_WORLD,
  &status);
    MPI_Recv(&last, 1, MPI_INT, partner, tag, MPI_COMM_WORLD,
  &status);
    range = last - first + 1;
    MPI_Recv(array+first, range, MPI_INT, partner, tag,
  MPI_COMM_WORLD, &status);
    /* ---------------- Send to others ------------------*/
    for (i = level; i < dim; i++) {
      middle = (first + last) / 2;
      find(array, first, last, middle);
      range = middle - first + 1;
      partner = NAME ^ (1 << i);
      MPI_Send(&first, 1, MPI_INT, partner, tag, MPI_COMM_WORLD);
      MPI_Send(&middle, 1, MPI_INT, partner, tag, MPI_COMM_WORLD);
      MPI_Send(array+first, range, MPI_INT, partner, tag,
    MPI_COMM_WORLD);
    first = middle + 1;
    }
    quicksortseq(array, first, last);
    for (i = (dim - 1); i >= level; i--) {
      partner = NAME ^ (1 << i);
      MPI_Recv(&first, 1, MPI_INT, partner, tag, MPI_COMM_WORLD,
    &status);
      MPI_Recv(&middle, 1, MPI_INT, partner, tag, MPI_COMM_WORLD,
    &status);
      range = middle - first + 1;
      MPI_Recv(array+first, range, MPI_INT, partner, tag,
    MPI_COMM_WORLD, &status);
    }
    /*---------- Reduction ------------------*/
    partner = NAME ^ (1 << (level - 1));
    MPI_Send(&first, 1, MPI_INT, partner, tag, MPI_COMM_WORLD);
    MPI_Send(&last, 1, MPI_INT, partner, tag, MPI_COMM_WORLD);
    range = last - first + 1;
    MPI_Send(array+first, range, MPI_INT, partner, tag,
  MPI_COMM_WORLD);
    } /* REP */
  } /* j */
} /* nodes */
MPI_Finalize();
return 0;
} /* main */
```

Apéndice A-2

Funciones básicas sobre secuencias en NESL

Funciones básicas sobre secuencias	
Operaciones básicas	
`#a`	Longitud de `a`
`a[i]`	i-ésimo elemento de `a`
`dist(a,n)`	Crea una secuencia de longitud `n` con `a` en cada elemento.
`zip(a,b)`	Une dos secuencias elemento a elemento formando una secuencia de pares.
`[s:e]`	Crea una secuencia de enteros en el rango de `s` a `e` (sin incluir `e`)
`[s:e:d]`	Igual que `[s:e]` pero con un 'stride' `d`.
Prefijos	
`plus_scan(a)`	Realiza una operación de prefijos sobre `a` utilizando el operador `+`
`min_scan(a)`	Realiza una operación de prefijos sobre `a` utilizando el operador mínimo
`max_scan(a)`	Realiza una operación de prefijos sobre `a` utilizando el operador máximo
`or_scan(a)`	Realiza una operación de prefijos sobre `a` utilizando el operador `or`
`and_scan(a)`	Realiza una operación de prefijos sobre `a` utilizando el operador `and`
Reducciones	
`sum(a)`	Suma los elementos de `a`
`max_val(a)`	Retorna el máximo valor de `a`
`min_val(a)`	Retorna el mínimo valor de `a`
`any(a)`	Retorna true si alguno de los valores de `a` son true.
`all(a)`	Retorna true sólo si todos los valores de `a` son true.
`count(a)`	Cuenta el número de valores true en `a`.
`max_index(a)`	Retorna el índice del valor máximo.
`min_index(a)`	Retorna el índice del valor mínimo.

Funciones básicas sobre secuencias	
Funciones de reordenación	
`read(a, i)`	Lee en `a` de los índices `i`
`write(a, iv_pairs)`	Escribe los valores en `a` usando los pares de enteros de `iv_pairs`
`permute(a,i)`	Permuta los elementos de a de acuerdo a los índices de `i`.
`rotate(a,i)`	Rota la secuencia `a` en `i` posiciones.
`reverse(a)`	Invierte el orden de la secuencia `a`.
`drop(a,i)`	Elimina los primeros `i` elementos de `a`.
`take(a,i)`	Devuelve los primeros `i` elementos de `a`.
`odd_elts(a)`	Retorna los elementos impares de `a`
`even_elts(a)`	Devuelve los elementos pares de `a`
`interleave(a,b)`	Entrelaza los elementos de `a` y `b`.
`subseq(a,i,j)`	Retorna la subsecuencia de `a` entre las posiciones `i` a `j` (excluyendo a `j`)
`a->i`	Equivalente a `read(a, i)`
`a<-pares_ev`	Equivalente a `write(a, pares_ev)`
Anidación/ Aplanamiento	
`flatten(a)`	Aplana la secuencia `a` un nivel.
`partition(a,l)`	Particiona la secuencia a en una secuencia anidada usando las longitudes de l.
`bottop(a)`	Divide una secuencia en dos y la retorna como secuencia anidada.

Apéndice A-3

Sintaxis básica de NESL

Sintaxis	Ejemplo
`FUNCTION name(args) = exp ;`	`FUNCTION double(a) = 2*a;`
`IF e1 THEN e2 ELSE e3`	`IF (a == 22) THEN a ELSE 5*a`
`LET binding* IN exp`	`LET a = b*6;` `IN a + 3`
`{e1 : pattern IN e2}`	`{a + 22 : a IN [2, 1, 9]}`
`{pattern IN e1 | e2}`	`{a IN [2, 1, 9] | a < 8}`
`{e1 : p1 IN e2 ; p2 in e3}`	`{a + b : a IN [2,1]; b IN [7,11]}`

Apéndice A-4

Funciones escalares en NESL

Funciones escalares	
Lógicas	`not or and xor nor nand`
Comparación	`== /= <= =`
Predicados	`plusp minusp zerop oddp evenp`

Aritméticas	`+ - * / rem abs max min`
	`lshift rshift`
	`sqrt isqrt ln log exp expt`
	`sin cos tan asin acos atan`
	`sinh cosh tanh`

Conversión	`btoi code_char char_code` `float ceil floor trunc round`
Números aleatorios	`rand rand_seed`
Constantes	`pi max_int min_int`

Bibliografías

Bibliografía

[Aba96] Abandah, G.A., Davidson E.S. *Modeling the Communication Performance of the IBM SP2*. Proc. 10th IPPS, 1996

[Abo91a] Abolhassan, F., Keller, J., Paul, W.J. *On the Cost-Effectiveness of PRAMs*. Proc. of SPDP'91, IEEE Comp. Soc., pp. 2-9, 1991

[Abo91b] Abolhassan, F., Keller, J., Paul, W.J. *On the Cost-Effectiveness and Realization of the* Theoretical PRAM Model. Technical Report SFB-09-1991, Universität des Saarlandes, 1991

[Abo93] Abolhassan, J., Drefenstedt, R., Keller, J. Paul W.J., Scheerer, D. *On the physical design of PRAMs*. Computer Journal, 36(8), pp. 756-762, 1993

[Aho90] Aho, A., Sethi, R., Ullman, J. *Compiladores. Principios, técnicas y herramientas*. Addison-Wesley Iberoamericana. 1990

[Akl89] Akl, S. G. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, 1989

[Ala94] Alasdair R., Bruce A., Mills J.G., Smith A.G.. *CHIMP/MPI User Guide*. Technical Report EPCC-KTP-CHIMP-V2-USER 1.2. Edimburgh Parallel Computing Centre, 1994

[Alm94] Almasi, G., Gottlieb, A., *Highly Parallel Computing*. Benjamin Cummings, 1994

[Alp] *Alphaserver 8400 System*. http://www.digital.com/hpc/systems/sy8400.html

[Alv90] Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A,. Smith, B., *The Tera computer system*. Proc. of the 1990 Int. Conf. on Supercomputing, ACM, pp. 1-6, 1990

[Ans90] ANSI American National Standard Institute, Inc., New York. *American National Standards for Information Systems, Programming Language C*. ANSI X3.159-1989, 1990

[Ape] *APERITIF. Automatic Parallelization of Divide and Conquer Algorithms* http://wwwmayr.informatik.tu-muenchen.de/personen/erlebach/aperitif.html

[Arr96] Arruabarrena, J.M., Arruabarrena A., Beivide R., Gregorio J.A. *Assesing the Performance of the New IBM-SP2 Communication Subsystem*. IEEE Parallel and Distributed Technology, pp 12-22, 1996

[Bac97] Bach, P., Braun, M., Formella, A., Friedrich, J., Grün, Th., Lichtenau C. *Building the 4 Processor SB-PRAM Prototype.* Proc. of the Hawaii 30th International Symposium on System Science HICSS-30, 5, pp. 14-23, 1997

[Bar86] Barnes, J., Hut, P. *A hierarchical O(n log n) force calculation algorithm.* Nature, 324, 1986

[Bar97] Barbero, J.M. *Desarrollo de un compilador de ll con las herramientas lex y yacc*. Memoria del Proyecto de Ingeniería en Informática. Universidad de La Laguna, 1997

[Bat82] Batcher, K.E. *Bit Serial Parallel Processing Systems*. IEEE Transactions on Computers. C-31(5), pp. 377-384, 1982

[Ble89] Blelloch, G.E., *Scans as Primitive Parallel Operations*. IEEE Transactions on Computers, 38(11), pp. 1526-1538, 1989

[Ble90a] Blelloch, G. E., Chatterjee, S. *VCODE: A data-parallel intermediate language.* Proc. of Symposium on The Frontiers of Massively Parallel Computation, pp. 471-480, 1990

[Ble90b] Blelloch, G., Sabot G. W. *Compiling collection-oriented languages onto massively parallel computers*. Journal of Parallel and Distributed Computing, 8(2), pp. 119-134, 1990

[Ble94] Blelloch, G.E., Hardwick, J.C., Sipelstein, J., Zagha, M., Chatterjee, S. *Implementation of a Portable Nested Data-Parallel Language.* Journal of Parallel and Distributed Computing, 21, pp. 4-14, 1994

[Ble95] Blelloch, G.E., *NESL: A Nested Data-Parallel Language.* Technical Report CMU-CS-95-170 Carnegie Mellon University, 1995

[Ble96] Blelloch, G. E., *Programming Parallel Algorithms*. Communications of the ACM, 39(3), pp. 85-97, 1996

[Bur94] Burns G., Daoud R., Vaigl J. *LAM: An Open Cluster Environment for MPI*. John W. Ross editor. Supercomputing Symposium'94, pp.379-386, 1994

[But92] Butler R., Lusk E. *User's Guide to the p4 Parallel Programming System*. University of North Florida, Argonne National Laboratory, 1992

[C4] *Centre de Computació i Comunicacions de Catalunya* (Cesca/Cepba) http://www.cccc.es/

[Car91] Carlini and Villano, *Transputers and Parallel Architectures*. Ellis Horowood Limited, 1991

[Cha94a] Chakravarty, M.M.T., Schröer, F.W., Simons M. *V Reference Manual.* GMD FIRST, Technical Report, 1994

[Cha94b] Chakravarty, M.M.T., Schröer, F.W., Simons M. *V User's Manual.* TU Berlin, Technical Report, 1994

[Cha95] Chakravarty, M.M.T., Schröer, F.W., Simons, M. V. *Nested Parallelism in C.* W.K. Giloi, S. Jähnichen, B. Shriver (eds.): Proc. Programming Models for Massively Parallel Computers 1995, IEEE CS Press, 1995

[Cie] *Centro de Investigaciones Energéticas Medioambientales y Tecnológicas, Ciemat*. http://www.ciemat.es/index.html

[Cra] *CRAY T3E Technology Profile*. http://www.cray.com/products/systems/crayt3e/technology.html

[Cro93] Cross, D., Drefenstedt, R., Keller, J., *Reduction of Network Cost and Wiring in Ranade's Butterfly Routing*. Information Processing Letters 45(2), pp. 63-67, 1993

[Cul93] Culler D.E., Karp R., Patterson D., Sahay A., Schauser K.E.,Santos E., Subramonian R., Eicken T. *LogP: Towards a Realistic Model of Parallel Computation*. Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1993

[Cul95] Culler D.E., Liu L.T., Martin R., Yoshikada C. *LogP Performance Assessment of Fast Network Interfaces*. Technical Report. Computer Science Division. University of California, Berkeley, 1995

[Cul96a] Culler D.E., Karp R., Patterson D., Sahay A., Santos E., Schauser K.E., Subramonian R., Eicken T. *LogP, a Practical Model of Parallel Computation*. Communications of the ACM, 39(11), 1996

[Cul96b] Culler D.E., Liu L.T., Martin R., Yoshikada C. *Assessing Fast Networks Interfaces*. IEEE Micro 16, 1996

[Eng93] Engelmann, C., Keller, J. *Simulation-based comparison of hash functions for emulated shared memory*. Proc. of PARLE'93, Parallel Architectures and Languages Europe, Springer Verlag LNCS, 694, pp. 1-11, 1993

[Epc] *Edinburgh Parallel Computing Centre*. http://www.epcc.ed.ac.uk/

[Fir] *German National Research Center for Information Technology. Institute for Computer Architecture and Software Technology.* http://www.first.gmd.de/

[For] *The PRAM Programming Language Fork95*. http://www.informatik.uni-trier.de/~kessler/fork95/

[For96] Formella, A., Keller, J., Walle, T. *HPP: A High-Performance PRAM*. Proc. 2nd Int. Euro-Par Conference. Springer LNCS, pp. 425-434, 1996

[For78] Fortune S. Wyllie, J.. *Parallelism in Random Access Machines*. Proc. 10th ACM Annual Symp. on Theory of Computing, pp. 114-118, 1978

[Fra91] Fraser, C.W., Hanson, D.R. *A code generation interface for ANSI C.* Software Practice and Experience, 21(9), pp. 963-988, 1991

[Fra95] Fraser, C.W., Hanson, D.R. *A retargetable compiler: Design and Implementation*. Benjamin Cummings Publishing Company, Inc., 1995

[Gar94] García F., Rodríguez C., León C., Sande F. *A High Level Parallel Language and Its Implementation on Transputer Networks*. Seventh Conference of the Nort American Transputer Users Group. (NATUG-7), Atlanta, USA, 1994

[Gei94] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V. *PVM 3 User's Guide and Reference Manual.* Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1994

[Gib88] Gibbons, A., Rytter, W. *Efficient Parallel Algorithms*. Cambridge University Press. 1988

[Got83] Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.P., Rudolph, L., Snir, M. *The NYU ultracomputer - Designing an MIMD shared memory parallel computer*. IEEE Trans. on Computers, C-32(2), pp. 175-189, 1983.

[Gro93] Gropp W.D., Smith B. *Chameleon parallel Programming tools users manual.* Technical Report ANL-93/23. Argonne National Laboratory, 1993

[Gro94] Gropp, W.D., Lusk, W. *User's Guide for MPICH, a Portable Implementation of MPI*. Technical Report ANL-96/6. Argonne National Laboratory, 1994

[Grü98] Grün, T., Hillebrand, M. *NAS Integer Sort on Multithreaded Shared Memory Machines.* Euro-Par'98 Parallel Processing. 4th International Euro-Par Conference. Springer Verlag LNCS, 1470, pp. 999-1009, 1998

[Hag92] Hagerup, T. Schmitt, A., Seidl, H. *FORK: A High-Level Language for PRAMs.* Future Generation Computer Systems 8, pp. 379-393, 1992

[Ham96a] Hambrusch, S., Khokhar, A. Models for Parallel Computation. Proceedings of Workshop on Challenges for Parallel Processing, International Conference on Parallel Processing, 1996.

[Ham96b] Hambrusch, S. Khokhar, A. C3: A Parallel Model for Coarse-grained Machines. Journal of Parallel and Distributed Computing, Vol. 32, Nr. 2, pp 139-154, 1996.

[Ham96c] Hambrusch, S., Khokhar, A., Liu. Y. *Scalable S-to-P Broadcasting on Message-Passing MPPs.* Proceedings of the 25th International Conference on Parallel Processing, I:69-76, August 1996

[Han85] Hansen, P. B., *Brinch Hansen on Pascal Compilers*. Prentice-Hall. 1985

[Han94] Hansen, P. B., *Do hypercubes sort faster than tree machines?*. Concurrency: Practice and Experience, 6(2), 143-151, 1994

[Har94] Harris, T.J., *A Survey of PRAM Simulation Techniques*. ACM Computing Surveys, 26(2), pp. 187-206, 1994

[Har96] Hardwick, J. C. *An Efficient Implementation of Nested Data Parallelism for Irregular Divide-and-Conquer Algorithms*. First International Workshop on High-Level Programming Models and Supportive Environments, 1996

[Hil97] Hill, J., McColl, B., Stefanescu, D., Goudreau, M., Lang, K., Rao, B., Suel, T., Tsantilas, Bisseling, R., *BSPLib: The bsp Programming Library*. Technical Report PRG-TR-29-97, Oxford University Computing Laboratory, 1997. www.bsp-worldwide.org

[Hoa61] Hoare, C. A. R. *Algorithm 64: Quicksort*. Communications of the ACM, 4, 321, 1961

[Hoa71] C. A. R. Hoare *Proof of a Program: Find*. Communications of the ACM, 14, 39-45, 1971

[Hoa85] Hoare, C.A.R. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science, 1985

[Hpf93] High Performance Fortran Forum. *High Performance Fortran Language Specifications*. 1993

[Hud90] Hudak, P., Wadler, P. *Report on the functional programming language HASKELL*. Technical Report, Yale University, 1990

[Iac] *Instituto de Astrofísica de Canarias*. http://www.iac.es/

[Ibm] *RS/6000 SP System*. http://www.rs6000.ibm.com/hardware/largescale/index.html

[Ibm95] IBM *PVMe for AIX User's Guide and Subroutine Reference Version 2, Release 1*. Document number GC23-3884-00. IBM Corp. 1995

[Inm88] INMOS Limited. *Occam2 Reference Manual.* Prentice Hall International Series in Computer Science, 1988

[Inm90] INMOS Limited. *ANSI C toolset reference manual.* INMOS limited, 1990

[Jaj92] JáJá J., *An Introduction to Parallel Algorithms*. Addison Wesley, 1992

[Kel94] Keller, J., Paul, W.J., Scheerer, D. *Realization of PRAMs: Processor Design*. Proc. WDAG94, 8th Int. Workshop on Distributed Algorithms, Springer LNCS, 857 pp. 17-27, 1994

[Kel95] Keller, J., Walle, T., *A Note on Implementing Combining Networks*. Information Processing Letters, 55(4), pp. 195-200, 1995

[Kes94] Kessler, C., Seidl, H. *Making FORK practical.* Technical Report 95-01, SFB-124-C1, Universität Saarbrücken, 1994

[Kes95a] Kessler, C., Seidl, H. *Fork95 language and compiler for the SB-PRAM*. Proc. of the 5th Workshop on compilers for Parallel Computers, pp. 28-40, 1995

[Kes95b] Kessler, C., Seidl, H. *Integrating Synchronous and Asynchronous Paradigms: The Fork95 Parallel Programming Language.* Proc. of MPPM-95 Conference on Massively Parallel Programming Models. IEEE CS Press, Berlin, 1995

[Kes96] Kessler, C., Träf, J. *A library of Basic PRAM Algorithms and its Implementation in FORK*. Proc. of the 8th Annual ACM Symp. on Parallel Algorithms and Architectures. ACM Press, pp. 193-195, 1996

[Kes97a] Kessler, C., Seidl, H. *The Fork95 Parallel Programming Language: Design, Implementation, Application*. Int. Journal of Parallel Programming 25(1) pp. 17-50, 1997

[Kes97b] Kessler, C., Seidl, H. *Language Support for Synchronous Parallel Critical Sections* Proc. of APDC-97 Int. Conf. on Advances in Parallel and Distributed Computing, IEEE CS Press, 1997

[Kes98] Kessler, C. *fcc: Fork95 Compiler Reference Manual.* Universität Trier, 1998

[Lei92] Leighton, F.T. *Introduction to Parallel Algorithms and Architectures. Arrays. Trees. Hypercubes*. Morgan Kaufman Pub., 1992

[Len92] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W., Gupta, A., Hennessy, J., Horowitz, M., Lam, M. *The Stanford DASH multiprocessor*. IEEE Computer, 25(3), pp. 63-79, 1992

[Leo92] León, C. *Un Compilador Pascal Paralelo para el Modelo PRAM*. Memoria de licenciatura. Universidad de La Laguna, 1992

[Leo93] León C., de Sande F., García F. Rodríguez, C., Martín, C. *Un entorno de programación para la enseñanza de algoritmos paralelos*. Jornadas sobre Nuevas Tecnologías en la Enseñanza de las Matemáticas, Valencia, 1993

[Leo95a] León, C., Sande, F., Rodríguez, C., García F. *A PRAM Oriented Language.* Proc. of the Euromicro Workshop on Parallel and Distributed Processing IEEE CS Press, 182-191, 1995

[Leo95b] León C., de Sande F., Rodríguez C. y García F. *Implementación de un entorno paralelo orientado al modelo PRAM*. Actas de las I Jornadas de Informática, Tenerife, 1995

[Leo96a] Léon C. *Diseño e Implementación de Lenguajes Orientados al Modelo PRAM*. Memoria de Tesis. Universidad de La Laguna. Centro Superior de Informática. 1996

[Leo96b] León, C., Rodríguez, C., Sande, F. y García, F. *Un Estudio Comparativo de Lenguajes Orientados al Modelo PRAM*. Actas de las II Jornadas de Informática. Granada, 1996

[Leo96c] León C., Rodríguez C., Sande F., García F. *Lenguajes con Paralelismo de Datos Anidado*. VII Actas de las Jornadas de Paralelismo. Santiago de Compostela, 1996

[Leo97] León, C., Rodríguez, C., García, F., Sande, F. *A PRAM Oriented Programming System*. Concurrency: Practice and Experience. 9(3), pp. 163-179, 1997

[Li93] Li, X., Lu, P., Schaefer, J., Shillington, J., Wong, P.S., Shi, H. *On the Versatility of Parallel Sorting by Regular Sampling*. Parallel Computing, 19, pp. 1079-1103, 1993

[Lun98] Luna, J. *llcc98: Una herramienta eficiente para el desarrollo de algoritmos paralelos*. Proyecto de final de Carrera. Centro Superior de Informática. Universidad de La Laguna, 1998

[Mil90] Milner, R., Tofte, M., Harper, R. *The Definition of Standard ML*. MIT Press, Cambridge, Mass., 1990

[Mpi] *Message Passing Toolkit: MPI Programmer's Manual,* SR-2197, 1.1, Online Software Publications Library. http://www.ciemat.es:8080/library/all/2197_1.1

[Mpi94] Message Passing Interface Forum. *MPI: A message-passing interface standard*. International Journal of Supercomputing Applications and High Performance Computing, 8(3/4), 1994

[Nes] *NESL: A Parallel Programming Language.* http://www.cs.cmu.edu/~scandal/nesl.html

[Pfa] Pfannenstiel, W., Dahm, M., Chakravarty, M.M.T., Jähnichen, S., Keller, G., Schröer, F. W. Simons, M.. *Aspects of the Compilation of Nested Parallel Imperative Languages*. J. Darlington, ed.: International Conference on Massively Parallel Programming Models, IEEE CS Press, 1997

[Pfi85] Pfister, G.F., Brantley, W.C., George, D.A., Harvey, S.L., Kleinfelder, W.J., McAuliffe, K.P., Melton, E.A., Norton, V.A., Weiss, J. *The IBM research processor prototype (RP3): Introduction and architecture*. Proc. of the 1985 Int. Conf. on Parallel Processing, IEEE, pp. 764-771, 1985

[Pra] http://ocre. csi.ull.es/pcgull/html/pram.html

[Pre85] Preparata, F. P., Shamos, M. I.. *Computational Geometry. An Introduction*. Springer-Verlag, New York, 1985

[Psr] Constantes experimentales para el problema PSRS. ftp://ftp.csi.ull.es/pub/ parallel/BSPWB/psrs.htm

[Qui90] Quinn, M.J., Hatcher, P.J. *Data Parallel programming on multicomputers*. IEEE Software 7(5), 1990

[Ran88] Ranade, A.G., Bhatt, S.N., Lennart, S., *The Fluent Abstract Machine.* Proc. 5th MIT Conference on Advanced Research in VLSI, MIT Press, pp. 71-93, 1988

[Ran91] Ranade, A.G., *How to emulate shared memory*. Journal of Computer and System Sciences, 42(3), pp. 307-326, 1991

[Rod98a] Roda J.L., Rodríguez C., Sande F., Morales D.G., Almeida F. *h-relation Models for Current Standard Parallel Platforms* Euro-Par'98 Parallel Processing. 4th International Euro-Par Conference. Springer Verlag LNCS, 1470, pp. 234-243, 1998

[Rod98b] Roda J.L., Rodríguez C., Sande F., Morales D.G., Almeida F. *A New Model for the Analysis of Asynchronous Parallel Algorithms*. 5th European PVM/MPI Users' Group Meeting. Springer-Verlag LNCS, 1497, pp. 387-394, 1998

[Rod97a] Rodríguez, C., Sande, F., León, C., García, F., *Expanding the Message Passing Library Model with Nested Parallelism*. Proc. of the 20th World Occam and Transputer User Group Technical Meeting. IOS Press.Enschede, The Netherlands, 1997

[Rod97b] Rodríguez, C., Sande, F., León, C., y García, F. *Ampliación del Modelo de Librería de Paso de Mensajes con Paralelismo Anidado*. III Jornadas de Informática. Cádiz, 1997

[Rod97c] Rodríguez C., Sande F., Roda J.L., Almeida F.,García Forte L., Sánchez, M., León C., García F., González D., *Very High Level and Efficient Programming with Message Passing Libraries*. VIII Jornadas de Paralelismo. Cáceres, 1997

[Rod98c] Rodríguez C., Sande F., León C., García L.. *Providing Nested Parallelism and Load Balancing on Message Passing Libraries*. 6th IEEE Euromicro Workshop on Parallel and Distributed Processing. Madrid, pp. 402-408, 1988

[Rod98d] Rodríguez, C., Sande, F., León, C. *Extending Processor Assignment Statements*. 2nd IASTED International Conference on Parallel and Distributed Systems. Vienna, Acta Press, pp. 228-233, 1998

[Rod98e] Rodríguez C., Sande F., León C., Coloma I., Delgado A. *Load Balancing and Processor Assignment Statements.* Euro-Par'98 Parallel Processing. 4th International Euro-Par Conference. Springer-Verlag. LNCS, 1470, pp. 539-544, 1998

[Rod98f] Rodríguez, C., Roda, J.L., Sande, F., Morales, D.G., Almeida, F., León, C. *The Design and Analysis of MPI Algorithms*. Actas de las IX Jornadas de Paralelismo. pp. 227-233. San Sebastián, septiembre 1998

[Rod98g] Rodríguez, C., Roda, J.L., Sande, F., Morales, D.G., Almeida, F., León, C. *The Design and Analysis of Parallel Combinatorial Algorithms Using Standard Message-Passing Libraries*. Proc. of PAREO'98, 1998

[Rod99a] Roda, J.L., Sande, F., León, C., González, J.A., Rodríguez, C. *The Collective Computing Model.* PDP'99 7th Euromicro Workshop on Parallel and Distributed Processing. (Aceptado para su publicación)

[Rod99b] Rodríguez, C., Sande F., León C., García, L. *Parallelism and Recursion in Message Passing Libraries: An Efficient Methodology.* Concurrency: Practice and Experience. Pendiente de Publicación.

[San93] Sande, F. *Realización de un entorno de Programación Paralela de alto nivel sobre Redes de Transputers*. Memoria de licenciatura. Universidad de La Laguna, 1993

[San96] Sande, F., García, F., León, C., Rodríguez, C. *The ll Parallel Programming System*. IEEE Transactions on Education. 39(4), pp. 457-464, 1996

[Sil] *Silicon Graphics Origin 2000*. http://www.sgi.com/origin/2000/

[Sbp] *SB-PRAM* http://www-wjp.cs.uni-sb.de/sbpram/sbpram.html

[Smi78] Smith, B.J. *A pipelined shared resource MIMD computer*. Proc. of the 1978 Int. Conf. on Parallel Processing, IEEE, pp. 6-8, 1978

[Skj] Skjellum A., Smith S.G., Doss N., Leung A.P., Morari M. *The Design and Evolution of Zipcode.* Parallel Computing, 20(4), pp. 565-596, 1994

[Swt] *Software Engineering Research Group*. http://swt.cs.tu-berlin.de/swt_e.html

[Top] *TOP500 Supercomputing Sites*. http://www.top500.org/top500.list.html

[Val90] Valiant, L. G. *A Bridging Model for Parallel Computation*. Communications of the ACM, 33(8), pp. 103-111, 1990