UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Departamento de Sistemas Informáticos y Programación

PROGRAMACIÓN FUNCIONAL PARALELA EFICIENTE EN EDEN

MEMORIA PARA OPTAR AL GRADO DE DOCTOR
PRESENTADA POR

Fernando Rubio Diez

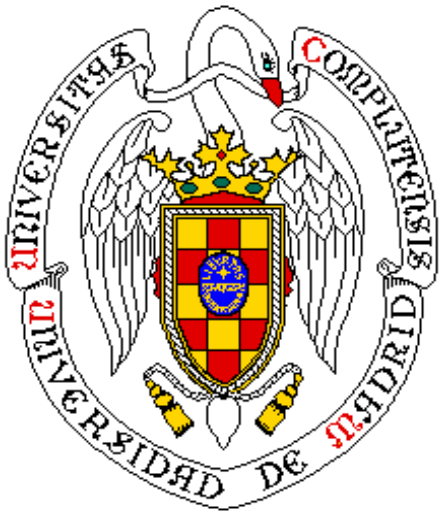
Bajo la dirección del Doctor:

Ricardo Peña Marí

Madrid, 2001

ISBN: 84--669-1852-3

Programación funcional paralela eficiente en Eden



TESIS DOCTORAL

Fernando Rubio Diez
Departamento de Sistemas Informáticos y Programación
Facultad de Informática
Universidad Complutense de Madrid
Octubre 2001

Programación funcional paralela eficiente en Eden

Memoria presentada para obtener el grado de
Doctor en Informática

Fernando Rubio Diez

Dirigida por el profesor

Ricardo Peña Marí

Departamento de Sistemas Informáticos y Programación
Facultad de Informática
Universidad Complutense de Madrid

Octubre 2001

Agradecimientos

Quisiera comenzar el presente trabajo mostrando mi agradecimiento a aquellas personas que han contribuido directamente al desarrollo del mismo. Primeramente me gustaría expresar mi gratitud a Ricardo Peña, sin el cual no hubiera podido llevarse a cabo la investigación. Él ha sido mi tutor desde que comencé mi participación en el proyecto Eden en Noviembre de 1996, y desde entonces siempre se ha mostrado dispuesto a clarificar mis innumerables preguntas acerca del lenguaje. Asimismo, fue él quien me propuso el área de trabajo, y quien ha dirigido esta tesis. Por todo ello, reitero mi agradecimiento por el tiempo que me ha dedicado durante los últimos años.

Las conversaciones con los distintos integrantes del grupo Eden me han proporcionado una realimentación muy productiva a la hora de desarrollar el trabajo expuesto en esta tesis. Por ello no quiero dejar pasar la oportunidad de mencionar a todos los miembros que forman o han formado parte de la sección española del proyecto Eden: Alberto de la Encina, Luis Antonio Galán, Félix Hernández, Mercedes Hidalgo, Manuel Núñez, Yolanda Ortega, Pedro Palao, Cristóbal Pareja, Ricardo Peña y Clara Segura. Del mismo modo, quisiera agradecer la realimentación obtenida de los miembros alemanes del grupo Eden: Silvia Breitinger, Ulrike Klusik, Rita Loogen y Steffen Priebe. En particular, quisiera mostrar mi más sincero agradecimiento a Ulrike Klusik, quien durante mis primeros pasos en el proyecto Eden siempre estuvo dispuesta a resolver todas mis dudas acerca de la implementación real del compilador de Eden.

También quisiera agradecer la paciencia mostrada por Natalia López para escuchar estoicamente mis explicaciones sobre mi tesis, a pesar de que su área de investigación es muy diferente. Gracias a que trabaja en otra área, sus comentarios siempre me han resultado muy útiles a la hora de mejorar la forma de explicar las ideas fundamentales de mi trabajo.

El trabajo desarrollado por Hans-Wolfgang Loidl y Kevin Hammond en el desarrollo de GranSim, forma la base de mi propio trabajo en el simulador Paradise, por lo que estoy en deuda con ellos. En particular, estoy doblemente en deuda con Hans-Wolfgang Loidl, ya que junto con él y Norman Scaife he tenido la oportunidad de realizar comparaciones precisas entre los lenguajes GpH, PMLS y Eden.

Por último, quisiera mencionar a Mario Weilguni, persona a la que no conozco, pero a la que agradezco enormemente el desarrollo de la versión 1.1 de Shisen-Sho, distribuida con Red Hat 6.0. Gracias a dicho programa han sido más llevaderos los meses de Julio y Agosto dedicados a escribir esta tesis.

Índice General

Capítulo 1

Introducción

Clásicamente, en el área de la programación paralela el principal objetivo ha sido conseguir la mayor eficiencia posible, explotando al máximo los recursos *hardware* disponibles. Para ello, se utilizaban lenguajes de muy bajo nivel que eran dependientes de la arquitectura concreta en la que se ejecutaran. Por dicho motivo, portar las aplicaciones paralelas a nuevas arquitecturas requería reescribir en gran medida las aplicaciones.

Con el paso del tiempo, la comunidad paralela ha procurado incrementar el grado de abstracción, buscando lenguajes que faciliten la portabilidad a distintas arquitecturas. El objetivo ideal es que no sólo sea posible portar el código, sino que al portarlo se mantenga también la eficiencia. Esto último resulta particularmente complejo en un entorno paralelo, ya que las topologías internas de los distintos sistemas pueden ser radicalmente distintas.

Por su parte, clásicamente, la programación funcional se ha centrado en proporcionar métodos genéricos de programación, sin dar demasiada importancia a la eficiencia en tiempo de ejecución, sino sólo a la eficiencia en tiempo de desarrollo y mantenimiento de los programas. Así, el paradigma ha mostrado ser muy útil desde el punto de vista docente (véase por ejemplo [LLR93, JvdBvdH93, Ker95, NPP95, POR99]), pero han sido pocas las aplicaciones reales que se han realizado en la industria, donde sólo ha tenido éxito el lenguaje funcional impuro Erlang [AWV93].

En los últimos años, la comunidad funcional ha comenzado a preocuparse en mayor medida por la eficiencia, realizándose tanto mejoras en los procesos de compilación (e.g. [San95, JS98]) como en los entornos de desarrollo. Particularmente relevantes son los perfiladores que permiten detectar y corregir ineficiencias relativas al tiempo de ejecución y al consumo de memoria de los programas (e.g. [RW91, San94, RR96a]), así como los trazadores [CRW01, NS97, SR97, Gil00, PPRS01] que permiten detectar y corregir fácilmente errores en los programas funcionales.

El ámbito de trabajo de la presente tesis se encuentra en la intersección de ambas líneas de evolución: la evolución hacia mayor abstracción de la pro-

gramación paralela, y la evolución hacia mayor eficiencia de la programación funcional. Por tanto, el objetivo principal de esta tesis es obtener un entorno de programación funcional paralela eficiente, entendiendo el concepto de eficiencia en sus dos vertientes:

- La eficiencia en tiempo de ejecución, tanto secuencial como paralela, de los programas desarrollados.
- La eficiencia en tiempo de desarrollo y mantenimiento de los programas, expresado en tiempo de programador.

Esta tesis muestra cómo obtener buenos resultados en ambos aspectos del concepto de eficiencia. Para ello, se mejorará la eficiencia del lenguaje Eden. Eden [BLOP98] es un lenguaje funcional paralelo que extiende al lenguaje funcional perezoso Haskell [PH99] con construcciones para permitir especificar programas paralelos, incluyendo la definición y concreción de procesos. Para lograr el objetivo de la mejora de su eficiencia, trabajaremos en tres frentes distintos:

1. Transformaciones automáticas en tiempo de compilación que mejoren la eficiencia del código generado.
2. Herramientas que proporcionen realimentación al usuario, informándole sobre las principales ineficiencias de sus programas.
3. Metodología de programación que conjuge rapidez de desarrollo con eficiencia de los resultados.

Compilador. Lógicamente, el mecanismo ideal para mejorar la eficiencia de cualquier programa es disponer de un compilador altamente inteligente, capaz de optimizar el código por sí mismo. Por dicho motivo, uno de los objetivos de la tesis será conseguir un compilador de Eden que optimice el código lo más posible, tanto en lo referente a las partes puramente secuenciales como a las que expresan el paralelismo. Tendremos como objetivo que nuestro compilador genere código secuencial tan eficiente como el más eficiente de los compiladores funcionales perezosos disponibles. Asimismo, nuestro objetivo también será conseguir que el código generado explote en la mayor medida posible todo el paralelismo que aparezca en la especificación de los programas de entrada. El Capítulo 5 se dedicará a introducir mejoras en el compilador de Eden.

Herramientas. Una de las principales razones de la falta de eficiencia de los programas paralelos es el hecho de que para los programadores resulta complicado razonar acerca del comportamiento paralelo de sus programas. Así pues, el programador necesita herramientas que le proporcionen información sobre el comportamiento real de su programa, de forma que pueda

determinar no sólo la presencia de ineficiencias, sino también sus causas. Por ello, otro de los objetivos de la tesis será diseñar e implementar herramientas fáciles de usar y que proporcionen dicha realimentación. Al incluir dichas herramientas en nuestro entorno de programación, conseguiremos no sólo el objetivo de facilitar el desarrollo de aplicaciones eficientes, sino también el objetivo de reducir los tiempos de desarrollo. En el Capítulo 4 se diseña e implementa una de estas herramientas, que permite simular ejecuciones de Eden sobre arquitecturas paralelas.

Esqueletos. Para conseguir el objetivo de desarrollar aplicaciones rápidamente, es fundamental disponer de una metodología de desarrollo. Los lenguajes funcionales perezosos han demostrado sobradamente su capacidad para desarrollar prototipos secuenciales rápidamente, por lo que no deberemos preocuparnos por las partes secuenciales, aunque sí de las paralelas. El desarrollo de aplicaciones paralelas es una tarea compleja, así que es fundamental disponer de mecanismos de abstracción que permitan obviar los detalles de implementación de bajo nivel. Durante la última década ha surgido el concepto de *esqueleto*, que permite especificar un esquema general de resolución de una familia de aplicaciones paralelas (como por ejemplo la familia de los problemas divide y vencerás), de modo que un programa basado en esqueletos simplemente necesite invocar al esqueleto o conjunto de esqueletos necesarios para crear la estructura paralela, sin preocuparse por como implementar la estructura paralela sobre la arquitectura subyacente. Un objetivo prioritario de la tesis será permitir el uso de esqueletos dentro de nuestro entorno de programación. Para ello, no sólo se proporcionará una librería de esqueletos para que pueda utilizarla el usuario, sino que también se desarrollará una metodología que permita a los programadores especificar sus propios esqueletos sin abandonar el lenguaje Eden. Gracias al uso de esqueletos, podrá lograrse no sólo el objetivo de desarrollar los programas más rápidamente, sino que también se mejorará la eficiencia en tiempo de ejecución, pues los usuarios normales podrán utilizar esqueletos previamente optimizados por programadores expertos. Los capítulos 6 y 7 confirmarán la utilidad de la metodología de Eden basada en esqueletos.

A continuación se detalla la estructura del resto de la tesis, donde los dos primeros capítulos serán básicamente recopilaciones de trabajos realizados por terceras personas, mientras que la parte original de la tesis comenzará en el Capítulo 4. En cada uno de los capítulos originales de la tesis se incluirá una sección final de conclusiones en la que se resumirán los resultados obtenidos.

El Capítulo 2 contiene una introducción a los conceptos básicos en programación paralela, así como sendos resúmenes sobre el estado del arte en las áreas de programación con esqueletos y programación funcional paralela. El objetivo de dichos resúmenes es encuadrar el contenido de la tesis dentro

del área general de investigación a la que pertenece, de modo que pueda apreciarse en mejor medida cuáles son las aportaciones del autor de la tesis.

El Capítulo 3 describe el lenguaje funcional paralelo Eden, que será el que se utilice durante el resto de la tesis para conseguir el objetivo de programación funcional paralela eficiente. Dicho capítulo no sólo describe el lenguaje propiamente dicho, sino que también da detalles acerca de la estructura del compilador, ya que serán relevantes en capítulos posteriores.

En el Capítulo 4, y tras una descripción del estado del arte en el área de las herramientas de caracterización del rendimiento en lenguajes funcionales paralelos, comienzan las contribuciones originales de la tesis. En dicho capítulo se diseña e implementa un simulador de arquitecturas paralelas capaz de predecir el comportamiento sobre distintas arquitecturas de los programas escritos en Eden. Esta herramienta proporcionará gráficas que representen la evolución en el tiempo del grado de paralelismo de los programas, por lo que permitirá detectar y corregir ineficiencias en la paralelización. Gracias a ello no sólo se mejorará la eficiencia de los programas, sino que también se reducirá el tiempo de desarrollo de los mismos, pues se empleará menos tiempo en tratar de corregir ineficiencias. Cabe resaltar que la herramienta no sólo será útil para el programador de Eden, sino también para los desarrolladores del compilador, pues facilitará la detección de ineficiencias en la implementación del compilador.

El Capítulo 5 se centra en la optimización automática de programas Eden. En él se desarrollan transformaciones que se aplican en tiempo de compilación para mejorar la eficiencia de los programas. Así, el objetivo de mejorar la eficiencia se consigue sin necesidad de involucrar al programador. Las principales aportaciones del capítulo serán: (1) definir un esquema de compilación que facilite futuras transformaciones; (2) proponer e implementar una transformación automática que incrementa el grado de paralelismo mediante un lanzamiento impaciente de procesos; (3) definir un conjunto de transformaciones que permitan realizar un análisis de *bypassing* que reduzca la cantidad de comunicaciones entre los procesos; y (4) reutilizar las optimizaciones secuenciales que realiza el compilador más eficiente de Haskell.

El Capítulo 6 presenta el punto de unión de la programación funcional paralela con la programación basada en esqueletos. En él se muestra cómo puede utilizarse el lenguaje Eden para implementar esqueletos de forma clara y concisa, a la vez que eficiente. De este modo, se implementa una librería de esqueletos que permitirá obtener las ventajas propias de la programación con esqueletos, pero sin heredar sus restricciones a la hora de expresar nuevos esquemas paralelos. Cabe resaltar que ésta es la primera implementación de una librería de esqueletos que se realiza en algún lenguaje funcional paralelo, y que cualquier programador Eden podrá extenderla con nuevos esqueletos que cubran sus áreas de trabajo habituales, lo que no puede hacerse en los lenguajes habituales basados en esqueletos.

El Capítulo 7 presenta un banco de pruebas formado por varios pro-

gramas Eden que utilizan distintos esqueletos de los definidos en el capítulo anterior. El objetivo es mostrar que, efectivamente, la programación en Eden es eficiente tanto en tiempo de programador como en tiempo de cómputo. El tiempo de desarrollo de las aplicaciones será muy bajo gracias tanto al alto nivel de abstracción del lenguaje como al entorno de desarrollo, que incluye una librería de esqueletos y un conjunto de perfiladores tanto para los cómputos secuenciales como para los paralelos. La eficiencia del lenguaje queda contrastada en la Sección 7.3, donde se realiza una comparación con otros dos lenguajes paralelos representativos del estado del arte en las áreas de programación paralela con esqueletos y programación funcional paralela.

Finalmente, el Capítulo 8 contiene las conclusiones generales de la tesis, así como las líneas de investigación que tiene previsto explorar en el futuro el autor.

Parte de los resultados que se muestran en esta tesis aparecen publicados en [**HPR00**, **PPRS00**, **KPR01**, **KLPR01**, **PR01**, **LOP+02**], mientras que otros resultados están actualmente sometidos a un proceso de revisores en [**PRS01**, **LRS+01**]. Para facilitar la identificación de estas publicaciones originales de la tesis, utilizaremos letra negrita para resaltarlas siempre que aparezcan tanto en el texto como en la bibliografía final.

En la medida de lo posible, a lo largo de la tesis se utilizarán términos castellanos en lugar de sus equivalentes anglosajones. Ahora bien, determinadas palabras se mantendrán en su versión original inglesa por falta de un término español comúnnmente aceptado. En dichos casos, la palabra aparecerá siempre en letra itálica. Una excepción será el término array, que se utilizará como una palabra normal de nuestro lenguaje. Son muchas las traducciones que aparecen en la literatura, incluyendo "vectores" y "matrices" para distinguir si son unidimensionales o no, "formaciones", "arreglos", etc. El autor considera que ninguno de estos términos es satisfactorio, y que a falta de una palabra propia de nuestro idioma, es mejor introducir en nuestro lenguaje el término original.

Capítulo 2

Preliminares y trabajo relacionado

El presente capítulo contiene una introducción al área de trabajo en la que se sitúa esta tesis, mostrando tanto los conceptos básicos como el trabajo relacionado. Presenta el estado del arte del área de la programación paralela en sus vertientes de programación con esqueletos y de programación funcional. Por tanto, no forma parte de las contribuciones originales de la tesis.

2.1 Conceptos básicos

En esta sección se introducen algunos conceptos básicos sobre programación paralela, con el objetivo de poder utilizarlos posteriormente a lo largo de la tesis. El lector conocedor del área probablemente preferirá pasar directamente a la siguiente sección.

Aceleración absoluta

La aceleración absoluta obtenida al paralelizar un programa se define como el tiempo de ejecución de la versión secuencial, dividido por el tiempo de ejecución de la versión paralela, donde el tiempo de ejecución secuencial debe obtenerse con el algoritmo más eficiente conocido, que no tiene que coincidir necesariamente con el algoritmo utilizado en la versión paralela:

$$A_{abs} = \frac{T_{sec}}{T_p}$$

Lógicamente, dicho valor puede depender del número de procesadores utilizados en la paralelización. Por dicho motivo, los datos sobre aceleración se muestran siempre como un par: número procesadores-aceleración. Por ejemplo, puede decirse que se ha obtenido una aceleración de 10.5 usando 20

procesadores, pero no tiene sentido decir simplemente que la aceleración es 10.5.

Podría pensarse que en lugar de utilizarse un par, podría verse como un porcentaje de aprovechamiento de los procesadores, es decir, como la fracción $\frac{aceleracion}{numero\ de\ procesadores}$ En este caso perderíamos información relevante, pues dicho porcentaje puede variar en gran medida dependiendo del número de procesadores (normalmente, a más procesadores menor suele ser el grado de aprovechamiento).

Aceleración relativa

La escalabilidad es la capacidad de adaptación de un mismo programa paralelo a un número creciente de procesadores. Es deseable, aunque no siempre posible, que no sea necesario modificar el código fuente del programa cada vez que se quiera ejecutar sobre un número distinto de procesadores, sino que éste se adapte automática y eficientemente a los recursos de que disponga.

Con el objetivo de comparar la escalabilidad de distintos programas, se define un nuevo tipo de aceleración (llamada *aceleración relativa*), que resulta de dividir el tiempo de ejecución del programa paralelo ejecutado sobre un monoprocesador entre el tiempo de ejecución de ese mismo programa cuando se ejecuta con n procesadores:

$$A_{rel} = \frac{T_1}{T_p}$$

Así pues, la única diferencia con la aceleración absoluta es el punto de referencia: en vez de ser la mejor versión secuencial, es la versión paralela pero ejecutada sobre un monoprocesador. Por tanto, el referente incluye los sobrecostes debidos a la introducción de paralelismo en el programa: creación de hebras, comunicaciones entre ellas, planificación de tareas, etc.

Paralelismo medio

El paralelismo medio (o grado de paralelismo), es la media aritmética del número de procesadores que se encuentran activos en cada instante de la ejecución. Expresa el grado de utilización de los procesadores disponibles. Por ejemplo, 13.5 con 15 procesadores expresa que de los 15 procesadores disponibles se está perdiendo completamente la capacidad de cómputo correspondiente a 1.5 procesadores. Más concretamente, siendo T_p el tiempo de ejecución del programa paralelo, podemos definir el paralelismo medio como

$$G_p = \frac{1}{T_p} \int_0^{T_p} num\_activas(t)\ dt$$

Nótese que el grado de paralelismo es notablemente diferente de la aceleración, puesto que se considera que un proceso también está activo cuando

está realizando un trabajo que no se realizaría en la versión monoprocesador (tanto secuencial como paralela), ya sea por motivos de especulación o simplemente por motivos de los sobrecostes debidos a la creación de procesos y a las comunicaciones entre los procesadores.

Isoeficiencia

La eficiencia de un algoritmo para un determinado número de procesadores es igual a la aceleración absoluta dividida por el número de procesadores. La función de isoeficiencia de un algoritmo mide en qué grado debe incrementarse el tamaño del problema de entrada para que se mantenga constante la eficiencia a medida que se incrementa el número de procesadores disponibles. Así, una función de isoeficiencia de $O(P)$ indica que la escalabilidad del algoritmo es buena, pues basta con que el tamaño del problema crezca linealmente con el número de procesadores para que se mantenga la eficiencia. Sin embargo, una isoeficiencia $O(P^2)$ indica que la escalabilidad es mala, pues el tamaño del problema debe crecer mucho para mantener la eficiencia.

Ley de Amdahl

Uno de los aspectos que limitan más seriamente la aceleración que puede alcanzarse al paralelizar un programa, es la cantidad de código inherentemente secuencial que existe en el algoritmo. Y esto es así incluso cuando pueda parecer que dicha parte secuencial es pequeña en comparación con la parte paralela. Este hecho se refleja claramente en la ley de Amdahl ([Amd67]), que establece cuál es la aceleración máxima que puede obtenerse para un algoritmo cualquiera. Así, si disponemos de p procesadores, y la proporción (en tanto por 1) inherentemente secuencial del algoritmo es f, entonces:

$$aceleracion\ maxima\ = \frac{1}{f + \frac{1\ -\ f}{p}}$$

Por ejemplo, si el 20% del cómputo que debe realizar nuestro algoritmo es inherentemente secuencial, entonces aunque dispusiéramos de un número infinito de procesadores, a lo sumo aceleraríamos el cómputo en un factor de 5, puesto que $\frac{1}{0.2+\frac{0.8}{\infty}}$ es 5.

Esta ley debe tenerse siempre en mente a la hora de interpretar los resultados obtenidos al paralelizar distintos algoritmos, pues de lo contrario, en muchas circunstancias podría parecer que el grado de paralelismo obtenido no es satisfactorio, cuando realmente puede estar muy próximo al máximo teórico alcanzable.

Tipo de comunicaciones

El tipo de comunicaciones que pueden establecerse entre distintos procesos puede clasificarse del modo siguiente:

Síncrona: tanto el receptor como el emisor del mensaje deben coincidir en determinados puntos de sus respectivas ejecuciones para que la comunicación pueda tener lugar.

Asíncrona: el emisor no necesita esperar a que el receptor del mensaje se encuentre en el punto de la ejecución en el que espere recibir el mensaje. Las comunicaciones asíncronas se dividen a su vez en dos tipos:

- Cuando el emisor debe esperar a recibir peticiones del receptor se dice que las comunicaciones son de tipo *pull*.
- En caso de que el emisor envíe datos sin necesidad de que el receptor los haya demandado se dice que las comunicaciones son de tipo *push*. Este mecanismo permite evitar tiempos de espera en el receptor, pues dispondrá de los datos antes de necesitarlos.

Especulación

Cuando se realiza un trabajo que realmente no va a ser necesario para el resultado final, se dice que dicho trabajo es *especulativo*. En programación funcional secuencial, el empleo de evaluación perezosa garantiza que en cada momento se computa sólo lo que realmente hace falta, por lo que no existe especulación. Ahora bien, dado que en un entorno paralelo disponemos de varios procesadores, suele ser útil comenzar cómputos antes de que se sepa si serán necesarios o no, con el fin de emplear procesadores que estarían ociosos en caso contrario.

Ubicación de tareas

La decisión sobre qué procesador debe computar cada una de las tareas que se generen, puede tomarse estática o dinámicamente. En el primero de los casos la decisión se toma en tiempo de compilación, para lo cual es preciso conocer en dicho momento tanto el número de tareas como el de procesadores. Por ello suele ser necesario restringir el tipo de algoritmos paralelos que pueden implementarse. Cuando la ubicación de tareas es dinámica, existen dos posibilidades para repartir nuevas tareas que se creen en tiempo de ejecución:

Activamente: el procesador que ha generado la nueva tarea decide qué procesador debe encargarse de resolverla.

Pasivamente: la nueva tarea permanece en el procesador hasta que otro procesador le pida trabajo.

En cualquiera de los dos casos, el objetivo debe ser que el reparto de carga entre los distintos procesadores sea lo más homogéneo posible, con el fin de que todos los procesadores terminen sus cómputos más o menos al mismo

tiempo. De este modo se aprovechará la capacidad de cómputo de todos ellos durante casi todo el tiempo de la ejecución.

Planificadores de tareas

Habitualmente cada procesador tiene varias tareas a realizar. Si el tiempo de cómputo del procesador se reparte equitativamente entre todas sus tareas, se dice que el planificador es *justo* o *expropiativo*, mientras que en caso contrario es *injusto*. Los planificadores justos deben realizar cambios de contexto frecuentes, para dar paso a las distintas tareas (de ahí el nombre de *expropiativos*), por lo que son menos eficientes que los injustos. Por su parte, los injustos asignan el procesador a una tarea hasta que dicha tarea finaliza o se bloquea, y sólo en ese momento se cambia de contexto para dar paso a otra tarea. En caso de que la tarea que se esté ejecutando sea especulativa, puede desaprovecharse mucho cómputo, resultando en una notable pérdida en la eficiencia general. De hecho, en el caso peor el cómputo de la tarea puede ser infinito, por lo que el programa puede que no termine debido a la elección de un planificador de tareas injusto, mientras que usando uno justo sí terminaría.

Tipos de paralelismo

Clásicamente se han distinguido dos tipos de paralelismo: paralelismo de datos y paralelismo de tareas. La idea subyacente en el paralelismo de datos es realizar operaciones globales sobre grandes estructuras de datos, de modo que las operaciones sobre los elementos individuales de la estructura puedan realizarse simultáneamente. Por tanto, con paralelismo de datos la tarea principal del programador consiste en decidir cómo distribuir las estructuras de datos entre los procesadores.

En el caso del paralelismo de tareas la fuente del paralelismo está en el control. La labor del programador es repartir entre los procesadores las distintas actividades que deban realizarse durante la ejecución. Por ejemplo, al paralelizar un esquema divide y vencerás, el programador debe especificar qué procesos se encargarán de realizar tareas de división del problema en subproblemas, y qué procesos resolverán tareas básicas. De entre los paralelismos de tareas cabe destacar el paralelismo de tipo *stream*. En dicho tipo existen listas de tareas de tamaño desconocido, que se generan dinámicamente, y deben repartirse también dinámicamente entre los distintos procesadores. Un ejemplo típico de este tipo de paralelismo es el paralelismo de tubería, en el que debe aplicarse una serie de etapas de cómputo a cada uno de los elementos de una lista de tamaño desconocido.

Tipos de arquitecturas paralelas

La clasificación más extendida de arquitecturas *hardware* fue propuesta en 1972 por Flynn [Fly72], y distingue cuatro tipos de arquitecturas:

SISD (*Single Instruction Single Data*). Se corresponde con las arquitecturas monoprocesador clásicas.

SIMD (*Single Instruction Multiple Data*). Una misma instrucción se ejecuta simultáneamente sobre múltiples datos de entrada. El ejemplo típico de este tipo de arquitecturas son los procesadores vectoriales, que permiten aplicar en paralelo operaciones sobre arrays.

MIMD (*Multiple Instruction Multiple Data*). Distintas instrucciones se ejecutan en paralelo, cada una de ellas operando sobre datos diferentes. Es el tipo de paralelismo más general, que viene representado por las arquitecturas multiprocesador.

MISD (*Multiple Instruction Single Data*). Es el tipo de arquitecturas menos convencional. De hecho, no existen máquinas reales que se adapten completamente a este tipo, si bien algunos autores consideran que los procesadores segmentados se encuadran dentro de esta categoría, pues sobre cada dato de entrada se ejecutan distintas operaciones en cada una de las etapas.

La clasificación de Flynn no sólo ha servido para clasificar *hardware*, sino que ha influido también en algunas clasificaciones de programas paralelos, generando los conceptos de programación SPMD (*Single Program Multiple Data*) y MPMD (*Multiple Program Multiple Data*). En principio, los programas SPMD se corresponden con los programas que sólo explotan paralelismo de datos, mientras que los MPMD cubren el tipo más general de paralelismo. En realidad la única exigencia de SPMD es que el mismo código resida en todos los procesadores, por lo que también puede cubrir cualquier tipo de paralelismo sin más que utilizar los identificadores de los procesadores para realizar bifurcaciones en el código del programa. También existe un equivalente a las arquitecturas MISD, que es el paralelismo de tipo MPSD (*Multiple Program Single Data*), que explota paralelismo de tipo tubería.

2.2 Tipos de lenguajes paralelos

Pueden distinguirse dos tipos de lenguajes en función de quién sea el responsable de detectar el paralelismo:

Lenguajes con paralelismo implícito: el programador sólo escribe programas secuenciales, y es el sistema quien detecta cómo puede extraerse paralelismo, empleando para ello análisis de dependencias de datos.

Lenguajes con paralelismo explícito: el programador es el responsable de decidir qué tareas deben realizarse en paralelo.

Dado que el diseño de programas paralelos es una tarea compleja, los lenguajes puramente implícitos no permiten obtener buenas aceleraciones, especialmente en arquitecturas en las que las latencias son altas comparadas con la capacidad de cómputo de los procesadores.

Los lenguajes de programación paralelos también pueden clasificarse en función del nivel de abstracción en el que deba moverse el programador a la hora de desarrollar sus programas. Una de las clasificaciones más relevantes es la realizada por Susanna Pelagatti en [Pel98], que clasifica los lenguajes de paralelismo explícito del siguiente modo:

Completamente abstractos: el programador sólo debe decidir el algoritmo paralelo a emplear, pero la implementación paralela es responsabilidad del sistema.

Parcialmente abstractos: el programador no sólo decide el algoritmo a emplear, sino que también debe fijar el grafo de procesos que se empleará, así como el reparto de tareas entre los procesos. Este nivel se subdivide en otros dos:

- **De alto nivel:** el programador sólo define explícitamente el grafo de procesos y el reparto de tareas entre los procesos, pero no las comunicaciones y sincronizaciones.
- **De bajo nivel:** el programador también debe manejar explícitamente las comunicaciones y sincronizaciones entre los procesos.

Dependientes de la máquina: el programador debe controlar todos los detalles de la implementación, incluyendo los dependientes de la arquitectura subyacente.

Ejemplos del nivel más abstracto son los lenguajes *data-parallel*, como High Performance Fortran (HPF) sin directivas [For93] o NESL (que se describirá en la sección siguiente). Dichos lenguajes proporcionan operaciones primitivas sobre estructuras de datos que pueden estar distribuidas físicamente en distintos procesadores, pero cuya distribución es completamente transparente para el programador, que las ve como si fueran estructuras locales. Por ejemplo HPF proporciona arrays distribuidos, permite aplicar una misma función a cada uno de los elementos del array, o combinar los valores de todos ellos mediante operadores suministrados por el programador.

Entre los lenguajes parcialmente abstractos de alto nivel se encuentran todos los lenguajes funcionales-paralelos que describiremos en la Sección 2.4.2, así como los lenguajes *data-parallel* que incluyen directivas para la descomposición explícita de las estructuras de datos. Cuando se incluyen directivas,

HPF vuelve a ser el ejemplo prototípico de este tipo de lenguajes. Por ejemplo, la directiva `PROCESSOR` permite especificar la estructura del grafo de procesadores, `DISTRIBUTE` permite especificar cómo se distribuye un array entre los distintos procesadores, mientras que `ALIGN` permite alinear dos arrays distribuidos, de modo que se garantice que determinadas partes del primero estarán en el mismo procesador que determinadas partes del segundo.

Entre los lenguajes parcialmente abstractos de bajo nivel se encuentran lenguajes como Linda [CG89], en los que el programador debe realizar las comunicaciones explícitamente mediante funciones como *send* y *receive*. En este mismo nivel se encuentran las librerías de paso de mensajes que no son específicas de una arquitectura concreta, como las conocidas librerías PVM [PVM93, GBDJ94] y MPI [Mes94, GLS94], que facilitan la portabilidad a distintas máquinas paralelas. Por dicho motivo, estas librerías están siendo utilizadas como lenguaje destino de la compilación de otros lenguajes paralelos de más alto nivel.

En el nivel más bajo de la jerarquía se encuentran lenguajes para arquitecturas específicas, como por ejemplo el uso de Occam sobre transputers [MSK86].

La elección del nivel de abstracción al que desarrollar los programas paralelos requiere establecer un compromiso entre eficiencia, portabilidad y capacidad expresiva. Los niveles más bajos permiten explotar más eficientemente los recursos *hardware*, pero no puede portarse a otras arquitecturas ni el código ni mucho menos la eficiencia. Los niveles más altos permiten portar trivialmente el código de unas arquitecturas a otras, pero sólo puede portarse la eficiencia cuando se restringe el tipo de paralelismo que puede explotarse. Por ejemplo, los lenguajes *data-parallel* permiten programar a un alto nivel de abstracción, consiguiendo portabilidad de código y eficiencia, pero restringiendo notablemente el tipo de programas que pueden desarrollarse, pues no sólo se restringen a paralelismo de datos, sino que las estructuras de datos deben ser siempre estáticas, y en muchas ocasiones incluso se restringen a estructuras unidimensionales.

2.3 Lenguajes de esqueletos

Los lenguajes basados en esqueletos aumentan el nivel de abstracción en el que se escriben los programas paralelos. La idea principal consiste en proporcionar un conjunto de esquemas básicos de paralelización, de modo que el programador se limite a concretar dichos esquemas para su problema específico, pero sin tener que descender al detalle de cómo implementar eficientemente los esquemas paralelos básicos. En principio, los lenguajes de esqueletos pueden clasificarse como completamente abstractos, pues normalmente sólo requieren que se especifique el algoritmo paralelo, pero su implementación es responsabilidad del sistema de soporte. Ahora bien, en la

práctica algunos lenguajes de esqueletos requieren la intervención del usuario cada vez que se quiere portar un programa, manteniendo la eficiencia, a una nueva arquitectura, por lo que dichos lenguajes se clasificarían como parcialmente abstractos de alto nivel. A continuación se exponen los conceptos básicos del enfoque, así como las aproximaciones más significativas que se han llevado a cabo en el área.

2.3.1 Conceptos básicos

En programación secuencial han surgido históricamente dos abstracciones importantes que han contribuido a crear una metodología de programación adecuada para abordar la construcción de grandes programas: la *abstracción funcional* y la *abstracción de datos*. Tanto en una como en otra se pueden distinguir dos aspectos:

- Su *especificación*, que determina su comportamiento observable para un usuario potencial.
- Su *implementación* en términos de abstracciones más simples, que determina su eficiencia. Para una misma especificación son posibles en general varias implementaciones.

Un esqueleto representa una abstracción en programación paralela y en general son posibles varias implementaciones del mismo atendiendo a factores tales como la arquitectura paralela subyacente, la granularidad de las tareas creadas, la estrategia de reparto de carga entre los procesadores, etc. Cada implementación tendrá en general una eficiencia diferente. Una de las características más importantes de un esqueleto es que debe ser posible predecir el *coste paralelo* de cada una de sus implementaciones. Por coste paralelo se entiende el tiempo de ejecución del algoritmo desde que el primer procesador empieza a trabajar hasta que termina de trabajar el último. La fórmula matemática que describe dicho coste se denomina *modelo de coste* de la implementación [Fos95].

La especificación describe como mínimo el valor de los datos producidos por el esqueleto en función del valor de los datos de entrada, esto es, su comportamiento funcional. Pero, normalmente, la especificación determina también la familia de problemas a la que es aplicable dicho esqueleto. Por ejemplo, existe un esqueleto *divide y vencerás* paralelo aplicable a problemas que admitan una función `split` para partir un problema en subproblemas, una función `combine` que combine los resultados de los subproblemas, etc. Afortunadamente, para casi todos los esqueletos existe una versión secuencial del mismo, como es el caso de *divide y vencerás*.

Resumiendo, un esqueleto consiste en las dos siguientes piezas de información:

Especificación Consta del *tipo* del esqueleto y del *algoritmo secuencial* que describe el comportamiento funcional del esqueleto.

Implementaciones Cada una de ellas consta a su vez de dos partes: (a) un **algoritmo paralelo** que describe la topología de procesos creada y la asignación de trabajo a cada uno de ellos, y (b) un **modelo de coste** que describe el tiempo paralelo esperado del algoritmo.

Los modelos de coste están parametrizados por ciertos valores que dependen o bien del *problema* concreto a resolver, o bien del *sistema de soporte a la ejecución*, o bien de la *arquitectura* hardware subyacente.

Los modelos de coste son en general sencillos de calcular si se conoce en qué orden y dónde se crean los procesos. Para describir el tiempo paralelo sólo es necesario tener en cuenta las actividades que suceden en el *camino crítico* del algoritmo y el coste de cada una. El camino crítico lo constituyen las actividades que necesariamente han de ejecutarse en secuencia para poner a trabajar a todos los procesadores, y las que se necesitan desde que acaba la última subtarea hasta obtener el resultado final. De las actividades que pueden ejecutarse en paralelo en varios procesadores, es necesario tomar en consideración la que termina más tarde. Para una panorámica de modelos de coste, puede consultarse [Ham00].

Las principales ventajas que ofrece la programación con esqueletos provienen de su alto nivel de abstracción, que no sólo permite desarrollar los programas más rápidamente y con menos errores, sino que también permite portar los programas conservando gran parte de la eficiencia, ya que para pasar a otras arquitecturas paralelas conservando la eficiencia basta con implementar eficientemente los esqueletos básicos, sin modificar las aplicaciones concretas.

2.3.2 Principales lenguajes de esqueletos

Cole El término "esqueleto" fue acuñado por Murray Cole en su tesis doctoral [Col88] (posteriormente publicada en [Col89]). En dicha tesis, propuso un lenguaje paralelo en el que sólo existían cuatro esqueletos básicos de paralelización. Cualquier aplicación debía poder expresarse en términos de dichos esqueletos, aunque se reconocía que el conjunto de esqueletos podría incrementarse a más de cuatro a medida que se detectara la conveniencia de incluir nuevos esquemas paralelos. Para cada uno de los esqueletos, se proponía una implementación sobre una rejilla bidimensional de *transputers*. El conjunto de esqueletos predefinido era el siguiente:

- FDCC *Fixed Degree Divide&Conquer*: Paraleliza el conocido esquema de programación divide y vencerás. Cada uno de los subproblemas en los que se divide el problema inicial se resuelve en paralelo, aplicando recursivamente el esquema FDCC. Para simplificar la implementación,

el número de problemas en los que se subdivide cada problema se restringe de forma que sea una constante del esqueleto.

- IC *Iterative Combination*: Este algoritmo sirve para aquellos problemas que se resuelven iterando una función hasta que se cumple una determinada propiedad. Básicamente, partiendo de un conjunto de datos `x`, la función `pares` decide qué pares de datos deben combinarse, mientras que `combinar` los combina.

```
ic :: (b -> a -> a) -> (a -> b) -> (a -> Bool) -> a -> a
ic combinar pares test x
  | test x    = x
  | otherwise = ic combinar pares t (combinar (pares x) x)
```

- C *Cluster Skeleton*: Este esqueleto es una generalización de los dos anteriores. Sirve para aquellos problemas que pueden dividirse en subproblemas (como FDCC), siendo dichos problemas paralelizables utilizando IC.

- TQ *Task Queue*: Este esqueleto sirve para aquellos problemas en los que inicialmente la entrada se divide en muchas tareas iniciales, de modo que dichas tareas puedan ser resueltas independientemente. Dichas tareas iniciales se introducen en una cola, y cada uno de los trabajadores puede extraer una tarea de la misma y resolverla, devolviendo su resultado. Durante la resolución de una tarea, los trabajadores pueden crear nuevas tareas, que se añaden dinámicamente a la cola de tareas global, para que cualquier trabajador pueda resolverla.

El principal mérito de la aproximación de Cole es el hecho de haber sido el primero en plantear un lenguaje puramente basado en esqueletos. Ahora bien, sus esqueletos son excesivamente complejos. Por ejemplo, no se proporciona ningún ejemplo de aplicación en el que sea útil el esqueleto C. Esta complejidad se debe en parte a que el lenguaje no permitía anidar esqueletos: como todo programa debía ajustarse a alguno de los esqueletos, los esqueletos complejos podían adaptarse más fácilmente a los problemas concretos.

Por otro lado, otra restricción del enfoque es que no hay ninguna forma de especificar cuántos procesadores usar, ya que se usan todos, lo cual puede ser contraproducente dependiendo de la relación cómputo/comunicaciones.

Darlington A diferencia del enfoque de Cole, esta aproximación [DFH+93, DGT93] incorpora un conjunto de esqueletos más simples, y en ellos no se incluyen detalles de implementación. Dichos detalles quedan relegados a las implementaciones que se hagan de los esqueletos para cada arquitectura destino.

La propuesta del grupo de Darlington se basa en tres pilares: los esqueletos, sus modelos de coste para las distintas arquitecturas, y un conjunto de reglas de transformación de programas.

Partiendo de una especificación del problema en términos de los esqueletos, las reglas de transformación permiten al programador optimizar la implementación para la arquitectura destino deseada. Para ello, deben consultarse los modelos de coste de los distintos esqueletos.

Existen dos tipos de reglas: las que traducen un esqueleto a otro, y las que permiten restringir los recursos a emplear (por ejemplo, reduciendo el número de etapas de una tubería para que coincida con el número de procesadores disponibles).

El conjunto de esqueletos es el siguiente:

- Una tubería en la que cada etapa de la misma puede ejecutarse en paralelo:

```
pipe :: [a -> a] -> (a -> a)
pipe = fold .
```

- Una simplificación del esqueleto TQ de Cole, en el que no se permite que los trabajadores creen nuevas tareas dinámicamente:

```
farm :: (a -> b -> c) -> b -> ([a] -> [c])
farm f env = map g
  where g x = f x env
```

  donde `env` representa un conjunto de datos fijos que van a ser necesarios para el cómputo de todas las tareas.

- Un divide y vencerás en el que no se restringe el número de subproblemas en el que se divide el problema original:

```
dc :: (a ->Bool) -> (a ->b) -> (a ->[a]) -> ([b] ->b) -> a -> b
dc test solve split combine x
  | test x    = solve x
  | otherwise = (combine .
                 map ((dc test solve split combine) . split) x
```

- El esqueleto RAMP *Reduce and Map Over Pairs* captura aquellos problemas en los que un conjunto de objetos evolucionan combinándose entre sí. Por cada uno de los elementos x de la lista se obtiene uno nuevo consistente en una combinación de x con el resto de elementos de la lista:

```
ramp :: (a -> a -> b) -> (b -> b -> b) -> [a] -> [b]
ramp f g xs = map h xs
  where h x = fold g (map (f x) xs)
```

- El esqueleto DMPA *Dynamic Message Passing Architecture* permite especificar cualquier topología de comunicación entre procesos. Es necesario especificar el estado inicial de cada proceso, qué cómputo realiza y cómo interacciona con cada uno de los otros procesos. Todo par de procesos puede interaccionar entre sí mediante paso de mensajes.

Al igual que la propuesta de Cole, el anidamiento de esqueletos sigue sin estar permitido: sólo pueden especificarse estructuras planas de paralelización.

El enfoque transformacional que fundamenta el enfoque es meramente metodológico, ya que no es un proceso automático, ni existe ningún tipo de soporte semiautomático para la optimización de los programas. La responsabilidad es completamente del programador, cuyo único punto de apoyo son los modelos de coste existentes para los distintos pares esqueleto/arquitectura. Además, dado que los modelos de coste para las distintas arquitecturas son distintos, portar un programa a otra arquitectura requiere repetir el proceso manual de optimización del algoritmo paralelo.

Los puristas de la programación basada en esqueletos critican fundamentalmente el último esqueleto, debido a que permite cualquier topología. De hecho, se considera una "puerta de escape" del mundo de los esqueletos.

La escuela de Heriot-Watt El lenguaje SkelML [Bra92, Bra93, Bra94a, Bra94b], desarrollado por Bratvold, es una versión paralela de un subconjunto del lenguaje funcional impaciente Standard ML [RTHM97]. SkelML identifica fuentes potenciales de paralelismo a partir del uso de las siguientes funciones de orden superior: `map`, `filter` y `fold`. La implementación paralela se basa en llamadas a Occam2 [Inm88].

Para determinar cuándo debe paralelizar un uso de dichas funciones, y cuándo no debe hacerlo, es preciso realizar una fase previa a la ejecución del programa. En dicha fase se utiliza un perfilador para determinar cuánto cómputo y cuántas comunicaciones se requieren en las aplicaciones de las funciones paralelizables. A partir de dichos datos y a partir de los modelos de coste disponibles para la arquitectura de destino, el compilador es capaz de decidir qué debe paralelizarse. Así pues, es el compilador quien decide en última instancia el paralelismo que se debe extraer.

Al igual que los enfoques de Cole y Darlington, SkelML tampoco permite la composición general de esqueletos. La única composición permitida es la composición en modo tubería, de modo que en cada etapa se utilice un esqueleto distinto. En particular, cuando una tubería está formada por un `fold` y un `map`, el compilador lo optimiza utilizando un esqueleto `foldmap`.

Posteriormente a SkelML, se han desarrollado distintos lenguajes basados en él. La principal aportación de los nuevos diseños ha sido introducir anidamiento de esqueletos.

Por ejemplo, en la tesis de Hamdam [Ham00] se presenta Ektran, un lenguaje funcional muy simple que permite utilizar esqueletos anidados. Siguiendo las ideas de SkelML, el paralelismo se extrae de la aplicación de ciertas funciones de orden superior, que en este caso son `map`, `fold` y `combine` (que representa una tubería). Dichas funciones pueden anidarse sin restricciones, y el compilador es capaz de detectar qué debe paralelizarse y qué no. Las principales restricciones son que el lenguaje base es excesivamente simple para utilizarse en aplicaciones reales (sólo se han desarrollado programas de unas pocas líneas de código), y que la introducción de un nuevo esqueleto implica modificar muy notablemente el compilador.

En la misma línea del trabajo de Hamdam, pero partiendo del lenguaje ML, se encuentra PMLS [MSBK01, SMH01] (Parallel ML with Skeletons), que extrae paralelismo de las funciones `map` y `fold`, y permite anidamiento arbitrario de esqueletos. La implementación de los esqueletos se realiza en C + MPI.

POPE El lenguaje POPE [RS94, SR96] (Paradigm-Oriented Programming Environment) está orientado hacia los algoritmos que se ajustan al esquema SIT (Static Iterative Transformation). Dicho esquema sigue básicamente la misma idea que el esqueleto IC de Cole. Los problemas se resuelven iterando una serie de pasos, donde cada paso puede consistir en operaciones con datos locales, en una multidifusión de datos globales desde el procesador principal, o en la combinación de datos recibidos de distintos procesadores.

POPE extiende Haskell con facilidades para expresar paralelismo iterativo, y genera código C con llamadas a PVM. Reutiliza GRIP [HJ92] para generar código.

SCL Tras su primera aproximación explicada previamente, el grupo de Darlington desarrolló el lenguaje SCL (Structured Coordination Language) [DGTY95a, DGTY95b, ADG+96]. SCL está concebido como un lenguaje de coordinación, de modo que el lenguaje secuencial a emplear puede ser cualquiera. SCL sólo define cómo coordinar la parte paralela de los programas. Para ello utiliza funciones de segundo orden que definen los esqueletos. Básicamente, el lenguaje pretende obtener lo mejor del mundo funcional y del imperativo: utiliza estilo funcional para los esqueletos, mientras que permite que el código secuencial sea imperativo para mejorar la eficiencia.

SCL está orientado hacia paralelismo de datos. Su tipo de datos más importante son los arrays distribuidos. Para manejar estos arrays, el lenguaje incluye distintos tipos de esqueletos:

- Los esqueletos de *configuración* permiten distribuir cómodamente los

arrays, y son muy similares a las directivas de HPF. El esqueleto `partition` transforma un array secuencial en un array paralelo de arrays secuenciales, de modo que cada subarray pueda ubicarse en un procesador distinto. Por su parte, `align` permite relacionar dos arrays distribuidos, estableciendo qué pares de subarrays deben residir en los mismos procesadores. Además de estos esqueletos básicos se incluyen otros que, combinando los básicos, facilitan definir cómo distribuir cualquier conjunto de arrays. Por último, un array distribuido puede convertirse en normal mediante `gather`.

- Los esqueletos *elementales* permiten operar con arrays distribuidos. Se incluyen dos versiones de `map` (para la segunda es relevante el índice de cada elemento), un `fold` y un `scan`. También se incluyen esqueletos tanto para realinear como para modificar arrays.

- Los esqueletos *computacionales* abstraen las estructuras de control más habituales. Incluyen esqueletos `pipe`, `farm` y `dc` como en el enfoque anterior del grupo de Darlington, así como versiones de un `iterUntil` similar al esqueleto IC de Cole. Por último los esqueletos `spmd` y `mpmd` abstraen, respectivamente, las características de los cómputos SPMD y de los MPMD.

A diferencia de los enfoques presentados hasta ahora, SCL sí permite anidamiento de esqueletos. Esto dota de una gran flexibilidad a las soluciones que pueden darse a los problemas. Al igual que en el primer enfoque del mismo grupo, el sistema incluye un conjunto de reglas de transformación de esqueletos que permiten optimizar las implementaciones. En este caso, el enfoque transformacional es semi-automático, ya que el sistema es quien aplica las reglas de transformación, pero en determinados casos es el usuario quien debe especificar qué reglas aplicar y en qué orden hacerlo.

P^{3}L Uno de los lenguajes más relevantes del area es P^{3}L (Pisa Parallel Programming Language) [Pel93, BDO$^+$95, Pel98, Pel02]. El lenguaje proporciona una serie de esqueletos que pueden anidarse sin ningún tipo de restricción. A partir de la especificación dada por el programador, el sistema es capaz de optimizar la estructura de procesos dependiendo de la arquitectura destino.

El lenguaje incluye construcciones para poder expresar tanto paralelismo de tipo *stream* como paralelismo de datos. Así, para el primer tipo incluye los esqueletos `farm` y `pipe`, similares a los introducidos en la primera aproximación del grupo de Darlington. Para el paralelismo de datos se incluyen esqueletos `map`, `reduce` y `comp`. El esqueleto `map` permite realizar operaciones sobre arrays, facilitando la especificación de cómo deben distribuirse dichos arrays entre los procesadores. El esqueleto `reduce` permite aplicar un *fold*

sobre un array, es decir, reduce un array mediante un operador binario asociativo. Por su parte, `comp` es la versión data-parallel de `pipe`, utilizándose para componer distintas aplicaciones de esqueletos data-parallel. Además de los esqueletos anteriores, también se incluye el esqueleto `loop` (similar al IC de Cole), para expresar paralelismo iterativo tanto de tipo *stream* como paralelismo de datos. El último esqueleto que se proporciona es un esqueleto para realizar ejecuciones secuenciales. De esta forma se consigue que absolutamente todas las construcciones del lenguaje sean esqueletos.

En principio, la programación de las partes secuenciales se podrían hacer en cualquier lenguaje secuencial, si bien la versión actual sólo permite utilizar C.

La primera fase que se realiza para optimizar un programa P^3L consiste en efectuar ejecuciones reales de las partes secuenciales, con el objetivo de poder concretar los modelos de coste con los datos específicos del problema en curso[1].

A partir de la especificación dada por el programador, el compilador de P^3L primero optimiza la estructura de procesos aplicando reglas generales de reescritura de árboles de procesos. Tras optimizar la estructura, se optimizan los recursos (procesadores) asignados a cada esqueleto. Tras una primera asignación de recursos en la que no se tienen en cuenta el número máximo de procesadores, en una segunda pasada se van eliminando recursos de los distintos esqueletos hasta que pueda ejecutarse en la máquina real. El objetivo final es minimizar el tiempo de ejecución, minimizando también (en la medida de lo posible) el número de procesadores.

Nótese que el proceso de transformaciones es completamente automático. Así pues, portar un programa a otra arquitectura sólo requiere realizar ejecuciones reales para obtener información de *profiling*, y después utilizar las transformaciones automáticas del compilador para que optimice la estructura de acuerdo a dicha información.

Para añadir un esqueleto al lenguaje, es preciso modificar el compilador. En particular, para cada una de las arquitecturas de destino es preciso añadir a las bibliotecas la siguiente información:

- Código fuente de los esqueletos.
- Estructura del grafo de procesos e información sobre cómo ubicar los procesos en procesadores.
- Modelo de coste.
- Para poder componer esqueletos apropiadamente hay que añadir información sobre:

[1] Si no se quieren realizar dichas ejecuciones, el sistema también permite que el usuario diga directamente cuáles son los tiempos.

- Cómo enlazar eficientemente las subpartes paralelas.
- Cómo combinar los modelos de coste de las subpartes paralelas para obtener un modelo global.

Actualmente P^3L es capaz de generar código para dos tipos de arquitecturas: Meiko CS1 (arquitecturas en malla tipo transputers), y arquitecturas que soporten PVM (y que por tanto permitan conexiones entre cualquier par de procesadores). Para cada tipo de máquina existe una máquina abstracta sobre la que se desarrollan los modelos de coste, que determina qué parámetros son relevantes.

Una de las principales restricciones del lenguaje es que sólo proporciona tipos estáticos. Así pues, es preciso conocer en tiempo de compilación el tamaño de todos los arrays. Esto facilita la predicción automática de costes del programa, pero restringe el abanico de problemas que pueden resolverse. En particular, no pueden resolverse problemas que requieran utilizar listas de tamaño indefinido. De hecho, el único tipo de datos del que se extrae paralelismo son los arrays n-dimensionales. Por tanto, realmente no existen *streams* en el sentido de un flujo continuo de datos de longitud desconocida, sino que el paralelismo de tipo *stream* sólo se puede utilizar con arrays de tamaño fijo.

Además de la restricción anterior, P^3L presupone que la máquina paralela a utilizar está dedicada por entero a su aplicación. Así pues, no puede utilizarse para programar tareas en máquinas que ejecutan varias aplicaciones simultáneamente.

HDC Otro lenguaje reciente basado en esqueletos es HDC [HLG+99, HL00] (Higher-order Divide and Conquer), desarrollado en la universidad de Passau. HDC toma como lenguaje de partida un subconjunto de Haskell, cambia el orden de evaluación perezoso por el estricto, y añade esqueletos para permitir la evaluación en paralelo. Como su nombre indica, HDC pretende explotar fundamentalmente el paralelismo de tipo divide y vencerás. De hecho, incluye una jerarquía de cinco esqueletos para el esquema divide y vencerás. Así, para cada problema se elegirá el esqueleto que mejor se comporte en ese caso.

Aunque el lenguaje se basa en el esquema divide y vencerás, también se incluyen esqueletos para `map`, `filter`, `scan` y `red` (fold). Ahora bien, dichos esqueletos también siguen un enfoque divide y vencerás. Por ejemplo, cuando el compilador decide paralelizar un `map`, divide la lista de tareas en dos sublistas, y asigna cada una de ellas a un procesador distinto. En cada uno de ellos se repite el proceso de división en dos listas, hasta que se decida resolver la lista de tareas en secuencial.

Por tanto, en principio sólo se permiten comunicaciones jerárquicas: cada bloque sólo comunica con su maestro o sus descendientes en modo divide y vencerás. Sólo existe una excepción: el lenguaje permite definir estructuras

de datos globalmente distribuidas, y en ese caso se puede usar acceso a memorias remotas para acceder a cualquier dato.

Para añadir un nuevo esqueleto al sistema, es necesario extender el preludio con su definición de tipos, para luego definir el código Haskell que genera la implementación del esqueleto. Dicho código Haskell es realmente una cadena de caracteres que incluye el código C (con llamadas a MPI) preciso para implementar el esqueleto.

Skil El lenguaje Skil [BK95, BK98, Bot98] extiende el lenguaje C con características funcionales: funciones de orden superior, aplicaciones parciales y tipos polimórficos. Estas características se eliminan en tiempo de compilación, generándose código C con llamadas a MPI. El lenguaje C también se extiende incluyendo estructuras de datos distribuidas, permitiendo definir arrays distribuidos (si bien las estructuras distribuidas no pueden anidarse).

Aunque permite paralelismo de tareas, Skil está especialmente orientado hacia el paralelismo de datos. En ese sentido, sus esqueletos permiten operar de forma distribuida con arrays. Así, el esqueleto `create` genera arrays distribuidos, `map` y `zip` permiten operar sobre arrays suponiendo que están idénticamente distribuidos, `fold` combina los elementos de un array distribuido para obtener un único resultado, `permute` permite variar la distribución de un array, y `broadcast` envía un bloque de un array a todos los procesadores.

A diferencia de otros lenguajes imperativos basados en esqueletos (como por ejemplo P^3L), los esqueletos de Skil no son una parte interna del lenguaje, sino que el programador puede tanto reutilizar los esqueletos existentes como crear esqueletos nuevos. Para esto último sólo hay que definir el programa C con las correspondientes primitivas de paso de mensajes. Ahora bien, Skil no permite anidamiento de esqueletos.

2.4 Programación funcional paralela

Durante la última década, se han propuesto diversas extensiones a lenguajes de programación funcionales con la intención de facilitar la especificación de sistemas concurrentes, así como la explotación de paralelismo (tanto introduciéndolo de forma explícita como explotando el paralelismo implícito), todo ello tomando como base un entorno de trabajo funcional. En [HM99] puede encontrarse una recopilación de las principales áreas de investigación que se encuentran abiertas actualmente, mientras que en [TLP01] aparece una descripción detallada de los principales lenguajes paralelos y/o concurrentes que extienden el lenguaje funcional Haskell [PH99].

En el presente trabajo nos centraremos en las extensiones paralelas, no sin antes mencionar brevemente algunos de los lenguajes funcionales concurrentes más relevantes. Cabe mencionar Facile [GMP89], Concurrent ML

[Rep91], Erlang [AWV93] y Concurrent Haskell [PGF96]. Estos lenguajes funcionales concurrentes proporcionan construcciones primitivas para la creación dinámica de procesos concurrentes (mediante *spawn* o *fork*) y para el intercambio de mensajes entre los procesos (*send* y *receive*). Por tanto, la concurrencia se maneja a un nivel de abstracción bastante bajo, lo cual viola el estilo de programación funcional, que aboga por utilizar altos niveles de abstracción. Otros lenguajes como Haskell with Ports [HN01] o GdH (Glasgow distributed Haskell [PTL00]) incluyen también primitivas para el desarrollo de sistemas distribuidos, utilizando también un bajo nivel de abstracción.

Otra aproximación completamente diferente sobre la que se ha trabajado, pretende explotar el paralelismo implícito de los lenguajes funcionales, de modo que se mejore el tiempo de ejecución del programa secuencial paralelizándolo automáticamente. Normalmente, el programador debe introducir anotaciones para optimizar la ejecución paralela, de modo que sea él quien controle que la granularidad sea la adecuada, así como que la ubicación de procesos en procesadores sea óptima. El resto de la sección está dedicada a comentar los lenguajes más relevantes de este enfoque, con la excepción de HDC y la familia de SkelML, que ya fueron descritos en la sección anterior. Una mención especial merecerá el lenguaje GpH (Glasgow Parallel Haskell), debido a que es un lenguaje muy cercano al lenguaje Eden, sobre el cual trata esta tesis. De hecho, la implementación de Eden reutiliza gran parte de la de GpH. Esta circunstancia permite hacer comparaciones muy precisas entre ambos lenguajes, ya que al utilizar la misma tecnología secuencial, las diferencias de eficiencia entre ambos lenguajes dependen fundamentalmente del diseño y metodología de uso de los lenguajes.

2.4.1 Lenguajes basados en arrays

SISAL Uno de los primeros lenguajes funcionales paralelos exitosos fue SISAL (Streams and Iterations in a Single Assignment Language) [CF90, Can92]. SISAL era un lenguaje estricto, sin orden superior y utilizaba notación imperativa, aunque fuera funcional. A pesar de utilizar asignaciones, el lenguaje era funcional debido a su condición de *single assignment* que garantiza que a cada variable sólo se le pueda realizar una asignación.

El principal área de destino del lenguaje era el desarrollo de programas para cálculo científico, donde consiguieron eficiencias competitivas con las del lenguaje Fortran. SISAL no sólo estaba pensado para multiprocesadores, sino que también permitía aprovechar el paralelismo de bajo nivel de los monoprocesadores: al garantizar que las asignaciones eran únicas, permitía detectar fácilmente las dependencias de datos, lo cual facilitaba la mejora de rendimiento en procesadores segmentados.

Desde el punto de vista de paralelismo de multiprocesadores, el lenguaje era implícito tanto en la extracción de paralelismo como en la interacción

de las hebras. El esquema básico era de paralelismo de datos, incluyendo operaciones sobre arrays distribuidos. También permitía extraer paralelismo de tubería comunicando productores y consumidores. Por último, permitía extraer paralelismo evaluando en paralelo subexpresiones independientes.

SAC De forma similar a SISAL, SAC (Single Assignment C) [Sch94, Sch96, Sch98] es un lenguaje funcional estricto, de primer orden, basado en paralelismo de datos, que emplea sintaxis imperativa, y cuya principal área de aplicación es el cómputo numerico. SAC es básicamente un subconjunto funcional de C, en el que se eliminan las variables globales y punteros, pero en el que se añaden operaciones de alto nivel sobre arrays de dimensiones variables. El lenguaje permite definir cualquier tipo de operación sobre arrays mediante el uso de *WITH-loops*, un mecanismo que generaliza la notación de listas intensionales de los lenguajes funcionales para permitir trabajar con arrays de dimensiones y formas desconocidas a priori. Este mecanismo permite aplicar operaciones sobre cualquier subrango de cualquier array, aunque el subrango concreto no se conozca a priori.

NESL NESL [Ble93, BG96, Ble96] es un lenguaje funcional paralelo basado en ML. Es estricto, fuertemente tipado, con paralelismo implícito e interacción implícita entre hebras. El modelo de paralelismo es el de paralelismo de datos, siendo su principal característica permitir el anidamiento de paralelismo.

Los tipos de datos básicos son los arrays unidimensionales (*sequences*). Sólo puede extraerse paralelismo con operaciones sobre este tipo. Así, el uso de un `map` sobre el mismo permite que el cómputo sobre cada componente pueda realizarse en paralelo. Además de `map`, permite el uso de otras muchas funciones paralelas, como para sumar o permutar elementos de una *sequence*. En general, se extrae paralelismo a partir del uso de funciones de orden superior y de construcciones similares a listas intensionales.

Como puede apreciarse, realmente es el programador quien debe especificar explícitamente dónde reside el paralelismo del algoritmo, si bien será el sistema quien decida cómo distribuir realmente el trabajo, y cómo sincronizar las distintas tareas.

El lenguaje no es completamente funcional, ya que no siempre mantiene la transparencia referencial. En particular, la construcción *write* sirve para escribir en paralelo sobre *sequences*, permitiendo que se modifique una misma posición dos o más veces. Dado que no se especifica cuál será la actualización que se conservará, el resultado es no determinista.

2.4.2 Lenguajes basados en anotaciones

Para-functional Programming Una aproximación bastante extendida es la de añadir anotaciones a los programas funcionales para especificar có-

mo ejecutarlos en paralelo. El enfoque original proviene de Para-functional programming, planteado por Paul Hudak [Hud86], e implementado posteriormente [MH95] tomando como lenguaje secuencial Haskell. A cada expresión se le puede asociar, mediante la palabra clave `sched`, una expresión de planificación. Dicha planificación puede definir un orden parcial entre eventos, especificando entre otras cosas qué debe suceder en secuencia y qué puede ser ejecutado en paralelo. En la planificación puede indicarse también en qué procesador concreto debe evaluarse cada parte de una expresión, proporcionando un alto grado de control sobre la implementación paralela.

Dado que la paralelización consiste en añadir anotaciones a los programas secuenciales, en principio puede aislarse completamente la parte paralela de la secuencial. Así, idealmente no sería necesario alterar el código secuencial, si bien en la práctica suele ser necesario hacerlo cuando se quieren obtener buenas aceleraciones.

El principal inconveniente del lenguaje es que el hecho de usar anotaciones de bajo nivel suele conducir a programas oscuros, difíciles de entender.

Concurrent Clean Concurrent Clean [PvE98, NSvP91, Kes95] es un lenguaje funcional perezoso muy similar a Haskell. Permite implementar programas paralelos añadiendo anotaciones a los programas secuenciales. Dichas anotaciones permiten indicar en qué procesador se quiere llevar a cabo la evaluación a forma normal de raíz de una expresión. Los argumentos no los evaluará dicho procesador, sino que deberán serles transmitidos en forma normal de raíz cuando los valores sean demandados. Para ello, cuando un procesador demanda datos de otro, en el procesador propietario de los datos se creará un nuevo proceso encargado de la reducción a forma normal de raíz de los mismos, a menos que ya estuvieran en forma normal. Existe una excepción a este modo de proceder: cuando una función es estricta en alguno de sus argumentos, dicho argumento se reduce a forma normal completa antes de que se cree el proceso remoto para la evaluación de la aplicación.

Nótese que sólo se transmiten valores en forma normal, de modo que el trabajo nunca se duplica en distintos procesadores.

Además de las anotaciones, el lenguaje incorpora como primitiva una función que devuelve en qué procesador está siendo evaluado un valor dado. La combinación de esta función y de las anotaciones permite un alto grado de control sobre la topología de procesos a crear.

Caliban La versión inicial de Caliban fue desarrollada por Paul Kelly [Kel87, Kel89], y posteriormente Frank Taylor lo redefinió y reimplementó [Tay97, KT99], pero manteniendo el esquema básico del lenguaje.

Siguiendo la idea de [GC92] de distinguir el modelo de cómputo del de coordinación, Caliban utiliza un lenguaje para especificar los cómputos y otro distinto para la coordinación de las tareas, siendo ambos lenguajes funciona-

les. El código de coordinación se añade mediante anotaciones que especifican cómo ejecutar en paralelo programas Haskell (el lenguaje de cómputo). Dichas anotaciones son una descripción declarativa del grafo de procesos a crear. Por cada proceso hay que indicar qué tareas va a realizar y qué arcos le van a comunicar con los otros procesos. Un proceso puede tener varias tareas a realizar, en cuyo caso debería repartir el tiempo entre ellas mediante un planificador justo y expropiativo[2] para garantizar la evolución de todos los cómputos.

Aunque el lenguaje de cómputo es perezoso, el lenguaje de coordinación introduce impaciencia para poder obtener paralelismo. Así, la evaluación de las salidas de un proceso se realiza aunque no exista demanda, de modo que el productor y el receptor puedan trabajar en paralelo. Dado que los procesos se comunican mediante *streams*, el consumidor puede utilizar los datos a medida que se van produciendo, sin necesidad de esperar a la finalización del cómputo completo, incrementándose el grado de paralelismo.

La principal restricción del lenguaje reside en el hecho de que todas las anotaciones se interpretan en tiempo de compilación, por lo que sólo pueden especificarse estructuras de procesos estáticas. Así, por ejemplo, al definir una granja de procesos, el número de trabajadores debe conocerse en tiempo de compilación. Por tanto, si quiere utilizarse el programa con distintos números de procesadores disponibles es necesario recompilar para cada nueva configuración.

2.4.3 GpH

GpH (Glasgow Parallel Haskell [THJ+96]) es una extensión del lenguaje Haskell que añade un nuevo combinador de composición paralela `par`, que al ser utilizado junto a la composición secuencial `seq` permite controlar el grado de paralelismo de los programas. La semántica de estos combinadores es la siguiente:

```
⊥ `seq` y = ⊥
x `seq` y = y
x `par` y = y
```

Operacionalmente, `seq` fuerza la evaluación a forma normal débil[3] de su primer argumento antes de evaluar el segundo argumento, mientras que `par` indica que la evaluación a WHNF de su primer argumento puede realizarse en paralelo con la evaluación del segundo argumento, aunque será en tiempo

[2] En la implementación actual el planificador no es expropiativo, por lo que no puede garantizarse la terminación del cómputo global.

[3] Una clausura está en forma normal débil cuando tiene un constructor en cabeza, o cuando es una lambda abstracción. En lo sucesivo usaremos WHNF (Weak Head Normal Form).

de ejecución cuando se decida si se crea o no una nueva hebra para evaluar el primer argumento de `par`. Normalmente, el primer argumento será necesario para la evaluación del segundo, pues de lo contrario se estaría realizando una evaluación completamente especulativa. Nótese que el uso de `par` es completamente transparente desde el punto de vista semántico, es decir, los resultados obtenidos serán los mismos independientemente de que se introduzcan o no combinadores `par`.

Ejemplo

A modo de ejemplo, consideremos el siguiente programa para calcular números de Fibonacci:

```
parfib :: Int -> Int
parfib 0 = 1
parfib 1 = 1
parfib n = nf2 `par` (nf1 `seq` (nf1+nf2+1))
           where nf1 = parfib (n-1)
                 nf2 = parfib (n-2)
```

Las definiciones de los casos base son evidentes, pero el caso general merece una explicación más detallada. Debe leerse como: puede crearse una nueva hebra para evaluar `nf2`, pero `nf1` se evaluará en el procesador actual antes de realizar la suma final. Si se creó una nueva hebra para `nf2`, entonces será la responsable de evaluarlo y de comunicar el resultado a la hebra principal. Si no se creó la hebra, entonces todo el trabajo lo realizará la hebra principal.

Realmente, el uso de las anotaciones `par` y `seq` proporciona una gran flexibilidad a la hora de realizar programas paralelos, pero tiene como contrapartida el hecho de que la labor de programación resulta demasiado laboriosa, y el comportamiento paralelo de los programas es difícilmente comprensible.

Implementación

Buena parte de la eficiencia de GpH se debe a que está implementado sobre el eficiente compilador GHC (Glasgow Haskell Compiler [JHHP92, Pey96]). De hecho, la implementación de GpH sólo modifica el sistema de soporte a la ejecución o *runtime system* (en adelante RTS) de GHC para introducir las estructuras necesarias para gestionar la distribución paralela del cómputo, conservándose por completo todas las fases de compilación de GHC.

El primer punto de la ejecución en el que se diferencian el RTS de GHC y el de GpH es en la evaluación de `par`. La traducción de las anotaciones `par` consiste en crear una *semilla* para evaluar el primer argumento, y

continuar evaluando el segundo[4]. Una semilla representa un trabajo que potencialmente puede realizarse en un procesador independiente.

Para conocer las semillas existentes, cada procesador necesita disponer de una lista de semillas. Asimismo, también dispone de una lista con las hebras que está evaluando localmente, de modo que cuando deja de evaluarse la hebra en curso (porque termine su cómputo o porque se bloquee a la espera de datos), se continúa con la evaluación de otra de las hebras de la lista local, realizándose el correspondiente cambio de contexto. Realmente, en vez de almacenar una lista de hebras, se almacena una lista de TSOs[5], siendo cada TSO un registro que contiene toda la información que se necesita sobre una hebra para poder realizar el cambio de contexto.

Ahora bien, en ningún caso el planificador de tareas interrumpe la ejecución de una hebra, sino que una hebra se ejecuta hasta que termina o hasta que se bloquea, es decir, el planificador es no-expropiativo.

En cuanto al reparto de carga entre procesadores, cuando un procesador no tiene ninguna hebra local que pueda ejecutar, si tiene alguna semilla local la convierte en hebra, y comienza a evaluarla. Si tampoco tiene ninguna semilla local, entonces envía mensajes al resto de procesadores (siguiendo un orden aleatorio) pidiendo que le envíen alguna de sus semillas. Si algún procesador dispone de semillas, le envía una de ellas, para lo cual debe enviarle una copia de la parte del grafo de clausuras que es necesaria para poder evaluar el cómputo asociado a la semilla. Finalmente, si nadie le envía ninguna semilla, el procesador permanece ocioso. Nótese que en ningún caso se permite que una hebra migre de un procesador a otro, sino que sólo pueden migrar las semillas.

La sincronización entre las distintas hebras viene determinada únicamente por las dependencias de datos: si una hebra necesita un dato que está produciendo otra hebra, entonces se bloquea hasta que haya sido producido (nótese que GpH no permite duplicar trabajo). Por ejemplo, si en

```
nf2 `par` (nf1 `seq` (nf1+nf2+1))
```

se ha generado una hebra para `nf2`, y el cómputo de `nf1` ha terminado antes que el de `nf2`, entonces la hebra encargada de hacer la suma debe bloquearse en espera de que la otra hebra termine de evaluar `nf2`.

Las comunicaciones son asíncronas, es decir, una vez que el enviante manda el mensaje, puede olvidarse por completo del mismo. Para establecer las comunicaciones entre los distintos procesadores, se utiliza la conocida librería de paso de mensajes PVM, lo cual facilita notablemente la portabilidad de la implementación. Además, para mejorar la eficiencia, los datos no se envían uno a uno, sino que se empaquetan por bloques, con el objetivo de

[4]Es decir, al evaluar `e1 `par` e2`, se generaría una semilla para `e1`, mientras se prosigue el cómputo de `e2`.

[5]Thread State Object.

reducir el impacto negativo que pueden provocar altas latencias entre los procesadores.

Estrategias

Con el objetivo de incrementar el grado de abstracción de los programas GpH, se introduce el concepto de *estrategias* [THLP98]. Las estrategias sirven para controlar el grado de evaluación de los cómputos. Son simples funciones que demandan la evaluación de su argumento hasta cierto grado, y que siempre devuelven `()` para indicar que la salida no es relevante. Se aplican mediante la función `using`, que primero aplica la estrategia al valor de entrada y luego devuelve el valor:

```
type Strategy a = a -> ()
using :: a -> Strategy a -> a
using x s = s x `seq` x
```

Un ejemplo simple de estrategia es la que demanda la espina de una lista de valores. En dicha estrategia, el ajuste de patrones fuerza a que aparezcan explícitamente todas las constructoras `(:)`, pero no se crea demanda para los valores de la lista:

```
spine :: Strategy [a]
spine []     = ()
spine (x:xs) = spine xs
```

Dado que las estrategias son simplemente funciones Haskell, pueden ser parámetros de otras estrategias. De esta forma es fácil definir estrategias sobre estructuras de datos, teniendo como parámetro la estrategia a aplicar sobre cada dato concreto. Por ejemplo, la siguiente estrategia permite implementar un `map` en paralelo:

```
parList :: Strategy a -> Strategy [a]
parList strat []     = ()
parList strat (x:xs) = strat x `par` parList strat xs

parMap s f xs = map f xs `using` parList s
```

El uso de estrategias permite implementar programas paralelos con un mayor grado de abstracción que cuando se emplean directamente los operadores `seq` y `par`. Así, idealmente permiten que el código secuencial sea completamente independiente del código de coordinación paralela. Ahora bien, en la práctica, para programas complejos suele ser necesario modificar la estructura del programa secuencial para obtener mejores aceleraciones.

Capítulo 3

El lenguaje Eden

Este capítulo describe el lenguaje funcional paralelo Eden, en el cual se basa esta tesis doctoral. El autor de la tesis no intervino en el diseño básico del lenguaje, sino que se incorporó al grupo Eden cuando el lenguaje ya existía, siendo su principal aportación al diseño de Eden la introducción del lanzamiento impaciente de procesos. La parte en la que ha intervenido más activamente el autor de la tesis ha sido en la implementación del lenguaje, pues junto a Ulrike Klusik ha sido el implementador del compilador. Ulrike Klusik ha implementado el *back-end*, mientras que el autor de la tesis ha implementado el *front-end* del compilador de Eden.

3.1 Fundamentos de Eden

El lenguaje Eden [BLOP98, BLOP96, BLOP97] extiende el lenguaje funcional perezoso Haskell con construcciones para permitir el desarrollo de programas tanto concurrentes como paralelos. Siguiendo la clasificación descrita en la Sección 2.2, Eden es un lenguaje paralelo parcialmente abstracto de alto nivel, es decir, el programador es el responsable de fijar el grafo de procesos y el reparto de tareas entre los mismos, mientras que las comunicaciones se realizan de forma implícita. Las principales características del lenguaje son las siguientes:

- Los procesos se comunican intercambiando valores a través de canales de comunicación modelados (fundamentalmente) mediante listas perezosas.
- La comunicación es implícita y asíncrona de tipo *push*.
- Permite la introducción de cómputos especulativos.
- Permite el desarollo de programas reactivos.
- Puede realizar cómputos no-deterministas.

- No asume la existencia de una memoria compartida, sino que está diseñado para trabajar eficientemente sobre arquitecturas distribuidas.

3.1.1 Procesos

Eden distingue entre *abstracciones de procesos* (que especifican mediante un estilo funcional el comportamiento de futuros procesos) y *concreciones de procesos* (en las que se proporcionan valores de entrada a las abstracciones de procesos, con el objetivo de crear procesos que se ejecuten realmente). Así pues, la relación entre las abstracciones y las concreciones de procesos es la misma que entre las λ-abstracciones y la aplicación de las mismas una vez que reciben los parámetros reales.

A este respecto, cabe resaltar que las abstracciones de procesos son valores de primera clase, es decir, pueden usarse igual que cualquier otro valor: pueden utilizarse como parámetros de una función, pueden almacenarse en estructuras de datos, pueden ser el resultado devuelto por una función, etc.

Un proceso que toma como entradas $in_1, \ldots, in_m$ y produce como salidas $exp_1, \ldots, exp_n$, se especifica mediante una expresión de *abstracción de proceso* de la siguiente forma:

process $(in_1, \ldots, in_m)$ -> $(exp_1, \ldots, exp_n)$
where $ecuacion_1$ ... $ecuacion_r$

cuyo tipo es `Process` $(\tau_1, \ldots, \tau_m)$ $(\widetilde{\tau_1}, \ldots, \widetilde{\tau_n})$, donde `Process` es un constructor de tipos binario predefinido, y $\tau_1, \ldots, \tau_n$ y $\widetilde{\tau_1}, \ldots, \widetilde{\tau_n}$ son, respectivamente, los tipos de las entradas y de las salidas. La parte `where` es opcional, y se utiliza para definir funciones auxiliares y subexpresiones comunes que se utilizan en la definición del proceso.

Realmente, no es necesario que se nombren explícitamente todos y cada uno de los canales de entrada y todas y cada una de las expresiones de salida, sino que para especificar los canales de entrada puede utilizarse cualquier ajuste de patrones (incluyendo una única variable), mientras que la salida del proceso puede ser cualquier expresión. La única restricción impuesta es que los tipos sean correctos:

process *entradas* -> *expresion*
where *ecuaciones*

Un proceso puede tener como entrada (respectivamente, salida) un único canal, una tupla de canales, o una lista de canales. Por tanto, para inferir tipos correctamente es preciso aplicar ciertos convenios:

- Si el tipo inferido para `a` (respectivamente, para `b`) en `Process a b` es una tupla, entonces cada uno de los componentes de dicha tupla se considera un canal independiente.

- Si un canal tiene tipo `[a]` (es decir, es una lista), entonces se tratará como una lista (potencialmente infinita) que transmite su contenido elemento a elemento.
- La anotación `<a>` expresa que `a` es un canal. Por ejemplo, la anotación `[<a>]` significa "lista de canales", que se utiliza para definir abstracciones de procesos con un número de canales que se desconoce en tiempo de compilación.

Ejemplo. La siguiente abstracción de proceso especifica un proceso que mezcla dos listas ordenadas para formar una nueva lista ordenada:

```
merger :: Ord a => Process ([a],[a]) [a]
merger = process (s1,s2) -> smerge s1 s2
         where smerge [] l = l
               smerge l [] = l
               smerge (x:l) (y:t) = if x <= y then x:smerge l (y:t)
                                              else y:smerge (x:l) t
```

□

La creación de procesos en tiempo de ejecución tiene lugar mediante *concreciones de procesos*, es decir, en el momento de aplicación de una abstracción de proceso a una tupla de expresiones de entrada. Dicha aplicación dará lugar a otra tupla de canales de salida para el nuevo proceso:

$$(out_1, \ldots, out_n) = p \ \# \ (input\_exp_1, \ldots, input\_exp_m)$$

Al igual que para las abstracciones de proceso, no es necesario que las concreciones mencionen explícitamente cada uno de los canales de entrada y de salida, y tampoco es necesario que p sea una variable que haya sido declarada directamente como una abstracción de proceso, sino que puede ser cualquier expresión. La única restricción que se impone es que los tipos sean correctos. Así pues, y teniendo en cuenta que el tipo de # es

```
(#) :: (Transmissible a,Transmissible b) => Process a b -> a -> b
```

donde la clase `Transmissible` se utiliza para garantizar que existe un método para transmitir a traves de canales los valores de entrada y salida del proceso, se permite cualquier expresión como

```
e1 # e2
```

siempre que esté bien tipada. De hecho, # es un valor de primera clase, por lo que, entre otras cosas, puede pasarse como parámetro de una función de orden superior, como por ejemplo en

```
zipWith (#) ps ins
```

Ejemplo. Puede crearse una red dinámica de ordenación concretando la abstracción de proceso `merger` del ejemplo anterior:

```
sortNet :: (Ord a, Transmissible a) => Process [a] [a]
sortNet = process list -> sort list
          where sort []  = []
                sort [x] = [x]
                sort xs  = merger # (sortNet # l1, sortNet # l2)
                           where (l1,l2) = unshuffle xs
                unshuffle []      = ([],[])
                unshuffle [x]     = ([x],[])
                unshuffle (x:y:t) = (x:t1,y:t2)
                                    where (t1,t2) = unshuffle t
                merger = ...
```

Aquellas listas que tienen más de dos elementos se dividen en dos sublistas, para las cuales se generan nuevas concreciones de los procesos `sortNet` y `merger`. La topología resultante es un árbol binario de procesos.

3.1.2 Sistemas reactivos

Las construcciones presentadas hasta el momento son suficientes para programar sistemas concurrentes deterministas, pero es necesario introducir nuevos conceptos no-funcionales para extender el poder expresivo del lenguaje de modo que puedan definirse también sistemas reactivos.

En Eden, la *reactividad* se introduce mediante la abstracción de proceso predefinida `merge`, que representa un proceso que mezcla de forma *reactiva* una lista de canales, donde cada uno de dichos canales es una lista:

```
merge :: Process [<[a]>] [a]
```

Tan pronto como un valor está disponible en alguna de las listas de entrada, el proceso `merge` lo copia a la salida. De esta forma puede conocerse el orden temporal en el que se producen determinados valores, pudiendo así reaccionar apropiadamente en función de dicho orden. Nótese que el proceso `merge` es *no determinista*, pues distintas ejecuciones pueden conducir a distintos órdenes en la lista de salida, debido a *carreras* en la producción de las listas de entrada del proceso.

Eden proporciona *canales dinámicos*. Un proceso puede generar un nuevo canal de entrada y enviar a otro proceso un mensaje que contenga el nombre de dicho nuevo canal. Entonces, el proceso receptor puede usarlo para devolver cierta información al proceso inicial (*recibir y usar*), o bien puede pasar el nombre del canal a otro proceso (*recibir y pasar*). Ambas posibilidades son mutuamente excluyentes, de forma que el programador debe asegurarse de que cada canal dinámico sólo se va a utilizar para establecer una conexión uno-a-uno entre un único emisor y un único receptor (el que creó el canal).

En caso contrario, en tiempo de ejecución se generará un mensaje de error notificando que se ha roto dicha condición, y se abortará la ejecución del programa.

Para los nombres de canales dinámicos, se introduce un nuevo constructor de tipos denominado `ChanName`, de modo que el nombre de un canal de tipo `a` es de tipo `ChanName a`. Para crear un nuevo canal dinámico se utiliza la siguiente construcción:

$$\textbf{new}\ (nombre\_canal, canal)\ expr$$

La expresión anterior declara un nuevo canal cuyo nombre es *nombre_canal*, y cuyo contenido podrá leerse accediendo a través de *canal*. El ámbito de visibilidad tanto de *nombre_canal* como de *canal* es el cuerpo de la expresión *expr*. Para poder establecer una comunicación, el nombre del canal se enviará a otro proceso a través de uno de los canales de salida ya existentes. El proceso que reciba el nombre del canal, y que quiera enviar algún dato a través de él, podrá hacerlo mediante una expresión de la forma:

$$nombre\_canal\ \textbf{!*}\ expr_1\ \textbf{par}\ expr_2$$

Antes de comenzar a evaluar $expr_2$, se crea una nueva hebra concurrente para la evaluación de $expr_1$, y el resultado de dicha hebra se transmitirá a través del canal dinámico. El resultado de la expresión completa es el de $expr_2$.

Ejemplo. En un sistema con un único proceso distribuidor y varios procesos trabajadores, la comunicación puede simplificarse introduciendo un canal dinámico. La idea consiste en que cada vez que un proceso trabajador pide que se le asigne un trabajo, debe crear un nuevo canal dinámico, a través del cual el distribuidor le comunicará el trabajo que debe realizar. Así, en el trabajador tendremos:

```
toManager = new (task_name, task)
                (AskForWork task_name : waitForWork task)
waitForWork (Work work) = doTheWork work : toManager
```

mientras que el distribuidor distribuirá el trabajo a través de los canales que se le comuniquen:

```
answerRequests workpool ((AskForWork chan): reqs)
  = chan !* (first workpool)
         par answerRequests (rest workpool) reqs
```

3.1.3 Topologías de comunicación y *bypassing*

Eden permite expresar fácilmente topologías de procesos mediante sistemas de ecuaciones, donde cada ecuación describe los canales de entrada y de salida de un proceso. De esta forma, se pretende facilitar las tareas de programación, de modo que puedan reutilizarse fácilmente esquemas de procesos. En [**PR01**] se describen distintos esqueletos para representar una amplia gama de topologías de procesos, como tuberías, toroides, etc.

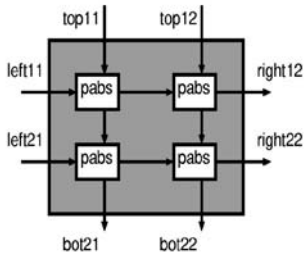


Figura 3.1: Cuadrícula de 4 procesos

Ejemplo. La abstracción de proceso `grid2` genera una cuadrícula de 4 procesos (véase Figura 3.1). La abstracción de proceso correspondiente a los procesos de las celdas es un parámetro del proceso `grid2`, de modo que el esqueleto es lo suficientemente genérico como para que pueda concretarse con cualquier abstracción de proceso del tipo adecuado:

```
grid2 :: Process (a,b) (a,b) -> Process (a,a,b,b) (a,a,b,b)
grid2 pabs = process (left11, left21, top11, top12) ->
                     (right12, right22, bot21, bot22)
  where (right11, bot11) = pabs # (left11,  top11)
        (right12, bot12) = pabs # (right11, top12)
        (right21, bot21) = pabs # (left21,  bot11)
        (right22, bot22) = pabs # (right21, bot12)
```

□

Ahora bien, en Eden, la comunicación se realiza a través de canales unidireccionales 1:1, de modo que cuando se crea un nuevo proceso, sus canales de salida se convierten en nuevos canales de entrada del proceso padre, y viceversa. El inconveniente es que este comportamiento no es deseable en el caso en el que el proceso padre no utiliza el contenido de dicho canal, sino que simplemente transfiere dicho contenido hacia/desde un tercer proceso.

El problema de establecer las conexiones directamente entre los nuevos procesos y su padre radica en que la topología resultante no sería la que se

aprecia en la figura 3.1, sino que todas las conexiones internas de la cuadrícula irían al padre, y de él volverían al hijo correspondiente. Por tanto, el proceso padre se convertiría en un cuello de botella, no sólo en este caso, sino en un gran número de topologías.

Por dicho motivo, Eden requiere un mecanismo automático denominado *bypassing* [KPS00] que sea capaz de redireccionar los canales de modo que se establezcan conexiones directas entre los productores y los consumidores, obviando a los intermediarios.

En el Capítulo 5 se trata de nuevo este problema. Allí se propone un esquema de compilación que permite la detección de situaciones de *bypassing*. Para aquellos esquemas complejos en los que el análisis no de buenos resultados, podrá utilizarse la metodología propuesta en la Sección 6.6 para derivar topologías no jerárquicas.

3.2 Semántica

La semántica operacional de Eden es una extensión de la semántica operacional estándar de Haskell, y refleja la distinción entre los sublenguajes de cómputo y de coordinación que conviven en Eden. Comprende dos niveles de sistemas de transiciones: el nivel inferior maneja aquellos efectos que son locales a un único proceso; mientras que el nivel superior describe los efectos que son globales al sistema completo (la creación de un nuevo proceso, el envío de un dato, la recepción de un dato, la creación de un canal dinámico, el establecimiento de una conexión usando un canal dinámico, y la finalización de una hebra.). La interfaz entre ambos niveles está formada por "acciones" que comunican al nivel superior la necesidad de un evento global, como por ejemplo, la creación de un proceso, el envío de un mensaje o la terminación de un proceso. Los detalles concretos y formales sobre dicha semántica pueden encontrarse en [BLOP98]. Recientemente, en [HO00] se ha desarrollado una nueva versión más formal de dicha semántica. En lo que resta de la presente sección se describen de un modo informal los aspectos fundamentales de la semántica de Eden.

Impaciencia v.s. pereza. Aunque el modelo computacional de Eden es un lenguaje perezoso (Haskell), se ha optado por no mantener dicha pereza para el modelo de coordinación, con el objetivo de mejorar el comportamiento paralelo de los programas. Así, la pereza se rompe en dos situaciones:

- Una vez lanzado un proceso, su cómputo está dirigido por la evaluación de sus expresiones de salida, para las cuales siempre existe demanda. De esta forma, se incrementa el grado de paralelismo, puesto que los procesos pueden producir datos de forma independiente, sin necesidad de esperar a que nadie los demande.

- Se permite que un proceso se lance antes de ser demandado, con el objetivo de acelerar la distribución del cómputo. Véase la Sección 5.1.1 para más detalles.

Nótese que al romper la evaluación perezosa, las dos reglas anteriores pueden conducir a que se realice trabajo innecesario para el cómputo final, pero es un riesgo que se acepta por la ventaja de mejorar el grado de paralelismo de los programas, si bien el programador deberá tenerlo siempre en mente a la hora de utilizar el lenguaje.

Reparto de trabajo. Con el objetivo de poder razonar acerca del comportamiento paralelo de los programas, debe quedar claro qué partes del cómputo deben realizarse en cada proceso. En Eden, al evaluarse

```
e1 # e2
```

el padre se encargará de evaluar `e2`, y de comunicar su valor a `e1`, mientras que todo el cómputo necesario para evaluar `e1` será llevado a cabo por el nuevo proceso hijo. Esto incluye no sólo la aplicación de `e1` a los valores de entrada correspondientes a `e2`, sino también la evaluación de las variables libres de `e1` (en caso de que sea preciso evaluarlas). A modo de ejemplo, en

```
p :: Int -> Int -> Process Int Int
p x y = process i -> x + y + i
result = p (fib 5) (fib 6) # (fib 7)
```

el padre sólo se encargará de evaluar `fib 7`, mientras que será el proceso hijo quien evalúe no sólo la suma final, sino también `fib 5` y `fib 6`.

Evaluación de las salidas de los procesos. El objetivo de un proceso es evaluar sus salidas. Excepto las listas que se transmiten en forma de *streams*, las expresiones se evaluarán *completamente* (es decir, a forma normal) antes de ser enviadas por un canal de salida. Así, la evaluación de las expresiones de salida debe realizarse siempre por el emisor, y nunca por el receptor.

Para cada una de las expresiones de salida de un proceso, se creará un flujo de ejecución concurrente distinto, puesto que, en principio, dichas evaluaciones son independientes. Por tanto, se distinguen dos niveles de concurrencia en Eden: la concurrencia y evaluación paralela de los procesos; y la evaluación concurrente de las distintas hebras de cada proceso.

Terminación. Dado que el cómputo de un proceso viene dirigido por la evaluación de sus canales de salida, un proceso terminará de forma inmediata en cuanto no tenga ningún canal de salida. Al terminar, sus canales de entrada dejarán de ser útiles, por lo que se eliminarán, y los correspondientes canales de otros procesos que alimentasen dichas entradas serán cerrados, propagándose así la terminación de unos procesos a otros.

Sincronización. Cuando una hebra concurrente de un proceso necesita un cierto valor de un canal de entrada, pero dicho valor aún no ha sido recibido, la evaluación de dicha hebra será suspendida hasta que el correspondiente emisor produzca y envíe el dato. Nótese que la comunicación a través de canales se realiza utilizando envío no-bloqueante, pero recepción bloqueante, y que la sincronización entre los procesos se realiza única y exclusivamente mediante el intercambio de información a través de los canales.

3.3 Implementación

Dado que Eden es una extensión de Haskell, parece razonable tratar de reutilizar un compilador de Haskell ya existente, y extenderlo con el objetivo de cubrir las características propias de Eden. Por dicho motivo, el primer compilador de Eden (MEC: Marburg-Madrid Eden Compiler) se ha realizado reutilizando el código fuente de GHC, de modo que soporte todas las funcionalidades de GHC.

La elección de GHC como punto de partida para el desarrollo de MEC se debe a su eficiencia, fiabilidad y disponibilidad, ya que puede obtenerse su código fuente en la página web del proyecto: http://www.haskell.org/ghc.

Actualmente está disponible una primera versión de Eden que implementa casi todas las características del lenguaje, con sólo dos excepciones: aún no se han implementado las anotaciones de canales que permiten generar estructuras de canales; y todavía no se realiza *bypassing* automático de canales.

El resto de la presente sección describe el proceso de compilación de MEC, para lo cual es preciso presentar con anterioridad el proceso de compilación de GHC. El lector interesado puede encontrar en [KOP99] la descripción detallada y a nivel abstracto del RTS de MEC, mientras que en [Klu98] pueden verse los detalles de la implementación real del RTS.

3.3.1 El proceso de compilación de GHC

El proceso de compilación de GHC está dividido en las fases que pueden apreciarse en la Figura 3.2. El código fuente está escrito por completo en Haskell, con la excepción del analizador sintáctico (escrito en Lex/Yacc y C) y del RTS (escrito en C).

Existe una primera fase para convertir *literate*[1] Haskell en Haskell, en el caso en el que el programa original estuviese escrito así. Acto seguido, el analizador sintáctico reconoce las fuentes Haskell, generando como salida el árbol de sintaxis abstracta correspondiente al programa de entada. Dado que

[1] El modo *literate* da preferencia a los comentarios sobre el código fuente, de modo que, por defecto, todo son comentarios, y para introducir código hay que utilizar notación especial.

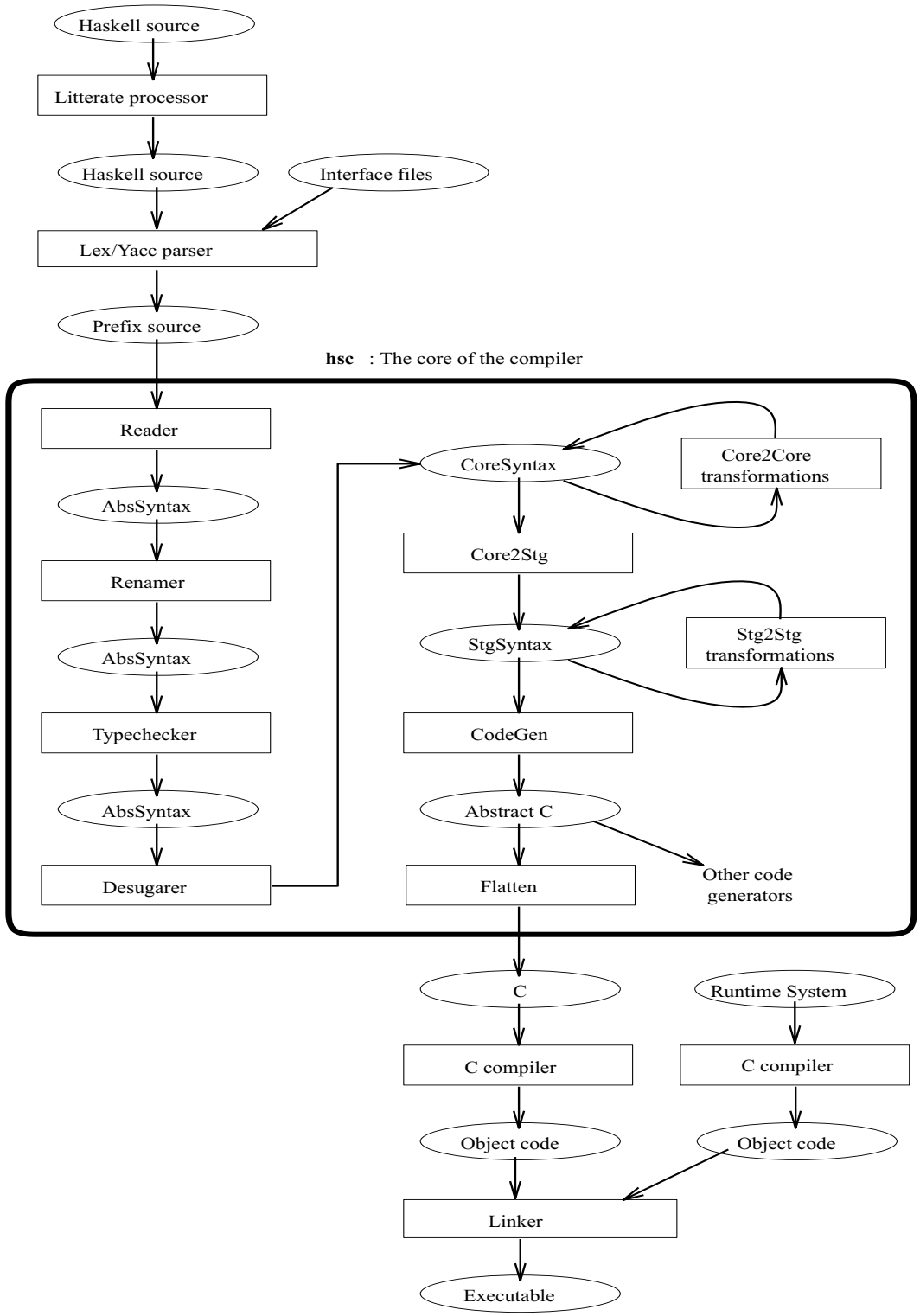


Figura 3.2: Estructura de GHC

el analizador genera C, y el resto del compilador está escrito en Haskell, es necesaria una fase intermedia (*reader*), que además se encarga de resolver las precedencias de los operadores infijos que se hayan utilizado en el programa.

El *renamer* resuelve conflictos debidos al ámbito de visibilidad de los identificadores, especialmente en lo concerniente a las importaciones y exportaciones de los módulos.

La fase de inferencia de tipos anota el programa con la información de tipos correspondiente, eliminando toda la sobrecarga que pudiera existir.

El "desdulcificador" traduce la rica sintaxis abstracta de Haskell a un lenguaje funcional mucho más simple denominado *Core*. Nótese que esta fase es posterior a la inferencia de tipos, de modo que los mensajes de errores de tipos pueden ofrecerse relacionados con el código fuente, haciéndolos más fácilmente comprensibles por el programador.

Posteriormente, se realizan (de forma opcional) un conjunto de transformaciones dentro del lenguaje Core, con el objetivo de mejorar la eficiencia del código generado.

A continuación, el programa Core se traduce a otro lenguaje funcional más simple: STG[2], y se realizan (opcionalmente) otras transformaciones dentro de dicho lenguaje.

Finalmente se realiza la generación de código. GHC genera C como lenguaje final, de modo que posteriormente es necesario procesarlo con un compilador de C. Ahora bien, con el objetivo de mejorar la eficiencia sin perder la portabilidad tanto a distintos compiladores de C como a distintas arquitecturas, en lugar de generarse directamente código C, se genera un árbol de sintaxis abstracta C. Así, una última fase (que puede ser distinta para cada tipo de arquitectura) lo traduce a C optimizando los recursos de que se dispongan en cada caso.

Una vez generado el código C, se enlaza con el código del RTS, generándose el programa ejecutable.

El código fuente correspondiente al RTS (escrito en C) es, en cierto sentido, independiente del proceso de compilación propiamente dicho. Existen distintos RTS en función del uso que se esté haciendo de GHC. Por ejemplo, existe un RTS para Haskell, otro que además incluye las características necesarias para realizar perfiles secuenciales, un RTS para cuando se utiliza GpH (Glasgow Parallel Haskell), otro para cuando se está trabajando con GranSim [Loi96], etc. Estos RTS comparten ciertas partes de su código fuente, mientras que otras son propias de cada uno de ellos.

3.3.2 El proceso de compilación de MEC

Con el objetivo de minimizar el esfuerzo en la labor de reingeniería, la idea fundamental en el desarrollo de MEC consiste en modificar lo mínimo

[2]Shared Term Graph Language.

posible GHC, de modo que sólo se modifica una etapa del compilador cuando es indispensable (véase la Figura 3.3).

La primera fase que hace falta modificar es, lógicamente, el analizador léxico y sintáctico, pues en Eden existen construcciones que no están presentes en Haskell. Ahora bien, con el objetivo de evitar que dicha modificación acarree como consecuencia modificaciones en las fases siguientes del compilador, no se modifica la sintaxis abstracta que puede generar el analizador. La idea para conseguir "esconder" las construcciones Eden sin abandonar la sintaxis correspondiente a Haskell consiste en utilizar funciones predefinidas escritas en Haskell. Así, por ejemplo, cuando el analizador detecta la construcción

```
process patron -> expresion
```

genera como salida la aplicación de la función predefinida `process` a una función:

```
process (\ patron -> expresion)
```

Es importante resaltar la gran diferencia existente entre el primer `process` y el segundo. El primero corresponde a una palabra reservada en Eden, mientras que en el segundo caso es simplemente el nombre de una función Haskell, que está definida en un módulo que puede considerarse como el preludio de Eden.

En dicho módulo, la definición de `process` (y del resto de funciones básicas) se realiza de modo que el sistema de tipos de GHC sea capaz de inferir correctamente los tipos de las expresiones de Eden. Para ello, basta conseguir las siguientes declaraciones de tipos:

```
data Process a b = Process (a->b)
process :: (a -> b) -> Process a b
(#) :: (Transmissible a, Transmissible b) => Process a b -> a -> b
```

De esta forma[3] se consigue engañar a GHC para que procese las construcciones de Eden. Ahora bien, debe llegar un punto en el que reaparezcan las construcciones Eden. La forma de conseguirlo pasa por definir las funciones del preludio Eden de modo que invoquen a otras funciones primitivas (escritas en C) que se introduzcan directamente en el código del RTS. Así, por ejemplo, cuando se evalúe

```
e1 # e2
```

se llamará a una función primitiva, que hará que el RTS cree un nuevo proceso para evaluar `e1`, así como que se creen nuevas hebras para evaluar `e2`. Véase la Sección 3.3.3 para más detalles sobre el RTS de Eden.

[3] Lógicamente, también se realizan traducciones similares para el resto de las construcciones de Eden.

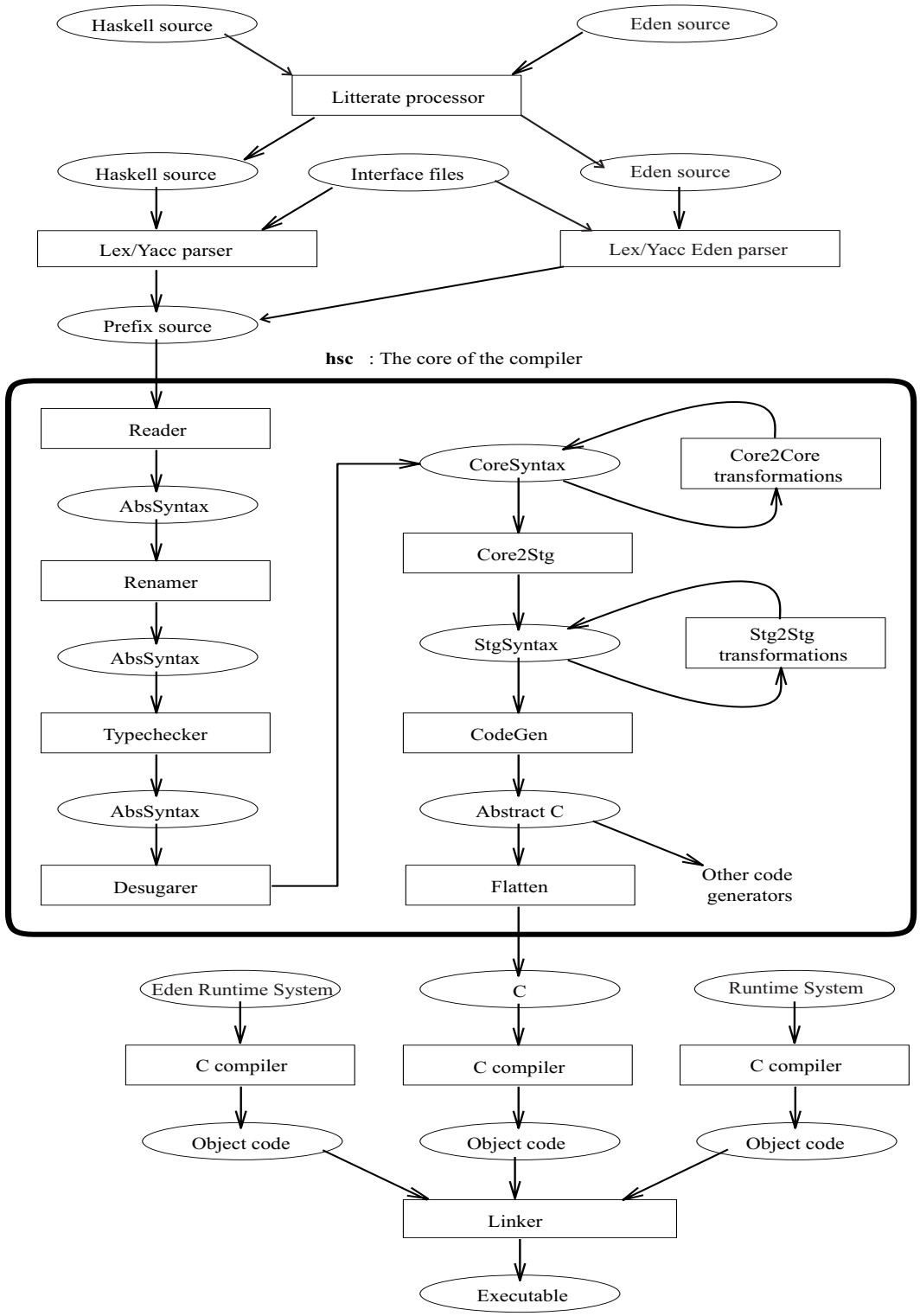


Figura 3.3: Estructura de MEC

Así se consigue un alto grado de reutilización del código fuente de GHC. Ahora bien, existen algunas transformaciones automáticas que serían deseables en Eden, pero que no están presentes en GHC, como es el caso del lanzamiento impaciente de procesos (véase la Sección 5.1.1), por lo que es preciso añadirlas en la fase de transformaciones a nivel Core. Además, desafortunadamente algunas de las transformaciones automáticas que GHC efectúa al nivel del lenguaje Core no mejoran la eficiencia de los programas Eden sino que, por el contrario, pueden reducirla. Es más, determinadas transformaciones no sólo pueden empeorar la eficiencia sino que, lo que es peor, pueden cambiar la semántica del programa Eden original. En el Capítulo 5 se muestra cómo se pueden mantener las transformaciones de GHC e incluir nuevas transformaciones propias de Eden, todo ello sin riesgos semánticos.

3.3.3 El RTS de Eden

Con el objetivo de reutilizar al máximo el compilador GHC, el RTS de Eden se ha desarrollado a partir del RTS de la versión paralela de GHC (GpH: Glasgow Parallel Haskell). Así, podemos reusar las unidades de cómputo de GpH (denominadas *hebras*), así como parte de sus mecanismos de comunicación, si bien siguen siendo precisas grandes modificaciones en el RTS. Así, por ejemplo, GpH está implementado sobre la librería de paso de mensajes PVM, mientras que Eden lo está tanto sobre PVM como sobre MPI.

Además de cuestiones de implementación como la anterior, existen otras muchas cuestiones conceptuales que diferencian ambos RTSs. A continuación se describen los aspectos fundamentales del RTS de Eden.

Planificador de tareas. Dentro de cada procesador, Eden requiere la existencia de un planificador de tareas justo, que permita que evolucionen de forma concurrente los cómputos asociados a las distintas hebras que residan en el mismo procesador.

El planificador de tareas distribuye el tiempo de cómputo entre aquellas hebras que no se encuentren bloqueadas, para lo cual se mantiene una lista con los TSOs[4] de dichas hebras, de modo que cada vez que alguna se bloquea se elimina de la lista, y cada vez que alguna se desbloquea, se añade a la lista.

Estructuras adicionales al RTS de GpH. Dado que en Eden existen conceptos que no aparecen en GpH, necesitamos registrarlos en el RTS. Así, para cada procesador existen tres tablas adicionales que recogen información acerca de los procesos y de los canales de Eden (véase la Figura 3.4):

[4]Thread State Object. Un TSO es un registro que representa una hebra, y que contiene toda la información necesaria sobre ella, de modo que puedan realizarse fácilmente los cambios de contexto, o la recogida de basura.

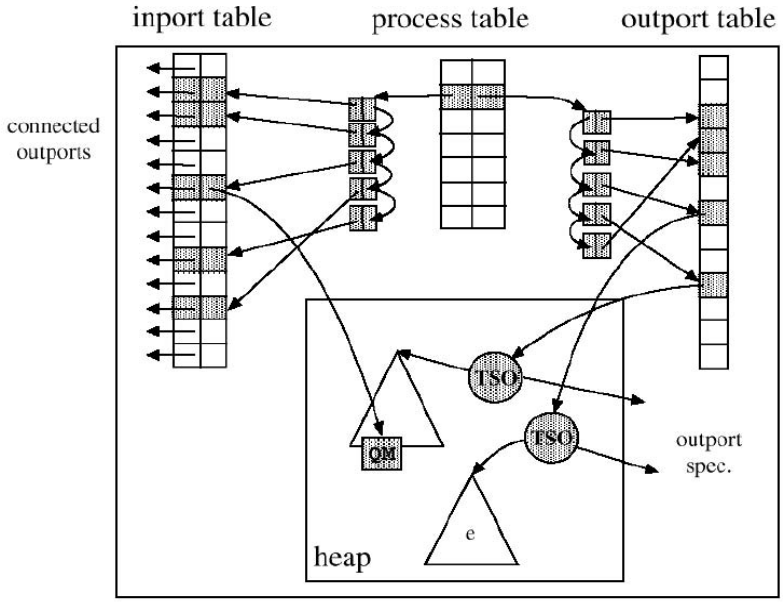


Figura 3.4: Tablas adicionales del RTS de Eden.

- La tabla de *inports* relaciona cada canal de entrada (que tiene asociado un identificador localmente único) con la posición de memoria sobre la que deben escribirse los datos que se reciban por el canal. Asimismo, almacena para cada canal de entrada el identificador global asociado al correspondiente canal de salida remoto, con el objetivo de poder propagar las condiciones de terminación.

- La tabla de *outports* relaciona cada canal de salida (que tiene asociado un identificador localmente único) con el TSO de la hebra asociada al mismo. La necesidad de esta tabla viene dada por cuestiones de detección de terminación, así como para facilitar las labores de recogida de basura.

- La tabla de procesos asocia a cada proceso local la lista enlazada de sus canales de entrada, así como la lista enlazada de sus canales de salida. La razón de ser de esta tabla es que varios procesos pueden residir en el mismo procesador, por lo que es preciso distinguir de algún modo qué hebras pertenecen a cada uno de ellos.

Comunicaciones. En Eden, todas las comunicaciones entre procesos se realizan a través de canales de comunicación 1 a 1, y todos los datos que se envían se reducen previamente a forma normal. Una vez que disponemos de un dato en forma normal que debemos enviar, se empaqueta en un mensaje[5] MPI toda la parte del grafo de clausuras asociada al dato, y se envía al procesador donde resida el proceso receptor, lo cual sabremos porque es una información que se almacena en el TSO de la hebra enviante. Una vez allí, se desempaqueta el mensaje, se determina a qué canal de entrada pertenece y se consulta la tabla de *inports* para determinar en qué posición de memoria debe escribirse el valor recibido.

Sincronización. Con el fin de evitar que dos hebras de un mismo proceso evalúen dos veces una misma clausura (con la consiguiente duplicación de trabajo), se utilizan un tipo especial de clausuras denominadas clausuras `QueueMe` (en lo sucesivo QM), cuyo modo de operación es el siguiente:

- Cuando una hebra necesita evaluar una clausura que puede estar compartida, la convierte en una QM, y procede a evaluarla.

- Cuando una hebra intenta consultar el contenido de una QM, dicha hebra queda bloqueada, en espera de que la QM sea sobreescrita con su valor.

- Cuando la hebra evaluadora concluye el cómputo de la clausura, actualiza la QM con el valor calculado, y "despierta" a todas aquellas hebras que estuvieran bloqueadas en espera de leer el valor de la QM. Para poder despertarlas, cada QM lleva asociada una cola en la que se registran los TSOs de las hebras que están suspendidas en la QM.

Además de la sincronización entre las hebras de un mismo proceso, es preciso sincronizar los distintos procesos entre sí. Para ello se utiliza la misma idea de las QMs. A cada canal de entrada se le asocia una QM, de modo que si una hebra intenta leer su contenido antes de que el emisor haya enviado el valor correspondiente, la hebra quedará bloqueada hasta que se reciba el dato. A este respecto, cabe diferenciar los dos tipos de canales que existen: los de tipo *stream* y el resto. Cada vez que se recibe un dato por un canal de tipo *stream*, no sólo se actualiza la QM correspondiente con el dato recibido, sino que es preciso crear una nueva QM para los siguientes datos que se recibirán por el canal, por lo que también debe actualizarse la tabla de *inports* para que apunte a la dirección de la nueva QM. Sin embargo, para el resto de canales no es preciso, pues sólo se envía un único mensaje.

[5] Realmente puede ser en varios mensajes MPI, en caso de que no haya espacio suficiente en uno único.

Lanzamiento de un proceso. Al crear un proceso, es preciso transmitir al procesador responsable de su evaluación toda la información precisa para que pueda evaluarlo de acuerdo a las reglas semánticas del lenguaje. Así, al evaluarse

```
e1 # e2
```

es preciso transmitir al procesador remoto no sólo la clausura asociada a `e1`, sino todas aquellas clausuras que "cuelguen" de ella, es decir, todas aquellas que puedan ser precisas para su evaluación. De no hacerse así, serían necesarias comunicaciones posteriores para obtener dichos datos, pero dichas comunicaciones se realizarían de un modo oculto para el programador, lo cual es un comportamiento que contradice los principios del lenguaje.

Así, por ejemplo, para

```
let
  x1 = fib 20
  x2 = fib 10
  fib x = ...
in
  (p x1 x2) # x1
```

deberían copiarse las clausuras asociadas a `p`, `x1`, `x2`, y `fib`, así como las que dependieran de `fib`. Es decir, se copian todas las variables libres, y todas las variables libres de dichas variables libres, y así sucesivamente[6].

Una vez determinada la información a enviar, se empaquetará en un mensaje MPI, y se enviará al procesador responsable de evaluar el nuevo proceso.

Nótese que, en el ejemplo anterior, el cómputo de `x1` se duplicará, pues deberá hacerlo tanto el padre como el hijo.

Lógicamente, además de la transmisión del proceso, es preciso añadir nuevas entradas en las tablas Eden, tanto en el procesador donde reside el padre como en el que vaya a residir el hijo. Así, no sólo se crearán nuevas hebras y sus correspondientes TSOs, sino que se crearán nuevos *inports* y *outports* (en ambos procesadores), y en las tablas se recogerán las conexiones precisas para que puedan establecerse las comunicaciones posteriores. Ahora bien, para que el padre pueda establecer dichas conexiones, es preciso esperar a la llegada de un mensaje de confirmación del procesador hijo, pues de lo contrario el padre no podría conocer los identificadores de los canales del nuevo hijo.

[6] Las variables que se definen en el nivel más externo del programa no se consideran variables libres, por lo que no es necesario copiarlas. La razón es que residen en todos los procesadores, por lo que es absurdo copiarlas, basta con indicar cuáles son.

Terminación. Los procesos que no tienen canales de salida abiertos deben terminar, puesto que no pueden contribuir al cómputo global del programa. Así, cada vez que se cierra un canal de salida se comprueba si el proceso aún tiene algún otro canal de salida, para lo cual basta consultar la tabla de procesos. Si no los tuviera, entonces terminaría el proceso, lo cual consiste en cerrar los canales de entrada, y en enviar mensajes de terminación a las hebras que se encuentren al otro extremo de dichos canales de entrada.

Existen dos situaciones en las que se cierra un canal de salida: cuando la hebra termina de enviar todos los datos de salida; y cuando el proceso receptor envía un mensaje de terminación debido a que ya no necesita más datos. Análogamente, hay dos situaciones en las que se cierra un canal de entrada: cuando muere el proceso y cuando deja de estar referenciado (caso en el cual el recolector de basura puede eliminarlo).

Ubicación de tareas. En Eden, la asignación de un proceso a un procesador se realiza en el instante de creación del proceso, y es una decisión que se toma internamente en el procesador en el que se encuentre el proceso padre, siguiendo la estrategia de reparto que se indique al RTS. Se proporcionan dos modos distintos:

- El modo *round-robin* en el que todos los procesos creados desde un mismo procesador p se crean en procesadores numerados consecutivamente, empezando en el procesador $p + 1$.
- El modo *aleatorio* en el que los procesos creados desde cualquier procesador se crean en procesadores escogidos aleatoriamente.

El primer modo es adecuado cuando se crean tantos procesos como procesadores, ya que garantiza que éstos se crearán en procesadores distintos, mientras que el segundo es útil cuando se crean muchos más procesos que procesadores, ya que proporciona una distribución aleatoria de la carga. El modo de distribución de procesos se escoge al comienzo de cada ejecución.

Otra decisión concerniente al reparto de carga que debe tenerse en cuenta es que no se permite la migración de procesos (ni mucho menos de hebras) de un procesador a otro. Ello no sólo implicaría detener momentáneamente la ejecución del proceso a migrar, sino que supondría una grandísima cantidad de información a transmitir de un procesador a otro. Además, sería preciso modificar las tablas de conexiones de todos los procesadores que compartieran algún canal de comunicación con el proceso migrado, de modo que potencialmente se verían implicados todos los procesadores.

Mejora. Dado que lo habitual es que no se pueda disponer de un procesador para evaluar cada proceso, varios procesos pueden residir en un mismo procesador. En dicho caso, si dos procesos de un mismo procesador deben comunicarse entre sí, es absurdo que lo hagan a través de mensajes MPI, y

también es absurdo que se pueda duplicar trabajo. Por tanto, lo razonable es que sus hebras compartan la memoria como si perteneciesen a un mismo proceso, y que se utilice el mismo mecanismo de sincronización que para las hebras de un proceso. Así, evitamos los costes de empaquetamiento y desempaquetamiento de clausuras, el envío de mensajes, y evitamos duplicaciones de trabajo dentro de un mismo procesador.

Lógicamente, aunque podamos tratar todas las hebras de un procesador como si pertenecieran a un mismo proceso, debe mantenerse una tabla en la que se registre a qué proceso pertenece cada hebra, con la intención de poder detectar las condiciones de terminación de los procesos.

Capítulo 4

El simulador Paradise

En este capítulo comienzan las contribuciones originales de esta tesis. La primera parte del capítulo no es aún original, sino que sólo presenta el estado del arte en el desarrollo de herramientas que proporcionan al usuario realimentaciones para mejorar la eficiencia de los programas. Primero se describen las técnicas básicas empleadas en lenguajes funcionales perezosos secuenciales, para luego mostrar las herramientas más relevantes que tratan lenguajes funcionales paralelos. Tras esas dos secciones, nos centraremos en la parte original del capítulo, formada fundamentalmente por el simulador Paradise, que ha sido diseñado e implementado como parte de esta tesis.

La parte original del capítulo ha dado lugar a la publicación [**HPR00**], en la que se describe el simulador Paradise.

4.1 Perfiladores secuenciales

Existen dos aspectos a tener en cuenta a la hora de evaluar el coste de un lenguaje de programación:

- La eficiencia en ejecución de los programas escritos utilizando dicho lenguaje.
- El coste de desarrollo y mantenimiento de los programas.

Al comparar la eficiencia en tiempo de ejecución entre los lenguajes funcionales y los imperativos, resulta evidente la supremacía de estos últimos. Los defensores de los lenguajes funcionales suelen basar la defensa de la eficiencia de su paradigma argumentando que la parte más importante es el coste en tiempo de programador, es decir, defienden que es preferible optimizar el tiempo de desarrollo y mantenimiento antes que el tiempo de ejecución. En este aspecto, la programación funcional resulta más potente que la programación imperativa, debido a que su mayor nivel de abstracción permite desarrollar aplicaciones más rápidamente y, sobre todo, facilita el mantenimiento de los sistemas, pues su mayor concisión y claridad permite entender

más fácilmente el código fuente, y la ausencia de efectos laterales facilita los razonamientos sobre la corrección de los programas, así como la transformación de los mismos.

Ahora bien, aunque ponga más énfasis en la defensa de la optimización del tiempo de programador, la comunidad funcional sigue trabajando para mejorar el tiempo de ejecución de sus programas, desarrollando cada vez mejores compiladores y desarrollando herramientas que permitan al programador razonar sobre la eficiencia de sus programas.

Una de las principales razones de la falta de eficiencia de los programas funcionales es el hecho de que para los programadores resulta muy complicado razonar acerca de la eficiencia de sus programas. En el mundo funcional, existe una gran diferencia entre el alto nivel de abstracción en el que se mueve el programador, y el bajo de nivel que corresponde a la ejecución real de la aplicación. Esta diferencia se incrementa de forma considerable al razonar sobre el orden de evaluación cuando trabajamos con lenguajes perezosos, en los que resulta realmente complejo determinar cómo evoluciona la ejecución de los programas.

Así pues, lo que el programador necesita es algún modo de realimentación, de forma que pueda determinar no sólo la presencia de ineficiencias, sino también sus causas. Así, deberíamos ser capaces de ver fácilmente cuál es la parte de código responsable del comportamiento ineficiente, y sólo tendríamos que reescribir dicha parte.

Este tipo de realimentación no sólo resulta interesante en los lenguajes funcionales. De hecho, en el ámbito imperativo, los beneficios de estas herramientas fueron expuestos en [Knu71] por Knuth, quien se dio cuenta de que habitualmente más de la mitad del tiempo de ejecución de los programas se debía únicamente a menos de un 4% del código fuente. De esta forma, identificando cuáles son los cuellos de botella de los programas resulta sencillo dirigir los esfuerzos de optimización a una pequeña parte del código, pudiéndose obtener así mejoras espectaculares dedicando tan sólo unas pocas horas de trabajo.

El empleo de herramientas de caracterización del rendimiento (en lo sucesivo "perfiladores") proporciona un enfoque distinto para desarrollar programas eficientes. En lugar de escribir directamente programas oscuros en aras de la eficiencia, el uso de estas herramientas conduce a un nuevo método de desarrollo:

1. Escribir la primera versión del programa de forma clara.
2. Utilizar la herramienta para buscar errores de rendimiento.
3. Si se han detectado errores, corregirlos (reescribiendo la parte correspondiente del programa) y volver a 2.

De esta forma sólo tenemos que preocuparnos por mejorar la eficiencia de aquellas partes del programa que realmente afectan de forma considerable

al rendimiento global, evitando así una dedicación excesiva de tiempo para optimizar el resto del código, al mismo tiempo que obtenemos programas más legibles. Por tanto, este método de trabajo mejora la eficiencia en tiempo de ejecución al mismo tiempo que mejoramos la eficiencia en tiempo de desarrollo. Por otro lado, este tipo de herramientas permite detectar no sólo ineficiencias, sino también qué partes de los programas merece la pena paralelizar y cuáles no. Así, si determinada parte consume la mayor parte del tiempo de ejecución, y no se debe a una ineficiencia, podríamos acelerar la ejecución con sólo paralelizar dicha parte del código.

La forma habitual de implementar estas herramientas consiste en modificar el compilador (o intérprete) de modo que la compilación del programa genere un ejecutable instrumentalizado, que debe comportarse igual que el ejecutable normal, sólo que además debe registrar estadísticas sobre la ejecución que se está realizando, y debe salvar dicha información en un fichero para que pueda ser analizada posteriormente.

A pesar de que en la programación funcional perezosa hay una mayor necesidad de obtener una realimentación sobre el comportamiento de los programas, el desarrollo de herramientas con dicho objetivo fue bastante tardío. La causa de esta tardanza se debe a que estas herramientas deben devolver resultados relacionados con el código fuente original (a fin de poder ser comprensibles por los programadores), lo cual es más complejo a medida que se incrementa el grado de abstracción, pues las construcciones del lenguaje se alejan más y más de la implementación real subyacente. En concreto, necesitamos tratar características como el polimorfismo, las funciones de orden superior, la evaluación perezosa y las transformaciones automáticas que realizan los compiladores sobre el código fuente.

El *Glasgow Workshop on Functional Programming* del año 1990 puede considerarse el punto de partida de los perfiladores de lenguajes funcionales perezosos. Allí, Colin Runciman y David Wakeling ([RW91]) plantearon la necesidad de desarrollar dichas herramientas, y expusieron cuáles deberían ser los resultados que se obtuvieran a partir del uso de las mismas:

- Qué funciones son las responsables de que se asigne más memoria.
- Qué funciones son las responsables de retener más cantidad de memoria sin que pueda ser recogida por el recolector de basura.
- Qué funciones son las que consumen más tiempo de ejecución.

Con respecto a los aspectos referentes al consumo de memoria, el trabajo más relevante lo ha llevado a cabo el grupo de Colin Runciman.

La primera herramienta ([RW93a, RW93b]) para perfilar el consumo de memoria de un lenguaje funcional fue desarrollada por Colin Runciman y David Wakeling sobre el compilador LML/HBC[1] ([AJ89]).Dicha herramienta

[1]Compilador basado en la máquina G. Inicialmente compilaba Lazy ML, y posteriormente se modificó para Haskell.

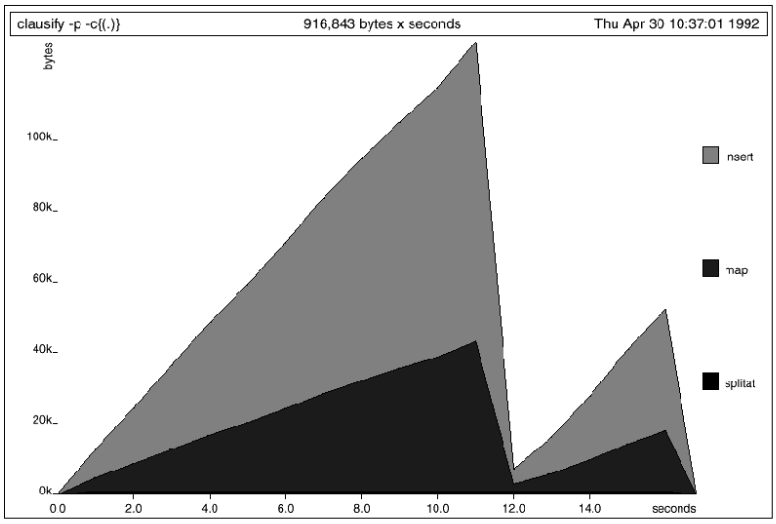


Figura 4.1: Gráfica de productores

permitía visualizar gráficamente la evolución en el tiempo del consumo de memoria mediante dos gráficas distintas, que permiten determinar cuáles son las funciones y constructores[2] que generan más memoria

Un ejemplo de la apariencia de estas gráficas puede apreciarse en la Figura 4.1, que muestra que la función que debemos optimizar para mejorar la residencia de memoria es `insert`.

Para realizar la implementación, cada objeto que se asigna en memoria incluye dos campos adicionales para indicar cuál fue la función responsable de su creación, y cuál fue su constructor. Durante la ejecución del programa, cada cierto tiempo (que puede fijar el usuario) se realiza un censo de la memoria, es decir, se detiene el cómputo y se recorre la memoria almacenando en un fichero la cantidad de memoria viva que se debe a cada función y constructor. Una vez terminada la ejecución, dicho fichero se postprocesa para poder representar de forma gráfica la evolución en el tiempo del consumo de la memoria.

Dicha herramienta fue modificada incrementalmente. Primero se modificó para obtener información sobre el tiempo de vida de las celdas de memoria

[2]Si la celda es un valor de un tipo algebraico, el constructor será la constructora que se encuentre en cabeza; si la celda es una función, el constructor será el nombre de la función, a menos que no tenga nombre, caso en el cual el constructor será UNKNOWN.

y sobre qué parte del programa es la responsable de que no se puedan recolectar [RR96b]. En la implementación, de este último caso se añade a cada celda una palabra extra para indicar qué conjunto de funciones la están reteniendo en memoria. Por otra parte, cada vez que se hace un censo de la memoria, hace falta anotar qué funciones retienen a cada celda. Para ello, por cada retenedor tendríamos que recorrer toda la memoria que está reteniendo, anotando que él es uno de los responsables de que siga viva.

Posteriormente, en [RR96a, RR96c] se incluyó información sobre si cada clausura iba a ser utilizada o no en el futuro, de modo que se detectase memoria inútilmente almacenada que no se iba a usar. Para ello, la implementación necesitaba realizar dos ejecuciones, a fin de poder obtener información sobre si se usuaría o no la memoria.

Con respecto al desarrollo de perfiladores de tiempo de ejecución, existen básicamente tres formas de implementarlos:

- Muestreo: Cada cierto tiempo fijo se interrumpe la ejecución del programa para ver qué se está ejecutando. De esta forma, tomando un número significativo de muestras podemos obtener unos resultados bastante fiables. Nótese que al estar realizando muestras aleatorias los resultados pueden variar de una ejecución a otra.

- Ejecución de bloques básicos: Cada vez que se ejecuta un bloque básico de instrucciones se incrementa el contador correspondiente a la parte del programa que esté ejecutándose. Sabiendo cuánto tiempo dura la ejecución de cada bloque básico podemos obtener el tiempo que se ha gastado para ejecutar cada parte del programa.

- Llamadas al reloj: Cada vez que el programa pasa de evaluar una función a evaluar otra, se llama al reloj del sistema para saber cuánto tiempo ha estado evaluando la función.

La última opción es poco fiable, debido a que acceder al reloj del sistema suele ser una operación costosa, y además la precisión de los relojes no suele ser suficientemente buena. Por ello, esta opción no es aconsejable para programas funcionales perezosos, pues debido a la pereza la función que estamos evaluando en cada momento cambia continuamente, por lo que estaríamos introduciendo considerables distorsiones al realizar multitud de llamadas al reloj. Además, debe notarse que dichas llamadas no se introducen a intervalos regulares, por lo que la distorsión en la atribución de costes no es homogénea. En el caso del muestreo, la distorsión en la atribución de costes es homogénea en todas las funciones, y como sólo estamos interesados en valores relativos sobre qué funciones consumen más que otras, entonces el resultado global es prácticamente equivalente a no tener distorsiones.

Uno de los perfiladores de tiempo más relevantes es el desarrollado por P. Sansom ([San94, SJ95, SJ97]), que se incluyó en GHC, y que es capaz

de medir tanto consumo de tiempo como de memoria. Básicamente, para indicar que quiere perfilar una expresión, el usuario sólo necesita anotar dicha expresión con un *centro de costes*[3]. Tras ejecutar el programa, el sistema indicará cuánto tiempo se ha empleado para la evaluación de cada uno de los centros de costes que se hayan declarado en el programa.

La implementación de los centros de costes es relativamente sencilla. Se utiliza un registro en el que se almacena el centro de costes que se encuentra en curso. Dicho centro de costes debe actualizarse cada vez que cambiamos de entorno léxico. Además, por cada centro de costes se mantienen una serie de contadores. Cada cierto tiempo fijo se detiene la ejecución para consultar cuál es el centro de costes actual, y se incrementa el contador de tiempo de ejecución asociado a dicho centro de costes. Además del contador de tiempo de ejecución, hay otros contadores para determinar, entre otras cosas, cuánta memoria ha asignado cada centro de costes.

Tras finalizar la ejecución, podemos obtener cuánto tiempo se ha gastado ejecutando código asociado a cada centro de costes, así como cuánta memoria han generado. El tiempo se muestra como un fichero de texto simple, donde a cada centro de costes se le asocia el tanto por ciento de tiempo que ha consumido. Para poder ver la evolución en el tiempo de la memoria consumida, se utiliza el mismo método de censos que en el LML/HBC (es decir, sólo atributos estáticos), pero dividiendo las celdas de memoria por el centro de costes que las generó, en lugar de por funciones[4]. El único cambio que hay que hacer es que cada vez que se genera una nueva celda, se marca con el centro de costes que se encuentra en curso.

El perfilador de Sansom fue extendido por R. G. Morgan y S. A. Jarvis ([MJ95, MJ98]) de modo que los costes no se atribuyan sólo al centro de costes más cercano, sino a toda la pila de centros de costes que contienen léxicamente a la expresión. Si queremos restringirnos al centro de costes más cercano, sólo tenemos que colapsar todas las pilas de centros de costes cuya cima sea el mismo centro de costes.

Para una visión más amplia sobre el estado del arte de los perfiladores, tanto de tiempo como de memoria, de los lenguajes funcionales perezosos secuenciales, véase el Capítulo 2 de [Rub99].

4.2 Perfiladores paralelos

A diferencia de las herramientas mencionadas en las secciones anteriores, en las que se tomaban datos sobre ejecuciones reales, en el ámbito paralelo se ha optado por dirigir los esfuerzos hacia el desarrollo de simuladores (lo más flexibles posible) de arquitecturas paralelas. Estos simuladores proporcionan

[3] cost-centre.

[4] También se permite dividir la memoria en función del tipo de datos al que pertenecen las celdas.

aproximaciones sobre los resultados que se obtendrían sobre arquitecturas muy diversas, e incluso inexistentes (como arquitecturas ideales de latencia cero y número infinito de procesadores), de forma que permiten hacer estudios de paralelización variando distintos parámetros de la arquitectura, todo ello sin necesidad de emplear máquinas reales.

Por otra parte, el hecho de realizar simulaciones en lugar de obtener directamente resultados de ejecuciones reales tiene la ventaja de que resulta sencillo relacionar los resultados de los distintos procesadores, lo cual resultaría bastante complejo (e ineficiente) en el caso de las ejecuciones reales. De todas formas, y como se expondrá en la última sección del capítulo, también es posible instrumentalizar las ejecuciones paralelas reales para obtener cierta información.

En el Capítulo 3 de [Rub99] se recoge una visión más amplia sobre el estado del arte de los perfiladores que se han desarrollado para lenguajes funcionales paralelos perezosos.

4.2.1 GranSim

Sin lugar a dudas, la herramienta más relevante que se ha desarrollado para analizar el rendimiento de programas funcionales paralelos es GranSim ([HLP94] [Loi96] [Loi98]). GranSim es un simulador de arquitecturas paralelas para ejecutar programas escritos utilizando GpH ([THJ+96]). GranSim posee unas buenas herramientas gráficas para visualizar el comportamiento paralelo de los programas, y la herramienta destaca especialmente por su gran flexibilidad, así como por su fiabilidad.

Su objetivo primordial era estudiar técnicas para analizar la granularidad de las hebras, así como para mejorar la eficiencia de los programas a partir de informaciones de granularidad. De hecho, el objetivo a largo plazo del proyecto es lograr paralelizar automáticamente los programas sin necesidad de anotaciones del programador. Por ese motivo, además de las anotaciones de GpH (`seq` y `par`), permite otras anotaciones que incluyen información de granularidad y sobre ubicación (para forzar que una hebra se evalúe en un determinado procesador, etc.). Asimismo, permite utilizar la información de granularidad para utilizar otras estrategias a la hora de convertir semillas en hebras, de forma que primero se conviertan aquellas semillas que tengan una anotación de granularidad mayor.

Visualización

GranSim genera dos tipos de gráficas: de actividad y de granularidad. Las gráficas de actividad muestran de tres formas distintas la evolución en el tiempo del número y del estado de las hebras, teniendo en cuenta que en GranSim una hebra puede encontrarse en cinco estados:

- *Running*: En ejecución.

- *Runnable*: Podría ejecutarse si hubiera procesadores disponibles.
- *Blocked*: Se encuentra bloqueada esperando datos.
- *Fetching* (este estado es propio de GpH): Está pidiendo clausuras a un procesador remoto, con el objetivo de evaluarlas localmente.
- *Migrating*: Está migrando de un procesador a otro.

Las gráficas generadas son las siguientes:

- Evolución global: Evolución en el tiempo del número de hebras que se encuentran en cada estado. Véase la Figura 4.2.
- Por procesador: Evolución en el tiempo del estado de cada procesador. Un procesador puede estar activo (color verde) o inactivo (color rojo). Además del estado del procesador, esta gráfica muestra información sobre cuántas hebras hay en el procesador en estado *runnable* (cuanto más oscuro es el verde, más hebras *runnable* hay en el procesador), y cuántas hay en estado *blocked* (además del color principal, existe una banda inferior de color azul, que es más gruesa cuantas más hebras estén bloqueadas en el procesador). Véase la Figura 4.3.
- Por hebra: Evolución en el tiempo del estado de cada hebra. Véase la Figura 4.4.

Las gráficas de granularidad muestran la distribución de las hebras en función de cuánto tiempo ha durado su ejecución, o en función de cuánta memoria han generado. Dichas gráficas pueden visualizarse de dos formas:

- En forma de histograma: La altura de cada barra indica cuántas hebras de las que se generaron tuvieron la granularidad que se indica en el eje x. Véase la Figura 4.5.
- De forma acumulativa: para cada granularidad, la altura de la gráfica indica cuántas hebras de las que se generaron tuvieron una granularidad igual o inferior a la granularidad que marca el eje x. Véase la Figura 4.6.

Flexibilidad de la herramienta

GranSim permite simular un amplio abanico de arquitecturas, para lo cual el usuario sólo necesita establecer una serie de parámetros que describan la máquina que quiere simular. Dichos parámetros no sólo incluyen el número de procesadores de la máquina, sino también el tiempo de latencia, tiempo para empaquetar un mensaje, tiempo para realizar un cambio de contexto, para crear una hebra, número de ciclos para realizar un operación aritmética,

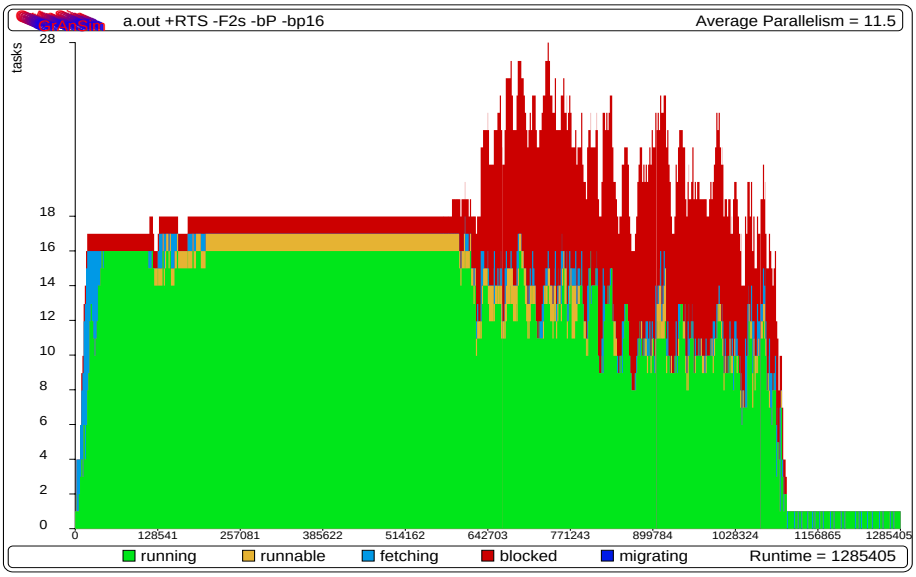


Figura 4.2: Gráfica de actividad global, simulando 16 procesadores.

una operación en punto flotante, una lectura de memoria, un almacenamiento en memoria y una instrucción de salto en memoria, etc.

Además, es capaz de simular arquitecturas ideales con un número infinito de procesadores (modo "Light" de GranSim). Pero, de todas formas, GranSim tiene restricciones en cuanto al tipo de arquitecturas que puede simular: no soporta arquitecturas no uniformes (con *clusters* de procesadores), ni tampoco procesadores vectoriales; asume que la latencia entre dos procesadores es independiente del tráfico, es decir, no tiene en cuenta el ancho de banda de los canales de comunicación; el número máximo de procesadores que puede simular está limitado (salvo en el modo Light) por la longitud de palabra de la computadora que se esté utilizando: si la longitud de palabra es de 32 bits, el número máximo de procesadores que podemos simular es 32.

La herramienta no sólo es flexible en cuanto a las arquitecturas, sino que también permite simular distintas estrategias:

- Migración: permite simular sistemas que autorizan la migración de hebras, y sistemas que no la autorizan.

- Modo de empaquetamiento: cuando se piden datos de un procesador a otro, se puede elegir entre empaquetamiento perezoso (sólo se empaqueta la clausura demandada) o especulativo (se empaqueta la clausura demandada, junto con un conjunto de clausuras que dependen de ella, con el objetivo de minimizar el número de comunicaciones).

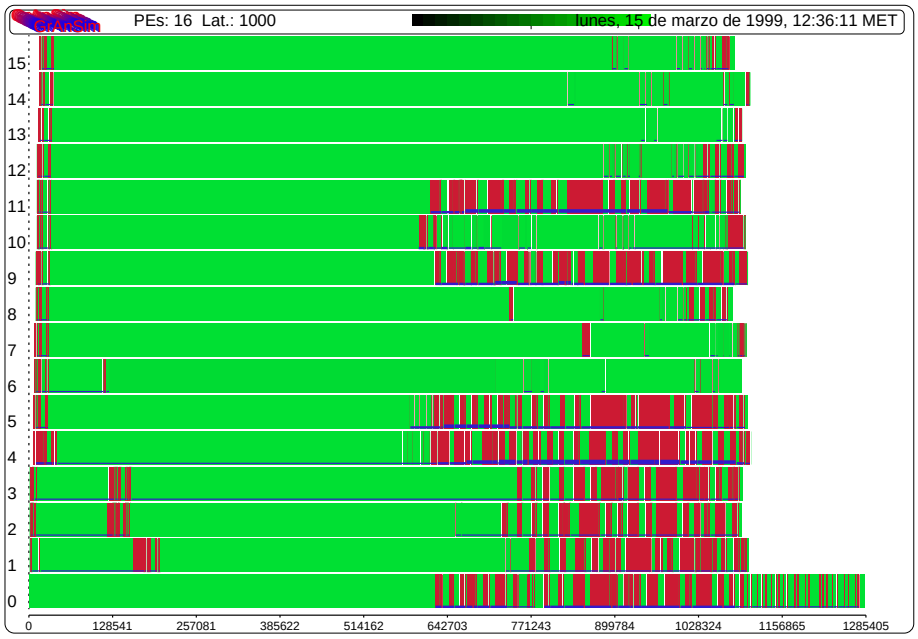


Figura 4.3: Gráfica de actividad por procesadores, simulando 16 procesadores

- Paquetes: permite fijar el tamaño máximo de los paquetes que se transmiten entre procesadores.
- Tipo de comunicaciones: cuando una hebra hace *fetching* a un procesador remoto, hay dos posibilidades: síncrona (la hebra se bloquea pero no pierde el control del procesador) o asíncrona (el procesador pasa a ejecutar otra hebra).

Implementación

Dado que GranSim pretende simular distintos tipos de arquitecturas, no pueden emplearse técnicas de muestreo para obtener tiempos de ejecución (como se hace en el perfilador secuencial de GHC comentado en la sección anterior), sino que la solución pasa por contar cuántos ciclos de reloj debería tardar en ejecutarse cada bloque básico de instrucciones. Para ello, por cada bloque básico se cuenta cuántas operaciones elementales (operaciones aritméticas, de punto flotante, lecturas de memoria...) realiza, y se calcula el número total de ciclos en función del tiempo necesario para realizar cada tipo de operación elemental[5].

[5]Dichos tiempos los puede variar el usuario en cada ejecución.

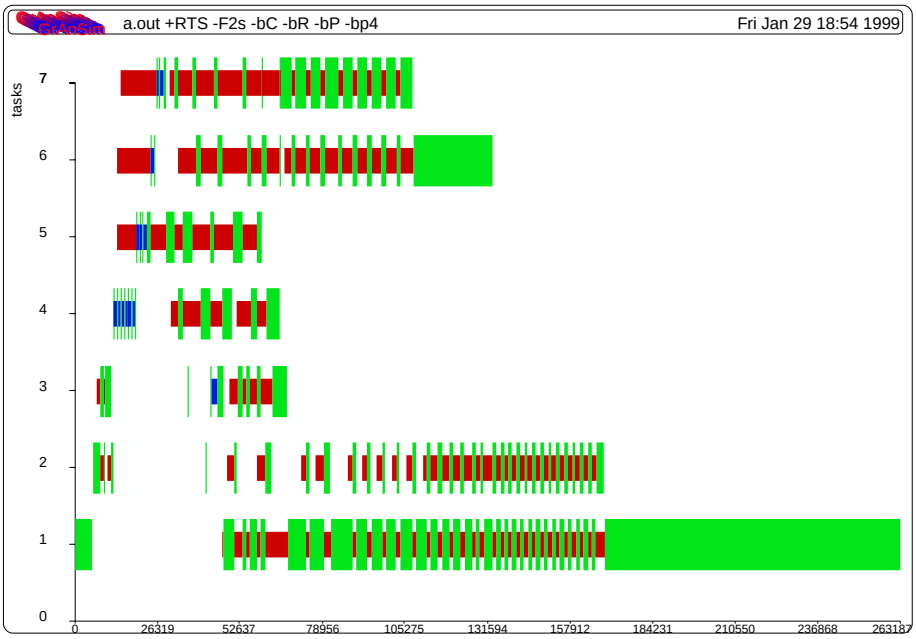


Figura 4.4: Gráfica de actividad por hebras

Por cada procesador es necesario disponer de un reloj que lleve la cuenta de cuántos ciclos de reloj han transcurrido en el procesador. Además, para gestionar el trabajo interno de cada procesador es necesario mantener dos colas: una cola de hebras y una cola de semillas. El tratamiento de dichas colas es análogo al que se hace en GpH: el planificador de hebras es injusto (una hebra se ejecuta hasta que termina o hasta que se bloquea), y una semilla sólo se convierte en hebra si no hay ninguna hebra que pueda ejecutarse. Para ser más exactos, el algoritmo que se sigue para buscar trabajo una vez que la hebra en curso ha terminado o se ha bloqueado, es el siguiente: primero se intenta ejecutar alguna hebra local, y si no la hay se busca una semilla local para convertirla en hebra. De no encontrarse, se intenta robar una semilla remota, y si no hubiera ninguna disponible se intenta robar una hebra remota. Este es el método empleado por defecto, si bien el usuario puede variarlo para estudiar distintas políticas.

Para simular varios procesadores utilizando uno único, es necesario una cierta justicia en cuanto al tiempo dedicado a simular cada procesador. Por otra parte, también es necesario que se simule correctamente la sincronización entre los procesadores, de modo que las comunicaciones se produzcan en los momentos precisos, sin que el tratamiento de ningún evento se adelante a otros eventos que deben ser anteriores.

La sincronización entre procesadores se consigue mediante una única cola

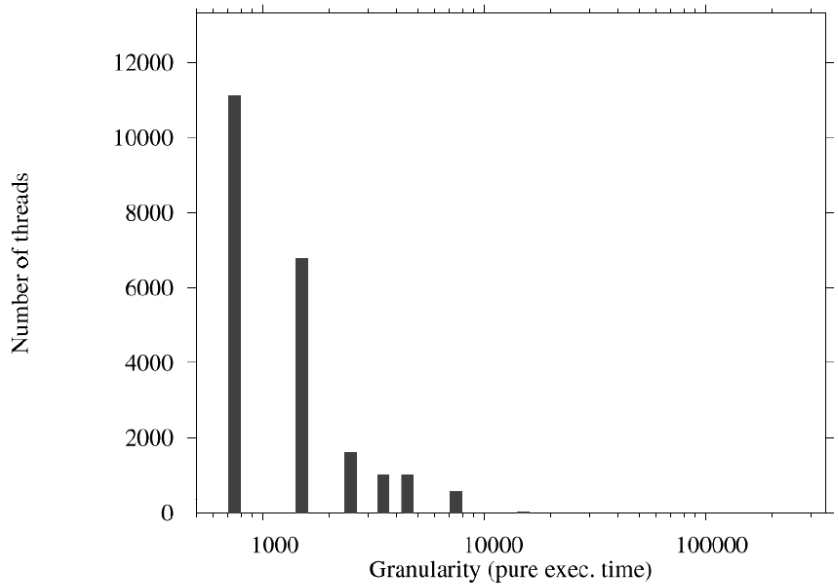


Figura 4.5: Histograma de granularidades

global de eventos (la Figura 4.7 muestra la estructura general de GranSim), que recoge cuáles son los eventos que van a suceder en el futuro, ordenados en función del instante en el que deberán tratarse. Por ejemplo, si en el instante t el procesador PE1 envía un dato al procesador PE2, y la latencia del sistema es de n ciclos, entonces se añadirá un nuevo evento a la cola de eventos, indicando que en el instante t+n PE2 recibirá dicho dato. Así se garantiza que las comunicaciones no se hacen efectivas hasta el momento preciso, de modo que PE2 no pueda utilizar los datos antes de que realmente los haya recibido. Análogamente se crean otro tipo de eventos para indicar cuándo debe crearse una hebra, cuándo debe morir, cuándo debe desbloquearse...

El reparto de tiempo entre los procesadores también se realiza a través de la cola global de eventos. Para ello, tras ejecutar cada bloque básico de instrucciones, se da paso a tratar otros eventos, no sin antes añadir a la cola un evento especial consistente en continuar la ejecución de la hebra actual. Básicamente, si prescindimos del tratamiento del resto de eventos, el sistema realiza una planificación *round-robin* para repartir el tiempo entre los procesadores.

Cada hebra lleva asociado un registro en el que se almacenan estadísticas sobre la hebra: en qué momento se creó, si ha migrado o no, cuántas palabras ha asignado en memoria, cuántos bloques básicos de instrucciones ha

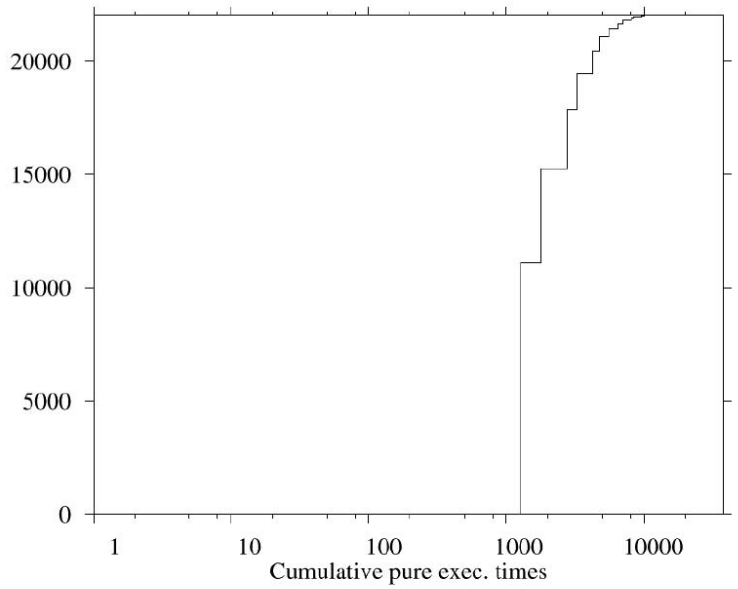


Figura 4.6: Gráfica de granularidades acumuladas

ejecutado, durante cuánto tiempo se ha estado ejecutando, cuánto tiempo (y cuántas veces) ha estado bloqueada, cuánto tiempo (y cuántas veces) ha estado leyendo datos remotos, etc. Estos campos se actualizan cada vez que ocurre un evento relevante, y se vuelcan en el fichero histórico en el momento en que muere la hebra.

Con el fin de poder generar todas las gráficas anteriormente mencionadas, la herramienta almacena en un fichero todos[6] los eventos relevantes que se producen: conversión de una semilla a hebra, activación de una hebra tras haberse bloqueado, migración de una semilla, migración de una hebra, *fetching* solicitado por una hebra, respuesta a un *fetching*, bloqueo de una hebra, robo de una hebra, robo de una semilla de otro procesador, etc.

La implementación del modo Light de GranSim es significativamente diferente. En el modo Light se supone que se dispone de un número infinito de procesadores, motivo por el cual no es necesario que mantengamos una cola de hebras por cada procesador, ya que cada procesador sólo se va a tener que encargar de una única hebra. Así pues, es suficiente con utilizar una única cola global de hebras y otra única cola global de semillas. Además,

[6]Realmente la herramienta permite seleccionar entre 4 niveles de información a registrar. De esta forma sólo se generan grandes ficheros cuando realmente se quieren utilizar para obtener todos los tipos de gráficas.

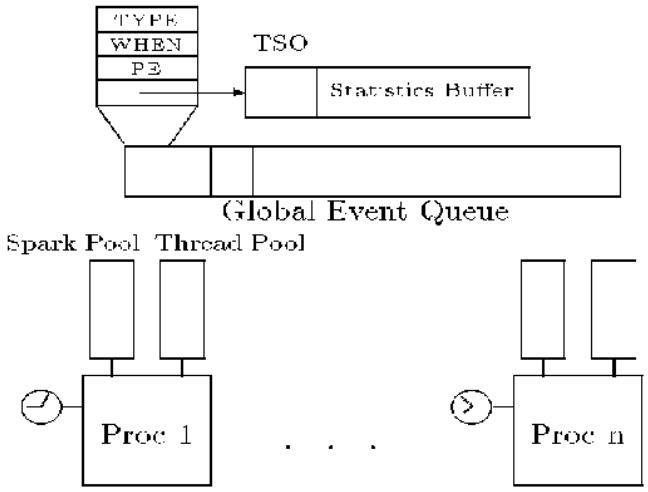


Figura 4.7: Estructura general de GranSim.

dado que disponemos de un número infinito de procesadores, las semillas siempre se convierten automáticamente en hebras.

Fiabilidad

Dado que GranSim es sólo un simulador, sus resultados no son completamente exactos. Por ejemplo, GranSim no tiene en cuenta todos los detalles de los costes de PVM. Tampoco calcula con exactitud el número de instrucciones máquina que se realizan de cada tipo, pues sólo hace estimaciones superficiales, pero no tiene en cuenta optimizaciones que puede realizar el compilador de C, como por ejemplo para minimizar el número de operaciones en punto flotante[7].

Para medir el grado de fiabilidad real de la herramienta, se ha utilizado GranSim para predecir el comportamiento que tendría un cierto programa (Linsolv [Loi97]) sobre la máquina GRIP ([HJ92]), que cuenta con 16 procesadores. Pues bien, GranSim preveía un paralelismo medio de 15, y una aceleración de 11.92, mientras que los datos reales obtenidos al utilizar GRIP fueron un paralelismo medio de 14.5 y una aceleración de 13.58. Así pues, las predicciones son bastante buenas, si bien debe tenerse siempre en mente que no son exactas.

[7]Los errores cometidos en el número de operaciones realizadas al utilizar procesadores SPARC no superescalares son del orden del 10% en operaciones aritméticas, 2% en lecturas/escrituras en memoria, 20% en saltos y del 14% en operaciones en punto flotante.

Restricciones

Además de las restricciones mencionadas previamente acerca de las arquitecturas que no podía simular, la principal restricción de GranSim es que sólo muestra cuántas hebras están en cada estado, pero no es capaz de devolver los datos en función del código fuente, es decir: podemos saber que hay muchas hebras bloqueadas, pero no podemos saber cuál es la parte del código fuente responsable de que se hayan generado esas hebras, ni tampoco podemos saber la causa por la que están bloqueadas. Las dos secciones siguientes comentan sendas extensiones a GranSim cuyo objetivo es solventar esta restricción.

4.2.2 Perfiles por estrategias

En [KHT98], y siguiendo el concepto de estrategia que se introdujo originalmente en [THLP98], D. King y cia. extienden el simulador GranSim de forma que el usuario pueda ver cuánto paralelismo está generando cada estrategia (e incluso cada uso distinto de la misma estrategia). De esta forma se facilita el modo de determinar cuáles son las partes del programa que están mal paralelizadas, pues serán aquellas correspondientes a las estrategias que no están generando paralelismo. Así se resuelve una de las principales restricciones de GranSim, que no permitía relacionar su salida con el código fuente.

A modo de ejemplo, consideremos el siguiente fragmento de código, en el que la función `using` aplica una estrategia, y `markStrategy` es la encargada de anotar las hebras con su estrategia correspondiente:

```
main = print ((a,b) `using` parPair rnf rnf)
   where
     a = . . . `using` markStrategy "A" stratA
     b = . . . `using` markStrategy "B" stratB
     stratA = . . .
     stratB = . . .
```

En este ejemplo, para calcular la tupla `(a,b)` se utiliza una estrategia que consiste en utilizar una hebra para reducir a forma normal[8] la primera componente, y otra hebra para reducir a forma normal la segunda componente. Más internamente, la evaluación de `a` se realiza mediante la estrategia `stratA`, y sus hebras se anotan con el nombre "A". Análogamente, `b` se evalúa mediante la estrategia `stratB`, y sus hebras se anotan con el nombre "B".

Una posible gráfica[9] sería la que puede apreciarse en la Figura 4.8. Nótese que en dicha figura se aprecia claramente cuánto paralelismo está generando

[8] rnf: reduce to normal form.

[9] Se pueden generar el mismo conjunto de gráficas que en GranSim.

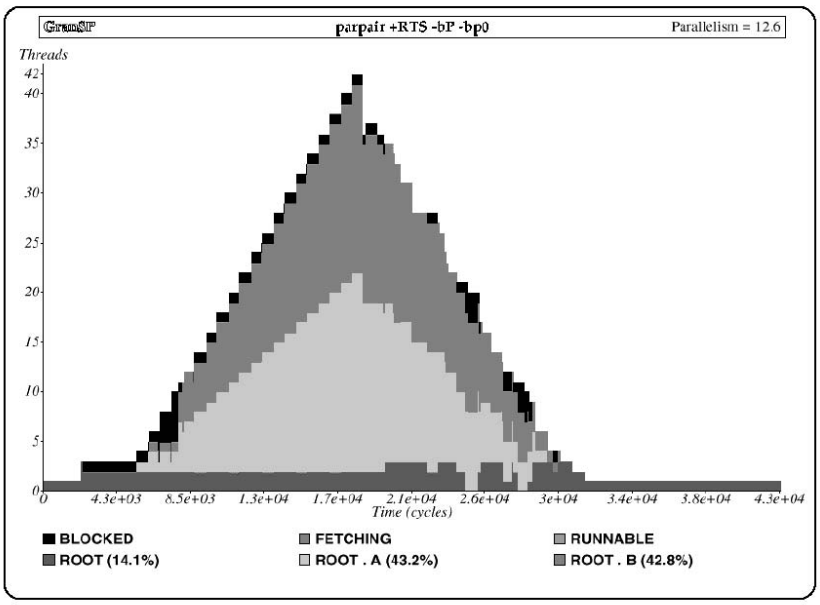


Figura 4.8: Perfil por estrategias

cada estrategia[10], de modo que si una de ellas no generara paralelismo, de inmediato sabríamos sobre qué parte del código fuente deberíamos dirigir nuestros esfuerzos.

La implementación es bastante sencilla. Sólo requiere añadir dos campos más a cada TSO (la estrategia y la hebra creadora), y definir una nueva función `markStrategy`. Al crear una nueva hebra, dichos campos se rellenan a partir del TSO del padre, y no se modificará el campo correspondiente a la estrategia hasta que se ejecute `markStrategy`. `markStrategy` tiene dos argumentos (una cadena de caracteres m y una estrategia e), y devuelve otra estrategia. Su ejecución se limita a devolver su segundo argumento, pero modificando en el TSO de la hebra actual el campo correspondiente a la estrategia creadora:

$$\text{TSOactual.marca} \leftarrow \text{TSOactual.marca} \mathbin{++} \text{"."} \mathbin{++} \text{m}$$

Nótese que en el TSO no se sustituye por completo la marca, sino que sólo se concatena la nueva marca. De esta forma, con el mismo esfuerzo

[10] En las gráficas se utiliza un color distinto para las hebras *running* correspondientes a cada estrategia. Sin embargo, las hebras *blocked*, *fetching* y *runnable* no se distinguen en función de la estrategia que las creó, pues sólo estamos interesados en las hebras que producen paralelismo real.

de implementación[11], no sólo se consigue saber qué estrategias generan más paralelismo, sino también cuáles son las secuencias de llamadas que generan el paralelismo.

4.2.3 GranCC

K. Hammond y cia. ([HHLT97, HKL+00]) han extendido el compilador GHC para mezclar el sistema GranSim y el perfilador de GHC (véase la Sección 4.1). De esta forma se pretende que no sólo podamos saber cuánto paralelismo se está generando, sino qué partes del programa (qué centros de costes) son las que generan más paralelismo, y qué partes menos, siendo el aspecto de las gráficas generadas muy similar al de la herramienta de la sección anterior. Además, y a diferencia de la herramienta anterior, no sólo podemos saber cuánto paralelismo genera cada parte del programa, sino que también podemos combinar dichos datos con los datos sobre cuánto tiempo y memoria consume cada centro de costes.

A nivel conceptual, la implementación de la herramienta sólo necesita extender cada TSO con un campo correspondiente al centro de costes al que pertenece. Dicho campo deberá rellenarse convenientemente en el momento de creación de cada hebra. Aunque conceptualmente la implementación de esta herramienta es sencilla, realmente el esfuerzo necesario para desarrollarla es de grandes proporciones, pues implica mezclar el código fuente del perfilador de GHC con el código fuente de GranSim.

El principal inconveniente de este enfoque es que cada vez que se produce un cambio de evaluación de un centro de costes a otro, es preciso registrarlo en el fichero que se usará para el postprocesamiento. Esto hace que dicho fichero adquiera habitualmente tamaños gigantescos, y que su postprocesamiento requiera no sólo mucho espacio en memoria, sino también mucho tiempo de cómputo.

4.3 Paradise

4.3.1 Introducción

Como ya se dijo en la Sección 4.2, resulta complicado razonar sobre el comportamiento paralelo real de los programas que escribimos, por lo que necesitamos obtener una cierta realimentación, lo cual puede lograrse usando un simulador como GranSim (véase la Sección 4.2.1).

En la presente sección se describe Paradise (*PARAllel DIstribution Simulator for Eden*) [**HPR00**], herramienta desarrollada como parte de esta tesis. Paradise es fundamentalmente una modificación del perfilador GranSim, de modo que trabaje sobre programas Eden en lugar de sobre programas GpH.

[11] El sobrecoste es mínimo, y el único riesgo es que haya una cadena de anidamientos de estrategias demasiado larga, lo cual no sucede habitualmente.

Primero se describen las gráficas que deseamos que se obtengan como resultado de utilizar la herramienta, y posteriormente se describen las distintas fases en las que se ha dividido el desarrollo de la misma.

4.3.2 Salidas que se espera que produzca Paradise

Queremos reutilizar GranSim de modo que trate programas Eden. Ahora bien, no sólo queremos obtener el mismo tipo de gráficas que GranSim, sino que queremos extender la herramienta de modo que podamos ver dichas gráficas de un modo "distinto". Además, también queremos poder generar otros tipos de gráficas que tienen sentido para lenguajes como Eden, pero no para GpH.

Gráficas reutilizadas

Paradise debe poder generar todas las gráficas que produce GranSim:

- **Comportamiento global**: Queremos poder obtener una única gráfica en la que se aprecie fácilmente la evolución en el tiempo del número de hebras que hay en el sistema, distinguiendo cuántas de ellas se encuentran en cada uno de los estados posibles (*running*, *runnable* y *blocked*[12]).

- **Por procesador**: Evolución en el tiempo del número de hebras que hay en cada procesador, distinguiendo las hebras en función del estado en el que se encuentren.

- **Por hebras**: Evolución en el tiempo del estado en el que se encuentra cada una de las hebras.

- **Gráficas de granularidad**: Representación en forma de histograma del número de hebras de cada nivel de granularidad (variable por el usuario).

A diferencia de GpH, ahora no estamos interesados en el modo *Light* del simulador. La razón por la que se introdujo en GpH radica en la metodología de uso del lenguaje, que se basa en comenzar la paralelización exponiendo el máximo grado de paralelismo posible, para luego ir reduciéndolo de modo que la granularidad se ajuste adecuadamente a la máquina real. En Eden se pretende controlar la granularidad desde un principio, de modo que no se creen falsas expectativas por la suposición de una máquina ideal con infinitos procesadores y, sobre todo, coste de comunicaciones cero.

[12] En Eden no existen los estados *fetching* y *migrating*.

Gráficas modificadas

GranSim proporciona diversas gráficas que permiten determinar si el grado de paralelismo de un programa es satisfactorio o no, pero no proporciona ningún tipo de relación entre las gráficas generadas y el código fuente del programa. Esta circunstancia dificulta notablemente el proceso de optimización de los programas, ya que no están claras las fuentes de las ineficiencias.

En Paradise (al igual que en las herramientas vistas en las Secciones 4.2.2 y 4.2.3), se pretende relacionar la salida del simulador con el código fuente. En Eden, el paralelismo está asociado a las abstracciones y concreciones de procesos. Así pues, lo razonable sería que las gráficas estuvieran relacionadas con dichas partes del código. Así, deberíamos poder obtener todas las gráficas del apartado anterior (*comportamiento global*, *por procesador*, *por hebras* y *granularidad*), pero *restringidas* para visualizar sólo las hebras correspondientes a un determinado grupo de abstracciones o concreciones de procesos.

Asimismo, también sería deseable poder *particionarlas*, de modo que, por ejemplo, para el comportamiento global no veamos sólo cuántas hebras hay en cada estado, sino también cuántas hebras de cada abstracción/concreción están en cada estado.

Por último, resultaría útil que la gráfica de comportamiento *por hebras* incluyera como información adicional en qué procesador se encuentra cada hebra, puesto que ello facilitaría razonar sobre ineficiencias debidas al reparto de carga entre los procesadores.

Nuevas gráficas

Eden no es un lenguaje funcional perezoso puro, puesto que cada hebra evalúa sus salidas de forma especulativa, sin necesidad de que nadie demande los valores que produce. Por tanto, si el programador no utiliza correctamente el lenguaje, puede generar programas que realizan grandes cantidades de trabajo completamente innecesario, y no le resultará sencillo detectar la causa de la ineficiencia, a menos que reciba algún tipo de información de ayuda por parte del simulador.

Por dicha razón, sería deseable una *gráfica de especulación*: para cada grupo de abstracciones o concreciones de procesos, queremos ver la evolución en el tiempo de la cantidad de memoria que se ha enviado como salida de sus hebras, pero que aún no ha sido utilizada en el correspondiente proceso receptor. Asimismo, no sólo queremos obtener la diferencia entre lo producido y lo usado en valores absolutos, sino también porcentualmente.

Además, para evitar suministrar demasiados datos al usuario, será posible restringir la salida de esta gráfica, de forma que sólo se visualice información acerca de aquellas partes del código que más especulen.

Como puede apreciarse, esta gráfica no refleja realmente cuánto trabajo

innecesario se ha realizado, sino cuánta memoria se ha generado de forma especulativa sin que nadie la use. Aunque no es exactamente lo que desearíamos, es un método indirecto que puede ayudarnos a detectar trabajo innecesario.

Finalmente, otra característica propia de Eden es que, a diferencia de GpH, es posible duplicar un mismo trabajo en varios procesadores. Por dicho motivo, además de la gráfica de especulación, se proporcionará un nuevo tipo de información (esta vez en modo texto) con el objetivo de determinar la cantidad de trabajo que se ha duplicado. Básicamente, por cada par de hebras se indicará cuántas clausuras comunes han sido evaluadas por separado en distintos procesadores. Ahora bien, con el objetivo de relacionar la información con el código fuente, en vez de producir información sobre cada par de hebras, se producirá información por cada par de partes del código fuente que el usuario estime oportuno. Como puede apreciarse, no se calcula realmente cuánto tiempo se ha empleado realizando trabajo duplicado (lo cual sería sumamente complejo), sino que sólo se calcula el número de repeticiones efectuadas.

Dado que la información anterior puede ser excesiva para una primera aproximación, inicialmente se proporcionará únicamente la cantidad total de repeticiones de trabajo que se han realizado. Posteriormente el usuario podrá ver cuánto trabajo repetido se ha hecho en cada marca (véase la Sección 4.3.4) y finalmente, si desea toda la información, podrá verla por pares de marcas.

4.3.3 Paradise-0.0

Como primer paso en el desarrollo del simulador, Paradise-0.0 sólo proporciona las mismas gráficas que proporciona GranSim, sin posibilidad de relacionarlas con el código fuente, y sin ninguna información acerca de la especulación de los programas. Aunque pueda parecer que esta es la parte más sencilla, realmente requiere una cuidadosa labor de reingeniería sobre el código fuente de GranSim.

Además de las diferencias de bajo nivel entre las implementaciones subyacentes de GpH y de Eden, las principales diferencias conceptuales que deben subsanarse para poder convertir GranSim en un simulador de Eden son las siguientes:

- **Planificador justo**: Eden es un lenguaje que necesita que el planificador de tareas sea justo, puesto que de lo contrario una hebra podría tomar el control del procesador y producir datos indefinidamente de forma especulativa, sin permitir la ejecución del resto de hebras que se encuentren en el mismo procesador. Por tanto, programas terminantes podrían convertirse en no-terminantes. Ahora bien, GpH no requiere justicia en el reparto de tiempo entre las hebras y, por consi-

guiente, GranSim tampoco utiliza un planificador justo, de modo que es necesario modificarlo para Paradise.

- **Semillas vs hebras**: En GpH el programador no es quien decide qué expresiones se van a calcular en paralelo, sino que sólo puede introducir anotaciones del estilo "esta expresión puede ejecutarse en una hebra independiente". Para cada una de las expresiones anotadas, se genera una *semilla*, que puede convertirse o no en una hebra, siendo el RTS el responsable de tomar esta decisión.

  Ahora bien, en Eden el programador es quien decide exactamente cuántas hebras deben generarse y qué debe ejecutarse en cada hebra. Así pues, es necesario modificar GranSim de modo que todas las semillas se conviertan siempre en hebras.

- **Duplicación de trabajo**: A diferencia de GpH, en Eden es posible duplicar la evaluación de una misma expresión en distintos procesadores. Por ejemplo, si tenemos un programa como

```
let
  x = ...
in
  p x # x
```

  entonces `x` se calculará tanto por el proceso padre como por el hijo (siempre y cuando `p` demande el valor).

  Dado que en GpH nunca se duplica trabajo, en GranSim no es necesario duplicar clausuras al moverlas o copiarlas de un procesador a otro, sino que basta con indicar en qué procesadores se encuentran. Ahora bien, en Paradise no puede permitirse que una misma clausura actualizable esté en dos procesadores, pues en tal caso, al evaluarla el primer procesador, el segundo ya la encontraría evaluada, y no se simularía fielmente Eden, pues no se duplicaría trabajo. Esto implica que es necesario modificar el algoritmo de empaquetamiento de grafo de clausuras de GranSim, de modo que se hagan copias reales y no sólo simulaciones.

- **Reparto de carga**: Las estrategias de reparto de carga de GpH y Eden son completamente diferentes. En GpH, los procesadores que no están ejecutando ninguna hebra, y que no tienen ninguna semilla local, envían mensajes a los otros procesadores preguntándoles si tienen alguna semilla, y en caso afirmativo se "roba" una de dichas semillas. Sin embargo, en Eden cada vez que se lanza un nuevo proceso se le asigna a un procesador, y no existe ninguna posibilidad de que posteriormente pase a ejecutarse en otro procesador.

- **Mensajes Eden**: En Eden existen varios tipos de mensajes que pueden enviarse de unos procesadores a otros (principalmente para crear nuevos procesos y para enviar datos de unos procesos a otros), por lo que es necesario añadir su simulación a GranSim. La mejor forma de hacerlo es utilizar el mismo mecanismo de GranSim, es decir, utilizar un tipo de evento distinto para cada tipo de mensaje que pueda transmitirse.

Implementación de los mensajes propios de Eden

Dado que en Eden existen tipos de mensajes que no aparecen en GpH, necesitamos introducir tantos nuevos eventos globales como nuevos tipos de mensajes tenemos. Los mensajes que pueden enviarse de un procesador a otro son:

- *createProcess*: Envía un mensaje con toda la información necesaria para que un procesador cree y evalúe un nuevo proceso.
- *Ack*: Confirmación de la recepción del mensaje anterior.
- *sendHead*: Envío (a través de un canal) de un valor de salida de un proceso.
- *sendVal*: Envío (a través de un canal) de un valor de salida de un proceso. A diferencia del mensaje anterior, tras enviar el dato se cierra el canal.
- *DCH*: Conexión a un canal dinámico.
- *sendTerm*: Notificación de que una hebra debe terminar como consecuencia de que el proceso receptor de sus mensajes ya ha concluido su cómputo.

En la implementación real (véase la Sección 3.3.3), dichos mensajes se envían utilizando la librería de paso de mensajes MPI, pero en GranSim sólo queremos simularlo, generando eventos en los lugares en que realmente deberían enviarse mensajes. Por ejemplo, para el mensaje *sendVal*, la hebra enviante debe:

1. Empaquetar en un mensaje MPI los datos que deben enviarse.
2. Enviar el mensaje MPI.
3. Cerrar el canal de salida.

Mientras que el receptor, cuando llega el mensaje debe:

1. Desempaquetar el mensaje MPI.

2. Escribir los datos recibidos en la posición adecuada de la memoria, y despertar a las hebras que estuvieran bloqueadas en espera de los datos que acaban de llegar.

3. Cerrar el canal de entrada.

En Paradise debe hacerse exactamente lo mismo, salvo por el hecho de que en lugar de enviar un mensaje MPI, se crea un nuevo evento global que indique que *lat* ciclos después del instante actual, al procesador receptor llegará un mensaje con los datos enviados, donde *lat* es la latencia del sistema. Por su parte, cuando el procesamiento de la cola global de eventos nos conduzca a tratar dicho evento, el proceso receptor ejecutará los mismos pasos (2 y 3) que en la versión real.

Además, para simular correctamente los tiempos, es necesario calcular el tiempo que se tarda en empaquetar y desempaquetar los datos, en cerrar un canal, etc.

Estado actual

Actualmente existe una versión de Paradise-0.0 desarrollada por el autor de esta tesis, e integrada en el compilador de Eden. Dicha versión soporta todas las características de Paradise-0.0. El número máximo de procesadores que puede simularse es 128.

En las figuras 4.9, 4.10 y 4.11 puede apreciarse el aspecto de las gráficas generadas. En el resto de la tesis se incluyen algunos ejemplos reales de uso de la herramienta.

4.3.4 Paradise-1.0

A diferencia de la versión anterior, Paradise-1.0 sí relaciona la salida producida con el código fuente. Es decir, es capaz de generar todos los tipos de gráficas propuestos en la introducción, salvo las gráficas de especulación.

Introducción de marcas

La idea para relacionar la salida con el código fuente es muy similar a la utilizada en el perfilador de estrategias (véase la Sección 4.2.2), y consiste en añadir una nueva función para introducir *marcas*, cuya semántica operacional se correspondería con la siguiente definición Haskell:

```
mark :: String -> a -> a
mark nombre expr = expr
```

Como puede verse, la semántica operacional de `mark` consiste simplemente en devolver su segundo argumento. Ahora bien, en tiempo de ejecución, su evaluación produce un efecto lateral (no representable en Haskell) consistente en modificar el nombre de la hebra que se encuentre en ejecución:

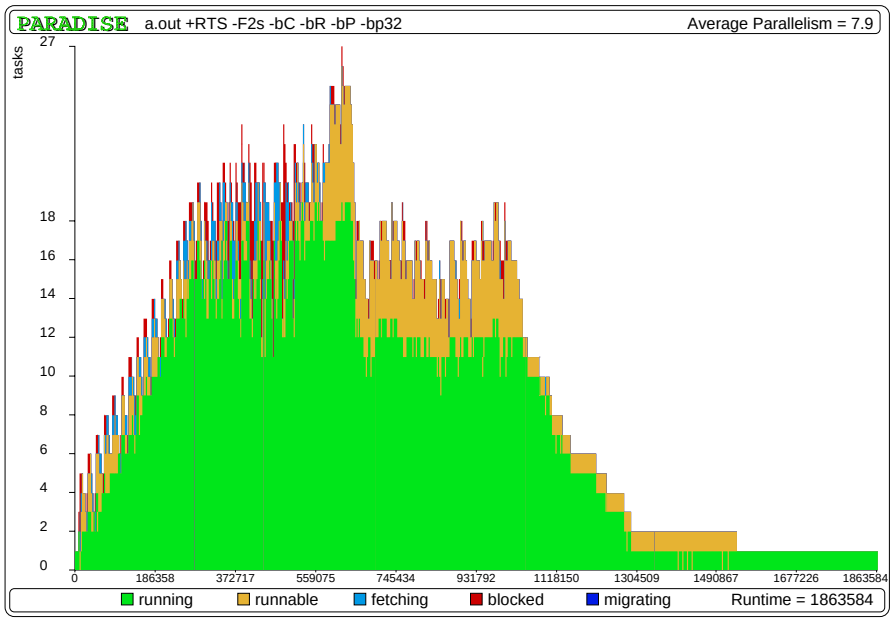


Figura 4.9: Gráfica de comportamiento global.

```
tso.marca <- tso.marca ++ "." ++ nombre
```

Como puede apreciarse, no se cambia el nombre de la hebra por completo, sino que sólo se le añade una nueva cadena. De esta forma, es posible recordar la cadena completa de lanzamiento de procesos que ha tenido lugar hasta que se ha creado la hebra. Así se consigue una mayor flexibilidad con un coste muy bajo[13].

Modo de uso

Como puede observarse, el tipo de `mark` es muy general, por lo que puede usarse en cualquier lugar del código fuente. Ahora bien, parece razonable restringirnos a usarlo sólo en los lugares en los que tenga más sentido. Dado que el objetivo es anotar el paralelismo, sólo deberíamos utilizarla en los puntos en los que aparece el paralelismo, es decir, en las abstracciones y/o en las concreciones de procesos La metodología recomendada es que sólo debería usarse justo después de que se cree una nueva hebra. Así, el programador

[13] A diferencia del perfilador secuencial de pilas de centros de costes (Sección 4.1), en programación paralela no tiene sentido que haya una cadena de anidamientos muy larga, pues conduciría a una granularidad demasiado fina. Por tanto, no se necesitan complejos algoritmos para minimizar las longitudes de las pilas de centros de costes, sino que basta con concatenar la nueva marca.

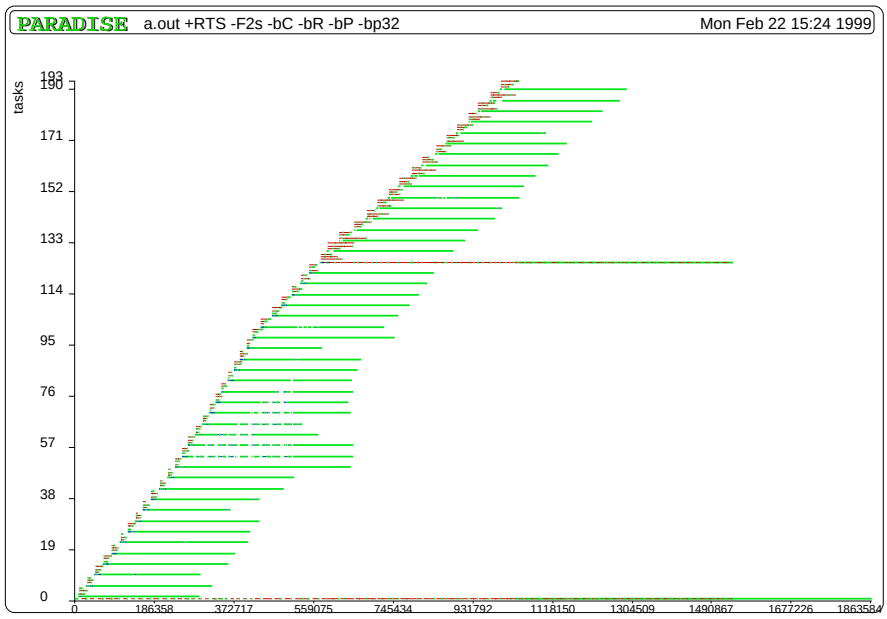


Figura 4.10: Gráfica por hebras.

podría introducir marcas en tres situaciones distintas, dependiendo del tipo de información que quiera obtener:

Abstracciones de procesos: Si se quiere saber cuánto paralelismo genera cada abstracción, puede asociarse una marca con cada proceso:

```
myProcess = mark "mp" (process x -> ...)
```

Dado que todas las hebras de la misma abstracción se marcan con el mismo nombre, podremos obtener la cantidad de paralelismo producida por cada abstracción.

Concreciones de procesos: A veces también estamos interesados en distinguir las distintas concreciones de una misma abstracción de proceso. Por ejemplo, al paralelizar el algoritmo de ordenación por mezcla podemos querer distinguir las concrecciones de las partes izquierdas de las de las derechas:

```
parMsort h = process xs -> ys
  where ys | h <= 0 = seqMsort xs
           | h > 0  = ordMerge (mark "L" (parMsort (h-1)) # x1)
                               (mark "R" (parMsort (h-1)) # x2)
        (x1,x2) = repartir xs
```

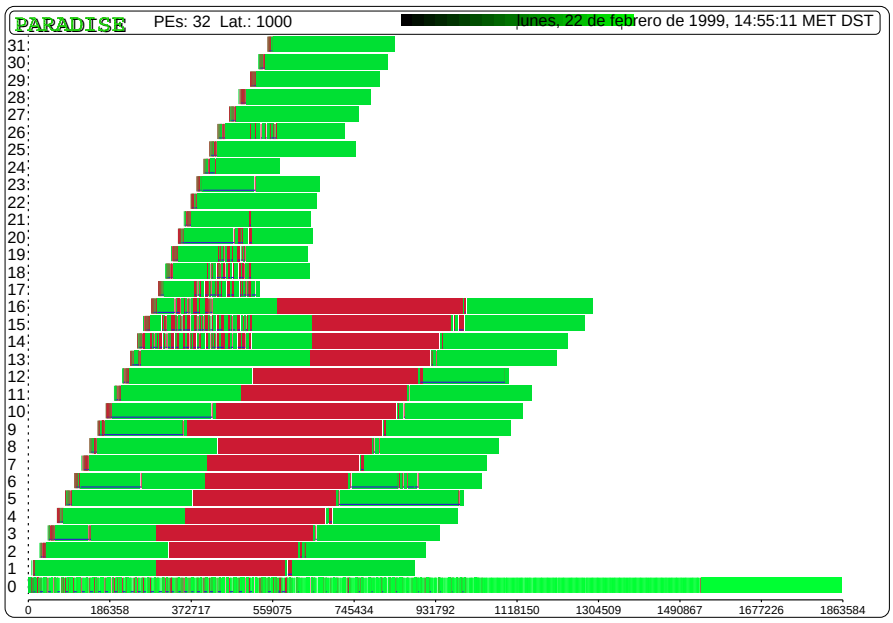


Figura 4.11: Gráfica por procesadores.

Nótese que no sólo es posible distinguir las concreciones de ambas ramas, sino que realmente podemos distinguir todas y cada una de las concreciones que se efectúen durante la ejecución del programa. La razón es que `mark` no cambia por completo el nombre de la hebra, sino que sólo añade un apéndice a la marca en curso, de modo que cada hebra recuerda su camino de concreciones. Si la profundidad del árbol de procesos es dos, los procesos se etiquetarían como muestra la Figura 4.12. Nótese que cada proceso tiene un nombre distinto, tal y como queríamos. Es más, al haberse generado siguiendo un patrón regular, no sólo es posible distinguir unos procesos de otros, sino también agruparlos por familias. De esta forma, podríamos distinguir cuánto trabajo se realiza en los hijos "izquierdos" y cuánto en los "derechos", así como cuándo se realizan los trabajos. Esto lo sabremos porque todas las hebras de los hijos izquierdos terminarán con la anotación "L", mientras que las de los derechos lo harán con "R". Asimismo, podríamos distinguir el paralelismo en función del nivel en el que se encuentren las hebras dentro del árbol de concreciones, de modo que veamos cuánto trabajo realizan las hojas (ordenando listas mediante `seqMsort`) y cuánto los nodos internos (repartiendo trabajo y mezclando listas ya ordenadas), para lo cual sólo hace falta contar el número de anotaciones

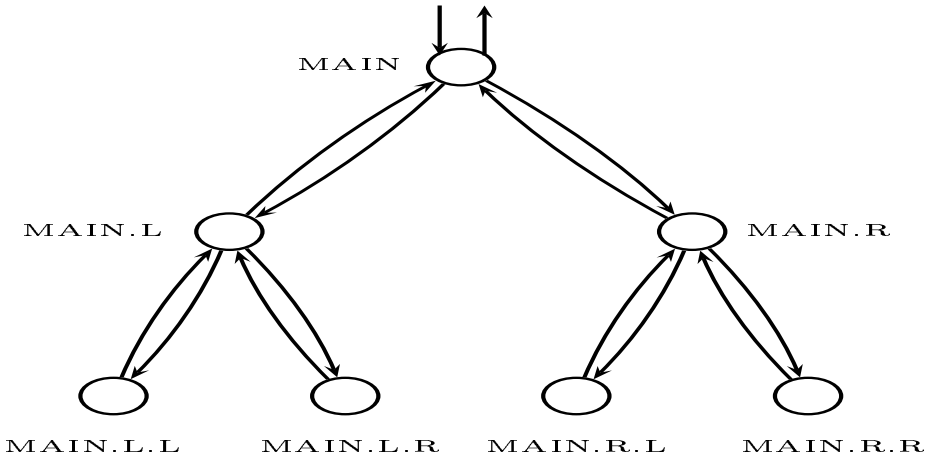


Figura 4.12: Árbol de procesos para mergeSort

anidadas. Por ejemplo, si tenemos 2 niveles de procesos, "MAIN.L.R" corresponderá a una hoja[14], mientras que "MAIN.R" corresponderá a un nodo intermedio.

Hebras: También podemos distinguir cada hebra concreta de una abstracción de proceso o de una concreción. La forma de hacerlo es trivial, pues sabemos que se creará una hebra por cada valor de la tupla de salida del proceso. Así, con

```
p = process (i1,i2,i3) -> (mark "o1" o1, mark "o2" o2)
```

tendremos que para todas las concreciones del proceso se distinguirán los cómputos de la primera hebra de los de la segunda, mientras que con

```
q # (mark "i1" e1, mark "i2" e2)
```

podremos marcar hebras concretas en las concreciones particulares.

Como hemos dicho, en principio `mark` sólo debe utilizarse para anotar hebras justo después de su nacimiento, aunque a veces también puede ser útil incluir la anotación en otros puntos. Así, podría ser interesante usar una anotación distinta para cada una de las alternativas de un proceso:

```
p x = case x of
       1 -> mark "1" (process (a,b) -> fact a)
       v -> mark "2" (process (a,b) -> fact b)
```

con el objetivo de saber cuál es la rama que se ha ejecutado.

[14] La hebra principal está anotada como "MAIN".

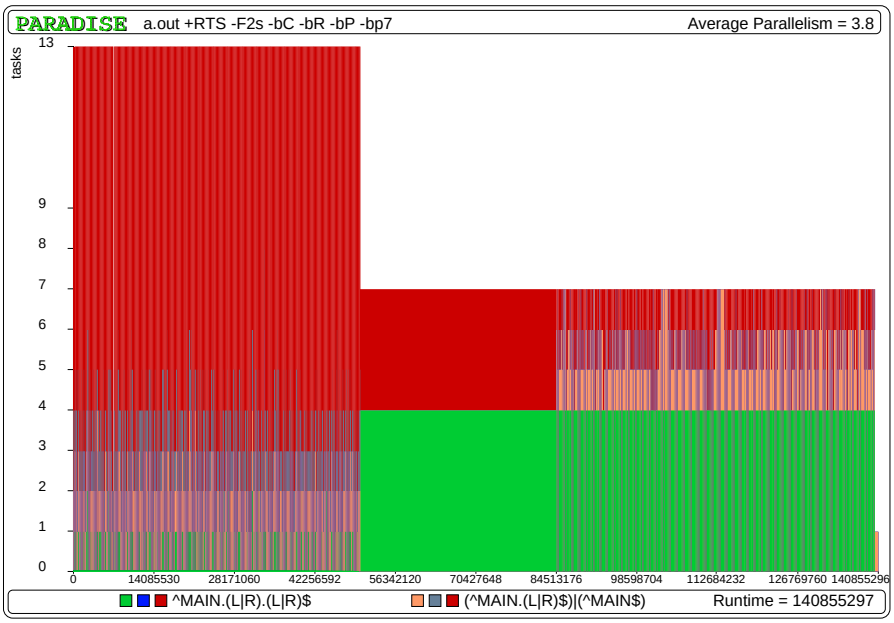


Figura 4.13: Gráfica de comportamiento global del algoritmo mergeSort con 7 procesadores y 2 niveles de árbol de procesos.

Herramientas de visualización

Las herramientas de visualización de GranSim han sido adaptadas a Paradise, por lo que pueden obtenerse exactamente el mismo tipo de gráficas. Ahora bien, las herramientas de Paradise también permiten aprovechar el sistema de marcas para mostrar más información al usuario. Así, ahora el comportamiento global de los programas no estará dividido en un número fijo de bandas, sino que habrá tres bandas (*running*, *runnable* o *blocked*) por cada marca diferente (véase la Figura 4.13). Del mismo modo, el comportamiento por hebras también incorpora el nombre de cada hebra, e incluso permite añadir el número de procesador en el que han sido ubicadas (véase la Figura 4.14).

Dado que la información recogida por las marcas puede ser excesivamente minuciosa, a menudo estaremos interesados en restringir la cantidad de información que queremos ver en las gráficas. Para ello, Paradise incorpora las siguientes facilidades:

Unificación: Todas las marcas que respondan a una expresión regular que suministre el usuario serán representadas como una única marca. Por ejemplo, si en el programa `mergeSort` queremos distinguir los procesos hoja de los nodos intermedios, basta utilizar expresiones regulares apro-

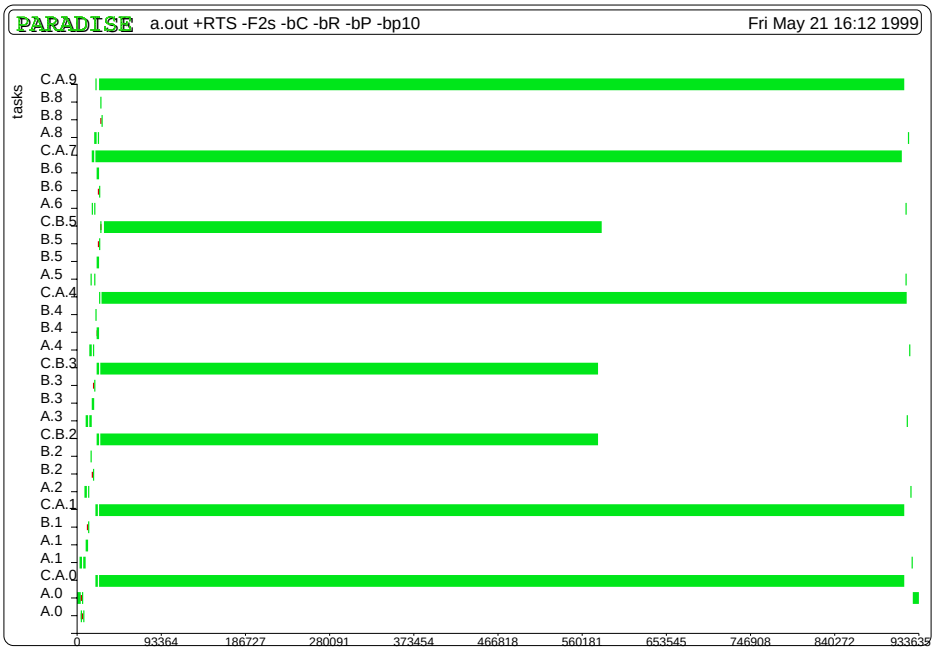


Figura 4.14: Gráfica de comportamiento por hebras, incluyendo información sobre el procesador en el que se ubican, y ordenadas por marcas.

piadas. Como sabemos que la profundidad del árbol de procesos es dos, las marcas de las hojas deberán ser de la forma `^MAIN.(L|R).(L|R)$`, donde `^` representa el comienzo de la cadena, `$` el final, y `|` la elección entre dos alternativas. En la Figura 4.13 puede verse el resultado que se obtiene con las anotaciones.

Filtro: Todas las marcas que encajen con una expresión regular que suministre el usuario no serán representadas en la gráfica. Esto nos permitirá ignorar las partes bien paralelizadas y centrarnos en aquellas que contengan la parte interesante a estudiar. Por ejemplo, en la Figura 4.15 nos restringimos a ver el paralelismo generado por los nodos internos de `parMsort`. Dado que genera poco cómputo, mejoraríamos la eficiencia global si sólo utilizásemos un nodo interno, y el resto fuesen hojas. Es decir, deberíamos aplanar el árbol de procesos.

Número de procesador: Las marcas de las hebras pueden extenderse con el número del procesador en el que residen.

Amplificación: La gráfica puede restringirse al período de tiempo que elija el usuario.

Colapsar bandas: Para evitar un excesivo número de bandas, se permite

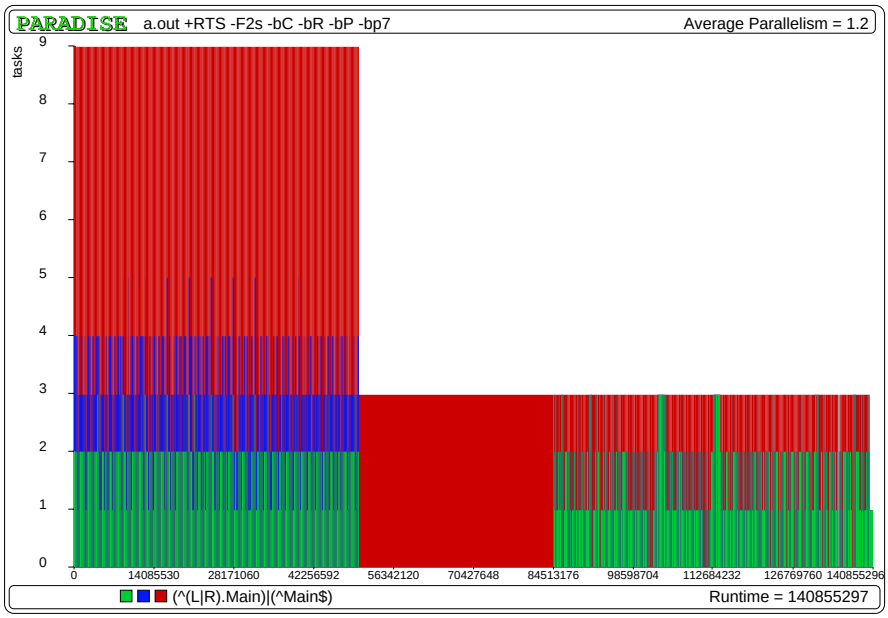


Figura 4.15: Gráfica de comportamiento global del algoritmo de mergeSort con 7 procesadores, 2 niveles de árbol de procesos y visualizando sólo los nodos intermedios.

colapsar en una única banda las hebras *blocked* de todas las marcas, así como las *running* y *runnable* de una misma marca.

Ordenación: La herramienta permite visualizar las hebras ordenadas lexicográficamente en función de sus marcas. De dicho modo se puede acceder más rápidamente a la información deseada.

Todas estas facilidades permiten extraer información de distintas formas a partir del mismo fichero de datos. De este modo, con simples postprocesamientos distintos podemos obtener incrementalmente la información que deseemos, sin necesidad de volver a compilar y ejecutar el programa con unas marcas distintas.

Implementación

Al igual que para la herramienta de la Sección 4.2.2, la principal modificación que debe realizarse en el RTS es añadir un campo extra a cada TSO. Dicho campo contendrá la marca asociada a la hebra, y sólo cambiará su valor cuando se ejecute la función `mark`. Además, para que la herencia de las marcas sea correcta, cada vez que se cree una nueva hebra, su correspon-

diente campo de marca se inicializará con el valor de la marca de su padre, siendo `MAIN` la marca de la hebra principal.

Finalmente, en el momento de la muerte de una hebra, en el fichero de historia se escribirá no sólo la información que se incluía hasta el momento, sino un campo adicional indicando la marca asociada a la hebra.

No es suficiente con modificar el RTS para que obtenga toda la información necesaria sobre las marcas, sino que también es preciso modificar las herramientas de visualización para que sean capaces de generar las nuevas gráficas.

Estado actual

Actualmente existe una versión de Paradise-1.0 desarrollada por el autor de esta tesis, e integrada en el compilador de Eden. Dicha versión soporta todas las características de Paradise-1.0, incluyendo todas las facilidades de postprocesamiento descritas.

4.3.5 Paradise-2.0

Finalmente, Paradise-2.0 es la versión que producirá todas las salidas esperadas para Paradise. Las únicas características que no estaban incluidas en Paradise-1.0 eran la generación de gráficas de especulación, y la generación de información sobre duplicación de trabajo.

Duplicación de trabajo

Para obtener información sobre el número de clausuras que se han evaluado por duplicado, necesitamos un generador de identificadores globales. Además, a cada clausura se le añadirá un nuevo campo que contenga cuál es su identificador global[15].

- Cada vez que se crea una nueva clausura duplicando otra:
  - Si la clausura original aún no tenía asociado un identificador global, entonces se le asigna uno nuevo, y a la nueva clausura se le asigna el mismo identificador global.
  - Si la clausura original ya tenía un identificador global, entonces se asigna ese mismo identificador a la nueva clausura.
- Cada vez que se evalúe una clausura que tenga asociado un identificador global, escribiremos en el fichero de historia un par (hebra evaluadora, identificador global).

De este modo, tras la ejecución se realizará un posterior postprocesamiento que permitirá detectar fácilmente cuántas clausuras se han evaluado dos o más veces, y qué hebras han sido sus evaluadoras.

[15] Si no lo tiene, el campo contendrá un valor predefinido que indique tal circunstancia.

Especulación

Con respecto a las gráficas de especulación, recordemos que Eden rompe la pereza de Haskell en dos aspectos: (1) una vez lanzado un proceso, siempre existe demanda para sus valores de salida; y (2) puede lanzarse un proceso antes de ser demandado (véase la Sección 5.1.1).

Por tanto, cabe la posibilidad de que se realice más trabajo del que sería estrictamente necesario. Además, aun cuando todo el trabajo realizado fuera útil, puede ser interesante conocer las velocidades de generación y consumo de los datos: si el productor es demasiado lento, puede convertirse en un cuello de botella, ralentizando la ejecución en el proceso receptor; si el productor es demasiado rápido, podría ser interesante modificar el programa fuente para tratar de ralentizarlo.

Además, las gráficas de especulación no sólo resultarían útiles para los programadores, sino también para los desarrolladores del compilador, pues podría ayudar a comparar el comportamiento de distintos planificadores de tareas.

Ejemplo Recordemos el ejemplo de la ordenación por mezcla introducido anteriormente:

```
parMsort h = process xs -> ys
  where ys | h <= 0 = seqMsort xs
           | h > 0  = ordMerge (mark "L" (parMsort (h-1)) # x1)
                               (mark "R" (parMsort (h-1)) # x2)
        (x1,x2) = repartir xs
```

En este caso, aunque no se hace ningún tipo de trabajo innecesario, sí que resultaría interesante comparar las distintas velocidades a las que se producen y consumen los datos. De este modo, sería sencillo determinar si la distribución de las listas a los procesos hijo es demasiado lenta o no.

Implementación Para obtener la gráfica de especulación mencionada en la Sección 4.3.2, añadiremos a cada clausura un campo extra de "especulación". Dicho campo servirá para determinar si la celda es producto de la especulación o no. Así, si la celda ha sido generada como salida de un proceso pero aún no ha sido utilizada, el campo de especulación contendrá el identificador de la hebra enviante (o la marca, si queremos utilizar Paradise-1.0), y en caso contrario tendrá un valor reservado (por ejemplo, cero). Para mantener correctamente los valores de dicho campo, basta tener en cuenta los siguientes casos:

- Cuando se recibe un dato a través de un canal de entrada, el campo de especulación de la celda correspondiente a dicho dato se marca con el identificador[16] de la hebra enviante.

[16] Puede usarse el identificador único de la hebra o su marca.

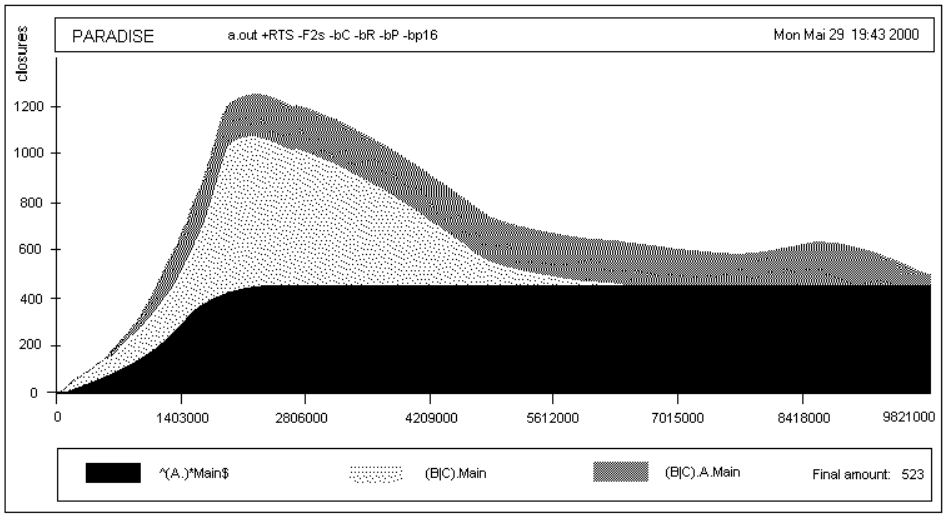


Figura 4.16: Gráfica de especulación.

- Cuando se crea una celda debido a cualquier otra circunstancia, el campo correspondiente a la especulación se marca con cero.
- Cada vez que se utiliza una celda, su campo de especulación se pone a cero, considerándose que una celda se ha utilizado cuando:
  - Realizamos un análisis de casos sobre su valor.
  - La aplicamos (si es una función).
  - La utilizamos como argumento de una función primitiva.
  - Cuando se actualiza el valor.

Con este campo, podemos hacer censos de la memoria a intervalos regulares, de modo que para cada hebra (o marca) podemos obtener cuánta memoria de la que ha generado (y que persiste en memoria) no ha sido aún utilizada. Así, si escribimos dicha información en un fichero, tras la ejecución sería sencillo realizar un postprocesamiento que mostrara las gráficas deseadas (véase la Figura 4.16).

Como ya se dijo, con estas gráficas sólo se obtiene una visión indirecta del trabajo innecesario que se ha realizado, pues sólo calculamos cuánta memoria se ha generado sin necesidad. Es más, ni siquiera se obtiene realmente eso, sino que sólo se visualiza cuánta memoria de la generada y no usada permanece en memoria sin ser recolectada.

Otro inconveniente de este enfoque es que no proporciona información sobre el tanto por ciento de datos aún no consumidos, sino sólo el valor absoluto. Por tanto, mostraría los mismos resultados si se han generado

1500 datos y se han consumido 1000, que si se han generado 550 y se han consumido sólo 50.

Para conseguir todos los resultados deseables, además del campo de especulación extra de cada celda, deberíamos añadir dos campos más a cada TSO. Dichos campos (llamémosles "producción" y "consumo") recogerían, respectivamente, la cantidad de memoria que ha enviado la hebra por su canal de salida, y la cantidad de dicha memoria que ya ha sido utilizada remotamente. Dichos campos se muestrearían periódicamente, y se escribirían en un fichero de salida. Tras la ejecución del programa, sería sencillo obtener cuáles son las hebras "sospechosas" de producir demasiados datos (tanto en valor absoluto como porcentual).

La forma de mantener correctamente los valores de los nuevos contadores es la siguiente:

- Cada vez que se envía un dato por un canal de salida se incrementa el contador de producción de la hebra.
- Cuando se recibe un dato por un canal de entrada, se anota su campo de especulación con el identificador de la hebra productora.
- Cuando se crea una nueva celda de otro modo, su campo de especulación se pone a cero.
- Cuando se usa una celda cuyo campo de especulación no estaba a cero, se pone a cero y se incrementa el contador de consumo de la hebra que envió el dato[17].

Nótese que aunque la información se recoja para cada hebra, es sencillo obtener la información acumulada por marcas, pues sabemos a qué marca pertenece cada hebra, así que sólo hay que acumular los contadores de cada hebra de la marca.

Estado actual

En este momento, el compilador de Eden está basado en la versión 3.02 del compilador GHC, por lo que actualmente la principal labor de implementación dentro del proyecto Eden es la migración a la versión actual de GHC (5.xx). El RTS de dicha versión de GHC ha sido completamente reescrito con respecto a la versión 3.02, por lo que la migración de Eden llevará bastante tiempo de trabajo. Dado que en estos momentos el desarrollo de Paradise-2.0 sólo podría llevarse a cabo sobre el compilador no actualizado, se ha decidido postponer su implementación hasta que se haya realizado la migración.

[17] Podemos acceder a dicha hebra gracias a que el campo de especulación contiene su identificador, y a que en una simulación se tienen accesibles todos los TSOs de cualquier procesador.

4.4 Perfiles paralelos reales

Para desarrollar programas paralelos eficientes resulta muy útil el uso de simuladores, pero también sería deseable disponer de perfiles reales sobre la máquina de destino. La razón es que un simulador sólo puede proporcionar aproximaciones sobre el comportamiento real, pero puede esconder características difícilmente simulables. Por ejemplo, en GranSim no se tiene en cuenta la jerarquía de memoria, ni el ancho de banda entre los procesadores, y se supone que la latencia es siempre la misma entre todos los procesadores.

Cuando se realizan perfiles reales, es fundamental evitar introducir distorsiones en el comportamiento del programa. Por este motivo no podemos obtener tanta información como con un simulador. Ahora bien, de todas formas puede obtenerse la suficiente información como para determinar si las predicciones del simulador fueron correctas. En el resto de la sección se describe cómo debería ser un perfilador real para Eden.

Grado de paralelismo

Para determinar el grado de paralelismo de un programa, deberíamos analizar los procesadores por separado, para posteriormente juntar los datos de todos ellos. Por cada procesador deberíamos mantener 4 contadores:

- **NP**: Número de procesos que hay en el procesador.
- **NH**: Número total de hebras que hay en el procesador.
- **NB**: Número de hebras del procesador que están bloqueadas.
- **NE**: Número de hebras que se encuentran en estado de ejecución en el procesador (0 ó 1).

El mantenimiento de estos 4 contadores es sencillo y barato, tanto en tiempo de ejecución como en memoria. Las únicas modificaciones que habría que introducir al compilador serían:

- Cada vez que se crea un nuevo proceso se incrementa el contador NP.
- Cada vez que muere un proceso se decrementa NP.
- Cuando se crea una hebra se incrementa NH.
- Cuando muere una hebra se decrementa NH.
- Cuando se bloquea una hebra se incrementa NB y se decrementa NE.
- Cuando un procesador pasa de estar inactivo a ejecutar una hebra, se incrementa NE.

- Cuando el planificador de tareas decide cambiar la hebra que se encuentra en ejecución, no hay que hacer absolutamente nada, pues no afecta a ninguno de los contadores.

- Cuando se recibe un valor a través de un canal, por cada una de las hebras que estuvieran bloqueadas en espera de dicho valor, se decrementa en una unidad NB.

Todos los casos anteriores están perfectamente identificados en el compilador, por lo que no hace falta introducir ningún tipo de comprobación en tiempo de ejecución, que pueda distorsionar las medidas. Simplemente se modifican unos contadores, lo cual requiere un tiempo mínimo. El único caso en el que podría parecer que debemos realizar comparaciones es en el último de los puntos, pues debemos tener en cuenta cuántas hebras estaban suspendidas en espera del dato. Ahora bien, el compilador ya necesita procesar una a una dichas hebras para "despertarlas", por lo que tampoco se introduciría ninguna distorsión significativa.

A intervalos regulares, dichos contadores serían muestreados y deberían escribirse en un fichero[18], de modo que al finalizar la ejecución pudiéramos postprocesar dicho fichero para obtener gráficas como la denominada "por procesador" en GranSim.

También sería deseable poder obtener una gráfica de comportamiento global. Para ello, sería necesaria una cierta sincronización entre los procesadores, pues de lo contrario no seríamos capaces de mezclar los datos producidos por cada uno de ellos. Ahora bien, introducir dichas sincronizaciones puede resultar complejo y lento, por lo que es preferible no introducirlas. Aún así, si la precisión de los relojes locales de cada procesador es similar, podemos combinar sin temor los datos generados por ellos, obteniéndose gráficas de comportamiento general bastante fiables.

Nótese que no es posible obtener información relacionada con el código fuente utilizando la función `mark`, pues ello requeriría recolectar mucha información durante la ejecución: al menos sería necesario un contador por cada marca y, sería necesario modificarlos cada vez que el planificador de tareas diese paso a otra hebra de otra marca distinta. Esto podría distorsionar gravemente los resultados obtenidos.

Análisis de granularidad

Para determinar la granularidad de las hebras creadas, puede utilizarse el mismo mecanismo que para los centros de costes de GHC (véase la Sección 4.1): simplemente tenemos que muestrear la hebra que se está ejecutando, en vez de muestrear el centro de costes en curso. Tras concluir la ejecución,

[18] Lógicamente debería existir un almacén intermedio, de modo que no fuera necesario escribir directamente en disco cada muestreo, sino que se escribiera en memoria principal.

aquellas hebras que se hayan ejecutado durante más tiempo también habrán sido muestreadas más veces.

Además, ahora sí que puede utilizarse la función `mark` para relacionar el grano de las hebras con el código fuente. Para ello basta con muestrear la marca de las hebras, en vez de la hebra propiamente dicha. De este modo puede obtenerse la misma información que con Paradise.

Especulación

En principio sería deseable poder obtener la misma información que con Paradise-2.0, pero eso implicaría recoger demasiados datos, por lo que los resultados obtenidos podrían estar distorsionados

Una posible simplificación que podría implementarse consistiría en utilizar sólo el campo de producción de las hebras, y muestrearlo sólo una vez: en el momento de muerte de la hebra. De esta forma, tendríamos datos exactos sobre cuánto ha producido cada hebra[19], aunque desconoceríamos cuánto se ha consumido. Ahora bien, es de esperar que los consumos totales sean equivalentes a los obtenidos con Paradise, pues dependen fundamentalmente del modelo de cómputo (un dato sólo llegará a utilizarse si realmente hay demanda para él), y no de cuestiones como el reparto de carga o el planificador de tareas. Así, combinando los datos reales con los de Paradise, podemos conocer la cantidad de memoria innecesaria que ha generado cada hebra (o marca).

De todas formas, debe recalcarse que sólo podemos obtener información sobre cuánto ha especulado cada hebra, pero no podemos saber en qué momentos de la ejecución ha sido más rápida dicha especulación. Es decir, no podemos ver la evolución en el tiempo de la especulación, sólo los datos finales.

Estado actual

Al igual que en el caso de Paradise-2.0, se ha decidio postponer la implementación del sistema hasta haber finalizado la migración del compilador de Eden a la nueva versión de GHC.

4.5 Perfiles secuenciales en Eden

Para que un programa paralelo sea eficiente, no basta con que la distribución del cómputo sea correcta, sino que es preciso que la versión secuencial original sea también eficiente, pues de lo contrario nunca obtendremos buenos resultados. Por ello, es fundamental disponer de un perfilador de programas secuenciales.

[19] O cada marca, pues podemos saber a qué marca pertenece cada hebra.

Así pues, Eden necesita también un perfilador secuencial. Afortunadamente, dado que Eden está implementado reutilizando GHC, podemos escribir la versión secuencial enteramente en Haskell, y utilizar el perfilador de GHC para mejorar la eficiencia, todo ello utilizando el mismo compilador. Pero también sería deseable que, una vez escrito el programa en Eden, pudiéramos seguir utilizando dicho perfilador. Para conseguirlo, basta compilar nuestros programas Eden añadiendo en tiempo de compilación la opción `-no-eden`. Dicha opción consigue que el programa Eden se convierta automáticamente a Haskell[20], de modo que pueden utilizarse todas las herramientas propias de GHC.

Básicamente, sólo hay que traducir

```
process ins -> outs
```

como

```
Process (\ins -> outs)
```

a la vez que se define `#` como

```
(Process f) # x = f x
```

donde `Process` es la única constructura del tipo `Process`:

```
data Process a b = Process (a -> b)
```

De este modo, todo programa Eden no reactivo se transforma en su equivalente programa secuencial Haskell, y puede utilizarse el perfilador de GHC sin ninguna restricción.

4.6 Conclusiones

En general no resulta sencillo comprender el comportamiento paralelo de los programas, por lo que las herramientas que proporcionan una realimentación al programador resultan muy útiles para detectar y corregir ineficiencias en su paralelización. Dichas herramientas son especialmente útiles en lenguajes que, como GpH, delegan en el RTS la mayor parte de las decisiones sobre la paralelización. En Eden no es tan necesario disponer de dicha realimentación, pero sigue siendo muy útil, especialmente para los programadores menos experimentados, que suelen tener problemas para entender porqué sus paralelizaciones no son tan satisfactorias como desearían. También resulta útil para programadores expertos, aunque sólo cuando se enfrentan a problemas complejos.

[20] Lógicamente existe una restricción, y es que el programa Eden original no debe ser reactivo, pues de lo contrario sería imposible escribirlo en Haskell.

Disponer de un simulador como Paradise no sólo resulta ventajoso para que los programadores mejoren la eficiencia de sus programas, sino que también es una herramienta muy valiosa desde un punto de vista didáctico. Cuando se forman nuevos programadores paralelos suelen tener problemas para entender qué no debe hacerse y porqué no. El uso de gráficas que muestran el comportamiento de los programas facilita la comprensión de los principios básicos que hay que tener en cuenta al paralelizar aplicaciones. Recuérdese que, como dice el refrán, *una imagen vale más que mil palabras*.

Cabe resaltar que la implementación de Paradise fue anterior a la implementación de la versión paralela del compilador de Eden. Gracias a ello, las primeras ejecuciones de programas Eden en las que realmente se podía apreciar la capacidad paralela del lenguaje se efectuaron con Paradise, y permitieron detectar problemas en el diseño del lenguaje. De este modo, dichos problemas pudieron corregirse antes de que se terminara el desarrollo del compilador completo.

Por último, cabe mencionar que un programa paralelo sólo puede ser eficiente si su correspondiente versión secuencial también lo es. Para ello es conveniente disponer de herramientas que proporcionen la realimentación necesaria para mejorar dicha eficiencia secuencial. En el caso de Eden, esto se ha conseguido gracias a la reutilización del compilador GHC, que no sólo es el compilador de Haskell que genera un código más eficiente, sino que también proporciona buenos perfiladores tanto de consumo de tiempo como de memoria.

Capítulo 5

Optimizaciones automáticas

Todo el contenido del capítulo es original de esta tesis. En él se describe cómo mejorar el proceso de compilación de Eden, introduciendo una fase intermedia que facilita la implementación de transformaciones automáticas que mejoren el código paralelo de los programas Eden. El capítulo comienza planteando qué tipo de mejoras serían deseables, para luego mostrar el esquema general de la solución. El resto del capítulo describe detalladamente la solución propuesta.

El trabajo aquí expuesto ha dado lugar a la publicación [**PPRS00**].

5.1 Motivación

En el proceso de compilación de Eden, las abstracciones y concreciones de procesos permanecen ocultas durante la fase de optimizaciones automáticas que se realizan en el lenguaje Core. Este hecho tiene dos graves inconvenientes:

- Las transformaciones automáticas de Haskell pueden violar la semántica de Eden.
- No pueden realizarse optimizaciones específicas de Eden, pues sus construcciones están ocultas.

En [PPP99, PS00] se estudian los problemas que pueden ocasionar las transformaciones automáticas de GHC sobre el código Eden, y se llega a la conclusión de que deben desactivarse mientras no se encuentre una solución a los problemas descubiertos.

En lo que respecta a las optimizaciones específicas de Eden, sería deseable poder establecer comunicaciones directas entre los productores y consumidores de mensajes (véase la Sección 5.1.2), y sobre todo sería conveniente poder lanzar impacientemente procesos (véase la Sección 5.1.1). Asimismo, debería dejarse abierto el camino para futuros análisis y transformaciones, como por

ejemplo para detectar qué partes de un programa son deterministas y cuáles no (véase [PS01a]).

En este capítulo se describe una solución que resuelve ambos problemas: permite introducir nuevas optimizaciones, y consigue inmunidad contra las transformaciones de GHC.

5.1.1 Lanzamiento impaciente de procesos

Supongamos que tenemos la siguiente implementación paralela de la función de Fibonacci:

```
fib = process x -> f x
  where fsec 0 = 1
        fsec 1 = 1
        fsec n = fsec (n-1) + fsec (n-2)
        f n = if n < 15 then fsec n else fib # (n-1) + fib # (n-2)
```

Debido a su sencillez se ha elegido para poner de manifiesto un problema general debido a la pereza: en la Figura 5.1 pueden observarse los perfiles de actividad general y por hebras de dicho programa al evaluar `fib # 18` sin utilizar lanzamiento impaciente de procesos. Es evidente comprobar que, salvo en el instante de lanzamiento de nuevas hebras, nunca hay más de una hebra ejecutándose, es decir, el programa no es paralelo. De hecho, lo único que se ha obtenido es una ejecución secuencial distribuida entre distintos procesadores.

¿Cuál es el problema? Si nos fijamos en la gráfica, se observa que no se lanza ningún proceso[1] hasta que no ha terminado de evaluarse el proceso anterior. La razón es que para evaluar en Haskell la expresión

```
fib # (n-1) + fib # (n-2)
```

primero se demanda la WHNF del argumento izquierdo[2] de la suma, y una vez que se ha obtenido, se demanda la del derecho, de modo que no existe demanda para evaluar el segundo proceso hasta que no ha terminado la evaluación del primero.

El problema radica en el hecho de que para que se lance un proceso es necesario que se demande su salida. La solución pasa por violar la pereza, de forma que el lanzamiento de procesos sea especulativo, es decir, un proceso no se lanza cuando se demandan sus salidas, sino que se lanza en cuanto se crea

[1] En este ejemplo, dado que cada proceso tiene sólo un canal de salida, cada proceso tiene sólo una hebra, además de las hebras que necesite para alimentar las entradas de sus procesos hijos.

[2] La razón por la que se evalúa primero el argumento izquierdo se debe a la definición de +, que primero hace ajuste de patrones sobre el primer argumento, y luego sobre el segundo. Otras funciones pueden demandar sus argumentos en otro orden, pero el problema será el mismo, ya que los argumentos se irán demandando uno a uno.

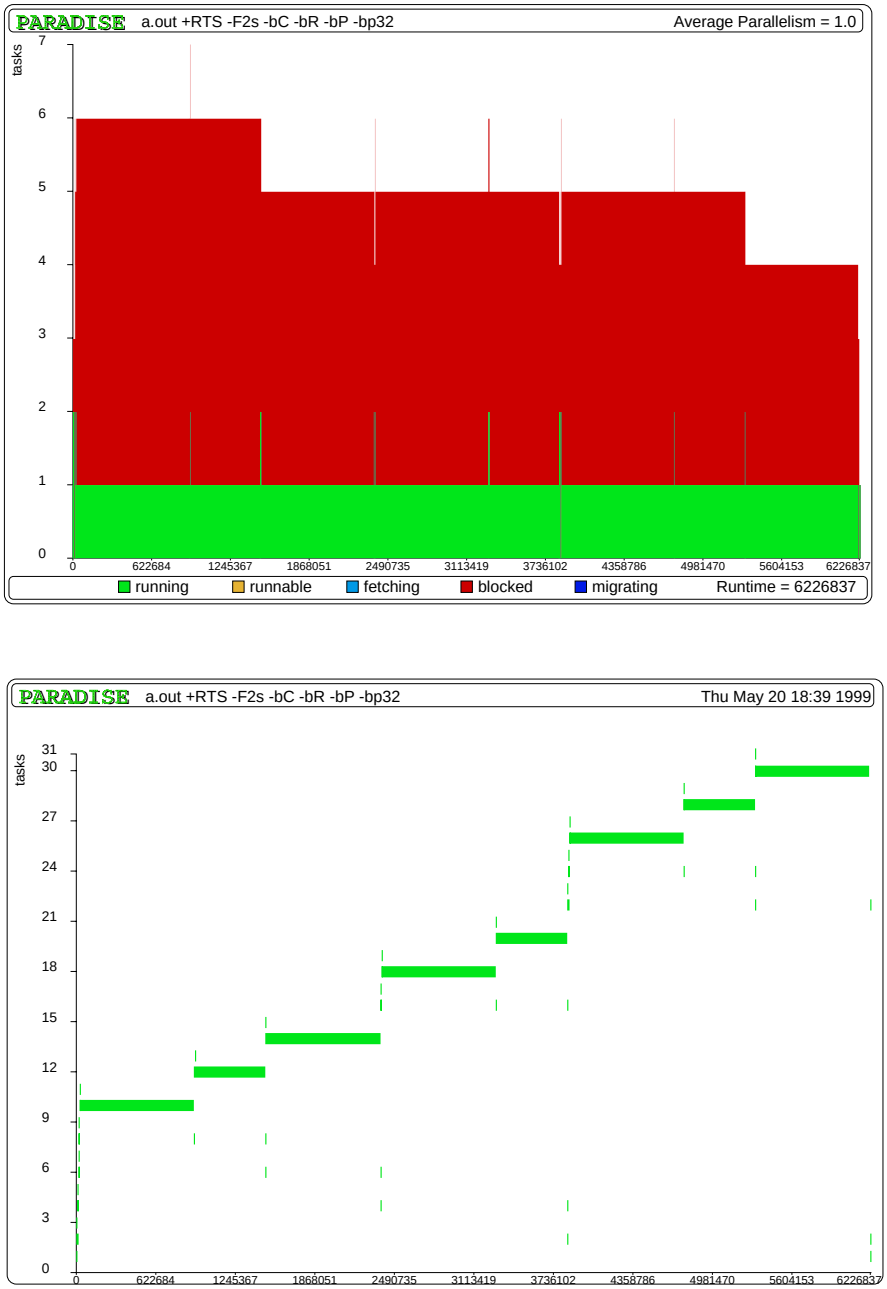


Figura 5.1: Comportamiento global y por hebras de la versión paralela de la función de Fibonacci

su correspondiente clausura, aun cuando nadie demande los valores. Aunque este hecho puede conducir a realizar trabajo extra de forma innecesaria, es un riesgo que puede y debe controlar el programador.

En un entorno paralelo la pereza resulta un grave inconveniente, pues limita el grado de paralelismo generado. Como se comentó en el Capítulo 3, Eden viola la pereza de Haskell en una situación: una vez que se ha lanzado un proceso, siempre hay demanda para sus salidas. Así pues, para conseguir que se evalúen en paralelo distintos procesos sólo necesitamos conseguir que se demande parte de su salida. Una vez demandada la WHNF de la salida del proceso, el resto de la evaluación del proceso no requiere demanda externa. El problema radica en que demandar la WHNF de un entero es equivalente a demandar su forma normal, por lo que no podemos proseguir con el cómputo normal hasta que haya terminado la evaluación del proceso.

El problema no se limita a los enteros (y demás tipos simples) sino que es extensible a cualquier tipo que se utilice como salida del proceso, ya que en Eden los valores producidos por un proceso se envían siempre en forma normal. Por tanto, demandar la WHNF es equivalente a demandar la forma normal, es decir, implica esperar a que el proceso termine su cómputo.

Las dos únicas excepciones son los procesos que devuelven tuplas y los que devuelven listas: devolver una tupla significa que la salida del proceso se divide en hebras que envían de forma independiente los datos que producen; los procesos que devuelven listas no las devuelven en forma normal, sino que envían una a una las componentes de la lista producida (eso sí, cada componente de la lista se envía en forma normal).

Por tanto, para demandar la salida de un proceso sin necesidad de esperar a que el proceso evalúe su resultado, habría que escribir todos los procesos de forma que devuelvan listas cuyo primer elemento se calcule de forma inmediata, de modo que la WHNF se obtenga lo antes posible. Esto es intolerable como metodología de programación. Es preciso que sea el propio compilador quien realice de forma automática una transformación que lance impacientemente los procesos, de modo que el programador pueda utilizar el lenguaje usando un estilo claro y natural. Para ello, debe quedar claro en qué momento exacto deben lanzarse los procesos: un proceso se lanzará en cuanto su correspondiente clausura se haya creado, independientemente de que exista demanda para su salida.

5.1.2 Conexión directa de canales

El *bypassing* automático es una optimización de Eden que reduce el número de mensajes y de hebras que se emplean en tiempo de ejecución. La idea principal consiste en conectar directamente los productores de los mensajes con los consumidores, de modo que se elimine la necesidad de retransmisión de mensajes a través de procesos intermedios. Por ejemplo, en el ejemplo de la Figura 5.2 se aprecia que la topología de comunicaciones que se crea no

```
q :: [Process a (a,a)] -> Process (a,a) (a,a)
q [ ]    = pid
q (p:pp) = process (v1,v2) -> (v2,o2)
  where (o2,o3) = (q pp) # (p # v1)
pid :: Process a a
pid = process x -> x
```

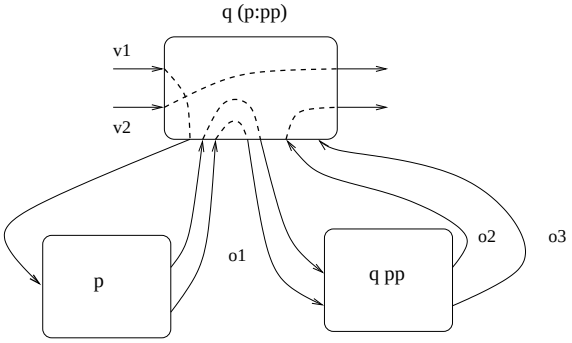


Figura 5.2: Un ejemplo de *bypassing*

es la realmente deseada, pues gran parte de las comunicaciones no se realizan directamente. Sería conveniente que cada uno de los canales se tratara de forma independiente para conectarlo con el proceso adecuado. Así, por ejemplo, en la Figura 5.2 `v1` debería comunicarse directamente a `p`, sin pasar por el proceso asociado a `q (p:pp)`. Para conseguir conexiones directas es preciso analizar todos los usos que se hacen de cada canal, de modo que únicamente se procede a la conexión directa cuando se detecta que un canal sólo lo usan un proceso emisor y uno receptor.

La estrategia de implementación del *bypassing* se basa en una combinación de un análisis en tiempo de compilación, y de una modificación al RTS para que aproveche las anotaciones obtenidas con el análisis (véase [KPS00] para una descripción detallada de ambas partes). El análisis debe decorar con anotaciones tanto las abstracciones como las concreciones de procesos, dependiendo del tipo de *bypassing* que deba realizarse en cada caso. Así, cuando deben conectarse procesos que corresponden a distintas generaciones (por ejemplo un "abuelo" con un "nieto") se anotan las abstracciones de proceso, mientras que cuando deben conectarse directamente distintos procesos hermanos hace falta anotar los `let`s en los que aparezcan las concreciones.

5.2 Esquema de la solución

Siguiendo la estructura de compilación expuesta en la Sección 3.3, tras la generación del código Core intermedio y antes de las transformaciones automáticas de Core, proponemos introducir una nueva fase de transformaciones

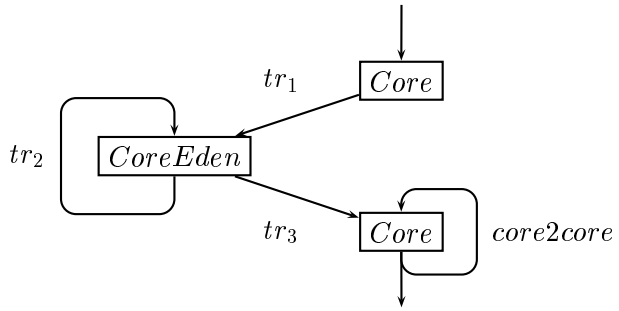


Figura 5.3: Esquema general de las transformaciones.

CoreEden (véase Figura 5.3). Lógicamente, es preciso realizar una traducción de Core a CoreEden que haga explícitos los procesos, y posteriormente deben realizarse las transformaciones que se estimen oportunas. Ahora bien, puede resultar extraño que posteriormente vuelva a generarse Core, y que se realicen las transformaciones core2core de GHC. El objetivo de este último paso es volver a "esconder" las construcciones de Eden, de modo que el proceso de compilación de GHC se vea afectado lo menos posible. Esto incluye conseguir que el compilador de Eden sea inmune a todas las transformaciones automáticas que realice GHC, y no tener que modificar el lenguaje STG ni la generación de código.

5.3 El lenguaje Core

Para poder explicar la transformación, es preciso conocer previamente algunos detalles del lenguaje Core ([San95]), puesto que es el lenguaje en el que se ha implementado la transformación. Core es un lenguaje funcional muy elemental (véase la Figura 5.4), que se utiliza como lenguaje intermedio en el proceso de compilación de GHC. Los programas Haskell se traducen a Core mediante un proceso de desazucaramiento. Una vez en Core, GHC realiza diversas transformaciones automáticas con el objetivo de mejorar la eficiencia, y después prosigue el proceso de compilación hasta la generación de código (véase la Sección 3.3 para más detalles sobre el proceso de compilación tanto de GHC como de MEC).

El lenguaje Core puede verse como un λ-cálculo de segundo orden[3] extendido con construcciones **let** y **case**. Básicamente, la construcción **let** es la encargada de generar clausuras en la memoria dinámica, mientras que **case** fuerza la evaluación a WHNF. Es decir, **let** es perezosa mientras que **case**

[3]El λ-cálculo de segundo orden es λ-cálculo con abstracciones de tipos.

Bindings	$binds$	$\rightarrow$	$bind_1; \ldots;\ bind_n$	$n \geq 1$
	$bind$	$\rightarrow$	**nonrec** $var\ =\ expr$	
		$\mid$	**rec** $var_1\ =\ expr_1;$	
			$\ldots;$	
			$var_n\ =\ expr_n$	$n \geq 1$
Expression	$expr$	$\rightarrow$	$expr_1\ expr_2$	Application
		$\mid$	$expr\ type$	Type application
		$\mid$	$\backslash\ var\ ->\ expr$	Lambda abstraction
		$\mid$	$/\backslash\ tyvar\ ->\ expr$	Type abstraction
		$\mid$	**case** $expr$ **of** $alts$	Case expression
		$\mid$	**let** $bind$ **in** $expr$	Local definition(s)
		$\mid$	$con\ expr_1 \ldots expr_n$	Saturated constructor
		$\mid$	$prim\ expr_1 \ldots expr_n$	Saturated primitive
		$\mid$	var	Variable
		$\mid$	$literal$	Literal
Alternatives	$alts$	$\rightarrow$	$calt_1; \ldots;\ calt_n; default\ ->\ expr$	$n \geq 0$ (Boxed)
		$\mid$	$lalt_1; \ldots;\ lalt_n;\ var\ ->\ expr$	$n \geq 0$ (Unboxed)
Constructor alt	$calt$	$\rightarrow$	$con\ var_1 \ldots var_n\ ->\ expr$	$n \geq 0$
Literal alt	$lalt$	$\rightarrow$	$literal\ ->\ expr$	

Figura 5.4: Sintaxis del lenguaje Core

es impaciente.

Así, la semántica de **let** *bind* **in** *expresion* consiste en crear una nueva clausura en memoria (para *bind*) y seguir evaluando *expresion*, pero suspendiendo la evaluación de la nueva clausura hasta el momento preciso en el que sea demandada. Realmente, *bind* no siempre es una sola ligadura, sino que puede ser un conjunto de ligaduras mutuamente recursivas. Así pues, en el caso general no sólo se genera un clausura, sino tantas como ligaduras tenga *bind*.

Por su parte, la semántica de **case** *expresion* **of** *alternativas* consiste en evaluar *expresion* de forma estricta hasta WHNF, y una vez alcanzada dicha WHNF, ejecutar la alternativa cuya guarda encaje con el valor obtenido.

Cuando se traduce un programa Haskell a un programa Core, inicialmente[4] sólo aparece **case** como traducción de los análisis por casos en el correspondiente programa Haskell[5]. En cuanto a los **let**, se generan tantos como haga falta para que las aplicaciones de funciones no se realicen sobre

[4]Posteriormente, algunas transformaciones Core2Core transforman **let** en **case** cuando se detecta que una función es estricta en alguno de sus argumentos. Ahora bien, nuestra transformación se realiza antes del resto de transformaciones, por lo que podemos ignorarlas.

[5]Realmente también puede aparecer como consecuencia de haber utilizado en el programa una anotación `seq`.

expresiones, sino sobre variables ligadas a expresiones.

A modo de ejemplo, dado un programa Haskell para calcular la función de Fibonacci

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

la correspondiente traducción a Core sería aproximadamente[6] la siguiente:

```
fib = \x -> case x of
               0 -> 1
               1 -> 1
               n -> let
                      f1 = let
                             n1 = (-) n 1
                           in
                             fib n1
                    in
                      let
                         f2 = let
                                n2 = (-) n 2
                              in
                                fib n2
                      in
                        (+) f1 f2
```

5.4 El lenguaje CoreEden

Como ya se dijo anteriormente, la principal transformación que es deseable en Eden pero que no es fácil de realizar sin extender el lenguaje Core, es el *bypassing* de canales (véase Sección 3.1.3). Para poder realizar un análisis de *bypassing*, necesitamos hacer explícitas las abstracciones y concreciones de procesos. En caso contrario, no podríamos manejar fácilmente la información necesaria: cuáles son los canales de entrada y de salida, qué conexiones hay entre las distintas concreciones de procesos, y qué conexiones hay entre una abstracción de proceso y las distintas concreciones que aparecen en su definición.

La definición del lenguaje CoreEden puede apreciarse en la Figura 5.5, donde las partes que no se muestran coinciden exactamente con la sintaxis del lenguaje Core. Básicamente es necesario introducir un nuevo tipo de expresión para las abstracciones de proceso, y un nuevo tipo de ligadura para las concreciones de proceso. Nótese que no existe ninguna construcción especial para los canales dinámicos ni para el proceso reactivo `merge`. La razón es que las transformaciones a realizar actualmente no necesitan tratar dichas

[6] Realmente el tratamiento de los enteros no es como se muestra en esta traducción.

$$
\begin{array}{lll}
\mathit{program} & \to & \mathit{binds}_1, \ldots, \mathit{binds}_n \\
\mathit{binds} & \to & \ldots \text{Core bindings} \ldots \\
 & | & \textbf{recpar}\ \mathit{bind}'_1; \ldots; \mathit{bind}'_n \\
\mathit{bind}' & \to & v = e \\
 & | & \mathit{channels} = v\ \texttt{\#\#}\ \mathit{channels} \\
\mathit{channels} & \to & \{v_1, \ldots, v_n\} \\
e & \to & \ldots \text{Core expressions} \ldots \\
 & | & \textbf{process}\ \mathit{channels} \to \mathit{body} \\
\mathit{body} & \to & [\textbf{let}\ \mathit{binds}\ \textbf{in}]\ \mathit{channels}
\end{array}
$$

Figura 5.5: Sintaxis de CoreEden

$$
\begin{array}{lll}
\mathit{binds} & \to & \ldots \text{Core bindings} \ldots \\
 & | & \textbf{recpar}\ \mathit{bind}'_1; \ldots; \mathit{bind}'_n \\
 & & [\textbf{bypass}\ \mathit{channels}] \\
e & \to & \ldots \text{Core expressions} \ldots \\
 & | & \textbf{process}\ \mathit{channels} \to \mathit{body} \\
 & & [\textbf{bypass}\ \mathit{channels}]
\end{array}
$$

Figura 5.6: Sintaxis de CoreEden con anotaciones de *bypassing*

construcciones, por lo que podemos mantenerlas ocultas. Ahora bien, en el futuro podrá extenderse el lenguaje CoreEden si se desea realizar algún tipo de análisis que necesite dicha información, como por ejemplo determinar qué partes de un programa son deterministas y cuáles pueden no serlo [PS01a].

Nótese que para las concreciones de procesos utilizamos ligaduras en lugar de expresiones. Esto se debe a que necesitamos nombrar explícitamente los canales de entrada y de salida, de modo que podamos decidir cuáles deben conectarse directamente. Así, existe un nuevo tipo de ligadura **recpar** que agrupará todo tipo de ligaduras, tanto paralelas como no. Y tenemos un nuevo tipo de expresión **process** que hace explícitos los canales de entrada y de salida de las abstracciones de proceso. De hecho, para poder realizar convenientemente el *bypassing*, es necesario decorar los procesos con anotaciones que indiquen cómo deben realizarse las conexiones. Dependiendo del tipo de *bypassing*, dichas anotaciones deben ligarse a las abstracciones o a las concreciones, por lo que la sintaxis de CoreEden debe extenderse como se muestra en la Figura 5.6. Véase [KPS00] para más detalles sobre los distintos tipos de *bypassing*, y [**PPRS00**] para los distintos tipos de anotaciones necesarias para identificarlos.

5.5 De Core a CoreEden

El primer paso que debemos llevar a cabo en nuestro esquema de compilación es traducir de Core a CoreEden, haciendo explícitas las abstracciones y concreciones de procesos. En Core, las abstracciones están ocultas como

$$
\begin{array}{lll}
 & tr_1 \;\; (\textbf{let } f \;=\; \lambda i.e \textbf{ in } process\ f) & \\
(a) & \quad = \;\; \textbf{process } \{i_1, \ldots, i_n\} \rightarrow & \textbf{let } i = (i_1, \ldots, i_n) \\
 & & \textbf{in let } o = tr_1\ (e) \textbf{ in } \{o\}
\end{array}
$$

$$
(b) \;\; tr_1 \left(\begin{array}{l}
\textbf{let} \\
\quad \ldots \\
\quad o \;=\; [\textbf{let } binds_1 \\
\qquad\quad \textbf{in let } binds_2 \textbf{ in } \ldots] \;\#\; p\ i \\
\quad \ldots \\
\textbf{in} \\
\quad e
\end{array} \right)
$$

$$
= \quad \begin{array}{l}
\textbf{let recpar} \\
\quad \ldots \\
\quad \{o_1, \ldots, o_m\} \;=\; p \;\texttt{\#\#}\; \{i\} \\
\quad o \qquad\qquad\;\; = \;\; (o_1, \ldots, o_m) \\
\quad [tr_1(binds_1 \texttt{++} binds_2 \texttt{++} \ldots)] \\
\quad \ldots \\
\textbf{in} \\
\quad tr_1(e)
\end{array}
$$

$$
(c) \;\; tr_1 \;\; (\;\#\; p\ i\;) \;\; = \textbf{let recpar } \{o_1, \ldots, o_m\} \;=\; p \;\texttt{\#\#}\; \{i\} \textbf{ in } (o_1, \ldots, o_m)
$$

Figura 5.7: Transformación de Core a CoreEden

aplicaciones de la función `process` a la función que representa el comportamiento del proceso (véase [BKL98]). Dicha función de comportamiento tendrá como entrada un parámetro que represente la tupla de canales de entrada. La traducción se muestra en la Figura 5.7a. Nótese que no sólo se ha hecho explícita la abstracción de proceso, sino también los canales de entrada, ya que ahora son independientes unos de otros. Los canales de salida no se han separado todavía, sino que se separarán en etapas posteriores.

Por lo que respecta a las concreciones de procesos, cada una de ellas se encuentra oculta como una aplicación de la función `#` a dos argumentos, que representan la abstracción de proceso y la tupla de valores de entrada. Ahora bien, en CoreEden las concreciones no son expresiones, sino ligaduras. Por dicho motivo, la traducción más sencilla consistiría en incluir las concreciones dentro de nuevos **let**s, como se muestra en la Figura 5.7c. Afortunadamente, habitualmente la concreción ya aparece dentro de un **let**, por lo que puede aplicarse la regla (b) de la Figura 5.7. Nótese que en ambos casos se crean variables frescas para los canales de salida, mientras que en la regla (b) también se mantiene el nombre original de la salida para mantener las referencias a él. Nótese también que en la regla (b) pueden aparecer muchas concreciones dentro del mismo **let**, en cuyo caso simplemente habría que aplicar la transformación a cada una de ellas, pero manteniendo un único **let**. Es más, en caso de que una concreción aparezca tras varios

lets anidados, los aplanaremos para poder tener todas las concreciones en un único **let**.

Lógicamente, el conjunto de reglas de la Figura 5.7 se completa con reglas triviales que exploran todas las posibilidades que ofrece la sintaxis del lenguaje Core. Por ejemplo, para los análisis por casos basta la siguiente regla:

$tr_1(\textbf{case}\ e\ \textbf{of}\ alts) \quad = \quad \textbf{case}\ tr_1(e)\ \textbf{of}\ tr'_1(alts)$

5.6 Transformaciones en CoreEden

Antes de realizar el análisis de *bypassing*, y una vez generado el árbol de sintaxis abstracta CoreEden de un programa, lo primero que debe hacerse es agrupar en construcciones **recpar** tantas ligaduras como sea posible, utilizando transformaciones *aplanadoras*. Posteriormente tendremos que hacer explícitos los canales de entrada y de salida de los procesos, mediante transformaciones *tupladoras*. Así, al tener juntas varias concreciones de procesos, y al tener acceso a los distintos canales, podremos determinar con mayor facilidad las interconexiones de los procesos.

5.6.1 Transformaciones aplanadoras

Las transformaciones aplanadoras tratan de juntar en un único **recpar** tantas concreciones de proceso como sea posible. Para ello basta aplicar las tres reglas de transformación siguientes:

$$\frac{\textbf{let}\ [\textbf{rec}|\textbf{recpar}]\ binds_1\ \textbf{in let recpar}\ binds_2\ \textbf{in}\ e}{\textbf{let recpar}\ binds_1; binds_2\ \textbf{in}\ e}$$

$$\frac{\textbf{case}\ (\textbf{let recpar}\ binds\ \textbf{in}\ e)\ \textbf{of}\ alts}{\textbf{let recpar}\ binds\ \textbf{in}\ (\textbf{case}\ e\ \textbf{of}\ alts)}$$

$$\frac{(\textbf{let recpar}\ binds\ \textbf{in}\ e)\ x}{\textbf{let recpar}\ binds\ \textbf{in}\ (e\ x)}$$

La principal transformación es la primera, mientras que la utilidad de las otras dos se limita a exponer nuevos casos en los que pueda aplicarse la primera regla. Con el fin de obtener el máximo aplanamiento posible, se iterará la aplicación de las reglas hasta que se alcance un punto fijo.

Nótese que la aplicación de las reglas preserva trivialmente la semántica de Haskell, y que también preservan la de Eden, puesto que no se cambia el momento en el que se lanzan los procesos. La razón es que, en el lenguaje Core, **case** es estricto en su discriminante, y la aplicación funcional es estricta en la función a aplicar, por lo que sólo se flotan aquellas expresiones que realmente van a demandarse en el siguiente paso de cómputo.

El aplanamiento que hemos especificado va en contra de las transformaciones de GHC que tratan de separar lo más posible las ligaduras para facilitar optimizar partes concretas del programa. Ahora bien, nuestra transformación no afecta negativamente a dichas optimizaciones, pues GHC realizará posteriormente un análisis de dependencias para detectar componentes fuertemente conexas [JM99], por lo que nuestro aplanamiento será finalmente deshecho, y sólo habrá sido utilizado por comodidad en etapas intermedias de la compilación.

5.6.2 Transformaciones tupladoras

Estas transformaciones tratan de hacer explícitos los canales individuales tanto de las abstracciones como de las concreciones de procesos. Sólo hacen falta para aquellos casos en los que deba realizarse *bypassing* entre procesos que usen tuplas como entradas o salidas. Es más, sólo hacen falta cuando deba hacerse *bypassing* de canales individuales: si todos los canales de una misma tupla deben conectarse a un mismo proceso, entonces no hace falta diferenciarlos, y a efectos de *bypassing* pueden tratarse como un único canal, por lo que el tuplamiento no es necesario.

En todas las reglas que mostramos a continuación supondremos que se han eliminado previamente todos los alias. Es decir, no existirán ligaduras del estilo $a = b$, sino que en todas las apariciones de a se usará b, eliminándose la definición de a.

Concreciones de procesos

Tras la traducción de una concreción de procesos de Core a CoreEden, se obtiene una expresión como la siguiente:

$$\textbf{let recpar } \ldots \ \{o_1, \ldots, o_m\} \ = \ p \ \#\# \ \{i\} \ \ \ldots \ \textbf{in } e$$

por lo que ahora tendremos que hacer explícitos los distintos canales de entrada, y deberán analizarse tanto las entradas como las salidas para tratar de aislar los usos que se realizan de cada uno de los canales.

Tuplamiento de las entradas. Sólo puede realizarse *bypassing* de canales individuales cuando no existe ningún cómputo que involucre la tupla entera de canales. Por tanto, si se usan los canales de modo independiente debe existir una clausura como la siguiente:

$$i \ = \ [\textbf{let } binds_1 \textbf{ in let } binds_2 \textbf{ in } \ldots] \ \ (i_1, \ldots, i_n)$$

Si no se encuentra esta clausura sólo será posible realizar *bypassing* de todos los canales juntos, pero no individualmente de cada uno de ellos. Por tanto el tuplamiento no será ni posible ni necesario.

Si se encuentra la clausura, deben flotarse las ligaduras del **let**, de modo que i se defina como una tupla $(i_1, \ldots, i_n)$. Tras ello, basta aplicar la siguiente transformación para hacer explícitos los canales de entrada:

$$\frac{\{o_1, \ldots, o_m\} \ = \ p \ \#\# \ i}{\{o_1, \ldots, o_m\} \ = \ p \ \#\# \ \{i_1, \ldots, i_n\}}$$

Por último, es necesario eliminar todas las referencias que existieran a la variable i, para lo cual hay que aplicar la siguiente regla en todos los lugares en los que sea posible:

$$\frac{\textbf{case } i \textbf{ of } (\ldots, v_j, \ldots) \ \rightarrow \ e}{e[i_j / v_j]}$$

Si la tupla asociada a i no se utilizaba de modo global, sino sólo para acceder a sus elementos individuales, entonces tras el paso anterior deberían haber desaparecido todas las referencias a i, por lo que bastaría usar la transformación *dead-code removal* de GHC para eliminar la clausura.

Los pasos descritos anteriormente pueden resumirse del modo siguiente:

1. Buscar la clausura $i \ = \ (i_1, \ldots, i_n)$ en el árbol sintáctico que representa el programa CoreEden.

2. Si se encuentra dicha clausura, entonces se aplica *maybe_ remove(i)*, que se define como sigue:

$$maybe\_remove(i) \stackrel{\text{def}}{=} \begin{cases} \text{- Sustituir usos de } i \text{ por el correspondiente } i_j \\ \text{- Eliminar la definición de } i \text{ (si fuera posible)} \end{cases}$$

Tuplamiento de las salidas. En la traducción de Core a CoreEden ya se introdujeron canales independientes para los canales de salida, por lo que la expresión que tendremos será de la forma

$$\begin{array}{l} \textbf{let recpar} \\ \quad \begin{array}{ll} \ldots & \\ \{o_1, \ldots, o_m\} & = \ p \ \#\# \ \{i_1, \ldots, i_n\} \\ o & = (o_1, \ldots, o_m) \\ \ldots & \end{array} \\ \textbf{in } e \end{array}$$

Al igual que en el caso de los canales de entrada, para que pueda hacerse *bypassing* de los canales individualmente, es necesario que no existan apariciones de o. Por ello, basta con realizar los mismos dos últimos pasos que efectuamos con los canales de entrada, es decir, sólo debe aplicarse *maybe_ remove(o)*.

Abstracciones de procesos

Tras la traducción de Core a CoreEden, las abstracciones de procesos aparecen como expresiones de la forma

$$\textbf{process} \quad \{i_1, \ldots, i_n\} \quad \rightarrow \quad \textbf{let} \ \ i \ = \ (i_1, \ldots, i_n) \ \textbf{in let} \ o \ = \ e \ \textbf{in} \ \{o\}$$

Tuplamiento de las entradas. Sólo puede realizarse *bypassing* de los canales de entrada individuales si no existen apariciones de i, por lo que la solución se limita a aplicar *maybe_ remove(i)*.

Tuplamiento de las salidas. Si la expresión e produce como valor final una tupla $(o_1, \ldots, o_m)$, y además puede realizarse *bypassing* de los canales de salida individuales, entonces debe ser posible acceder directamente a los canales individuales. Por tanto, la definición de la salida del proceso debe ser de la forma

$$\textbf{let} \ o = [\textbf{let} \ldots \textbf{in}] \ (o_1, \ldots, o_m)$$

En dicho caso, si o no se usa, entonces la abstracción de proceso se traducirá por

$$\textbf{process} \ \{x1 \ldots x_n\} \rightarrow \textbf{let} \ldots \textbf{in} \ \{o_1, \ldots, o_m\}$$

5.6.3 Análisis en CoreEden

Tras las transformaciones anteriores tenemos el camino allanado para que pueda implementarse el análisis de *bypassing*. La descripción de dicho análisis se sale del tema de esta tesis, por lo que el lector interesado debe consultar [KPS00] para más detalles. Asimismo, esta etapa es también la adecuada para realizar cualquier otro tipo de análisis, como por ejemplo el análisis de no-determinismo [PS01a].

5.7 De CoreEden a Core

Tras haber realizado los análisis pertinentes en el lenguaje CoreEden, es necesario volver a traducir al lenguaje Core, para proseguir con el proceso de compilación de GHC. Los principales objetivos de esta etapa son los siguientes:

- Encapsular la información obtenida en los análisis efectuados.
- Lanzar impacientemente procesos.
- Permitir que se reutilicen la mayor parte de las optimizaciones que efectúa GHC.

$$tr_3\ (\textbf{process}\ \{i_1,\ldots,i_n\} \to body) \quad = \quad \textbf{let}\ f = \lambda\, i_1.\ \ldots \lambda\, i_n.\ tr_3(body)\ \textbf{in}\ processB\ f$$

$$tr_3\ (\textbf{let recpar}\ bs\ \textbf{in}\ e) = \quad \textbf{let rec}\ bs' \ \textbf{in case}\ o'_1\ \textbf{of}\ \_ \to \ldots\ \textbf{case}\ o'_n\ \textbf{of}\ \_ \to tr_3(e)$$

where $(bs', os') = trbs\ bs$, $v_i = vs!!i$

$$trbs\ bs\ =\ (concat\ bss,\ concat\ oss') \quad \text{where}\ (bss,\ oss')\ =\ unzip\ (map\ trb\ bs)$$

$$trb\ (\{o_1,\ldots,o_m\} = p\ \#\#\ \{v_1,\ldots,v_n\})\ cs\ =\ (bs, [o'])$$

where $bs = [o = \textbf{case}\ o'\ \textbf{of}\ Lift\ o'' \to o'',\ o' = instantiate\ p\ (v_1,\ldots,v_n),\ o_1 = \textbf{case}\ o\ \textbf{of}\ (o'_1,\ldots,o'_m) \to o'_1,\ \ldots,\ o_m = \textbf{case}\ o\ \textbf{of}\ (o'_1,\ldots,o'_m) \to o'_m]$

Figura 5.8: Traducción de CoreEden a Core

En esta tesis nos centraremos sólo en los dos últimos puntos. La codificación de la información de los análisis se describe en [**PPRS00**]. La Figura 5.8 muestra las partes relevantes de la función tr_3 que realiza la traducción a Core. Se utilizan dos nuevas funciones primitivas, `processB` e `instantiate`, que se utilizan para ocultar respectivamente las abstracciones y las concreciones de procesos. La primera sólo tiene un argumento, que es la función que contiene el comportamiento del proceso, mientras que la segunda tiene dos argumentos: la abstracción de proceso y el valor de entrada del proceso.

Abstracciones de procesos Cada abstracción de proceso se oculta mediante la aplicación de la función predefinida `processB` a la función que caracteriza el comportamiento del proceso. En principio, dicha función `processB` podría ser exactamente la misma que la que se usaba previamente (`process`) para ocultar las abtracciones de procesos en Core. Ahora bien, utilizamos una función distinta con el objetivo de poder añadir información obtenida en los análisis efectuados a nivel de CoreEden. De hecho, `processB` tiene un parámetro extra, que no se muestra aquí, y que recoge información de *bypassing* (véase [**PPRS00**] para más detalles).

Concreciones de procesos Una expresión **let recpar** puede contener varias concreciones de procesos, y hay que conseguir que todas ellas se lancen impacientemente. La idea principal es que deberían demandarse sus salidas sin necesidad de esperar a que produzcan ningún resultado. Esto puede conseguirse si utilizamos una función que siempre devuelva trivialmente el mismo constructor en cabeza. Así, el operador de lanzamiento de procesos se define como sigue:

```
(#) p v = case (instantiate p v) of Lift a -> a
```

Nótese que la función `instantiate` siempre devuelve en cabeza el constructor `Lift`, por lo que podremos demandar la salida de cualquier aplicación de `instantiate` sin necesidad de tener que esperar a que se evalúe ningún proceso. Ahora bien, en cuanto se demande la salida de una aplicación de `instantiate`, la función primitiva que lanza el proceso se aplicará, con lo que el proceso se creará.

En la Figura 5.8 la función *trb* realiza la traducción de cada una de las concreciones de procesos. Para ello se generan varias ligaduras: una para mantener referencias a la salida o original, m (número de salidas) para referenciar directamente cada uno de los canales de salida, y una más para el valor intermedio o' que incorpora el constructor `Lift`. El lanzamiento impaciente se produce en la función tr_3 gracias al uso de expresiones `case` aplicadas a la variable o'. Tan pronto como se fuerza la evaluación de `instantiate`, la función primitiva toma el control de la evaluación del proceso, y demanda sus salidas sin necesidad de utilizar ningún otro `case`. Nótese que *trbs* simplemente aplica la función *trb* a cada una de las ligaduras de *bs*, recolectando así todas las nuevas ligaduras y todas las variables o' que deban usarse para lanzar procesos.

En [PS00] se muestra que el lanzamiento impaciente de procesos no se ve afectado por ninguna de las transformaciones automáticas que realiza posteriormente GHC.

5.8 De Core a Core

Al llegar a la última etapa de nuestra modificación al proceso de compilación disponemos de código Core no optimizado. En [JM99, JS98] se muestra que el código no optimizado es en media 2.7 veces más lento que el código GHC optimizado, por lo que resulta evidente que debemos estar interesados en mantener la mayoría de las transformaciones que realiza GHC. En [PPP99, PS00, **PPRS00**] se han estudiado qué transformaciones de GHC son peligrosas para Eden. En esta sección primero presentaremos qué transformaciones podrían ser peligrosas según dichos estudios, para luego analizar cuáles serían los problemas reales que conllevaría su uso.

transformación	antes	después
(a) Full laziness	**let** $g = \lambda y.$**let** $x = e$ **in** e' **in** $\ldots$	**let** $x = e$ **in let** $g = \lambda y.e'$ **in** $\ldots$
(b) Static arguments	$foldr\ f\ z\ l =$ **case** l **of** $[\,] \rightarrow z$ $(a : as) \rightarrow$ **let** $v = foldr\ f\ z\ as$ **in** $f\ a\ v$	$foldr\ f\ z\ l =$ **let** $foldr'\ l =$ **case** l **of** $[\,] \rightarrow z$ $(a : as) \rightarrow$ **let** $v = foldr\ f\ z\ as$ **in** $f\ a\ v$ **in** $foldr'\ l$
(c) Specialization	$g = \Lambda ty.\lambda dict.\lambda y.$ **let** $f = \Lambda ty.\lambda dict.e$ **in** $f\ ty\ dict\ (f\ ty\ dict\ y)$	$g = \Lambda ty.\lambda dict.\lambda y.$ **let** $f = \Lambda ty.\lambda dict.e$ **in let** $f' = f\ ty\ dict$ **in** $f'\ (f'\ y)$
(d) *let* floating from *let* rhs	**let** $x =$ **let** $bind$ **in** e **in** b	**let** $bind$ **in let** $x = e$ **in** b
(e) *case* floating from *let* rhs	**let** $v =$ **case** e_v **of** $\ldots$ $C_i\ x_{i1} \ldots x_{ik} \rightarrow e_i$ $\ldots$ **in** e	**case** e_v **of** $\ldots$ $C_i\ x_{i1} \ldots x_{ik} \rightarrow$ **let** $v = e_i$ **in** e $\ldots$
(f) *let* to *case*	**let** $v = e_v$ **in** e	**case** e_v **of** $v \rightarrow e$
(g) Unboxing *let* to *case*	**let** $v = e_v$ **in** e	**case** e_v **of** $C\ v_1 \ldots v_n \rightarrow$ **let** $v = C\ v_1 \ldots v_n$ **in** e

Figura 5.9: Reglas peligrosas de GHC

5.8.1 Transformaciones peligrosas

En la mayoría de reglas peligrosas que se detectaron en [PPP99, PS00] el principal problema residía en que las concreciones de proceso aparecían ligadas en construcciones **let**. Gracias a la transformación de CoreEden a Core hemos conseguido que ahora las concreciones aparezcan en el discriminante de construcciones **case**, por lo que gran parte de las reglas peligrosas dejaron de serlo. En [**PPRS00**] se revisaron los dos trabajos anteriores, teniendo en cuenta la nueva estrategia de compilación. Los posibles riesgos que se detectaron en este último trabajo se agrupan en tres tipos distintos, que se describen a continuación.

Transformaciones que afectan al no-determinismo

Algunas reglas pueden cambiar el comportamiento no-determinista expresado por el programador. El motivo de dicho cambio es un incremento en la compartición de clausuras: antes de la transformación, distintas evaluaciones de una expresión no-determinista pueden conducir a distintos valores,

mientras que tras la transformación la expresión no-determinista puede que se evalúe sólo una vez, por lo que todas sus apariciones tendrán el mismo valor. Las reglas que producen este efecto son las siguientes:

Full Laziness Supongamos que en la Figura 5.9a *e* es una expresión no-determinista. Antes de aplicar esta regla, `x` podría producir distintos valores en cada aplicación de `g`, mientras que tras usar la regla se garantiza que `x` tomará un único valor.

Static Argument Transformation En este caso (véase la Figura 5.9b) el problema se presenta cuando la aplicación parcial de la función a sus argumentos estáticos es no-determinista.

Specialization Supongamos que en la Figura 5.9c *e* es una función no-determinista. Entonces las dos aplicaciones parciales `f ty dict` que aparecen en `g` denotan dos funciones potencialmente distintas. Ahora bien, tras aplicar la regla se garantiza que ambas apariciones de `f'` denotan la misma función.

Transformaciones que afectan al número de procesos creados

Las tres reglas anteriores pueden reducir el número de veces que un proceso se lanza. Por ejemplo, si suponemos que en la regla *full laziness* *e* contiene concreciones de procesos, antes de aplicar la regla se lanzarán los procesos tantas veces como veces se aplique la función `g`, mientras que tras aplicar la regla sólo se lanzarán procesos la primera vez que se aplique la función.

Transformaciones con otros efectos peligrosos

Existen otras reglas que pueden afectar a la eficiencia de los programas paralelos Eden. Las posibles modificaciones son:

- Cambiar trabajo entre el proceso padre y el hijo (reglas e, f y g en la Figura 5.9). Por ejemplo, al flotar un `case` fuera de un `let`, puede forzarse un cómputo antes de lanzarse un proceso, por lo que el proceso padre será el encargado de efectuar dicho cómputo, mientras que antes podía ser un trabajo de un hijo.

- Cambiar el trasiego de datos en las comunicaciones (reglas d, e, f y g en la Figura 5.9). Al modificar la disposición de las ligaduras, la cantidad de clausuras que deben comunicarse a los procesos hijo al crearse puede verse modificada, tanto para bien como para mal.

- Cambiar la memoria necesaria (reglas d, e, f y g en la Figura 5.9). Al igual que en el caso anterior, la modificación de la disposición de las ligaduras puede hacer que procesos que realmente no necesitaban una clausura hayan tenido que crearla y almacenarla temporalmente en su memoria.

5.8.2 Discusión sobre los riesgos

No-determinismo. Incrementar el no-determinismo de un programa Eden es claramente incorrecto, puesto que podrían obtenerse soluciones que no estaban entre el conjunto de posibles soluciones de la especificación del problema. Ahora bien, reducir el no-determinismo no es ningún problema, pues toda solución que se obtenga con el programa menos no-determinista también podría haberse obtenido con el programa original. Podría argumentarse que el hecho de reducir el número de posibles soluciones del programa es incorrecto, pero esta argumentación sólo tendría sentido si la semántica del lenguaje garantizara una cierta probabilidad a cada una de las posibles soluciones del programa.

El lenguaje Eden no incorpora no-determinismo porque sea deseable, sino que lo que realmente se desea obtener es reactividad. Así, el proceso `merge` se introduce debido a sus características reactivas, que desgraciadamente también llevan asociadas no-determinismo. Por tanto, lo único que deben preservar las transformaciones son las propiedades reactivas, no la cantidad de no-determinismo. Así pues, los posibles riesgos debidos a la reducción de no-determinismo no son tales, y pueden aplicarse sin ningún temor las reglas que en [**PPRS00**] se consideraban peligrosas por reducir el no-determinismo.

Número de procesos. El hecho de reducir el número de procesos utilizados modifica claramente la semántica operacional del programa, pero no la denotacional, pues el resultado final del cómputo será el mismo. Así pues, lo único que alterará será la eficiencia, que en general mejorará por realizar menos cómputos.

Otros problemas. Los otros riesgos descritos previamente (cambiar trabajo entre procesos, cambiar el trasiego de datos y cambiar el consumo de memoria) no modifican la semántica del lenguaje, pero pueden afectar a la eficiencia del compilador. Ahora bien, aunque a veces afectarán negativamente, otras veces mejorarán la eficiencia del programa, pues sólo "cambian" trabajo de unos procesadores a otros. Dado que no son realmente negativas, y dado que consiguen mejorar sustancialmente la eficiencia de las partes secuenciales del programa, se ha decidido mantener todas las optimizaciones de GHC, aunque puedan modificar ligeramente la eficiencia de las partes paralelas.

5.9 Resultados obtenidos

Actualmente está completamente implementada la transformación que permite lanzar procesos impacientemente. A pesar de que en principio siempre debería utilizarse el lanzamiento impaciente, hemos decidido añadir una

opción de compilación (`-eeager`) para que el usuario pueda activar o desactivar a su antojo la transformación. De todos modos, en todos los programas que mostraremos a partir de ahora, supondremos que la transformación está activada, y también usaremos siempre todas las optimizaciones automáticas que proporciona GHC, con el fin de obtener la mejor eficiencia posible.

A modo de ejemplo, veamos el resultado que se produce cuando volvemos a compilar, utilizando lanzamiento impaciente de procesos, el programa que calculaba en paralelo la función de Fibonacci. Sin modificar ninguna línea de código se obtiene el resultado de la Figura 5.10. Nótese que el número de hebras que se crea es exactamente el mismo que antes, pero con la diferencia de que ahora se crean al principio del cómputo.

Con lo que respecta al resto de transformaciones, aún no se han implementado. El motivo es que todavía no se ha implementado el análisis de *bypassing*, por lo que no son necesarias las transformaciones aplanadoras ni las tupladoras, que se limitan a allanar el camino a dicho análisis. De todos modos, dado que ya se ha implementado la transformación que lanza impacientemente los procesos, hemos adquirido todos los conocimientos de bajo nivel precisos para realizar cualquier otra transformación fácilmente.

5.10 Conclusiones

Las principales contribuciones de este capítulo han sido las siguientes:

1. Introducir el lanzamiento impaciente de procesos.
2. Reutilizar las optimizaciones automáticas de GHC.
3. Facilitar el proceso de introducción en el compilador de nuevas transformaciones y análisis propios del lenguaje Eden.

El lanzamiento impaciente de procesos que se ha descrito e implementado ha probado ser fundamental para conseguir buenas paralelizaciones en el lenguaje Eden.

Dado que estamos interesados en mejorar la eficiencia de Eden, es imprescindible poder reutilizar en el mayor grado posible las optimizaciones que realiza automáticamente GHC. Hemos mostrado que dicha reutilización puede ser total, gracias a lo cual se mantiene para Eden las mejoras del código secuencial que obtenía GHC, es decir, un factor de 2.7.

Gracias a la introducción del lenguaje intermedio CoreEden, se ha facilitado el camino para posteriores transformaciones o análisis automáticos que se deseen efectuar. Además, se han adquirido los suficientes conocimientos de bajo nivel de dicha fase de la compilación como para poder implementar rápidamente cualquier nueva transformación.

Además de las tres principales contribuciones expuestas, cabe resaltar en este capítulo la utilidad del simulador Paradise (descrito en el capítulo

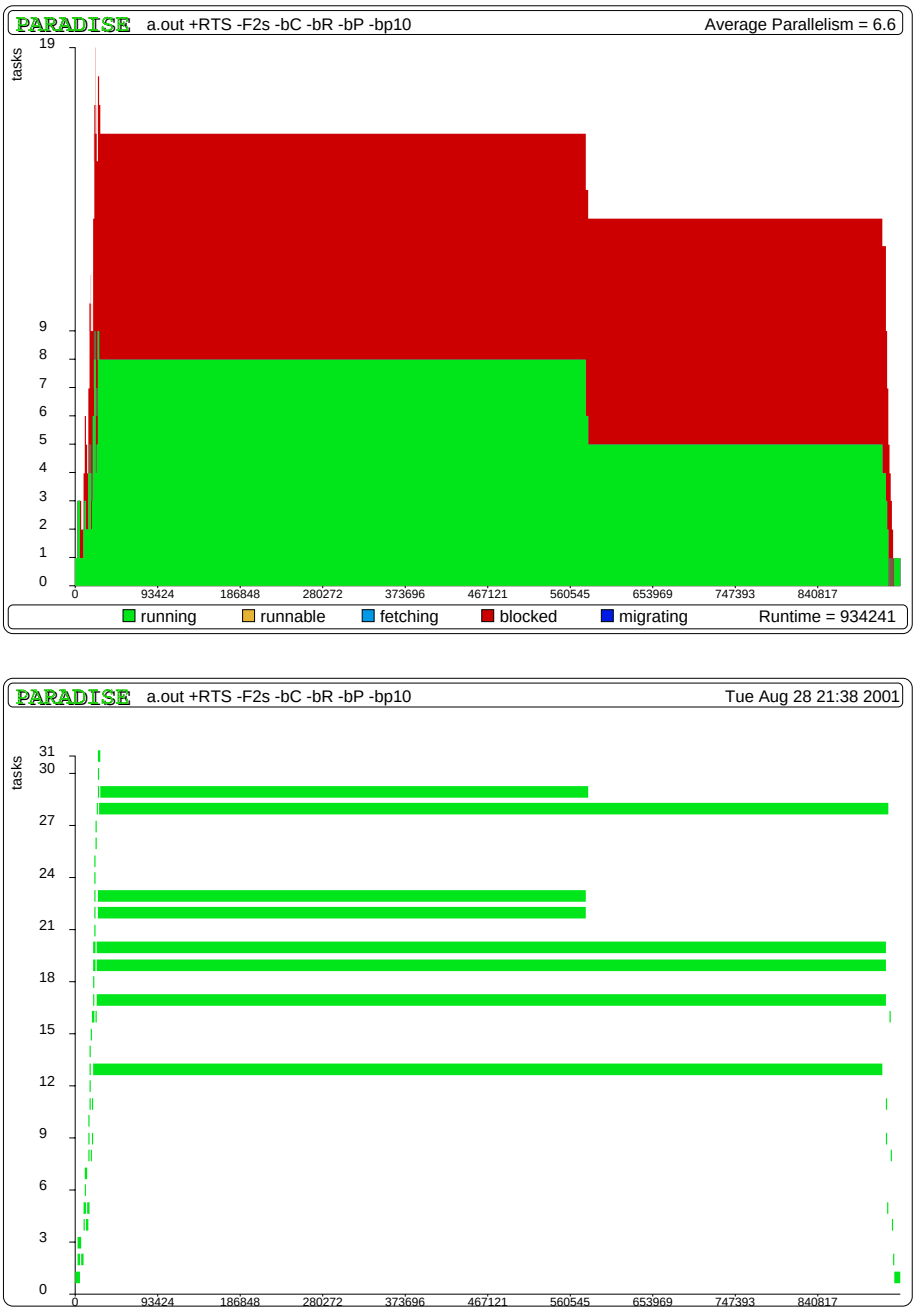


Figura 5.10: Comportamiento global y por hebras de Fibonacci usando lanzamiento impaciente de procesos

anterior) para detectar problemas en el compilador. El ejemplo del programa de Fibonacci que se expuso en la Sección 5.1.2 fue realmente el primer programa que se ejecutó en Paradise. El autor de la tesis, tras implementar la primera versión de Paradise decidió probar un ejemplo sencillo para comprobar el correcto funcionamiento del simulador, y no sólo comprobó dicho correcto funcionamiento, sino también el *incorrecto* funcionamiento del propio lenguaje Eden. Gracias a Paradise, pudo introducirse el lanzamiento impaciente de procesos antes de que se completara el desarrollo de la versión paralela del compilador de Eden.

Capítulo 6

Esqueletos en Eden

Todo el contenido de este capítulo pertenece a la parte original de la tesis. El capítulo desarrolla una librería de esqueletos escritos en el lenguaje Eden. A diferencia de los lenguajes basados en esqueletos, en los que el conjunto de esqueletos era fijo, en este capítulo mostramos cómo puede utilizarse Eden para definir nuevos esqueletos utilizando únicamente las características del propio lenguaje, sin necesidad de utilizar ningún otro lenguaje auxiliar de más bajo nivel. Gracias a ello, el programador Eden podrá disponer de las ventajas de los lenguajes de esqueletos, pero sin verse limitado a tener que expresar sus programas en función de un número fijo de esqueletos predefinidos: cuando existan esqueletos que se ajusten a sus necesidades, los usará, y cuando no los haya los podrá definir él mismo.

Cabe destacar que el único lenguaje funcional paralelo en el que se ha implementado una librería de esqueletos es Eden. De hecho, la originalidad del capítulo radica en buena parte en este punto: las especificaciones de los esqueletos presentados no son especialmente novedosas, sino que la novedad está en el hecho de poder implementarlas eficientemente utilizando un lenguaje de programación funcional de alto nivel. Gracias a ello, la librería de esqueletos puede crecer fácilmente a medida que el usuario lo necesite, sin necesidad de depender de los desarrolladores del compilador.

Parte del trabajo aquí expuesto ha sido publicado en [**KPR01**, **KLPR01**, **PR01**, **LOP$^+$02**, **PRS01**].

6.1 Introducción

En el presente capítulo mostramos cómo puede utilizarse el lenguaje Eden para implementar esqueletos. A diferencia de los lenguajes basados en esqueletos (véase la Sección 2.3) en los que el conjunto de esqueletos disponibles era fijo para cada lenguaje, Eden es de los pocos lenguajes de alto nivel en el que los esqueletos se pueden definir y usar en el mismo lenguaje e incluso en el mismo programa. En la inmensa mayoría de los restantes enfoques, la

creación de esqueletos se considera una actividad del *programador de sistemas* o incluso una actividad asociada a la construcción del compilador. El sistema ofrece un número limitado de esqueletos y la creación de un nuevo esqueleto acarrea normalmente un esfuerzo considerable. El *programador de aplicaciones* puede utilizar los esqueletos disponibles pero no crear otros nuevos.

Así, en Eden podemos obtener los beneficios de tener esqueletos, pero con la ventaja de que los esqueletos se definen en el propio lenguaje. Por tanto, los programadores no necesitan cambiar de lenguaje para crear y usar esqueletos, sino que pueden añadir paulatinamente nuevos esqueletos a la librería de esqueletos, pudiendo definir esqueletos específicos de las áreas de trabajo en las que el programador trabaje habitualmente. Hay una gran similitud entre crear esqueletos en Eden y crear funciones de orden superior en un lenguaje funcional secuencial: en Eden se puede programar en paralelo utilizando explícitamente abstracciones y concreciones de procesos, al igual que en programación secuencial puede programarse directamente usando funciones recursivas. A veces es más conveniente el uso de recursión explícita que el uso de funciones de orden superior, pero los programadores experimentados siempre tratarán de usar orden superior en la medida de lo posible, con el objetivo de reducir tando la cantidad de trabajo empleado como los errores introducidos. Del mismo modo, el programador Eden tratará de usar esqueletos en la medida de lo posible. Así pues, Eden puede verse como un lenguaje con dos niveles, y con dos áreas de aplicación (véase la Figura 6.1):

Programación con procesos: Los procesos se definen y lanzan de forma explícita. A este nivel, cuando nos encontremos en el área de *aplicaciones* el programador expresará su algoritmo con topologías de procesos *ad-hoc*. Esto podrá ser necesario cuando no exista ningún esqueleto que encaje adecuadamente con el algoritmo que deba implementarse. En este mismo nivel, el programador de *sistemas* podrá abstraer un esquema de resolución paralela de problemas escribiendo un esqueleto mediante usos explícitos de procesos.

Programación de orden superior: El programador de aplicaciones diseñará sus algoritmos paralelos utilizando concreciones de los esqueletos disponibles, mientras que el programador de sistemas podrá definir nuevos esqueletos reutilizando los previamente existentes.

En aplicaciones complejas puede ser necesario mezclar ambos niveles y áreas. Esto será posible gracias a que el lenguaje no impone barreras entre ellos.

En las Secciones 6.3 a 6.5 se presenta una librería de esqueletos en Eden, clasificados según el tipo de paralelismo que explotan. Por cada uno de los esqueletos se incluyen una o varias implementaciones, junto con sus correspondientes modelos de coste. Dichos modelos estarán basados en los

	Área de sistemas	Área de aplicaciones
Nivel superior	Esqueletos en términos de esqueletos	Programas en términos de esqueletos
Nivel procesos	Esqueletos en términos de procesos	Programas en términos de procesos

Figura 6.1: Eden como lenguaje de dos niveles y dos áreas

conceptos que se muestran en la siguiente sección.

6.2 Modelos de coste en Eden

Como ya se dijo en la Sección 2.3.1, un modelo de coste es una fórmula matemática que describe el tiempo de ejecución de un programa paralelo. En Eden, los modelos de coste estarán parametrizados por ciertos valores que dependen o bien del *problema* concreto a resolver, o bien del RTS, o bien de la *arquitectura* hardware subyacente. En la Figura 6.2 se detallan dichos parámetros. En lo que respecta a los parámetros que dependen del problema, los dos primeros (N y t_f) caracterizan el tiempo de cómputo de la aplicación, definiendo tanto el tiempo necesario para evaluar una tarea como el número de tareas. Los otros dos parámetros (nwI y nwO) caracterizan la cantidad de comunicaciones del programa, especificando el número de palabras de cada mensaje.

Los parámetros que dependen del RTS son los referentes al tiempo de creación de los procesos. Existen dos parámetros debido a que la creación de un proceso involucra a dos procesadores, por lo que es necesario poder asignar tiempos a cada uno de ellos de forma correcta. El procesador en el que reside el proceso padre del nuevo proceso será el encargado de asumir el coste t_{create}, mientras que al procesador en el que se cree el proceso se le asignará el coste $t_{\#}$.

Obviamente, los parámetros t_f, t_{create} y $t_{\#}$ también dependen del procesador empleado, pero por claridad no lo incluimos en el apartado referente a la arquitectura. En dicho apartado, además del número de procesadores P, y de la latencia δ, resulta fundamental conocer el tiempo que se tardará en enviar y recibir los mensajes. Para ello hará falta conocer el sobrecoste inicial λ que se aplica a todos los mensajes, así como el coste β de enviar cada palabra adicional. La Figura 6.3 (proveniente de [Fos95]) muestra los valores de estos parámetros para varias arquitecturas típicas.

Para simplificar la exposición, en los modelos que se presentan en las secciones posteriores se utilizan los siguientes parámetros que son combinación de los que aparecen en la figura:

Parámetros dependientes del problema	
N	Tamaño de los datos de entrada
t_f	coste secuencial de la función f
nwI	número de palabras de mensaje de entrada a un hijo
nwO	número de palabras de mensaje de salida de un hijo
Parámetros dependientes del RTS	
t_{create}	tiempo de CPU en un procesador-padre para crear un proceso-hijo
$t_{\#}$	tiempo de CPU en un procesador-hijo para crear un nuevo proceso
Parámetros dependientes de la arquitectura	
P	Número de procesadores
δ	latencia de un mensaje, desde comienzo de envío a comienzo de recepción
λ	coste fijo por mensaje para enviar o recibir un mensaje
β	coste por palabra para enviar o recibir un mensaje

Figura 6.2: Parámetros de los modelos de coste

Arquitectura	λ	β
IBM SP2	40	0.11
Intel Paragon	121	0.07
Meiko CS-2	87	0.08
nCUBE-2	154	2.4
Thinking Machines CM-5	82	0.44
Workstations con Ethernet	1500	5.0

Figura 6.3: Parámetros (en μsegundos) de algunas arquitecturas.

$t_{unpackI} = \lambda + \beta.nwI$	coste de desempaquetar un mensaje de entrada a un hijo
$t_{unpackO} = \lambda + \beta.nwO$	coste de desempaquetar un mensaje de salida de un hijo
$t_{packI} = \lambda + \beta.nwI$	coste de empaquetar un mensaje de entrada a un hijo
$t_{packO} = \lambda + \beta.nwO$	coste de empaquetar un mensaje de salida de un hijo

A la hora de calcular los modelos de coste, habrá que tener en cuenta cómo se ubican procesos en procesadores. Recuérdese que, como se dijo en la Sección 3.3.3, en Eden existen dos modos de ubicación de tareas: de forma *round-robin* y de forma aleatoria. En la mayoría de los casos supondremos el uso del modo *round-robin*, pues permitirá controlar mejor cómo se reparten las tareas entre procesadores. En el caso de utilizar el modo aleatorio, no podría predecirse con exactitud la aceleración, pues ejecuciones distintas podrían tener tiempos de ejecución bastante diferentes.

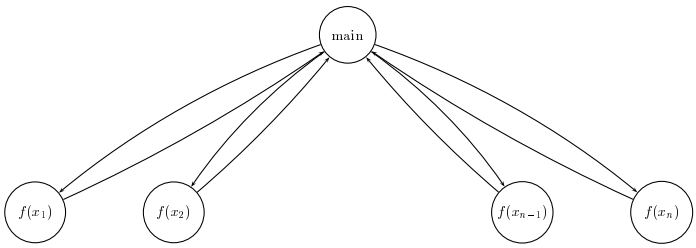


Figura 6.4: Topología de procesos de `map`

6.3 Paralelismo de datos

6.3.1 Map

En numerosos algoritmos paralelos es necesario aplicar una misma operación a un conjunto de datos del mismo tipo, por ejemplo calcular el determinante de todas las matrices de una lista. No existen dependencias entre unos datos y otros, es decir, el resultado de cada aplicación sólo depende de su correspondiente argumento. En ese caso es indiferente el orden en que se calcule el conjunto de resultados. En particular pueden calcularse en paralelo. Dado que la fuente del paralelismo está en los datos, es el ejemplo prototípico de paralelismo de datos. Estructuras apropiadas para aplicarlo son los vectores o las matrices y las listas.

En la terminología de esqueletos, el paralelismo de datos para listas está expresado por la función `map` cuyo tipo y algoritmo en Haskell son los siguientes:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

El esqueleto `map` ha de aplicar una misma función a todos los elementos de una lista. Describimos a continuación distintas posibles implementaciones paralelas en Eden. En la literatura, este esqueleto aparece continuamente y normalmente se implementa a bajo nivel. En Eden se pueden expresar varias implementaciones distintas utilizando un alto nivel de abstracción. Dado que Eden es un lenguaje de programación orientado al paralelismo de tareas, todos los esquemas tienen un aspecto maestro-esclavo en el que un proceso gestor controla el trabajo de los restantes procesos.

Implementación ingenua

Es la más simple y consiste en crear un proceso para evaluar cada elemento de la lista. El operador `using` aplica la estrategia de evaluación `spine` (véase la Sección 2.4.3) a la lista de lanzamientos de procesos. El resultado

es que se crean impacientemente todos los procesos indicados en la lista y cada uno comienza a ejecutarse en paralelo con los demás (si hay suficientes procesadores). Cada proceso simplemente aplica la función `f` del esqueleto al elemento correspondiente de la lista. Esta solución sólo es apropiada cuando la granularidad de las tareas es gruesa, y además el tiempo de cómputo de todas ellas es similar, pues de lo contrario el reparto de carga no sería satisfactorio. Para más detalles sobre estrategias puede verse [THLP98].

```
map_naive :: (Transmissible a,Transmissible b) => (a->b)->[a]->[b]
map_naive f xs = [(process y -> f y) # x | x <- xs] `using` spine
```

Para calcular el modelo de coste, suponemos el modo *round-robin* de distribución de procesos, y suponemos también que la granularidad de los procesos (es decir, el coste de la función `f`) es uniforme. Nótese que de no hacer dichas suposiciones no podría obtenerse un modelo de coste fiable, pues dependería de dos distribuciones aleatorias: la distribución de procesos en procesadores y la distribución de granularidades de los distintos procesos. Con las suposiciones mencionadas, el modelo sería el siguiente:

$$\begin{aligned} t_{map\_naive} &= L_{init} + t_{worker} + L_{final} \\ L_{init} &= P(t_{create} + t_{packI}) + \delta \\ L_{final} &= \delta + t_{unpackO} \\ t_{worker} &= \lceil \tfrac{N}{P} \rceil (t_{\#} + t_{unpackI} + t_f + t_{packO}) \end{aligned}$$

El tiempo total de ejecución viene dado por la suma de los tiempos de las tareas que se encuentran en el camino crítico. Distinguimos entre el tiempo de inicialización (L_{init}), el tiempo de cómputo realmente paralelo (L_{worker}), y el tiempo de finalización (L_{final}), distinción que también realizaremos en el resto de los modelos que mostraremos a lo largo del presente capítulo.

La inicialización del esqueleto contabiliza el tiempo que se tardará hasta que todos los procesadores empiecen a trabajar. Para ello, en el camino crítico han de crearse P procesos en el procesador padre y también han de empaquetarse P mensajes, para enviar una tarea a cada uno de los procesos creados. Después de enviar una tarea a cada procesador, habrá que esperar un tiempo δ hasta que el procesador al que se le envió la última tarea la reciba realmente y pueda empezar a realizar cómputos. Esa misma latencia δ tendrá que contabilizarse también en el tiempo de finalización del esqueleto (L_{final}), de modo que se espere hasta que llegué el último mensaje desde el último procesador. A dicha latencia habrá que sumarle el coste $t_{unpackO}$ de desempaquetar los datos recibidos en ese último mensaje.

El tiempo de cómputo paralelo t_{worker} será el tiempo de cómputo del procesador más cargado, que recibirá $\lceil \frac{N}{P} \rceil$ tareas. Para resolver cada una de esas tareas, primero se tendrá que crear el proceso correspondiente, para lo cual empleará un tiempo $t_{\#}$. Posteriormente, en un tiempo $t_{unpackI}$ se desempaquetará la tarea asociada al proceso, para luego efectuar los cómputos

que resuelven la tarea en un tiempo t_f. Finalmente, el resultado del cómputo se empaquetará y se enviará al proceso principal, hecho que se contabiliza en el tiempo t_{packO}.

El resto de las actividades del procesador padre (es decir, el envío de $N - P$ mensajes y la recepción de $N - 1$ resultados) se consideran fuera del camino crítico o de coste despreciable. Si no fuera así, habría que añadir los costes correspondientes a t_{worker}, pues el procesador con mayor carga sería el principal, que además de resolver sus propias tareas debería distribuir tareas y recolectar resultados.

Implementación mediante una granja

Cuando hay menos procesadores que elementos en la lista, se puede reducir la sobrecarga asociada a la creación de procesos creando tan sólo uno por procesador y repartiendo la lista de tareas entre ellos. De esta forma se consigue una granularidad mayor para cada proceso. Esta configuración se denomina *granja*. En ella, el proceso padre o "granjero" es responsable de la distribución de los datos a los procesos "trabajadores" y de la recolección de los resultados. Para ello admite dos parámetros adicionales, las funciones `unshuffle` y `shuffle` que realizan respectivamente el reparto y la recolección. Han de ser tales que, para toda lista *xs* y para todo natural n, `xs == (shuffle . unshuffle n) xs`. Al igual que en la versión ingenua, el modelo de coste supone que la distribución de procesos en procesadores es *round-robin* y que la granularidad de las tareas es uniforme. Si el número de procesadores es elevado, las tareas de distribución y recolección pueden suponer un coste no despreciable por lo que es más eficiente dedicar un procesador al proceso granjero. Para ello, el esqueleto admite un parámetro más `threshold` que especifica el umbral a partir del cual debe crearse un proceso trabajador menos para dedicar un procesador al granjero. La constante `noPe` está reservada en Eden y toma en tiempo de ejecución el número de procesadores disponibles.

```
map_farm :: (Transmissible a,Transmissible b) =>
            (Int->[a]->[[a]]) -> ([[b]]->[b]) ->Int ->
            (a->b) ->[a] ->[b]
map_farm unshuffle shuffle threshold f tasks
  | noPe > threshold = farm (noPe-1)
  | otherwise        = farm noPe
  where farm np  = shuffle (map_naive (map f) (unshuffle np tasks))
```

Las siguientes funciones `shuffle` y `unshuffle` realizan una distribución y recolección *round-robin* de las tareas entre los procesos trabajadores:

```
unshuffle :: Int -> [a] -> [[a]]
unshuffle n ins
 | lf < n    = take n (map (:[]) firsts ++ repeat [])
```

```
 | otherwise = mzipWith' (:) firsts (unshuffle n rest)
 where (firsts, rest) = splitAt n ins
       lf             = length firsts
       mzipWith' f [] ys = []
       mzipWith' f (x:xs) ~(y:ys) = f x y : mzipWith' f xs ys

shuffle :: [[a]] -> [a]
shuffle = concat . transpose
```

El modelo de coste para `map_farm` es el siguiente:

$$\begin{aligned} t_{map\_farm} &= L_{init} + t_{worker} + L_{final} \\ L_{init} &= P(t_{create} + t_{packI} + t_{unshuffle_1}) + \delta \\ L_{final} &= \delta + t_{unpackO} + t_{shuffle_1} \\ t_{worker} &= t_{\#} + \lceil \frac{N}{P} \rceil (t_{unpackI} + t_f + t_{packO}) \end{aligned}$$

La diferencia esencial con el modelo de `map_naive` es que ahora el coste $t_{\#}$ de creación de un proceso-hijo sólo se carga una vez a cada trabajador. Se añaden tambien en el camino crítico el coste de `unshuffle` para los primeros P mensajes y el de `shuffle` para el último mensaje recibido por el proceso granjero.

Implementación mediante trabajadores replicados

La implementación mediante una granja no es adecuada en las tres situaciones siguientes:

- Cuando la granularidad de las tareas no es uniforme.
- Cuando los procesadores no tienen todos la misma potencia de cálculo.
- Cuando el esqueleto comparte los procesadores con otros programas.

El problema en todos los casos es el mismo: el reparto de carga entre los distintos procesadores no es equitativo. En el primer caso, las tareas que recibe uno de los procesadores pueden ser de grano grueso, mientras que las de otro pueden ser de grano fino. La Figura 6.5 muestra un ejemplo de un reparto de carga que es deficiente debido a este hecho, donde resulta evidente que se desaprovecha tiempo de cómputo de los procesadores con tareas de grano fino (los procesadores 2 y 3 en la figura). El comportamiento ideal sería el de la Figura 6.6.

En los otros dos casos el problema es exactamente el mismo: si los procesadores tienen distinta potencia de cálculo, los más potentes deberían recibir más trabajo, pues de lo contrario se desaprovecharían. Por último, si los procesadores no están dedicados únicamente a nuestra aplicación, otros trabajos pueden "robarnos" tiempo de CPU, y en general no lo robarán homogéneamente a todos los procesadores. El resultado es que, desde el punto

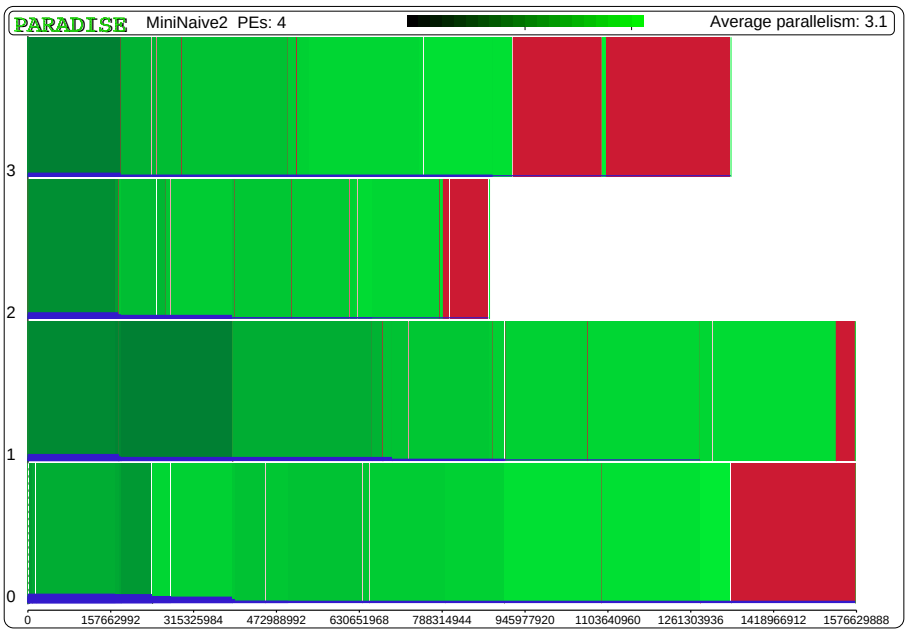


Figura 6.5: Comportamiento por procesadores con mal reparto de carga.

de vista de nuestra aplicación, la potencia de cómputo de los procesadores será diferente.

En los tres casos la solución consiste en no realizar una distribución *a priori* de las tareas entre procesadores, sino en irlas distribuyendo bajo demanda a medida que se completan las anteriores. Esta implementación recibe el nombre de *trabajadores replicados* [**KPR01**]. En ella, al igual que en la granja, se crean tantos procesos trabajadores como procesadores, o uno menos si se desea reservar un procesador para la distribución y recolección. El proceso "gestor" asigna inicialmente una o más tareas a cada procesador. Si se asignan al menos dos, se conseguirá solapar el tiempo de comunicación del resultado de la primera tarea y de la recepción de una nueva, con el cálculo de la segunda, mejorando así la utilización de los procesadores al disminuir los tiempos ociosos. Cada vez que un trabajador finaliza una tarea, envía su resultado al gestor, el cual asigna inmediatamente una nueva tarea al trabajador. El cómputo termina cuando el gestor ha recibido los resultados de todas las tareas.

La clave de este esqueleto está en su naturaleza *reactiva*. No es posible escribir una función que distribuya la lista de tareas entre los trabajadores sin tener en cuenta el *orden temporal* de terminación de las tareas previas. Este orden temporal sólo se conoce en tiempo de ejecución, dado que depende de

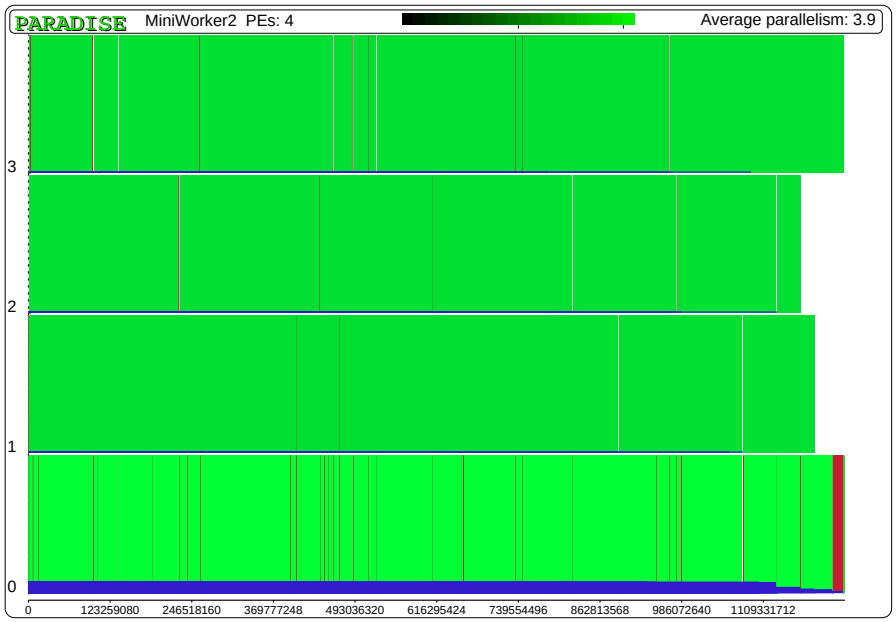


Figura 6.6: Comportamiento por procesadores con buen reparto de carga.

la granularidad de las tareas y de la potencia de cálculo de los procesadores. Un lenguaje funcional puro no puede computar dicho orden a partir de los resultados de los trabajadores. Pero Eden *sí* puede, ya que dispone del proceso reactivo `merge` que produce una lista en la que el orden de los valores a la salida es precisamente el orden temporal en que dichos valores estaban disponibles en las listas de entrada. Por tanto el resultado es no-determinista, lo que hace de `merge` un proceso no funcional.

La implementación de `map` con trabajadores replicados, que llamaremos `map_rw`, utiliza la función `rw` que recibe como parámetros: (1) el número de trabajadores a crear; (2) el número de tareas iniciales asignadas a cada trabajador; (3) la función de cómputo; y (4) la lista de tareas a distribuir. Las principales ideas de la implementación son las siguientes:

- La lista `tasksAndIds` asigna un número diferente a cada tarea. Este número se utilizará posteriormente por `sortMerge` para ordenar los resultados de modo que el orden coincida con el de las tareas iniciales. Dado que los resultados producidos por cada proceso ya están ordenados, la función de ordenación es simplemente una generalización de la típica función de mezcla del algoritmo mergesort.

- Se define un nuevo tipo de datos `ACK` para devolver datos al proceso principal. Dicho tipo no sólo incluye el resultado computado, sino

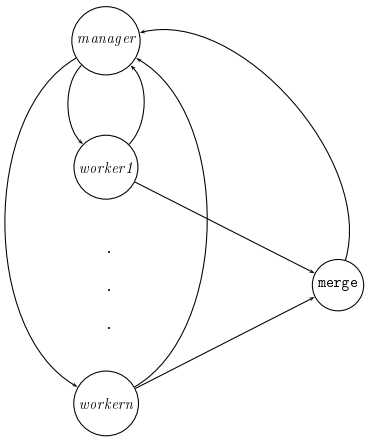


Figura 6.7: Topología de procesos generada con trabajadores replicados.

también el identificador del proceso trabajador (para que pueda asignársele una nueva tarea) y el identificador de la tarea (para que pueda ordenarse el resultado).

- Los resultados se mezclan de forma no determinista generándose la lista `unordResult`, y se utilizan en `distribute` para distribuir nuevas tareas tan pronto como un proceso trabajador termina una tarea.

La topología de procesos creada puede verse en la Figura 6.7. Nótese que los resultados que produce cada trabajador llegan al gestor a través de dos caminos: (1) el directo, que se usa para extraer los resultados finales; y (2) a través de `merge`, para poder asignar nuevas tareas dinámicamente a medida que se finalizan las anteriores. El código fuente es el siguiente:

```
map_rw :: (Transmissible a,Transmissible b) => Int->(a->b)->[a]->[b]
map_rw thr | noPe > thr = rw (noPe-1) 2
           | otherwise  = rw noPe 2
rw :: (Transmissible a,Transmissible b) => Int->Int->(a->b)->[a]->[b]
rw np prefetch f tasks =  results     where
  results      = sortMerge outsChildren
  outsChildren = [(worker f i)#inputs |
                  (i,inputs)<- zip [0..np-1] inputss]
  inputss      = distribute tasksAndIds
                            (initReqs++(map owner unordResult))
  tasksAndIds  = zip [1..] tasks
  initReqs     = concat (replicate prefetch [0..np-1])
  unordResult  = merge # outsChildren
  distribute [] _          = replicate np []
  distribute (e:es) (i:is) = insert i e (distribute es is)
    where insert 0 e ~(x:xs)     = (e:x):xs
```

```
          insert (n+1) e ~(x:xs) = x:(insert n e xs)
worker f i = process ts -> map f' ts
  where f' (id_t,t) = ACK i id_t (f t)

data ACK a = Transmissible a => ACK Int Int a
owner (ACK p _ _)  = p
result (ACK _ _ b) = b
taskId (ACK _ i _) = i

-- Extract a sorted output from an unordered
-- list of lists of acknoledgments
sortMerge :: [[ACK a]] -> [a]
sortMerge xss = map result (sortN xss)
-- Assuming each of the lists are sorted,
-- it merges them in order to form a sorted list
sortN :: (Ord a) => [[a]] -> [a]
sortN [] = []
sortN [xs] = xs
sortN xss = sort2 (sortN xss1) (sortN xss2)
  where (xss1,xss2) = unshuffle xss
sort2:: (Ord a) => [a] -> [a] -> [a]
sort2 [] ys = ys
sort2 xs [] = xs
sort2 (x:xs) (y:ys) = if x < y then x:(sort2 xs (y:ys))
                               else y:(sort2 (x:xs) ys)
unshuffle :: [a] -> ([a],[a])
unshuffle = foldr (\ x ~(xs1,xs2) -> (x:xs2,xs1)) ([],[])
```

Merece la pena resaltar cómo interaccionan las distintas características de Eden en este esqueleto para obtener el comportamiento deseado:

Pereza La conexión circular entre el proceso principal y los trabajadores se describe elegantemente mediante ecuaciones mutuamente recursivas. El esquema funciona porque las listas son perezosas y todas las funciones utilizadas son *productivas*, siendo funciones productivas aquellas que pueden producir parte de su salida antes de consumir por completo sus entradas (véase [Sij89] para una descripción más formal de productividad). Ejemplos de este tipo de funciones son `zip`, `map`, `concat`, `++` y `distribute`. En [PS01b] se demuestra que este y otros esqueletos son productivos.

Impaciencia En general, la pereza se opone al paralelismo. Nuestro esquema es paralelo gracias a que los procesos del esqueleto se lanzan impacientemente, y gracias a que una vez lanzados producen sus salidas sin necesidad de demanda. En el esqueleto, tanto `outsChildren` como `unordResult` se producen impacientemente por los procesos correspondientes. Ademas, siempre hay demanda para la lista `inputss`,

puesto que se han creado hebras en el proceso padre para alimentar a los trabajadores.

No determinismo reactivo El proceso `merge` es el corazón del esqueleto. Sin este proceso reactivo sería imposible reaccionar a los mensajes de los trabajadores en el orden en el que se producen.

Nótese que, gracias al uso de la función `sortMerge`, el no determinismo no contamina el exterior del esqueleto. Por tanto, y a pesar de que el esqueleto internamente se comporta de manera no determinista, el resultado de `map_rw` es completamente determinista para un usuario del mismo. De hecho, su comportamiento es exactamente igual al de la función `map`.

El modelo de coste es similar al de la implementación mediante una granja, con la importante salvedad de que ahora, a pesar de que las tareas pueden tener distinta granularidad, se considera que cada trabajador recibe el número promedio de tareas $\frac{N}{P}$ (nótese la ausencia del operador $\lceil . \rceil$ de redondeo por exceso), cada una de ellas con un coste promedio t_{comp} que es la media aritmética del coste individual t_{f_i} de cada tarea i. Es decir, el modelo permite que la granularidad de las tareas sea irregular, y aun así permite suponer que la carga está perfectamente equilibrada entre los procesadores. Consideramos que $t_{distribute_1}$ acumula los costes de `zip`, `concat` y `replicate` para una tarea. Por último, y al igual que en los modelos anteriores, se supone que se usa el modo *round-robin* de distribución de procesos, para garantizar que los procesos se ubican en procesadores diferentes.

$$\begin{aligned}
t_{map_rw} &= L_{init} + t_{worker} + L_{final} \\
L_{init} &= P(t_{create} + t_{packI} + t_{distribute_1}) + \delta \\
L_{final} &= \delta + t_{unpackO} + t_{sortMerge_1} \\
t_{worker} &= t_{\#} + \frac{N}{P}(t_{unpackI} + t_{comp} + t_{packO}) \\
t_{comp} &= \frac{1}{N}\sum_{i=1}^{N} t_{f_i}
\end{aligned}$$

Implementación con datos fijos compartidos

Frecuentemente, las implementaciones que hemos mostrado tanto con la granja de procesos como con los trabajadores replicados no son convenientes. Cuando existen estructuras de datos fijas y que deben usar todas las tareas, debería enviarse dicha estructura una única vez a cada proceso, de modo que las tareas de cada proceso compartan la misma estructura. De lo contrario las comunicaciones entre el proceso principal y los trabajadores podrían incrementarse drásticamente. Un ejemplo de esta situación puede verse en el trazador de rayos descrito en [**KLPR01**], donde todas las tareas comparten una escena fija.

La solución es muy simple: los nuevos esqueletos tienen un parámetro extra (los datos compartidos) que se envía a los trabajadores una única vez a través de un canal independiente para que la comunicación sólo se efectúe

una vez. Para el caso de los trabajadores replicados, el código fuente sería el siguiente:

```
rwF :: (Transmissible a, Transmissible b, Transmissible fixed) =>
       Int -> Int -> fixed -> (fixed -> a -> b) -> [a] -> [b]
rwF np prefetch fixed fw tasks =  results     where
  ...
  outsChildren =  [(worker fw i) # (fixed,inputs) |
                   (i,inputs) <- zip [0..np-1] inputss]

worker :: (Transmissible a, Transmissible b, Transmissible fixed) =>
          (fixed ->a ->b) -> Int -> Process (fixed,[(Int,a)]) [ACK b]
worker f i = process (fixed,ts) -> map f' ts
  where f' (id_t,t) = ACK i id_t (f fixed t)
```

y para las granjas se procedería de forma análoga. Nótese que estas nuevas implementaciones no se ajustan exactamente al esquema secuencial de `map`, sino a la siguiente redefinición del mismo:

```
mapF :: fixed -> (fixed -> a -> b) -> [a] -> [b]
mapF fixed f []     = []
mapF fixed f (x:xs) = f fixed x : mapF fixed f xs
```

Esta definición no tiene mucho sentido en secuencial, ya que se podría haber utilizado simplemente `map (f fixed) xs`. Ahora bien, en el caso paralelo necesitamos hacer explícito que queremos enviar `fixed` a través de un canal, pues de lo contrario se pasaría sin evaluar a los procesos hijo, y se repetiría su evaluación tantas veces como procesos trabajadores tuviéramos.

Con respecto a los modelos de coste, las únicas diferencias con los introducidos anteriormente son dos: (1) ahora t_{worker} tiene un coste extra desempaquetando el mensaje que contiene la estructura compartida; y (2) L_{init} tiene un coste extra empaquetando ese mismo mensaje, y debe empaquetarlo P veces.

6.3.2 Map & reduce

Un esquema típico que proporcionan gran cantidad de lenguajes de esqueletos es el denominado *map & reduce*. La idea paralela subyacente consiste en ejecutar un paso `map` paralelo seguido de una combinación también paralela de dichos resultados. La especificación secuencial es una combinación de las funciones `map` y `foldl`, donde se recibe una función a aplicar a todos los elementos de una lista, y otra función asociativa, conmutativa y con elemento neutro `neutral` que se usará para combinar los resultados:

```
mr f comb neutral tasks = foldl comb neutral (map f tasks)
```

Implementación

Este esquema puede paralelizarse aplicando primero en paralelo la etapa correspondiente al `map`, para después aplicar en secuencial el `fold`. Ahora bien, sabiendo que el operador es asociativo y que tiene elemento neutro, cada proceso puede encargarse de aplicar la función `fold` a la parte del resultado que ha calculado, de modo que cada proceso devuelva un único valor. Así, la etapa secuencial se reducirá a aplicar `fold` sobre una lista mucho más pequeña (un elemento por cada proceso), eliminándose posibles cuellos de botella. En la implementación en Eden incluiremos dos parámetros adicionales: el primero será el número de procesos a crear, mientras que el segundo permitirá especificar cómo quieren repartirse las tareas entre los procesos. Dado que el reparto es determinista, la granularidad de las tareas debe ser regular, pues de lo contrario el reparto de carga podría degradarse. En dicho caso la solución pasaría por utilizar un esquema similar al de los trabajadores replicados.

```
mrPM :: (Transmissible a, Transmissible b) =>
        Int -> (Int -> [a] -> [[a]]) ->
        (a -> b) -> (b -> b -> b) -> b -> [a] -> b
mrPM np unshuffle f comb neutral tasks = foldl' comb neutral results
  where results = [(workerPM f comb neutral) # mtasks
                   | mtasks <- unshuffle np tasks] `using`  spine
workerPM f comb neutral = process tasks -> foldl' comb neutral results
  where results = map f tasks
```

Nótese que, a diferencia de lo que ocurría con las granjas de procesos del esqueleto `map`, ahora sólo es necesario suministrar una función `unshuffle`, pero no la correspondiente `shuffle`. El motivo es que el operador de combinación es asociativo y conmutativo, por lo que da igual el orden en que se combinen los resultados. Nótese también que utilizamos `foldl'`, es decir, la versión estricta de `foldl`. Gracias a ello podemos mejorar la eficiencia en consumo de memoria del esqueleto.

Con esta implementación hemos conseguido paralelizar la etapa `fold`, y reducir las comunicaciones que van desde los trabajadores al proceso padre, puesto que sólo deben comunicar un resultado en lugar de una lista. Ahora bien, al comienzo de la ejecución el proceso padre debe comunicar las listas de tareas a los trabajadores. En muchas ocasiones (véase por ejemplo la Sección 7.2.3) la generación de la lista de tareas es muy poco costosa, mientras que la comunicación de las mismas lo es mucho más. En dichos casos sería preferible que cada trabajador generara él mismo su propia lista de tareas, de modo que se redujeran las comunicaciones iniciales entre el padre y los trabajadores. Esta idea general, que recibe el nombre de *autoservicio* (véase [**KLPR01**]), se implementaría del siguiente modo para el esqueleto *map & reduce*:

```
mrAS :: Transmissible b => Int -> (Int -> [a] -> [[a]]) ->
```

```
        (a->b) -> (b->b->b) -> b -> [a] -> b
mrAS np unshuffle f comb neutral tasks = foldl' comb neutral results
 where results = [(workerAS f comb neutral mtasks) # ()
                  | mtasks <- unshuffle np tasks] `using`  spine

workerAS f comb neutral tasks = process () -> foldl' comb neutral results
  where results  = map f tasks
```

Nótese que la lista de tareas no se pasa a los procesos trabajadores a través de un canal de entrada, sino a través de un parámetro. Dado que los parámetros no se reducen a forma normal en el padre (véase la Sección 3.2), sólo será necesario comunicar la clausura que representa cómo obtener la lista de tareas. Este es el motivo por el que en el contexto de `mrAS` sólo aparece `Transmissible b`, y no aparece `Transmissible a`.

Al igual que en el caso de `map`, el número de procesos que deben emplearse dependerá del de procesadores disponibles. Dado que habitualmente la distribución de tareas entre procesos será un simple reparto *round-robin*, la incluiremos por defecto en la implementación:

```
map_reduce_as :: Transmissible b =>
                 Int -> (a->b) -> (b->b->b) -> b -> [a] -> b
map_reduce_as thr
  | noPe > thr = mrAS (noPe-1) unshuffle
  | otherwise  = mrAS noPe unshuffle

unshuffle :: Int -> [a] -> [[a]]
unshuffle n xs = [takeEach n (drop i xs) | i <- [0..n-1]]
 where takeEach :: Int -> [a] -> [a]
       takeEach n []     = []
       takeEach n (x:xs) = x : takeEach n (drop (n-1) xs)
```

Nótese que la definición de `unshuffle` está pensada para su ejecución paralela. La versión para ejecución secuencial podría hacerse más eficiente garantizando que la lista de tareas se recorre sólo una vez. Ahora bien, en nuestra implementación cada proceso podrá calcular sus tareas de forma independiente al resto de procesos, y cada proceso sólo recorrerá una vez la lista de tareas.

El modelo de coste de `map_reduce_as` es el siguiente, donde se supone que el tiempo t_f necesario para evaluar una tarea es igual para todas ellas:

$$\begin{aligned} t_{map\_reduce\_as} &= L_{init} + t_{worker} + L_{final} \\ L_{init} &= Pt_{create} + \delta \\ L_{final} &= \delta + t_{unpackO} + t_{foldl_P} \\ t_{worker} &= t_{\#} + t_{takeEach} + \lceil \tfrac{N}{P} \rceil (t_f + t_{packO}) + t_{fold_{\lceil \frac{N}{P} \rceil}} \end{aligned}$$

Nótese que las diferencias con los modelos de `map` radican en que L_{init} no cuenta costes por comunicaciones del padre a los hijos, L_{worker} cuenta costes para generar las tareas ($t_{takeEach}$) y combinar los resultados ($t_{fold_{\lceil \frac{N}{P} \rceil}}$), y

L_{final} cuenta el coste de combinar los resultados producidos por los trabajadores (t_{foldl_P}).

6.4 Paralelismo de tareas

Otra fuente de paralelismo en los programas es el llamado *paralelismo de tareas*, en el cual hay que identificar un conjunto de tareas posiblemente distintas y trabajando cada una de ellas con datos diferentes tales que, combinando sus resultados, se puede calcular el resultado del problema original. Presentamos en esta sección tres esqueletos: el de *divide y vencerás*, el de *ramificación y poda* y la *tubería* de procesos.

6.4.1 Divide y vencerás

En el esquema divide y vencerás es necesaria una función que determine si el problema es suficientemente sencillo o no. En caso de que lo sea, se aplicará otra función que resuelve el caso base, mientras que si no lo es hace falta dividir el problema en una lista de subproblemas, aplicar a cada uno de ellos el método divide y vencerás y posteriormente combinar todos los resultados parciales para obtener el resultado global. La especificación de este esqueleto es simplemente el esquema secuencial, que expresado en Haskell es:

```
dc :: (a->Bool) -> (a->b) -> (a->[a]) -> (a->[b]->b) -> a -> b
dc trivial solve split combine x | trivial x = solve x
                                 | otherwise = combine x children
  where children = map (dc trivial solve split combine) (split x)
```

Nótese la suposición de que la función `combine` puede utilizar el argumento `x`, lo cual sucede a menudo en la aplicación de este esquema. Nótese también que el árbol de llamadas no es necesariamente homogéneo y que las soluciones triviales pueden aparecer en cualquier nivel del árbol, como se aprecia en la Figura 6.8.

Implementación ingenua

La solución más evidente consiste en crear un árbol dinámico de procesos que se corresponda con el árbol de llamadas del esquema secuencial. Un parámetro natural permite al programador controlar hasta qué nivel del árbol desea que se creen nuevos procesos, de modo que a partir de ese nivel se utilizará el algoritmo secuencial. De no hacerse así, la granularidad de las hojas sería excesivamente pequeña.

```
dc_naive :: (Transmissible a,Transmissible b) => Int -> (a->Bool) ->
            (a -> b) -> (a -> [a]) -> (a -> [b] -> b) -> a -> b
```

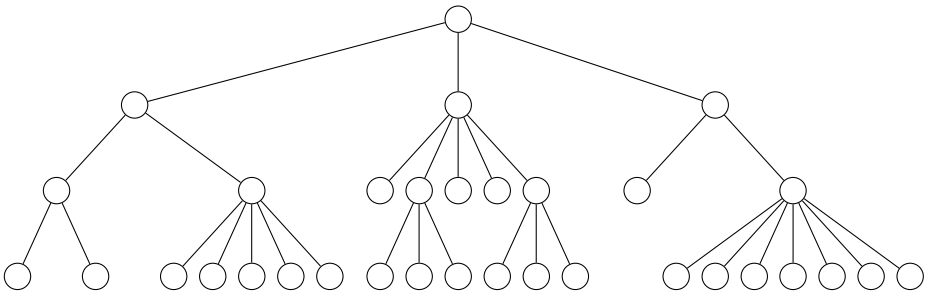


Figura 6.8: Ejemplo de topología de procesos generada con un divide y vencerás

```
dc_naive 0 triv solve split comb = dc triv solve split comb
dc_naive d triv solve split comb x | triv x    = solve x
                                   | otherwise = comb x children
  where children = map_naive (dc_naive (d-1) triv solve split comb)
                             (split x)
```

Nótese la utilización del esqueleto `map_naive` en cada nivel recursivo. Se crean tres tipos de procesos: (1) los que corresponden a un nivel intermedio del árbol y cuya tarea consiste en descomponer un problema en subproblemas, y en componer un resultado a partir de los subresultados; (2) los que se dedican a la resolución de un problema trivial; y (3) los que están a profundidad d y utilizan el esquema secuencial. Ésto da lugar a una gran diversidad de granularidades. A diferencia de los esqueletos precedentes, aquí no existe un proceso central que distribuya el trabajo entre los restantes: cada proceso (excepto la raíz) es "hijo" de algún otro y "padre" a su vez (excepto los de profundidad d y los de las hojas del árbol) de otros procesos.

Dado que el número de procesos y sus granularidades son desconocidos *a priori*, el modo apropiado para el RTS es el aleatorio (véase la Sección 3.3.3). Es difícil dar un modelo de coste para esta implementación porque habría que suponer conocidas dos distribuciones: la de procesos en procesadores y la de granularidades entre los procesos. Esta segunda es además dependiente del problema concreto a resolver. La combinación de dos distribuciones distintas en un mismo modelo produciría resultados estadísticamente poco significativos.

Implementación mediante trabajadores replicados

Esta implementación persigue dos objetivos: (1) disminuir la sobrecarga debida a la creación de procesos, creando tan solo uno por procesador; y (2) distribuir la carga equilibradamente entre los procesadores. Para ello, un proceso gestor descompone el árbol hasta una profundidad prefijada d y crea

una lista de tareas, donde cada una de ellas se resolverá utilizando el algoritmo secuencial. Para aplicarlo se puede emplear cualquier implementación del esqueleto `map`, aunque la recomendable es la `rw`, pues será capaz de repartir mejor la carga. El proceso gestor, que ha conservado los argumentos de cada nivel en una estructura de datos de tipo árbol, combina cada subconjunto de resultados con su argumento correspondiente hasta alcanzar la raíz del árbol de llamadas, donde calcula el resultado del problema inicial. Esta implementación es apropiada cuando el trabajo de descomposición y combinación es pequeño en comparación con el de resolución de subproblemas, de forma que un solo procesador lo pueda realizar sin convertirse en un cuello de botella. Si ese no fuera el caso, podría adoptarse una solución intermedia de modo que la combinación de resultados se realizara entre varios procesos. El código fuente de la versión con trabajadores replicados es la siguiente, donde el primer parámetro se utiliza para decidir a partir de cuántos procesadores merece la pena que el proceso principal resida en un procesador propio:

```
dc_rw :: (Transmissible a,Transmissible b) => Int -> Int -> (a->Bool)
         -> (a -> b) -> (a-> [a]) -> (a-> [b] -> b) -> a -> b
dc_rw thr d trivial solve split combine x
  = combineTop combine levels results
  where (tasks,levels) = generateTasks d trivial split x
        results        = map_rw thr (dc trivial solve split combine)
                                    tasks
```

Además de generar una lista de tareas, la función `generateTasks` creará un árbol que permitirá a la función `combineTop` combinar adecuadamente la lista de soluciones, de modo que se sepa qué soluciones deben combinarse con qué otras:

```
data Tree a = Node a [Tree a]

generateTasks :: Int -> (a -> Bool) -> (a -> [a]) -> a -> ([a],Tree a)
generateTasks 0 _ _ a = ([a],Node a [])
generateTasks n trivial split a
  | trivial a = ([a],Node a [])
  | otherwise = (concat ass,Tree a ts)
  where assts = map (generateTasks (n-1) trivial split) (split a)
        (ass,ts) = unzip assts

combineTop :: (a -> [b] -> b) -> (Tree a) -> [b] -> b
combineTop c t bs = fst (combineTop' c t bs)
combineTop' :: (a->[b]->b) -> (Tree a) -> [b] -> (b,[b])
combineTop' _ (Node a []) (b:bs) = (b,bs)
combineTop' combine (Node a ts) bs = (combine a (reverse res),bs')
  where (bs',res) = foldl f (bs,[]) ts
        f (olds,news) t = (remaining,b:news)
          where (b,remaining) = combineTop' combine t olds
```

Puede utilizarse el modelo de coste de `map_rw`, cargando en el parámetro $t_{distribute_1}$ el coste promedio de generar P tareas y en $t_{sortMerge_1}$ el coste promedio de combinar un resultado. Asimismo, es necesario conocer el número de tareas N a profundidad d y el tiempo promedio t_{comp} de resolución de cada una. Ambos datos pueden obtenerse a partir del algoritmo secuencial.

6.4.2 Ramificación y poda

Los algoritmos de ramificación y poda (e.g. véase [HS78, Capítulo 8]) no pueden expresarse utilizando ninguno de los esqueletos mostrados hasta el momento, pues ninguno permite explorar un espacio de búsqueda orientando dicha búsqueda en función de qué nodos son más prometedores. El primer parámetro `f` de nuestro esqueleto de ramificación y poda es la función que expande el espacio de búsqueda, generando una nueva lista de tareas o un resultado si se ha llegado a una solución. Dicha función tendrá dos parámetros de entrada: la tarea actual y la cota superior en curso, de modo que la función no genere tareas con cotas inferiores peores que dicha cota superior. El segundo parámetro `iniUB` del esqueleto será la cota superior de coste del problema, que típicamente se habrá obtenido mediante algún algoritmo devorador. El tercer y cuarto parámetros `lbf` y `ubf` extraen respectivamente la cota inferior de una tarea y la cota superior de un resultado. El último parámetro `iniTasks` es una lista de tareas iniciales, que típicamente será una lista unitaria que contenga la tarea raíz del árbol de búsqueda. Su especificación secuencial se muestra a continuación. Nótese que se utiliza una cola de prioridades con el objetivo de que en cada paso pueda extraerse la tarea más prometedora de las que se hayan generado hasta el momento:

```
bb :: ((task,Int) -> Either [task] result) -> Int ->
      (task -> Int) -> (result -> Int) -> [task] -> result
bb f iniUB lbf ubf iniTasks = last (bb' f iniUB lbf ubf tq)
  where tq = foldr insertPQueue emptyPQueue iniTasks

bb' :: ((task,Int) -> Either [task] result) -> Int ->
       (task -> Int) -> (result -> Int) -> (PQueue task) -> [result]
bb' f ub lbf ubf tq
  | (isEmptyPQueue tq) || (lbf promising > ub) = []
  | isLeft nextsOrResult = bb' f ub lbf ubf tq'
  | ubResult < ub        = result : bb' f ubResult lbf ubf tq1
  | otherwise            = bb' f ub lbf ubf tq1
 where promising     = minPQueue tq
       nextsOrResult = f (promising,ub)
       Left nexts    = nextsOrResult
       Right result  = nextsOrResult
       ubResult      = ubf result
       tq1           = elimMinPQueue tq
       tq'           = foldr insertPQueue tq1 nexts
isLeft (Left _ ) = True
```

```
isLeft (Right _) = False
```

Implementación

Un posible enfoque para paralelizarlo consistiría en dividir el espacio de búsqueda en muchos subárboles independientes, utilizar un `map` paralelo para resolver cada uno de ellos, y tras recoger todos los posibles resultados se elegiría el mejor de ellos. Cada trabajador debería seguir un algoritmo típico de ramificación y poda que busque la mejor solución en su subárbol.

Obviamente este enfoque es completamente ingenuo. En los algoritmos de ramificación y poda las soluciones que se van encontrando sirven como cota superior a la solución óptima, y se utilizan para podar ramas del árbol de búsqueda que se sabe que nunca obtendrán soluciones mejores. Ahora bien, si se implementara la opción anterior, las soluciones obtenidas en un trabajador sólo servirían para podar ramas en su propio subárbol, pero no en los de los demás trabajadores, ya que las tareas de un `map` paralelo son completamente independientes. La consecuencia sería que el algoritmo paralelo podaría muchas menos partes del árbol de búsqueda, es decir, realizaría mucho más cómputo que la versión secuencial. Este hecho ha sido confirmado en la práctica en [**KPR01**].

Una solución más inteligente debería permitir a todos los trabajadores conocer la mejor cota superior encontrada por cualquiera de los trabajadores hasta el momento. Hemos implementado esta idea modificando el esquema de trabajadores replicados que incluía estructuras de datos compartidas (`rwF`). La solución no será óptima, pues cada vez que se mejore la cota superior a utilizar, los trabajadores no podrán utilizarla con la tarea que esté en curso, sino que sólo la podrán utilizar con las siguientes tareas que reciban. A este esqueleto lo denominaremos `rwBB_naive`. Tendrá dos parámetros adicionales con respecto a `rwF`: `iniUB` será la cota superior inicial con la que empezará la evaluación; y `ubf` será una función que extraerá el coste a partir de un resultado. Existirá una lista `ubs` de cotas superiores que se realimentará desde la salida de los trabajadores a sus entradas mediante la función `flushUB`. Las modificaciones con respecto a `rwF` son las siguientes:

```
rwBB_naive :: (Transmissible a,Transmissible b,Transmissible fixed) =>
              Int -> Int -> fixed -> (fixed -> (a,Int) -> Maybe b) ->
              Int -> (b -> Int) -> [a] -> [b]
rwBB_naive nPE prefetch fixed fw iniUB ubf tasks =  results     where
  ...
  ubs = replicate (nPE * prefetch) iniUB ++
        flushUB ubf iniUB unordResult
  distribute :: Ord a => [a] -> [Int] -> [Int] -> [[(a,Int)]]
  distribute [] _ _                 = replicate nPE []
  distribute (t:ts) (i:is) (ub:ubs) = insert i (t,ub)
                                               (distribute ts is ubs)
```

```
flushUB :: (result -> Int) -> Int -> [ACK result] -> [Int]
flushUB ubf c (r:rs)  | c' < c    = c' : flushUB ubf c' rs
                      | otherwise = c  : flushUB ubf c  rs
                      where (ACK _ _ r') = r
                            c' = ubf r'
```

Una diferencia entre este enfoque y el algoritmo secuencial es que en el secuencial siempre se expande el nodo más prometedor del árbol de búsqueda, es decir, el que tiene la menor cota inferior de todos los nodos vivos del árbol. Ahora bien, en el algoritmo paralelo cada trabajador expandirá el nodo más prometedor de su subárbol. Nótese que esto es claramente peor que expandir los n nodos más prometedores del árbol completo, siendo n el número de trabajadores. Así pues, la versión paralela desperdiciará trabajo expandiendo nodos menos prometedores

La siguiente idea para solventar este problema consiste en introducir una cola *global* de tareas pendientes, y permitir que los trabajadores añadan tareas a dicha cola. De hecho, en la literatura (e.g. véase [Les93, Capítulo 10]) el esquema de trabajadores replicados es más general que los esqueletos que hemos presentado hasta el momento, ya que permite que los trabajadores generen nuevas tareas dinámicamente, de modo que se añadan a un almacen global. Este enfoque más general parece adecuado para resolver algoritmos de ramificación y poda, puesto que permite que el proceso principal envíe a los trabajadores las tareas ordenadas por sus cotas mínimas. Lógicamente, para obtener buenas aceleraciones, la granularidad de las tareas debería ser grande en comparación con las comunicaciones entre la cola global y los trabajadores. El esqueleto `rwBB` generaliza al anterior mediante estas ideas:

- Al igual que en `rwBB_naive`, los trabajadores reciben cada tarea unida a la mejor cota superior obtenida hasta el momento por el sistema completo. Por tanto, no debería generar ninguna tarea con una cota inferior igual o peor que la mejor cota superior.
- Por cada tarea recibida, cada trabajador puede devolver una solución junto con su coste, o una lista de nuevas tareas donde cada una llevará asociada su cota inferior.
- El proceso principal eliminará aquellas tareas cuya cota inferior supere o iguale el coste de la mejor solución obtenida hasta el momento.
- Cuando un trabajador termine una tarea, el proceso principal le asignará la tarea con la menor cota inferior de las tareas pendientes.
- El resultado del esqueleto será la mejor solución de todas las encontradas por los trabajadores.

Una primera versión de `rwBB` es la siguiente:

```
rwBB :: (Transmissible fixed,Transmissible task,Transmissible result,
         Ord task, Ord result) => Int -> Int -> fixed ->
         (fixed -> (task,Int) -> Either [task] result) -> Int ->
         (task -> Int) -> (result -> Int) -> [task] -> [result]
rwBB nPE pref fixed fw iniUB lbf ubf iniTasks =  minimum results
 where
  results      = (map (\(ACK _ (Right r)) -> r) . filter isResult)
                     unordResult
  unordResult  = merge # outsChildren
  outsChildren = [(worker fw i) # (fixed,inputs) |
                    (i,inputs) <- zip [0..nPE-1] inputss]
  inputss      = distribute tasks
                            (initReqs ++ (map owner unordResult)) ubs
  tasks        = iniTasks ++ unordTasks
  initReqs     = concat (replicate pref [0..nPE-1])
  unordTasks   = (concat . map (\(ACK _ (Left ts)) -> ts) .
                  filter isTask) unordResult
  ubs          = replicate (min n1 (np*pref)) iniUB ++
                 flushUB ubf iniUB unordResult
  n1           = length iniTasks

  distribute :: Ord a => [a] -> [Int] -> [Int] -> [[(a,Int)]]
  distribute (t:ts) (i:is) (ub:ubs)
    | lbf t >= ub = distribute ts (i:is) (ub:ubs)
    | otherwise   = insert i (t,ub) (distribute ts is ubs)

flushUB :: (result -> Int) -> Int ->
           [ACK (Either [task] result)] -> [Int]
flushUB ubf c (r:rs) | isResult r && c' < c = c' : flushUB ubf c' rs
                     | otherwise            = c  : flushUB ubf c  rs
                     where (ACK _ (Right r')) = r
                           c' = ubf r'

worker :: (Transmissible fixed,Transmissible task,Transmissible res)
          => (fixed -> (task,Int) -> res) -> Int ->
          Process (fixed,[(task,Int)])  [ACK res]
worker f i = process (fixed,ts) -> map f' ts
  where f' t = ACK i (f fixed t)
```

Los primeros cuatro parámetros tienen el mismo significado que en el esqueleto `rwF`, si bien la función a utilizar en los trabajadores ha cambiado: ahora el trabajador recibe una cota superior junto con cada tarea, y puede devolver tanto un resultado como una lista de nuevas tareas. El siguiente parámetro es la cota superior inicial (como en `rwBB_naive`). Las siguientes dos funciones sirven respectivamente para extraer la cota inferior de una tarea y para extraer el coste de una solución. El último parámetro es la lista de tareas iniciales, que en los algoritmos de ramificación y poda suele consistir en una única tarea que representa la raíz del árbol de búsqueda. La principal

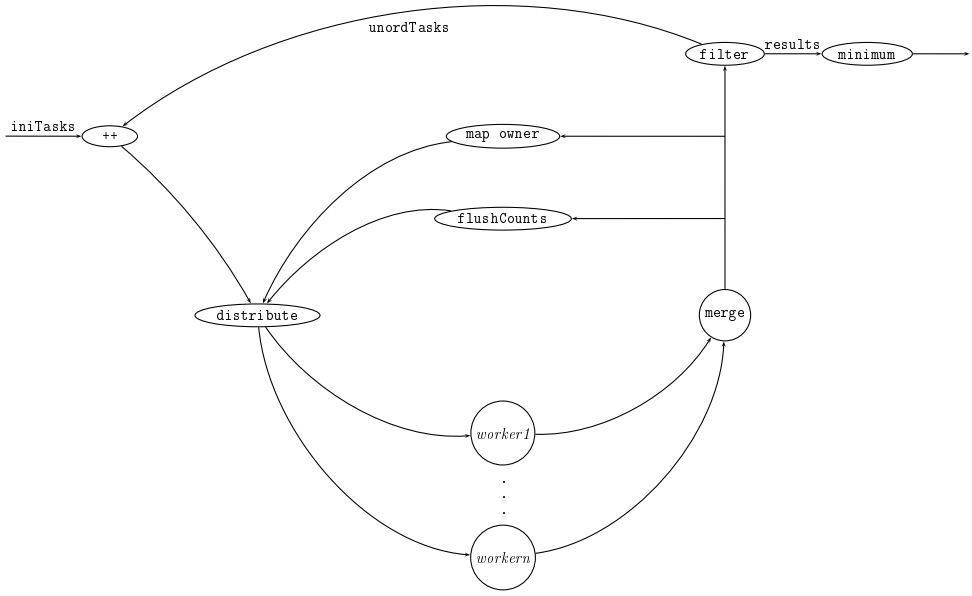


Figura 6.9: Flujo de datos del esqueleto `rwBB`

novedad con respecto a `rwBB_naive` es que ahora las tareas devueltas por los trabajadores se añaden a la lista de tareas iniciales.

Al tratar de aplicar este enfoque aparecen dos problemas:

1. Las tareas se envían desordenadas a los trabajadores, degradando terriblemente la eficiencia de la búsqueda.

2. Cuando no quedan más tareas disponibles, el algoritmo termina en un interbloqueo.

Por lo que respecta a la no terminación, dado que la función `distribute` en ningún momento cierra la lista de listas `inputss`, los trabajadores no podrán saber si aún van a recibir más tareas o no. Además, tampoco está claro dónde debería detectarse la terminación del algoritmo: podría ser tanto en la función `distribute` cuando todas las tareas se hayan consumido, o en `flushUB` cuando todos los trabajadores hayan devuelto sus respuestas a todas las tareas que se les asignaron. De hecho, la terminación en un entorno distribuido no sólo es un problema delicado en Eden, sino que también lo es en las versiones imperativas del esqueleto (véase [Les93, Capítulo 11]).

La solución al primer problema tampoco está nada clara en un entorno funcional: ¿cómo puede ordenarse, o calcularse el mínimo valor de una lista que está siendo producida por otros procesadores? Semánticamente equivale a ordenar una lista infinita o una lista con una cola indefinida. Operacionalmente, si un proceso trata de evaluar un valor aún no producido se quedará

suspendido en una clausura `queueMe` (véase la Sección 3.3.3), causando un interbloqueo.

La solución a los dos problemas es la misma: en el momento en que se aplica la función `distribute`, se cuentan el número nc de tareas consumidas y el número nd de tareas enviadas a trabajadores. Por su parte, al aplicar la función `flushUB` se cuentan el número np de tareas producidas por el algoritmo completo, así como el número na de respuestas recibidas desde los trabajadores. Inicialmente $np = n1$, siendo $n1$ el tamaño de la lista inicial de tareas. Cada vez que un trabajador devuelve un `ACK` con nuevas tareas, np se incrementa con el tamaño de dicha lista, y na se incrementa en una unidad. La relación $np \geq nc \geq nd \geq na$ es un invariante del algoritmo. Si la función `flushCounts` (la equivalente a `flushUB` en `rwBB_naive`) suministra a `distribute` los contadores np y na, entonces `distribute` dispondrá de los cuatro contadores.

Así, si $np > nc$ sabremos que pueden extraerse de forma segura $np - nc$ tareas de la lista `unordTasks`, sin riesgo de quedarse bloqueado. Esto resuelve el primero de los problemas, puesto que dicho segmento inicial puede introducirse en una cola de prioridades de la que se puede sacar trivialmente el menor elemento.

La condición de terminación es simplemente $(np = nc \wedge nd = na)$, es decir, todas las tareas producidas han sido enviadas a algún trabajador o eliminadas de la cola (por ser peores que la mejor solución encontrada hasta el momento), y todos los trabajadores han terminado de evaluar las tareas que tenían asignadas.

Los únicos detalles que quedarían por solventar son los relativos a disponer de suficientes mensajes `ACK` para poder distribuir las tareas pendientes. En `rw` cada vez que un trabajador terminaba una tarea y enviaba el correspondiente `ACK`, el proceso padre le enviaba una nueva a menos que no quedara ninguna. En este último caso ya se sabía que no se le enviaría ninguna más, y podía cerrarse la entrada al proceso. Ahora bien, en `rwBB` puede ser que cuando un trabajador envíe un `ACK` no haya ninguna tarea disponible para él, pero que posteriormente sí que la haya: dado que las tareas se generan dinámicamente, otro trabajador puede generar nuevas tareas después de que el primero enviara su `ACK`. Este problema se resuelve mediante las funciones productivas `expand` y `expandOneAck`, que guardan memoria de los `ACK` recibidos y no tratados.

En la Figura 6.9 se muestra el diagrama de flujo de la implementación. El código fuente es el siguiente, donde las definiciones no mostradas son las mismas que en la versión anterior:

```
rwBB nPE prefetch fixed fw iniUB lbf ubf iniTasks =  minimum results
 where
  ...
  inputss = distribute emptyPQueue 0 0 tasks
            (initReqs ++ (map owner unordResult)) counts
```

```
  counts = replicate (min n1 (nPE*prefetch)) (iniUB,n1,0,0)
           ++ expand moreACKs (flushCounts n1 0 ubf iniUB unordResult)
  moreAcks = max (np*prefetch - n1) 0

  distribute :: Ord a => PQueue -> Int -> Int ->
                [a] -> [Int] -> [(Int,Int,Int,Int)] -> [[(a,Int)]]
  distribute q nc nd ts w@(i:is) ((ub,np,na,nt):cs)
   | np > nc && lbf tmin > ub = if na == nd  then replicate nPE []
                                else distribute emptyPQueue np nd
                                                ts' w (expandOneACK cs)
   | np > nc              = insert i (tmin,ub)
                                (distribute q' (nc+1) (nd+1) ts' is cs)
   | np == nc && na == nd = replicate nPE []
   | np == nc && na < nd  = distribute q nc nd ts w (expandOneACK cs)
   where (ts1,ts') = splitAt (np - (nc + longPQueue q)) ts
         qaux      = foldr insertPQueue q ts1
         tmin      = minPQueue qaux
         q'        = elimMinPQueue qaux

flushCounts :: Int -> Int -> (result -> Int) -> Int ->
               [ACK (Either [task] result)] -> [(Int,Int,Int,Int)]
flushCounts np na ubf c (r:rs)
 | isTask r  = (c,np+nt,na+1,nt) : flushCounts (np+nt)(na+1) ubf c  rs
 | isResult r && c' < c
             = (c',np,na+1,0)    : flushCounts np     (na+1) ubf c' rs
 | otherwise = (c,np,na+1,0)     : flushCounts np     (na+1) ubf c  rs
 where (ACK _ (Right r')) = r
       (ACK _ (Left ts))  = r
       c' = ubf r'
       nt = length ts

expand 0 xs = xs
expand n (x@(_,_,_,nt):xs)
  | nt>1 && nt<=n+1 = (replicate nt x) ++ expand (n-nt+1) xs
  | nt>1            = (replicate (n+1) x) ++ xs
  | otherwise       = x:expand n xs
expandOneACK [] = []
expandOneACK (x@(a,b,c,nt):xs)
  | nt > 1    = (a,b,c,nt-1):(a,b,c,1):xs
  | otherwise = x:expandOneACK xs
```

Para poder utilizar un modelo de coste necesitaríamos saber el número de nodos que se exploran en el árbol de búsqueda en la versión paralela. Desgraciadamente esto no puede saberse *a priori*, puesto que en las distintas ejecuciones explorará un conjunto de nodos diferentes, ya que depende de las velocidades de cómputo de los distintos trabajadores. Es más, el número de nodos a explorar por la versión paralela puede ser incluso menor que en la versión secuencial, lo que puede conducir a aceleraciones superlineales: al tener que realizar menos cómputos totales, es posible que con p procesadores

se obtenga una aceleración superior a p.

Por tanto, no es posible determinar el tiempo paralelo de nuestros programas. En el supuesto de que fuéramos capaces de calcular el número de nodos que se expanden, podríamos utilizar un modelo similar al de `rwF`, con algunas pequeñas modificaciones para reflejar más exactamente las características de `rwBB`:

$$\begin{aligned} t_{rwBB} &= L_{init} + t_{worker} + L_{final} \\ L_{init} &= P(t_{create} + t_{packF} + t_{packI} + t_{distribute_1}) + \delta \\ L_{final} &= \delta + t_{unpackO} + t_{minimum} \\ t_{worker} &= t_{\#} + t_{unpackF} + \frac{N}{P}(t_{unpackI} + t_{comp} + t_{packO}) \\ t_{comp} &= \frac{1}{N}\sum_{i=1}^{N} t_{f_i} \end{aligned}$$

donde suponemos que el proceso principal no comparte procesador con ningún trabajador, y que sus cómputos no suponen un cuello de botella a la hora de distribuir el trabajo.

6.4.3 Tuberías de procesos

Una descomposición paralela muy conocida es la *tubería de procesos* cuya idea es similar a la del trabajo en una cadena de montaje. Cada etapa de la tubería recibe una secuencia de datos procesados en la etapa anterior, aplica a cada uno una función fija, y los envía a la etapa siguiente. Las funciones de cada etapa son normalmente diferentes entre sí. El paralelismo se consigue cuando, una vez "cargadas" todas las etapas, cada una trabaja sobre un dato distinto de la lista. La especificación en Haskell es un caso particular de la función de orden superior *foldl*: `pipe` recibe una lista no vacía de funciones y devuelve su composición funcional. Es necesario alterar el orden habitual del operador de composición `(.)` mediante `flip` para que las funciones se apliquen de izquierda a derecha.

```
pipe :: [[a] -> [a]] -> [a] -> [a]
pipe  = foldl1 (flip (.))
```

Nótese que cada función recibe una lista de valores y produce otra lista, pues si las etapas de la tubería no trabajasen sobre listas no sería posible extraer paralelismo alguno.

Implementación

La implementación más inmediata consiste en crear un proceso por cada una de las funciones de la lista:

```
pipe :: Transmissible a => [[a]->[a]] -> [a] -> [a]
pipe [f] xs = process xs -> f xs
pipe fs xs  = (ppipe fs) # xs
```

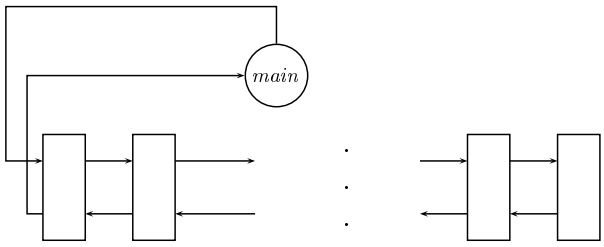


Figura 6.10: Topología generada con `pipe`

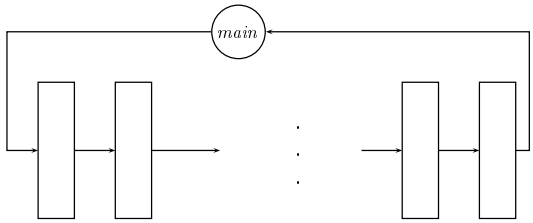


Figura 6.11: Topología generada con `pipeD`

```
ppipe :: Transmissible a => [[a]->[a]] -> Process [a] [a]
ppipe [f]    = process xs -> f xs
ppipe (f:fs) = process xs -> (ppipe fs) # (f xs)
```

Dado que las conexiones entre procesos en Eden se establecen entre el proceso creador y sus procesos hijo, esta definición no consigue la topología de tubería sino una en la que el último proceso envía su resultado al penúltimo, éste a su vez al anterior, y así hasta llegar al proceso-padre (véase la Figura 6.10). Este problema es un ejemplo de lo que en [KPS00] se denomina *bypassing between generations*. Sería deseable que el compilador fuera capaz de detectar y corregir automáticamente este problema, generándose la topología deseada (véase la Figura 6.11), pero desgraciadamente aún no es posible. Ahora bien, el programador también puede corregir el problema utilizando los canales dinámicos descritos en la Sección 3.1.2):

- El proceso padre debe crear un canal dinámico cuyo nombre será transmitido a través de la tubería hasta el último proceso de la misma. El resultado global del esqueleto será el valor que se reciba por dicho canal.

- El último proceso enviará su resultado a través del canal dinámico, por

lo que será recibido directamente por el proceso-padre.

Nótese que para llevar a cabo este diseño es preciso modificar las abstracciones de proceso para que dispongan de un canal extra de entrada, por el que se recibirá el nombre del canal que ha creado el padre. El código fuente es el siguiente:

```
pipeD :: Transmissible a => [[a]->[a]] -> [a] -> [a]
pipeD [f] xs = process xs -> f xs
pipeD fs xs = new (cn,c) let dummy = (ppipeD fs) # (xs,cn) in c

ppipeD :: Transmissible a => [[a]->[a]] -> Process ([a],ChanName [a]) ()
ppipeD [f]    = process (xs,cn) -> cn !* (f xs) par ()
ppipeD (f:fs) = process (xs,cn) -> (ppipeD fs) # (f xs,cn)
```

Como puede apreciarse, las similitudes entre ambos programas son notables. De hecho, esta segunda versión puede derivarse a partir de la primera utilizando la metodología que se propone en la Sección 6.6.

El modelo de coste para esta implementación es el siguiente:

$$\begin{aligned} t_{pipe\_naive} &= L_{init} + t_{worker} + L_{final} \\ L_{init} &= F(t_{create} + t_{\#} + t_{packI} + \delta) \\ L_{final} &= \delta + t_{unpackO} \\ t_{worker} &= \lceil \tfrac{F}{P} \rceil N(t_{unpackI} + max\{t_{comp_i}\}_{i=1}^{F} + t_{packO}) \end{aligned}$$

siendo F el número de funciones de la tubería, N el número de datos de la lista de entrada y t_{comp_i} el coste de la función de la etapa i aplicada a un dato de la lista. Suponemos $F > P$ y el modo *round-robin* en el RTS. Nótese el uso de la función max en el modelo. Expresa el hecho bien conocido de que la velocidad de proceso de una tubería viene determinada por el procesador más lento (es decir, por la etapa más costosa).

6.5 Esqueletos sistólicos

Los algoritmos paralelos llamados *sistólicos* se caracterizan porque los procesos que los componen alternan un paso de cómputo paralelo con una sincronización global entre todos los procesos del programa, a semejanza de la alternancia entre *sístole* y diástole que sucede en los corazones de los mamíferos. Los esqueletos que se presentan en esta sección responden a este esquema. En ellos, cada proceso reside en un procesador distinto, con lo que la topología de comunicación entre procesos debe entenderse también como la topología de comunicación entre procesadores. No damos para ellos una especificación en términos de un programa secuencial ya que estas topologías sólo tienen sentido como programas paralelos. De todos modos, si se deseara disponer de una versión secuencial, en Eden siempre puede conseguirse una a partir de cualquier programa paralelo simplemente reemplazando las

abstracciones de proceso `process x -> e` por lambda-abstracciones `\ x -> e` y los lanzamientos de proceso `e1 # e2` por aplicaciones `e1 e2`. De hecho, existe un modo del compilador de Eden que realiza esta transformación automáticamente, como se vio en la Sección 4.5

6.5.1 Método iterativo

Esta topología consiste en un proceso gestor y un conjunto de procesos trabajadores idénticos. Tanto uno como otros mantienen un estado local propio. En la fase de comunicación el gestor recibe resultados de los trabajadores, los procesa, y envía a cada uno una nueva tarea. En el paso de cómputo cada trabajador realiza en paralelo su tarea. La diferencia con el esqueleto de los trabajadores replicados de la Sección 6.3.1 es que ahora todos los trabajadores han de sincronizarse a la vez con el gestor tras cada tarea, pero la topología de procesos es la misma de la Figura 6.4. El esqueleto es apropiado para algoritmos que exigen iterar un cálculo hasta que se alcanza una condición de convergencia. Ejemplos típicos son la resolución de sistemas lineales de ecuaciones o la resolución de ecuaciones diferenciales.

El gestor se inicializa con un dato de tipo `inp` (el problema de entrada) y con el estado local del gestor, de tipo `ml`. Cada trabajador se inicializa con un dato de tipo `wl` (estado local del trabajador) y con una tarea de tipo `t`. En cada iteración, cada trabajador computa un subresultado de tipo `sr`, así como un nuevo estado local que se utilizará en la siguiente iteración. Al gestor sólo se le comunica el subresultado, pero no el nuevo estado local. El gestor combina los subresultados y produce o bien un nuevo conjunto de tareas y un nuevo estado local, o bien un resultado global, caso en el cual el cómputo finaliza. Los parámetros del esqueleto son los siguientes:

- Una función `split` que usará el gestor para obtener el estado inicial y la tarea inicial de cada trabajador, así como el propio estado local del gestor. El primer parámetro de esta función es un entero que indica en cuántas partes debe dividirse el trabajo, y que dependerá del número de procesadores disponibles.
- Una función `wf` que se utiliza por los trabajadores, y que dados un estado local y una tarea devolverá un subresultado y el nuevo estado local a emplear en la siguiente iteración.
- Una función `comb` que usará el gestor para combinar los subresultados producidos por los trabajadores. Dependiendo de si el cómputo global ha finalizado o no, devolverá el resultado final o el nuevo conjunto de tareas y el nuevo estado local.
- El problema de entrada, de tipo `inp`.

El código fuente Eden es el siguiente:

```
iterUntil :: (Transmissible wl, Transmissible t,
              Transmissible sr) =>
              (inp -> Int -> ([wl],[t],ml)) ->     -- split function
              (wl -> t -> (sr, wl)) ->             -- worker function
              (ml -> [sr] -> Either r ([t],ml)) -> -- combine function
              inp -> r

iterUntil split wf comb x = result    where
  (result, moretaskss)  = manager comb ml (transpose' srss)
  srss                  = map_naive (worker wf) (zip wlocals taskss)
  taskss                = transpose' (initials : moretaskss)
  (wlocals,initials,ml) = split x noPe

manager :: (ml->[sr]->Either r ([t],ml)) -> ml -> [[sr]] -> (r,[[t]])
manager comb ml (srs : srss) = case comb ml srs of
            Left  res      -> (res, [])
            Right (ts,ml') -> let (res',tss) = manager comb ml' srss
                              in  (res',ts:tss)

worker ::  (wl -> t -> (sr, wl)) -> (wl, [t]) -> [sr]
worker wf (local, [])  = []
worker wf (local,t:ts) = sr : worker wf (local',ts)
    where (sr, local') = wf local t

transpose' = foldr (mzipWith' (:)) (repeat [])
mzipWith' f [] _           = []
mzipWith' f (x:xs) ~(y:ys) = f x y : mzipWith' f xs ys
```

Nótese que para obtener una versión secuencial bastaría con utilizar `map` en vez de `map_naive`. Al disponer de sólo un procesador, la función `split` no dividiría el problema en subproblemas, y el algoritmo se comportaría como se esperaría de una versión pensada para ejecución secuencial.

El modelo de coste de la implementación es el siguiente:

$$\begin{aligned}
t_{iterUntil} &= L_{init} + I\ t_{worker} + (I-1)t_{parent} + L_{final} \\
L_{init} &= t_{split} + P(t_{create} + t_{packI}) + \delta \\
L_{final} &= P\ t_{unpackO} + t_{combine} \\
t_{parent} &= P\ t_{unpackO} + t_{combine} + P\ t_{packI} + \delta \\
t_{worker} &= t_{unpackI} + t_{compW} + t_{packO} + \delta
\end{aligned}$$

donde I es el número de iteraciones del algoritmo. Nótese que ahora los costes de los trabajadores y del gestor se alternan estrictamente en el camino crítico, por lo que debemos sumar los tiempos de cómputo de ambos en el tiempo total $t_{iterUntil}$.

6.5.2 Toroide

El toroide (Figura 6.12) es una topología bien conocida en la que cada proceso está conectado con cuatro vecinos. Se puede entender como una

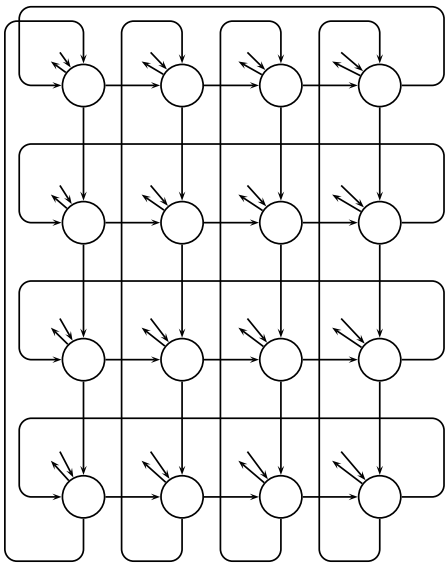


Figura 6.12: Ejemplo de topología de procesos en toroide.

matriz bidimensional de procesos en la que se consideran también vecinos al primero y al último de cada fila y al primero y al último de cada columna. Adicionalmente, existe un proceso padre que envía datos iniciales a todos los procesos del toroide y recibe los resultados finales de todos ellos. Una vez recibidos y procesados los datos iniciales, se sucede una secuencia en la que se alterna un paso de comunicación y un paso de cómputo. En el paso de comunicación todo proceso del toroide envía datos hacia sus vecinos del Sur y del Este y los recibe de sus vecinos del Norte y del Oeste.

En la implementación, la función `torus` es la responsable de crear la topología toroidal creando las dependencias de datos adecuadas entre las entradas y salidas de los distintos procesos `ptorus`. Para ello, las entradas a los procesos se obtienen desplazando circularmente las salidas de los mismos. Así, para las comunicaciones horizontales basta con indicar que la entrada del primer proceso corresponde a la salida del último proceso de la fila, de modo que la entrada del segundo corresponda a la salida del primero, la entrada del tercero a la salida del segundo y así sucesivamente.

El primer parámetro del esqueleto será el tamaño que se desea que tenga el toroide (típicamente será $\lfloor\sqrt{\texttt{noPe}}\rfloor$), el segundo proporciona el método para repartir los datos de entrada entre los distintos procesos, mientras que el tercero especifica cómo combinar los datos enviados al proceso padre para obtener el resultado global del esqueleto. El cuarto parámetro será la función a utilizar en cada trabajador, y el último será el problema de entrada.

Cada `ptorus` recibe una entrada de tipo `c` del padre, otra de tipo `[a]`

de su vecino del Norte y otra de tipo `[b]` de su vecino del Este. Forzamos a que las comunicaciones con los vecinos se realicen mediante listas para que el esquema sea realmente sistólico:

```
-- the torus topology
torus :: (Transmissible a,Transmissible b,
         Transmissible c,Transmissible d) =>
         Int -> (Int -> c -> [[c]]) -> ([[d]] -> d) ->
         ((c,[a],[b]) -> (d,[a],[b])) -> c -> d
torus size dist comb f input = comb outssToParent   where
 toChildren = dist size input
 outss   =  [[(ptorus f) # outAB | outAB <- outs'] | outs' <- outss']
 (outssToParent, outssA, outssB) = unzip3 (map unzip3 outss)
 outssA' = mzipWith (:) nrows (map last outssA) (map init outssA)
 outssB' = last outssB : init outssB
 outss'  = mzipWith3 mzip3 toChildren outssA' outssB'
 nrows   = length toChildren

-- each individual process of the torus
ptorus :: (Transmissible a,Transmissible b,
           Transmissible c,Transmissible d) =>
           ((c,[a],[b])->(d,[a],[b])) -> Process (c,[a],[b]) (d,[a],[b])
ptorus f = process (fromParent,inA,inB) -> f (fromParent,inA,inB)

mzip3 (x:xs) ~(y:ys) ~(z:zs) = (x,y,z) : mzip3 xs ys zs
mzip3 _ _ _ = []

mzipWith3 f (x:xs) ~(y:ys) ~(z:zs) = f x y z : mzipWith3 f xs ys zs
mzipWith3 _ _ _ _ = []

mzipWith f 0 _ _ = []
mzipWith f n ~(x:xs) ~(y:ys) = (f x y) : mzipWith f (n-1) xs ys
```

Las sucesivas barreras de sincronización vienen expresadas por las dependencias de datos de la función `f`, la cual ha de producir el primer elemento de las listas de salida antes de consumir el primer elemento de las listas de entrada, pues de lo contrario la topología podría bloquearse; ha de producir el segundo tras haber consumido el primero; y así sucesivamente. Nótese que las funciones `mzip3`, `mzipWith` y `mzipWith3` son simplemente versiones más perezosas de la familia de `zip`. Esta pereza es necesaria para poder romper las dependencias circulares que aparecen con las definiciones mutuamente recursivas que conectan los distintos procesos.

Al igual que ocurría con las tuberías de procesos, la topología creada con el programa anterior no es la deseada, pues ninguna de las comunicaciones se producirán directamente entre vecinos, sino que se realizarán a través del proceso padre, que se convertirá en un cuello de botella. El motivo vuelve a ser el hecho de que sólo se generan topologías jerárquicas de procesos. En este caso estaríamos interesados en realizar una conexión directa entre hermanos

(denominada *bypassing between siblings* en [KPS00]), que el compilador no es capaz de realizar por sí sólo. Para establecer conexiones directas entre los procesos del toroide sin involucrar al proceso padre en ellas, utilizamos de nuevo los canales dinámicos de Eden.

La idea principal vuelve a ser que por cada canal que queramos conectar directamente deberemos crear un nuevo canal dinámico. El proceso lector del canal original debería ser quien creara el canal dinámico, y quien enviara el nombre del canal al proceso que deba escribir a través de él. Para poder enviar dichos nombres de canales necesitaremos que sea el proceso padre quien reenvíe adecuadamente los nombres de canales a los destinatarios adecuados. Nótese que, dado que queremos que los receptores de datos envíen mensajes (nombres de canales) a los emisores de datos, lo que realmente hay que hacer es crear la topología inversa a la que teníamos originalmente. Para ello habrá que modificar los procesos que forman el toroide, siguiendo estos pasos:

- Los canales de salida que se utilizaban para enviar valores a los vecinos se sustituyen por dos canales de entrada adicionales. Por dichos canales se recibirán los nombres de los canales dinámicos que se utilizarán para enviar valores a los vecinos.
- En lugar de utilizar los canales de salida antiguos, se enviarán los datos a través de los canales dinámicos que se reciban por los canales de entrada.
- Se eliminan los canales de entrada que se utilizaban para recibir valores de los vecinos. En su lugar, deben crearse localmente dos nuevos canales dinámicos, y sus nombres deben enviarse a través de los dos nuevos canales de salida.
- En lugar de utilizar los valores que se recibían originalmente por los canales de entrada, se utilizan los valores de los canales dinámicos que se han creado localmente.

Así pues, ahora cada proceso crea dos canales por los que espera recibir los mensajes de sus vecinos del Norte y del Oeste, y recibe otros dos canales por los que envía sus resultados a sus vecinos del Sur y del Este:

```
ptorus :: (Transmissible a,Transmissible b,
           Transmissible c,Transmissible d) =>
           ((c,[a],[b]) -> (d,[a],[b])) ->
           Process (c,ChanName [a],ChanName [b])
                   (d,ChanName [a],ChanName [b])
ptorus f = process (fromParent,outChanA,outChanB) -> out
 where out = new (inChanA, inA) new (inChanB, inB)
             let (toParent,outA,outB) = f (fromParent,inA,inB)
```

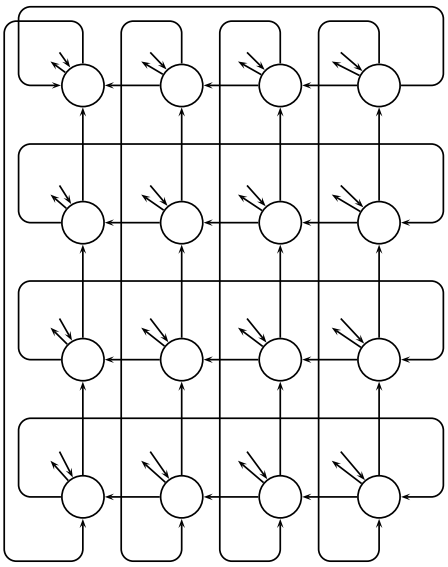


Figura 6.13: Toroide con comunicaciones invertidas.

```
in outChanA !* outA par outChanB !* outB par
   (toParent,inChanA,inChanB)
```

Nótese que aún no ha sido necesario modificar la función `torus` que se encarga de establecer las conexiones entre los procesos. Si no modificamos la función que genera la topología de conexión, lo que obtendremos es lo que se aprecia en la Figura 6.13, donde las comunicaciones se realizan en el sentido contrario. Ahora bien, desde el punto de vista del diseño de un toroide resulta completamente irrelevante en qué sentido se transmitan los datos. Así pues, sin necesidad de modificar la función `torus` hemos obtenido el toroide deseado: puesto que teníamos que invertir la topología original, y dicha topología era circular, la topología inversa es equivalente a la original.

De todas formas, si realmente quisiéramos obtener exactamente la misma topología tendríamos que modificar `torus` para invertir el orden en el que se realizan las comunicaciones. Lo único que habrá que cambiar es la forma en la que se realiza el desplazamiento de las salidas de los procesos antes de suministrarlas como entradas a los otros procesos. Así, bastará con modificar las definiciones de `outssB'` y `outssA'` para obtener exactamente la topología de la Figura 6.12:

```
outssB' = tail outssB ++ [head outssB]
outssA' = mzipWith (++) nrows (map tail outssA)
                              (map ((:[]).head) outssA)
```

Nótese que simplemente se han desplazado las salidas hacia la izquierda y

hacia arriba en vez de hacia la derecha y hacia abajo. En el caso general, serían necesarias reestructuraciones más significativas del código, pero afortunadamente la mayoría de las topologías interesantes (anillos, toroides, n-cubos, etc.) son circulares, por lo que no hace falta ninguna modificación. Además, la mayoría de las que no son circulares son muy regulares, como por ejemplo las rejillas, por lo que es sencillo invertirlas.

El modelo de coste del toroide es el siguiente:

$$\begin{array}{lcl} t_{torus} & = & L_{init} + t_{worker} + L_{final} \\ L_{init} & = & t_{dist} + P(t_{create} + t_{packC}) + \delta \\ L_{final} & = & \delta + P\ t_{unpackD} + t_{comb} \\ t_{worker} & = & t_{\#} + t_{unpackC} + t_{comp} + \\ & & N(t_{packA} + t_{packB} + t_{unpackA} + t_{unpackB} + t_{comp}) \\ & & + t_{packD} \end{array}$$

donde las letras A, B, C y D denotan el tipo de los mensajes transmitidos, es decir, A y B se usan para los mensajes que se transmiten entre hermanos, mientras C y D se usan respectivamente para los mensajes del padre a los hijos y de los hijos al padre. N es el número de pasos de cómputo, es decir la longitud de las listas `[a]` y `[b]`, y t_{comp} es el coste de un paso de cómputo. Se supone que dicho paso cuesta los mismo en todos los hijos. Asimismo, se supone que cada proceso tiene un procesador dedicado, incluido el proceso padre. Por tanto, el número de procesadores ha de ser $m \times n + 1$ siendo $m \times n$ las dimensiones del toroide.

6.5.3 Anillo

Un anillo (unidireccional) es un caso particular del toroide, donde cada proceso está conectado sólo con sus vecinos de la izquierda y de la derecha, además de con el proceso padre (véase la Figura 6.14). Al igual que en el caso del toroide, para crear la topología de procesos basta con utilizar definiciones mutuamente recursivas que establezcan las dependencias de datos adecuadas, de modo que las entradas de los procesos coincidan con las salidas de los mismos pero desplazadas.

```
-- the ring topology
ring :: (Transmissible a,Transmissible b,Transmissible c) =>
         Int -> (Int -> b -> [b]) -> ([c] -> c) ->
         ((b,[a]) -> (c,[a])) -> b -> c
ring n dist comb f input = comb toParent  where
  toChildren       = dist n input
  outs             =  [(pring f) # outA' | outA' <- outs']
  (toParent,outsA) = unzip outs
  outsA'           = last outsA : init outsA
  outs'            = mzip2 toChildren outsA'
```

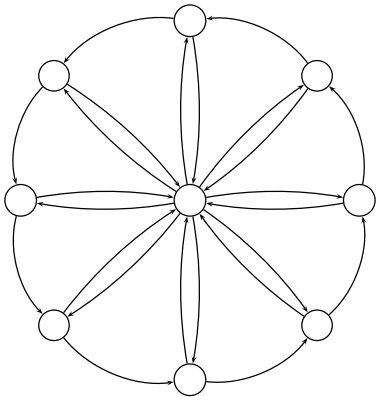


Figura 6.14: Ejemplo de topología de procesos en anillo.

```
-- each individual process of the ring
pring ::(Transmissible a,Transmissible b,Transmissible c) =>
         ((b,[a]) -> (c,[a])) -> Process (b,[a]) (c,[a])
pring f = process (fromParent, inA) -> out
  where out = f (fromParent, inA)

mzip2 (x:xs) ~(y:ys) = (x,y) : mzip xs ys
mzip2 _ _  = []
```

Pero, al igual que en el caso del toroide, la topología que se creará no será adecuada, puesto que todas las comunicaciones se realizarán a través del proceso padre. La solución vuelve a ser utilizar canales dinámicos, siguiendo los mismos pasos que con el toroide. Dado que el anillo es también una topología circular, no hará falta redefinir la función que crea la topología de procesos, sino sólo la definición de las abstracciones de procesos, sustituyendo los canales normales por los dinámicos, siguiendo los mismos pasos que en el toroide:

```
pring ::(Transmissible a,Transmissible b,Transmissible c) =>
         ((b,[a]) -> (c,[a])) ->
         Process (b,ChanName [a]) (c,ChanName [a])
pring f = process (fromParent,outChanA) -> out
 where out= new (inChanA, inA)
        let (toParent,outA) = f (fromParent,inA)
        in outChanA !* outA par (toParent,inChanA)
```

Al igual que con el toroide, si realmente quisiéramos que las comunicaciones se efectuaran en el mismo sentido que en la versión sin canales dinámicos

tendríamos que invertir las conexiones, lo cual puede hacerse trivialmente modificando en `ring` la definición de `outsA'` :

```
outsA' = tail outsA ++ [head outsA]
```

El modelo de coste es el siguiente:

$$\begin{array}{lcl} t_{ring} & = & L_{init} + t_{worker} + L_{final} \\ L_{init} & = & P(t_{create} + t_{packB}) + t_{dist} + \delta \\ L_{final} & = & \delta + P\ t_{unpackC} + t_{comb} \\ t_{worker} & = & t_{\#} + t_{unpackB} + t_{comp} + N(t_{packA} + t_{unpackA} + t_{comp}) + t_{packC} \end{array}$$

siendo P el número de trabajadores (cada uno en un procesador diferente), N el número de etapas sistólicas del sistema, y t_{comp} el tiempo que emplea cada trabajador en realizar una etapa. Al igual que en el caso del toroide, las letras A, B y C denotan el tipo de los mensajes transmitidos, usándose A para los mensajes entre hermanos, B para los mensajes del padre a los hijos y C de los hijos al padre.

6.5.4 Rejilla

Una rejilla es una topología bidimensional en la que cada proceso está conectado con sus cuatro vecinos. La diferencia con el toroide es que en la rejilla no se consideran vecinos al primero y al último proceso de una fila o columna. Además, los procesos no tienen conexiones adicionales para conectarse con el proceso principal: sólo los nodos de la primera fila o de la primera columna reciben entradas del padre, y sólo los de la última fila o columna envían datos al padre (véase la Figura 6.15), por lo que el proceso padre puede considerarse vecino de todos ellos. Nótese que una rejilla puede verse también como una tubería bidimensional.

Al igual que el toroide y el anillo, las rejillas también suelen usarse en algoritmos sistólicos. En cada paso, cada proceso trabajador recibe datos de sus vecinos del Oeste y del Norte, realiza un cómputo y posteriormente envía datos a sus vecinos del Este y del Sur. Como en los demás algoritmos sistólicos, en lugar de barreras de sincronizacion se utilizan listas para simular las distintas etapas del cómputo. La función `grid` es la encargada de crear las dependencias adecuadas para formar la rejilla que conecte los distintos procesos `pgrid`. Cada `pgrid` recibe dos entradas provenientes de sus vecinos, y produce como salida otras dos. Visto desde el exterior, las entradas a la rejilla son los datos que debe suministrar el proceso principal a los trabajadores de la primera fila y de la primera columna, y las salidas del sistema son las salidas de la última fila y de la última columna. La función que deben aplicar los procesos será un parámetro del esqueleto. El código Eden completo es el siguiente:

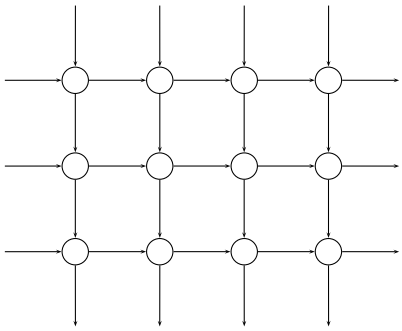


Figura 6.15: Ejemplo de una topología en rejillla.

```
-- the grid topology
grid :: (Transmissible a,Transmissible b) =>
         (([a],[b])->([a],[b])) -> ([[a]],[[b]]) -> ([[a]],[[b]])
grid f (insA,insB) = (outsA,outsB)   where
  outss           = [[(pgrid f) # outAB | outAB <- outs']
                     | outs' <- outss'] `using` (spine.concat)
  (outssA,outssB) = unzip (map unzip outss)
  outssA'         = mzipWith (:) insA (map init outssA)
  outssB'         = insB : init outssB
  outss'          = zipWith zip outssA' outssB'
  outsA           = map last outssA
  outsB           = last outssB

-- each individual process of the torus
pgrid :: (Transmissible a, Transmissible b) =>
         (([a],[b])->([a],[b])) -> Process ([a],[b]) ([a],[b])
pgrid f = process (inA,inB) -> (outA,outB)
          where (outA,outB) = f (inA,inB)
```

Donde la estrategia `spine.concat` se utiliza para demandar la salida de la lista de listas de procesos.

Al igual que sucedía con las topologías anteriores, sigue siendo necesario realizar una conexión directa entre hermanos, pues de lo contrario todas las comunicaciones se realizarán a través del proceso padre. Así pues, debemos utilizar canales dinámicos para establecer realmente la topología deseada. A diferencia de los dos casos anteriores, ahora la topología no es circular, pues el proceso principal sólo manda datos a los primeros procesos, y sólo los recibe de los últimos. Por tanto, ahora no bastará con modificar la definición de la abstracción de proceso, sino que también será preciso reescribir la función `grid` que conecta los procesos. Por lo que respecta a la abstracción `pgrid`, la metodología de modificación es exactamente la misma que para el toroide, de modo que es trivial obtener el código que se muestra a continuación. Nótese que cada `pgrid` recibe dos nombres de canales dinámicos que utiliza para

enviar datos directamente a sus vecinos, y que crea internamente dos nuevos canales dinámicos y envía sus nombres para que los vecinos puedan mandarle datos a través de ellos.

```
pgrid :: (Transmissible a, Transmissible b) =>
         (([a],[b])->([a],[b])) -> Process (ChanName [a],ChanName [b])
                                           (ChanName [a],ChanName [b])
pgrid f = process (outChanA,outChanB) -> nothing
 where nothing = new (inChanA,inA) new (inChanB,inB)
                 let (outA,outB) = f (inA,inB)
                 in  outChanA !* outA par
                     outChanB !* outB par
                     (inChanA,inChanB)
```

Para modificar `grid` debemos adaptar las comunicaciones del proceso padre a los nuevos procesos. Dado que todas las comunicaciones de los hijos se realizan mediante canales dinámicos, el padre también deberá comunicarse con ellos del mismo modo. Para ello seguirá los siguientes pasos:

- Deben crearse tantos nuevos canales dinámicos como conexiones directas se necesiten para recibir datos de los hijos.
- En lugar de utilizar los valores que se recibían de los hijos, se utilizarán los nuevos canales creados.
- En lugar de enviar datos a los hijos, se enviarán los nombres de los nuevos canales dinámicos.
- Los datos que el padre debía enviar a los hijos se enviarán a través de los nombres de canales dinámicos que los hijos le habrán enviado.

El código fuente completo se incluye a continuación. Nótese que la estructura básica sigue siendo la misma salvo renombramientos, si bien es preciso añadir una cantidad de código considerable para que todos los detalles de la implementación se cubran adecuadamente:

```
grid :: (Transmissible a,Transmissible b) =>
         (([a],[b])->([a],[b])) -> ([[a]],[[b]]) -> ([[a]],[[b]])
grid f (insA,insB) = everything   where
  nr = length insA  -- number of rows of the grid
  nc = length insB  -- number of columns of the grid

  outss           = [[(pgrid f) # outAB | outAB <- outs']
                     | outs' <- outss'] `using` (spine.concat)
  (outssA,outssB) = unzip (map unzip outss)
  outssA'         = mzipWith (:) outsAToParent (map init outssA)
  outssB'         = outsBToParent : init outssB
  outss'          = zipWith zip outssA' outssB'
```

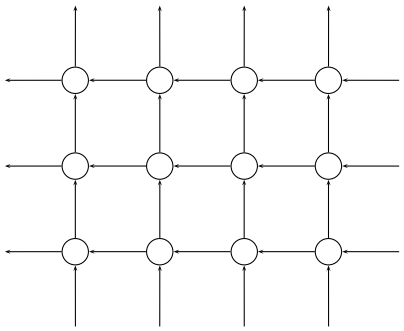


Figura 6.16: Rejillla con comunicaciones invertidas.

```
-- The parent creates new channels to receive values from
-- the last column and the last row children of the grid
channelsA             = generateChannels nr
channelsB             = generateChannels nc
(outsAToParent,outsA) = unzip channelsA
(outsBToParent,outsB) = unzip channelsB

-- The parent sends values to the first column and first row
-- of the grid, and waits until receiving values from the
-- last column and last row
everything = sendInitAs
sendInitAs = sendValues (zip insB (last outssB)) sendInitBs
sendInitBs = sendValues (zip insA (map last outssA)) (outsA,outsB)
```

donde las funciones `generateChannels` y `sendValues` son funciones generales que pueden utilizarse para definir cualquier otra topología no simétrica:

```
-- Generates and returns a list of dynamic channels
generateChannels :: Transmissible a => Int -> [(ChanName a, a)]
generateChannels 0 = []
generateChannels n = new (cn,c) ((cn,c) : generateChannels (n-1))

-- Sends a list of values through their associated dynamic
-- channels, and continues evaluating expresion e
sendValues :: Transmissible a => [(a, ChanName a)] -> b -> b
sendValues [] e            = e
sendValues ((v,ch):more) e = ch !* v par sendValues more e
```

Al igual que ocurría en el caso del toroide, al no haber modificado la estructura básica de interconexión obtendremos la topología de comunicaciones inversa (véase Figura 6.16) a la original. Como en los ejemplos anteriores, para obtener la misma topología deberíamos cambiar el sentido en el que se

desplazan las salidas de los procesos. En el caso de la rejilla esto implica modificar las siguientes definiciones:

```
outssA' = mzipWith (++) (map tail outssA) (map (:[]) outsAToParent)
outssB' = tail outssB ++ [outsBToParent]

sendInitAs = sendValues (zip insB (head outssB)) sendInitBs
sendInitBs = sendValues (zip insA (map head outssA)) (outsA,outsB)
```

Dado que una rejilla es una tubería bidimensional, el modelo de coste tendrá un gran parecido con el de la tubería, con la diferencia de que ahora es el proceso padre el encargado de crear todos los procesos de la topología. La latencia inicial tendrá que contar todo el tiempo que transcurra hasta que todos los procesos hayan recibido sus datos de entrada. Suponiendo que hay n filas y m columnas, para que el proceso de la última fila y columna reciba un dato habrán tenido que realizarse $n + m - 2$ pasos de cómputo. Suponiendo que se realizan I iteraciones, el modelo sería el siguiente:

$$\begin{aligned}
t_{grid} &= L_{init} + t_{worker} + L_{final} \\
L_{init} &= (n+m)t_{create} + n\ t_{packA} + m\ t_{packB} + \\
&\quad (n+m-2)(t_{unpackA} + t_{unpackB} + t_{comp} + t_{packA} + t_{packB} + \delta) \\
L_{final} &= \delta + t_{unpackA} + t_{unpackB} \\
t_{worker} &= I(t_{unpackA} + t_{unpackB} + t_{comp} + t_{packA} + t_{packB})
\end{aligned}$$

6.6 Derivación de topologías no jerárquicas

Como se ha podido apreciar en varios de los esqueletos anteriores, en muchas situaciones el simple empleo de abstracciones y concreciones de procesos no permite especificar la topología de comunicaciones deseada. Para poder implementar topologías no jerárquicas siempre ha sido necesario el uso de canales dinámicos. Ahora bien, también se ha visto que existía una gran similitud entre las especificaciones que no usaban canales dinámicos y las implementaciones reales que sí los usaban. En la presente sección se resume el método a seguir para derivar una topología no jerárquica a partir de una especificación que sólo haga uso de procesos, también presentado en [**PRS01**]. Dependiendo del tipo de conexión directa a resolver (véase [KPS00]) aplicaremos el método correspondiente.

6.6.1 Conexión directa entre hermanos

Un anillo, un toroide o una rejilla son ejemplos en los que es necesario establecer comunicaciones directas entre distintos procesos hermanos que han sido generados por un mismo proceso padre. Como ya hemos explicado, es preciso generar la topología inversa a la utilizada en la especificación, de modo que los receptores de mensajes puedan enviar los nombres de los

canales dinámicos a los emisores de mensajes. Los pasos a seguir son los siguientes:

- Para cada abstracción de proceso que se vaya a usar, se crea una nueva siguiendo estos pasos:
  - Para cada canal de salida `out` de tipo `t` que deba comunicarse directamente:
    - Se elimina ese canal de salida.
    - Se añade un nuevo canal de entrada `cn` de tipo `ChanName t`.
    - Se envían a través de `cn` los valores que antes se enviaran a través de `out`.
  - Por cada canal de entrada `inc` de tipo `t` que deba comunicarse directamente:
    - Se elimina ese canal de entrada.
    - Se añade un nuevo canal de salida `ocn` de tipo `ChanName t`.
    - Se crea un nuevo canal dinámico `(cn,c)`, siendo `c` de tipo `t` y `cn` de tipo `ChanName t`.
    - Se envía `cn` a través de `ocn`.
    - Se leen de `c` los valores que antes se leían de `inc`.
- En el proceso padre hace falta crear la topología de conexión inversa. Si la topología era circular (como en el toroide o en el anillo) no hace falta hacer nada, mientras que en el caso general debería reescribirse completamente el código. Las únicas reglas generales que pueden darse establecen cómo ajustar el proceso padre para que pueda comunicarse con los canales de sus hijos que han pasado a ser canales dinámicos debido a los pasos anteriores:
  - El proceso padre debe crear tantos canales dinámicos como conexiones directas necesite para recibir datos de los hijos, y enviarle los nombres a éstos.
  - En lugar de utilizar los canales por los que recibía datos de los hijos, se utilizarán los correspondientes canales dinámicos.
  - En lugar de enviar valores a los hijos, recibe de éstos los nombres de los nuevos canales dinámicos.
  - Los valores a enviar a los hijos se mandan a través de los nombres de canales dinámicos que los hijos le han enviado previamente.

6.6.2 Conexión directa entre generaciones

En [KPS00] se distinguían dos clases dentro de este tipo de conexión directa, dependiendo de si la comunicación debía realizarse desde un descendiente a un ascendiente o viceversa.

Comunicaciones desde un descendiente a un ascendiente

Un ejemplo de este tipo de *bypassing* se vio en la tubería. El último proceso de la tubería enviaba un mensaje al anterior, éste simplemente lo reenviaba al anterior, y éste al anterior y así hasta llegar al proceso padre. La solución general a este problema es la siguiente:

- Para cada abstracción de proceso que se vaya a usar, se crea una nueva siguiendo estos pasos:
  - Para cada canal de salida `out` de tipo `t` que deba comunicarse directamente:
    - Se elimina ese canal de salida.
    - Se añade un nuevo canal de entrada `cn` de tipo `ChanName t`.
    - Si el proceso es quien debe enviar realmente mensajes a través del canal, los envía a través de `cn`. En caso contrario envía `cn` al proceso hijo correspondiente.
- En el proceso principal, por cada canal de entrada `inc` que deba establecer una comunicación directa desde un descendiente al proceso principal para comunicar valores de tipo `t`:
  - Se crea un nuevo canal dinámico `(cn,c)`, siendo `c` de tipo `t` y `cn` de tipo `ChanName t`.
  - Se envía `cn` al hijo correspondiente.
  - Se leen de `c` los valores que antes se leían de `inc`.

Comunicaciones desde un ascendiente a un descendiente

Este tipo de *bypassing* resulta poco frecuente en las aplicaciones reales. De hecho, no sólo no se necesita en ninguno de los esqueletos mostrados en esta tesis, sino tampoco en ninguno de los programas que se han desarrollado hasta la fecha en el lenguaje Eden. De todas formas, por completitud incluimos el método a seguir, que utilizará unas pautas similares a las del apartado anterior, con algunas complicaciones adicionales. Ahora es un descendiente quien debe recibir datos directamente, por lo que debe ser él quien cree el canal dinámico y envíe el nombre de dicho canal al ascendiente correspondiente, atravesando tantos procesos intermedios como sea menester. En el programa original los procesos intermedios deberían tener un parámetro de entrada adicional que sólo se utilizará para transmitir valores desde el ascendiente al descendiente. Dicho parámetro deberá desaparecer, y en su lugar aparecerá un canal de salida por el que los procesos transmitirán a su ascendiente el nombre del canal dinámico por el que debe establecerse la comunicación directa:

- Para cada abstracción de proceso que se vaya a usar, se crea una nueva siguiendo estos pasos:
  - Para cada canal de entrada `inc` de tipo `t` que deba comunicarse directamente:
    * Se elimina ese canal de entrada.
    * Se añade un nuevo canal de salida `ocn` de tipo `ChanName t`.
    * Si el proceso es quien debe recibir realmente mensajes a través del canal, debe crear un nuevo canal dinámico `(cn,c)`, leer de `c` los datos que antes leyera de `inc`, y enviar `cn` a través del canal de salida `ocn`. En caso de no ser el receptor real de los mensajes, sólo debe enviar a través de su canal `ocn` el nombre `cn` que haya recibido del correspondiente canal `ocn` de su hijo.
- En el proceso principal, por cada canal de salida `out` que deba establecer una comunicación directa hacia un descendiente para comunicar valores de tipo `t`:
  - Se recibirá `cn` del hijo correspondiente.
  - Los valores que debieran enviarse a través de `out` se enviarán a través de `cn`.

6.7 Anidamiento de esqueletos

Una de las mayores dificultades con las que se encuentran los diseñadores de lenguajes de esqueletos es el anidamiento de los mismos. Como se vio en la Sección 2.3, una gran cantidad de dichos lenguajes no permitían que los esqueletos se anidaran, debido a las dificultades que ello entraña a la hora de implementar los compiladores. Aquellos que sí lo permiten suelen requerir complejas y laboriosas técnicas para hacerlo. Así, puede llegar a ser necesario tener que especificar cómo se combina cada posible par de esqueletos que soporte el sistema, aunque algunos lenguajes como P^3L proporcionan soluciones bastante satisfactorias.

En principio, en Eden la implementación del anidamiento de esqueletos es trivial, puesto que los esqueletos son simples funciones, y como tales pueden combinarse a gusto del programador. Ahora bien, aunque el anidamiento pueda implementarse de forma trivial, ello no quiere decir que la eficiencia de los programas obtenidos sea óptima, ni mucho menos. En este aspecto Eden sufre los mismos problemas que el resto de enfoques basados en esqueletos, puesto que no es fácil decidir cómo repartir los procesadores en presencia de anidamientos. De hecho, una importante área de investigación que se encuentra abierta actualmente pretende optimizar el reparto de procesadores de forma automática mediante combinaciones de modelos de coste, si bien

sólo existen resultados satisfactorios para casos muy simples como el caso en el que cada etapa de una tubería pueda ejecutarse en paralelo. Dicha situación es fácil de optimizar, pues sólo hay que atacar a los cuellos de botella de la tubería: a las etapas más costosas se les asignarán tantos recursos como sea posible hasta que dejen de ser los cuellos de botella, y a ninguna etapa se le asignarán recursos que no contribuyan a reducir el tiempo de ejecución global.

Desde nuestro punto de vista, la solución al problema del anidamiento debe pasar por una involucración del programador en el reparto de los recursos, pues no creemos que en el caso general sea posible hacerlo automáticamente. Como ha podido apreciarse a lo largo del presente capítulo, los esqueletos mostrados permiten especificar cuántos procesadores quieren emplearse, de modo que cuando se tienen varios esqueletos el programador puede decidir cómo repartirles los recursos. Desgraciadamente el control del que dispone se limita a indicar cuántos procesos se quieren utilizar, pero podría ser deseable un control más fino.

En [Fos95] se clasifican en tres grupos las formas en las que se pueden componer programas paralelos: (1) secuencialmente; (2) en paralelo; y (3) concurrentemente. Aplicando esta clasificación a la composición de esqueletos, en el primer caso la computadora paralela se utiliza por esqueletos distintos en diferentes momentos, por lo que la optimización es trivial. En el segundo caso la máquina se divide en dos o más subconjuntos disjuntos de procesadores, y a cada esqueleto se le asigna uno de dichos subconjuntos. El último caso es similar al segundo, salvo por el hecho de que los subconjuntos pueden no ser disjuntos. Las construcciones actuales de Eden no permiten dividir en subconjuntos los procesadores disponibles, si bien estamos pensando en añadir un mecanismo similar al de los *communicators* de la librería MPI, que permite que a cada subconjunto de procesadores se le asigne un identificador de grupo, siendo *all* el identificador del conjunto completo de procesadores. Si incorporásemos esta característica, bastaría con añadir al operador # de lanzamiento de procesos un parámetro opcional g (el identificador del grupo), de modo que se garantizara que el nuevo proceso se lance dentro del subconjunto de procesadores g.

Siendo más precisos, extenderíamos el lenguaje con un nuevo tipo abstracto predefinido `Commm` que represente los comunicadores. Cada valor de dicho tipo contendrá la lista de procesadores que forman parte del bloque. El tipo debe proporcionar un elemento distinguido `allPEs` que contenga todos los procesadores disponibles. Asimismo, serán necesarias funciones para generar nuevos comunicadores. El siguiente conjunto de funciones sería suficiente para nuestros objetivos:

```
allPEs :: Comm
sizeComm :: Comm -> Int
splitCommAt :: Int -> Comm -> (Comm, Comm)
splitCommRel :: Float -> Comm -> (Comm, Comm)
```

La función `sizeComm` devolverá el número de procesadores que forman parte del comunicador que se le pase como parámetro. La función `splitCommAt` genera dos nuevos comunicadores a partir de uno inicial, donde su primer parámetro indica cuántos procesadores se quieren asignar al primer comunicador, de modo que el resto de procesadores se asignen al segundo. Por último, la función `splitCommRel` es igual a la anterior, salvo por el hecho de que la división en comunicadores no se realiza en valores absolutos, sino relativos: por ejemplo, si el primer parámetro es `0.25`, la cuarta parte de los procesadores se asignarán al primer comunicador, y las restantes tres cuartas partes al segundo. Nótese que los comunicadores deben tener siempre al menos un procesador, pues de lo contrario no podrían ejecutar ninguna tarea. Por dicho motivo, al realizar las divisiones habrá que garantizar que al menos se asigna un procesador a cada parte, lo cual implica que si el comunicador original sólo disponía de un procesador, entonces los dos nuevos comunicadores deberán compartir el mismo único procesador. La implementación completa del tipo necesario para los comunicadores es la siguiente:

```
newtype Comm = Comm [Int]   -- lista de procesadores

allPEs :: Comm
allPEs = Comm [0..noPe-1]

sizeComm :: Comm -> Int
sizeComm (Comm xs) = length xs

splitCommAt :: Int -> Comm -> (Comm, Comm)
splitCommAt n c@(Comm xs)
  | sizeComm c > n = (Comm xs1, Comm xs2)
  | sizeComm c > 1 = (Comm (init xs), Comm [last xs])
  | otherwise      = (c,c)
  where (xs1,xs2) = splitAt n xs

splitCommRel :: Float -> Comm -> (Comm, Comm)
splitCommRel f c = splitCommAt n c
  where s = fromInt (sizeComm c)
        n = round (s*f)
```

Con este nuevo tipo disponible, podría definirse un nuevo operador de lanzamiento de procesos que tuviera en cuenta el comunicador en el que quiere lanzarse el proceso:

```
(##) :: Process a b -> a -> Comm -> b
```

de modo que el operador `#` original se redefiniría trivialmente en función del nuevo operador:

```
p # x = (p ## x) allPEs
```

Para poder anidar esqueletos utilizando el mecanismo de los comunicadores, sería suficiente redefinir todos los esqueletos añadiéndoles un parámetro extra que represente el comunicador en el que se ejecutarán, y utilizar dicho parámetro en todos los lanzamientos de proceso que se produzcan en el esqueleto. Por ejemplo, en el caso de la versión más simple de `map` bastaría con redefinirla como sigue:

```
map_naive :: (Transmissible a, Transmissible b) =>
             Comm -> (a -> b) -> [a] -> [b]
map_naive c f xs = [((process y -> f y) ## x) c | x <- xs]
                   `using` spine
```

Una vez que los esqueletos disponen de un comunicador, la composición de esqueletos sólo necesita pasar a cada esqueleto el comunicador adecuado, previo cómputo de cuál es el mejor reparto de procesadores.

A modo de ejemplo, reconsideremos el problema de la Sección 7.2.8 en el que se calculaban las fuerzas existentes entre un conjunto de partículas. Supongamos que queremos extenderlo para que no trabaje sobre un único conjunto de partículas, sino que queremos que reciba varios conjuntos de partículas, y que para cada uno de ellos no sólo compute las fuerzas que existen entre ellas, sino que también utilice las fuerzas calculadas para efectuar otro tipo de cómputos posteriores. Este problema podría resolverse mediante una tubería de dos etapas en la que la primera etapa fuera un anillo, y la segunda un `map`. Para cada conjunto de datos de entrada a la tubería, el anillo calcularía las fuerzas existentes, y el `map` usaría dichas fuerzas para calcular, por ejemplo, cuál sería el tiempo que aguantaría cada partícula sin destruirse bajo la fuerza calculada.

Suponiendo que sabemos que la primera etapa requiere 4 veces más tiempo de cómputo que la segunda etapa, y suponiendo que queremos reservar dos procesadores para los procesos principales de cada etapa, la combinación de esqueletos se expresaría del modo siguiente:

```
newForcess particless = pipe cp [ring cr ... , map_rw cm ... ]
(cp, crest) = splitCommAt 2 allPEs
(cr, cm)    = splitCommRel 0.8 crest
```

Para que el mecanismo descrito pueda llevarse a la práctica, sólo es necesario introducir dos modificaciones en el RTS de Eden: (1) introducir tablas en las que por cada comunicador se almacene la lista de procesadores que lo componen; y (2) modificar las estrategias de ubicación de procesos de modo que cuando se cree un nuevo proceso se garantice que se crea dentro del comunicador adecuado.

6.8 Conclusiones

Hemos mostrado que el lenguaje Eden permite la implementación concisa, y sin embargo eficiente, de esqueletos bien conocidos. Como prueba de tal concisión, los algoritmos que aparecen en este trabajo ocupan cada uno unas pocas líneas, lo cual confirma la utilidad de Eden como lenguaje *de sistema* para la creación de esqueletos. Como se ha visto a lo largo del capítulo, hay dos características de Eden que merece la pena resaltar para desarrollar esqueletos versátiles y eficientes:

- La existencia del proceso reactivo `merge`, que permite un reparto de carga óptimo en presencia de granularidades no homogéneas.
- La existencia de canales dinámicos, que permite implementar cualquier topología de interconexión entre procesos.

Estas dos construcciones confieren al lenguaje el suficiente bajo nivel como para ser un lenguaje *de sistema*, pero sin necesidad de perder el alto nivel de abstracción propio de un lenguaje funcional. De hecho, el no-determinismo de `merge` no suele "contaminar" el exterior de los esqueletos. Por su parte, aunque los canales dinámicos puedan considerarse construcciones de bajo nivel, y su empleo directo pueda parecer complejo, hemos visto que en la práctica sólo se utilizan para implementar especificaciones realizadas en el propio lenguaje utilizando abstracciones y concreciones de procesos, existiendo una metodología que dirige el modo de llevar a cabo dichas implementaciones.

Nótese que en el tipo de ninguno de los esqueletos que se han mostrado aparece el constructor de tipos `Process`. Gracias a ello el programador Eden podrá escribir sus programas paralelos sin más que invocar a determinados esqueletos pasándoles como parámetros simples funciones Haskell. Por tanto, no necesitará utilizar abstracciones ni concreciones de procesos, ni mucho menos el proceso `merge` o los canales dinámicos, consiguiéndose así un alto grado de abstracción.

Cabe destacar que el único lenguaje funcional paralelo perezoso en el que se ha implementado una librería de esqueletos es Eden: los lenguajes que como GpH no incluyen el concepto de proceso no permiten crear estructuras de procesos controlables por el programador; y los lenguajes como Caliban que sí incluyen tal concepto, deben decidir en tiempo de compilación la topología de comunicaciones a crear, restringiendo notablemente la capacidad expresiva de los mismos (sólo se ha implementado el esqueleto `map`). Además, ninguno de ellos tiene facilidades reactivas.

Comparado con los lenguajes basados en esqueletos, la capacidad expresiva de Eden es muy superior, al permitir definir cualquier nuevo esqueleto fácilmente. El conjunto de esqueletos que suministran los lenguajes convencionales suele ser muy restringido y, por ejemplo, en ninguno se proporciona un esquema de ramificación y poda, ni topologías en anillo o rejilla. Por otra

	Esqueletos	Anidamiento	Topologías dinámicas	Nuevos esqueletos	Distribuir datos	*streams*	Reactivo	Lenguaje cómputo
P^3L	Sí	Sí	No	Modificar compilador	Sí	No	No	C
PMLS	Sí	Sí	No	Modificar compilador	Sí	Sí	No	ML
Skil	Sí	No	Sí	C + MPI	Sí	Sí	Sí	C extendido
GpH	No	-	Sí	-	No	Sí	No	Haskell
Caliban	`map`	-	No	Caliban	Sí	Sí	No	Haskell
Eden	Sí	Sí	Sí	Eden	Sí	Sí	Sí	Haskell

Figura 6.17: Comparación de lenguajes paralelos (basados en esqueletos o funcionales paralelos)

parte, en la mayoría de lenguajes de esqueletos es necesario que la estructura de procesos se conozca en tiempo de compilación, por lo que no pueden describir topologías dinámicas. Además, suelen restringir también el tipo de paralelismo que explotan. Por ejemplo, dado que P^3L no posee ningún tipo de estructuras dinámicas, no permite explotar paralelismo de tipo *stream*.

La Figura 6.17 resume las características de que disponen tanto los lenguajes basados en esqueletos más significativos, como los lenguajes funcionales paralelos más relevantes. Puede apreciarse que todos los lenguajes, salvo Eden, carecen de alguna característica que les hace perder capacidad expresiva a la hora de escribir programas paralelos. Sólo Skil proporciona casi todas las características examinadas, pero lo consigue a expensas de permitir que se desarrollen nuevos esqueletos usando lenguajes de bajo nivel (C+MPI), así como prohibiendo el anidamiento de esqueletos. Por contra, Eden es el único capaz de reunir en un único lenguaje una gran capacidad expresiva y un alto nivel de abstracción.

Capítulo 7

Aplicaciones paralelas

El contenido de la primera parte del presente capítulo, que muestra un conjunto de aplicaciones desarrolladas en Eden, es completamente original de esta tesis, y ha sido publicado en [**KPR01**, **PR01**, **LOP$^+$02**]. La segunda parte del capítulo contiene una comparación de los resultados obtenidos al implementar un mismo conjunto de aplicaciones en tres lenguajes funcionales paralelos distintos. El trabajo de comparación se realizó entre tres personas, una por cada lenguaje, de modo que la parte original de la segunda sección se restringe a la concerniente al lenguaje Eden. Aún así, se incluye la comparación completa, con el objetivo de poder comparar el trabajo del autor de la tesis con el de otros investigadores del mismo área. El trabajo expuesto en la segunda parte del capítulo ha dado lugar al artículo [**LRS$^+$01**].

7.1 Introducción

A diferencia del capítulo anterior en el que utilizábamos Eden como lenguaje de sistemas, en el presente capítulo mostraremos ejemplos de uso de Eden como lenguaje de desarrollo de aplicaciones. Comenzaremos mostrando un conjunto de programas paralelos escritos en Eden y que utilizan los esqueletos que definimos en el capítulo anterior. Para cada uno de ellos describiremos el problema, mostraremos el código fuente relevante e incluiremos las aceleraciones reales obtenidas en máquinas paralelas del tipo Beowulf [RBMS97].

La segunda parte del capítulo está dedicada a comparar la programación en Eden con la programación en otros dos de los lenguajes funcionales paralelos más relevantes en estos momentos: PMLS y GpH. El objetivo es comparar la eficiencia de las aplicaciones desarrolladas, la capacidad expresiva de los lenguajes, la facilidad de uso y la portabilidad. Para ello se utilizarán tres ejemplos típicos de algoritmos paralelos, que se han implementado en los tres lenguajes.

7.2 Programación en Eden

En esta sección se muestran los resultados obtenidos para varios ejemplos que utilizan los esqueletos del capítulo anterior. Para cada ejemplo se incluirán tanto aceleraciones reales como predicciones de los modelos de coste. Las medidas se han llevado a cabo en un Beowulf de 64 procesadores del que dispone la universidad de St. Andrews. Cada uno de sus nodos es un Pentium II a 450MHz, con 348MB de RAM y con sistema operativo Linux RedHat 5.2. Por lo que respecta a la interconexión de los nodos, se lleva a cabo mediante un *switch full duplex fast Ethernet* 100Mb/s CISCO 2984G, siendo la latencia del sistema $\delta = 142\mu$s. Dicho Beowulf es un entorno paralelo de bajo coste, por lo que es una arquitectura de destino deseable para Eden, pero también es una arquitectura con altas latencias, por lo que es un reto para un lenguaje distribuido de alto nivel como Eden.

Aunque el Beowulf dispone de 64 procesadores, no es posible utilizarlos todos a la vez, debido fundamentalmente a conflictos con otros usuarios. Por ello nuestras medidas suelen utilizar sólo la mitad de los procesadores, siendo 43 el número máximo de procesadores que hemos usado en alguno de los experimentos.

El método seguido en todos los experimentos ha sido el mismo. La compilación siempre se ha realizado con todas las optimizaciones activadas, para obtener el mejor código posible, y siempre se ha utilizado la misma librería de paso de mensajes: PVM 3.4.2. Se han realizado medidas para todo número de procesadores menor o igual al máximo disponible, efectuando al menos 3 ejecuciones por cada número de procesadores. Por cada número de procesadores se extrae la media de las medidas realizadas, eliminando los resultados que sean muy irregulares, con el fin de aislarnos de interferencias de otros usuarios o de procesos del sistema operativo.

En todos los ejemplos mostraremos gráficas de aceleraciones relativas a la versión paralela más rápida con 1 procesador. No usaremos aceleraciones absolutas para ignorar los sobrecostes del RTS de Eden, que aún no ha sido completamente optimizado. En general, la versión paralela con un procesador es aproximadamente un 10% más lenta que la ejecución secuencial pura.

7.2.1 Conjuntos de Mandelbrot

Descripción del problema

Los conjuntos de Mandelbrot (véase e. g. [Bra89]) se utilizan para generar fractales como el de la Figura 7.1. Por cada píxel de la pantalla es necesario calcular su color, que se corresponde con su distancia al conjunto de Mandelbrot. Dicho color dependerá de las coordenadas, y se calculará iterando una función hasta que, o bien se obtenga una aproximación suficientemente buena, o bien se alcance un número máximo de iteraciones (4000 en nuestras

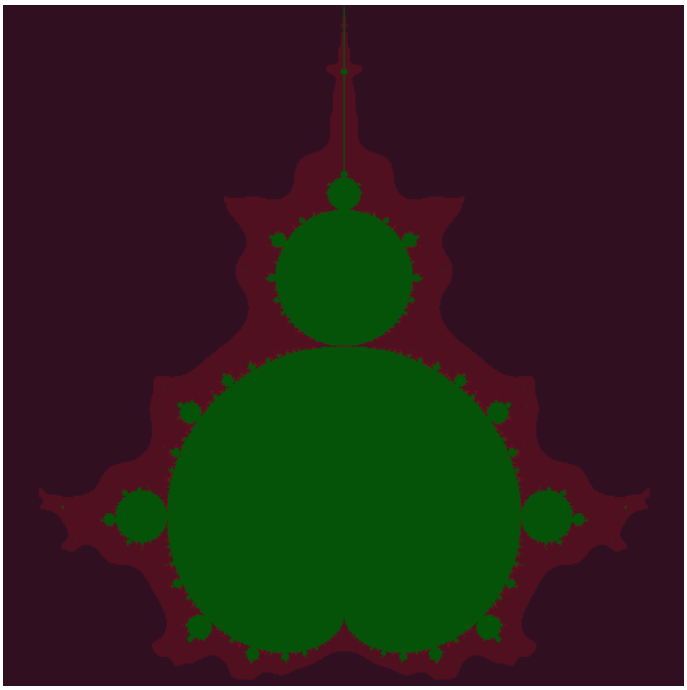


Figura 7.1: Ejemplo de salida del programa de Mandelbrot

medidas). Así pues, para algunos pixels es suficiente realizar unas pocas iteraciones, mientras que para otros harán falta 4000 iteraciones. El núcleo del algoritmo secuencial es el siguiente:

```
mandel sizeX sizeY = map aPixelOfMandel (range ((0,0),(sizeX,sizeY)))
aPixelOfMandel (x,y) = iterate (firstAprox (x,y)) 0
iterate v its
  | its == maxIts || goodAprox v = extractColor v
  | otherwise = iterate (nextAprox v) (its+1)
```

Algoritmo paralelo

Dado que el cómputo de cada píxel puede realizarse en paralelo, puede utilizarse el esqueleto `map`. Ahora bien, para incrementar la granularidad de las tareas asignaremos una línea completa de la pantalla a cada tarea, en lugar de un único píxel.

El algoritmo es altamente paralelo, pues todos los cómputos son completamente independientes, y por tanto la relación cómputo-comunicaciones es muy buena. Pero desgraciadamente la complejidad de las tareas es muy variable, como ya se anticipó en la descripción del problema. Por tanto, paralelizaciones ingenuas del problema pueden conducir a malos repartos de carga que degraden la aceleración obtenida. Por ejemplo, si simplemente dividimos la pantalla en p regiones (siendo p el número de procesadores) y

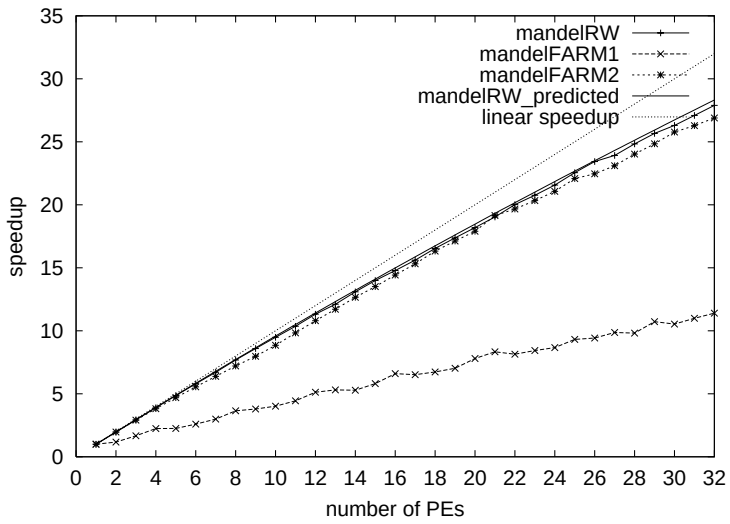


Figura 7.2: Aceleraciones para conjuntos de Mandelbrot

utilizamos un proceso para cada una de las regiones, no podemos garantizar un reparto de carga equitativo. Además, es sabido que los pixels que requieren más cómputo suelen estar próximos entre sí, por lo que no es recomendable asignar a un mismo procesador zonas "vecinas" entre sí. Puede utilizarse la implementación `map_farm`, pero en ese caso habrá que tener mucho cuidado a la hora de elegir las funciones `shuffle` y `unshuffle`, pues las aceleraciones variarán drásticamente dependiendo de ellas. Por si fuera poco, las aceleraciones dependerán del conjunto de Mandelbrot concreto que se calcule, y para conjuntos distintos pueden convenir distintas funciones de reparto. Por todo lo anterior, la solución más inteligente es el empleo de la implementación `map_rw`, ya que equilibra el reparto de carga por sí misma. Los cambios necesarios en el código fuente son sólo los siguientes:

```
mandel sizeX sizeY      = map_rw 20 (oneLineOfMandel sizeY) [1..sizeX]
oneLineOfMandel sizeY x = map aPixelOfMandel (range ((x,0),(x,sizeY)))
```

donde el primer parámetro de `map_rw` indica que a partir de 20 procesadores es mejor que el proceso principal no comparta su procesador con un proceso trabajador.

Resultados obtenidos

Hemos generado un conjunto de Mandelbrot para una pantalla de 1024×1024 pixels, siendo el tiempo de ejecución secuencial 459 segundos. La figura 7.2 muestra los resultados obtenidos tanto con `map_rw` como con `map_farm`. En el caso de `map_farm` hemos empleado dos estrategias distintas para repartir tareas entre procesos. La primera estrategia simplemente dividía la

pantalla en tantas regiones como procesadores hubiera, pero nos condujo a un mal reparto de carga con el que sólo se llegó a una aceleración de 11.5 con 32 procesadores. La otra estrategia asignaba las líneas $i + n * p$-ésimas al procesador i-ésimo (siendo p el número de procesadores), y obtuvo unos resultados muy próximos a los de `map_rw`, con una aceleración de 26.9 frente a los 27.9 de `map_rw`.

Las medidas confirman que el esqueleto `map_rw` es el más apropiado para este problema, al igual que lo será para cualquier otro en el que las granularidades sean variables. Aunque los sobrecostes de `map_rw` sean ligeramente superiores que los de `map_farm`, la aceleración absoluta que se obtiene es mayor debido al reparto óptimo de la carga. Nótese que `map_farm` puede acercarse a `map_rw` con elecciones apropiadas de `unshuffle` y `shuffle`, pero esto no sólo implica más trabajo por parte del programador, sino que puede depender del problema de entrada del programa. Además, en caso de que la máquina paralela esté siendo compartida con otros usuarios, el comportamiento de `map_farm` se verá muy afectado al no poder equilibrar la carga, mientras que `map_rw` se adaptará para extraer el máximo partido de los tiempos de CPU que se le asignen. Por ello, en el resto de ejemplos usaremos sólo `map_rw`, descartando `map_farm`.

Nótese también que las predicciones efectuadas para `map_rw` son bastante precisas, y que la escalabilidad es bastante buena. Para facilitar la exposición, tanto en esta aplicación como en las siguientes, simplificaremos el modelo de coste de los trabajadores replicados, colapsando parámetros de modo que la expresión que define el tiempo paralelo sea más manejable, y sólo distinga los costes que son proporcionales al número de procesadores (p) de los que son fijos:

$$t_p = \frac{t_{sec}}{p} + p * c_1 + c_2$$

En nuestro caso, el coste c_1 es muy pequeño, pues es sólo de 0.015 segundos, que es el tiempo necesario para crear un proceso y enviarle una tarea. La principal ineficiencia está en el coste c_2, que es de 1.2 segundos. Dicho cuello de botella se debe a la necesidad de combinar adecuadamente los resultados enviados por los trabajadores, de modo que pueda generarse la gráfica deseada. En el caso de `map_farm` no proporcionamos predicciones debido a que el modelo de coste supone que las granularidades de las tareas deben ser regulares, y ello no es cierto para los conjuntos de Mandelbrot.

7.2.2 Trazador de rayos

Descripción del problema

El programa `raytracer` calcula una imagen bidimensional a partir de una escena con objetos tridimensionales, y a partir de la posición de la cámara. Para ello, traza todos los rayos posibles a través de una ventana bidimensional perpendicular a la dirección de orientación de la cámara. La resolución

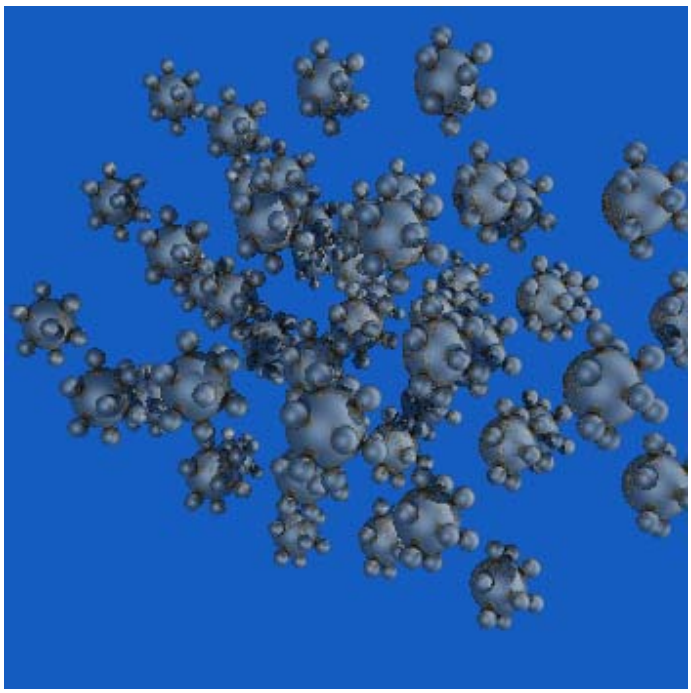


Figura 7.3: Escena utilizada por el trazador de rayos

de la ventana viene dada a través de un parámetro que indica el número de pixels de cada línea. Al trazar un rayo se calculan las intersecciones con los objetos. Cuando se encuentra una intersección se refleja el rayo, y el color del punto de intersección se calcula a partir de la fuerza del rayo y de la textura del objeto interceptado.

El programa original proviene del banco de programas de Impala [Imp], y fue traducido a Haskell por el grupo que desarrolla GpH. El código fuente del núcleo del programa Haskell se muestra a continuación. La función `ray` toma como parámetros el tamaño de la ventana en las dimensiones `x` e `y`, y la escena `world` que está formada por una lista de esferas. El cómputo a realizar se estructura con dos `map`s anidados, donde el más externo trabaja sobre líneas de la ventana, mientras el interno aplica la función `tracepixel` a cada punto de la ventana.

```
ray :: Int -> Int -> [Sphere] -> [[((Int, Int), Vector)]]
ray x y world = map (do_line world) sizes_y
  where do_line :: Int -> [((Int,Int), Vector)]
        do_line world i = map (\ j -> ((i,j), f world i j)) sizes_x
        sizes_x = [0..x-1]
        sizes_y = [0..y-1]
        f world i j = tracepixel world lights i j firstray scrnx scrny
        (firstray, scrnx, scrny) = camparams x y
```

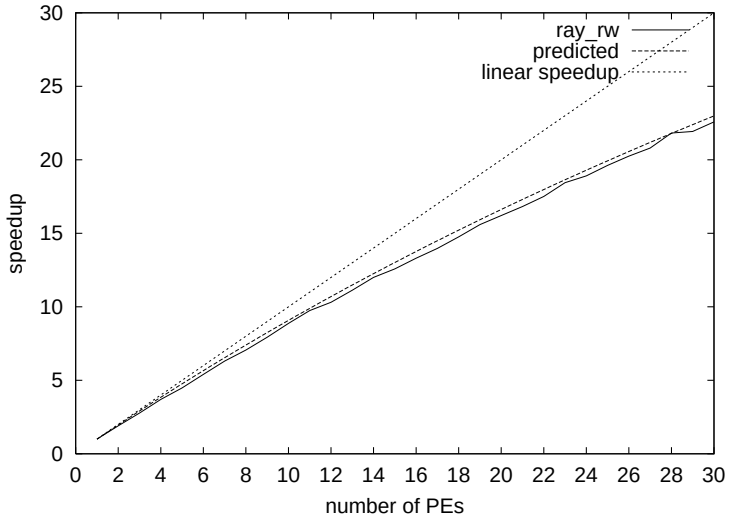


Figura 7.4: Aceleraciones del trazador de rayos

Algoritmo paralelo

Dado que los cómputos de todos los pixels son independientes entre sí, podemos aplicar el esqueleto `map`. Al igual que en el caso de los conjuntos de Mandelbrot, para incrementar la granularidad cada tarea calculará los impactos de una línea completa. Ahora bien, dado que todas las tareas necesitarán conocer la escena a trazar, es absurdo que haya que enviar dicha escena cada vez que se asigna una nueva tarea a un proceso. Por ello utilizaremos la versión del esqueleto `map` que incorporaba un parámetro extra para estructuras de datos fijas y compartidas por todas las tareas. Más concretamente utilizaremos la versión basada en trabajadores replicados, para garantizar un buen reparto de la carga. La modificación del código fuente es trivial:

```
ray x y world = map_rwF 20 world do_line sizes_y    where ...
```

Resultados obtenidos

Las aceleraciones que se muestran en la Figura 7.4 han sido obtenidas para una imagen de 350×350 pixels, con una escena formada por 640 esferas (véase la Figura 7.3), y su tiempo de ejecución secuencial fue 176.99 segundos. Como era de esperar, los resultados obtenidos son muy satisfactorios, ya que el problema es altamente paralelo. Además, el modelo de coste predice acertadamente las aceleraciones. En este caso, y utilizando de nuevo el modelo simplificado

$$t_p = \frac{t_{sec}}{p} + p * c_1 + c_2$$

observamos que c_1 vuelve a valer 0.015 segundos, pues las tareas a realizar por cada procesador son básicamente las mismas que en el problema anterior: crear un proceso y enviarle una tarea inicial. Al igual que con los conjuntos de Mandelbrot, la principal ineficiencia es un cuello de botella secuencial de 1.3 segundos, debido a la combinación de los resultados obtenidos por los trabajadores.

7.2.3 Suma de números de Euler

Descripción del problema

El número de Euler de un valor x dado es el número de enteros menores que x que son primos relativos con respecto a x. La suma de los números de Euler consiste en calcular la suma de los números de Euler de los n primeros naturales. La versión secuencial es trivial:

```
sumEuler :: Int -> Int
sumEuler n = sum (map euler [n,n-1..1])

euler   :: Int -> Int
euler x = length (filter (relprime x) [1..(x-1)])
```

Este sencillo programa ha sido propuesto recientemente en [TLP01] para comparar los distintos estilos de programación de diferentes lenguajes funcionales paralelos basados en Haskell, por lo que merece la pena ver cómo se implementaría en Eden. Nótese que la lista de números se genera de mayor a menor. El motivo es que la versión original fue propuesta para el lenguaje GpH, en el que si se genera la lista de menor a mayor se degrada la eficiencia de la versión paralela. En el caso de Eden es irrelevante el orden de generación de los números, así que mantenemos el orden de la versión original.

Algoritmo paralelo

El problema encaja perfectamente en el esquema *map & reduce* (véase la Sección 6.3.2, pues primero se aplica un `map` con la función `euler`, para luego sumar todos los resultados obtenidos. Dado que `sum` es un caso particular de `fold`, la utilización del esqueleto genérico será muy simple. Además, la lista de tareas puede calcularse de modo trivial en cada trabajador, con lo que se reducirán las comunicaciones. Así pues, la paralelización simplemente requiere cambiar la definición de `sumEuler` del siguiente modo:

```
sumEuler     :: Int -> Int -> Int
sumEuler th n = map_reduce_as th euler (+) 0 [n,n-1..1]
```

donde el parámetro `th` indicará a partir de cuántos procesadores interesa que el proceso principal no comparta procesador con un trabajador. Dado

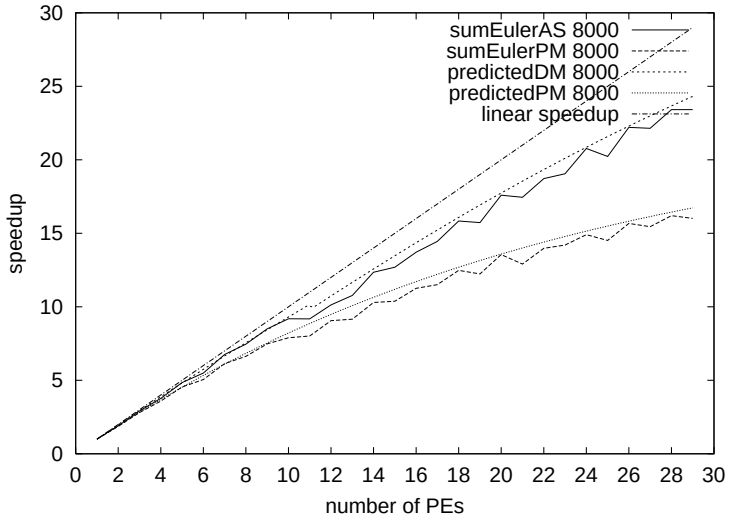


Figura 7.5: Aceleraciones obtenidas para los números de Euler

que la granularidad de la función `euler` depende directamente de su valor de entrada, será importante tener cuidado a la hora de distribuir las tareas entre procesadores. La granularidad se decrementa a medida que el valor de entrada es menor, por lo que lo razonable es emplear un reparto *round-robin*, que evite que a un procesador le toquen más tareas costosas que a otros.

Resultados obtenidos

La Figura 7.5 muestra los resultados obtenidos utilizando tanto la versión `map_reduce_as` como la `map_reduce_pm`, usando 8000 como valor de entrada, y siendo el tiempo de ejecución secuencial 80.39 segundos. El valor usado para el parámetro `th` ha sido 10. Puede apreciarse que la escalabilidad de la primera es bastante buena: dado que hemos reducido las comunicaciones al mínimo, la principal imperfección en la paralelización será el tiempo necesario para que el proceso principal cree e inicialice al resto de procesos, que es de 0.015 segundos por cada procesador. Así pues, en el modelo simplificado

$$t_p = \frac{t_{sec}}{p} + p * c_1 + c_2$$

tendremos que c_1 vale 0.015 segundos, mientras que c_2 será despreciable, ya que el único cómputo secuencial consiste en una suma de p elementos. Por lo que respecta a la versión que usa `map_reduce_pm`, existe un cuello de botella secuencial de 1.6 segundos, debido a que el proceso principal tiene que enviar 8000 tareas. A partir de la gráfica resulta obvia la importancia que tiene minimizar las comunicaciones.

Nótese que, a diferencia de los resultados obtenidos en los ejemplos anteriores, las gráficas no son completamente regulares. Esto se debe a que ahora

no se utiliza el esquema de trabajadores replicados para equilibrar la carga, por lo que el algoritmo es más sensible a números concretos de procesadores, así como a breves interferencias por parte de otras aplicaciones, tanto de otros usuarios como del propio sistema operativo.

7.2.4 Algoritmo de Karatsuba

Descripción del problema

El algoritmo de Karatsuba [KO62] calcula el producto de dos enteros de longitud arbitraria. Para ello utiliza un enfoque divide y vencerás. Siendo n la longitud de los enteros en su base de representación física, la complejidad del algoritmo ingenuo es $O(n^2)$, mientras que el algoritmo de Karatsuba tiene una complejidad $O(n^{log_2 3})$.

Si queremos multiplicar dos enteros x e y representados en base b, el algoritmo funciona del modo siguiente:

- Sea n la mitad de la longitud del número más largo de entre x e y.
- Sean $x_1 = x/b^n$, $x_2 = x \,\mathbf{mod}\, b^n$, $y_1 = y/b^n$ e $y_2 = y \,\mathbf{mod}\, b^n$.
- Sea $u = x_1 * y_1$, $v = x_2 * y_2$, $w = (x1 + x2) * (y1 + y2)$.
- Entonces el producto es $u * b^{2*n} + (w - u - v) * b^n + v$.

Nótese que para obtener x_1, x_2, y_1 e y_2 no ha sido necesario realizar ninguna división, sino que ha sido suficiente cortar las listas que representan a x y a y. Análogamente, el producto por b^n y por b^{2*n} no necesita multiplicaciones, sino sólo añadir ceros a las representaciones de los enteros largos. Por tanto, sólo son necesarias tres multiplicaciones, es decir, el problema de partida se divide en tres subproblemas. Dado que el tamaño de cada uno de los subproblemas es la mitad del original, y la complejidad de la combinación de los subresultados es $O(n)$, tenemos que la complejidad total del algoritmo es $O(n^{log_2 3})$.

Algoritmo paralelo

Este algoritmo encaja en el esquema divide y vencerás, pero nótese que la granularidad de las tareas puede ser variable debido a que los tamaños de los enteros pueden ser muy distintos. Por ejemplo, si uno de los enteros es de tamaño 1000 y el otro es de tamaño 600, generaremos tres problemas: uno con enteros de longitudes 500 y 100, y otros dos con ambos enteros de tamaño 500. La implementación del algoritmo de Karatsuba en términos del esqueleto divide y vencerás es la siguiente:

```
type MyInteger = [Int]
kara :: Int -> Int -> MyInteger -> MyInteger -> MyInteger
```

```
kara is1 is2 = dc trivial multSeq split combine (is1,is2)

trivial (is1,is2) = limit > min (length is1) (length is2)

split (is1,is2) = [(is1a,is2a),(is1b,is2b),
                   (addSeq is1a is1b,addSeq is2a is2b)]
  where ldiv        = (max (length is1) (length is2)) `div` 2
        (is1b,is1a) = splitAt ldiv is1
        (is2b,is2a) = splitAt ldiv is2

combine (is1,is2) [u,v,w] = result
  where ldiv    = (max (length is1) (length is2)) `div` 2
        u0s     = replicate (2*ldiv) 0 ++ u
        (wuv,s) = subSeqS w (addSeq u v)
        wuv0s   = (replicate ldiv 0 ++ wuv,s)
        result  = addSeqS (addSeq u0s v) wuv0s
```

donde las funciones no mostradas son simples funciones Haskell que implementan las sumas, restas y productos secuenciales (`addSeq`, `subSeqS` y `multSeq`).

Resultados obtenidos

Hemos probado tanto la versión `dc_rw` como la `dc_naive` sobre el mismo problema de entrada. El tiempo de ejecución secuencial de 440 segundos. La Figura 7.6 muestra tanto las aceleraciones predichas como las obtenidas realmente. Como era de esperar, la versión ingenua del esqueleto no sólo es peor, sino también mucho más irregular que la otra. La versión ingenua tiene más sobrecostes debido a que crea más procesos, pero la principal razón de su peor eficiencia es el mal reparto de carga que se produce. Nótese que la predicción realizada para `dc_rw` es bastante aproximada al comportamiento real, mientras que no proporcionamos ninguna predicción para la versión ingenua debido a que no disponemos de ningún modelo de coste suficientemente bueno.

7.2.5 Problema del viajante

Descripción del problema

El problema del viajante es un problema NP-completo clásico que requiere calcular caminos mínimos en un grafo. Partiendo de una ciudad concreta, un viajante de comercio debe visitar un conjunto de ciudades y volver a la ciudad de origen, de modo que el tiempo empleado en realizar el recorrido completo sea mínimo. El conjunto de caminos que conectan unas ciudades con otras se da como parámetro de entrada, así como el tiempo necesario para recorrer cada uno de dichos caminos. Nótese que el problema sólo tiene

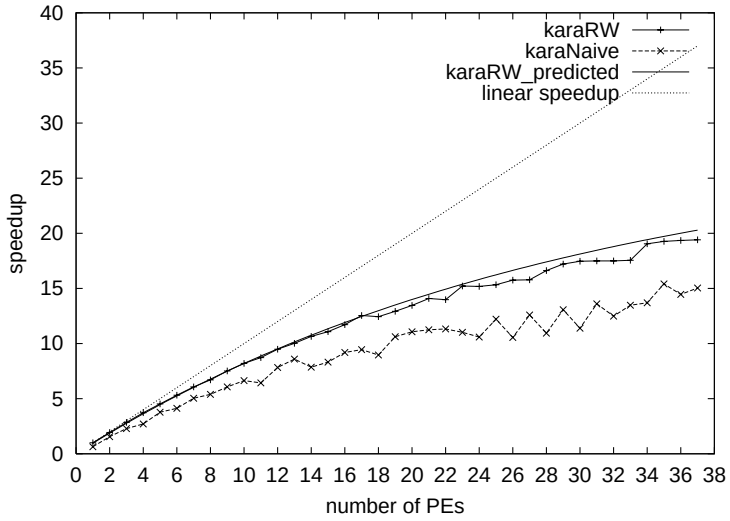


Figura 7.6: Aceleraciones del algoritmo de Karatsuba

sentido cuando el grafo generado por el conjunto de caminos es conexo, pues de lo contrario sería imposible realizar el recorrido pedido.

Algoritmo paralelo

Existen muchas soluciones a este problema. Aquí utilizaremos dos de ellas, ambas basadas en el esquema de ramificación y poda. La primera solución es la clásica propuesta en [HS78], donde en cada paso se extiende un camino parcial con un nuevo vértice. Para calcular la cota inferior de un camino parcial se emplea un algoritmo voraz de coste $O(n^2)$, por lo que en cada nodo se resuelve un problema $O(n^2)$, siendo n el número de vértices.

En la versión paralela, el árbol de búsqueda se divide inicialmente en $(n-1)(n-2)(n-3)$ subárboles, para lo cual basta descender tres niveles en el árbol. A continuación puede utilizarse el esqueleto `rwBB_naive` para distribuir las tareas entre los trabajadores y para realimentarles con la mejor cota superior obtenida hasta el momento por cualquiera de ellos:

```
type Graph = Array (Int,Int) Int -- Adjacency matrix with costs
type Path  = [Int]               -- A list of consecutive vertices
type Tour  = (Int,Path)          -- The integer is the tour cost
salesman :: Int -> Graph -> Tour
salesman np g = minimum (rwBB_naive np 1 g worker iniUB fst paths)
  where iniUB = ... cost of a non optimal solution
        paths = [[1,i,j,k] | i <- [2..n], j <- [2..n] \\ [i],
                             k <- [2..n] \\ [i,j]]
        worker :: Graph -> (Path,Int) -> Tour
        worker g (p,ub) = ... -- BB starting at p, using ub to prune
```

Nótese que utilizamos un `prefetch` de 1 con el objetivo de obtener la mejor cota superior lo antes posible.

La segunda paralelización es una adaptación del algoritmo de Little [LMSK63], siguiendo las indicaciones de [Qui94, Capítulo 13]. Ahora cada nodo representa un conjunto de restricciones a los caminos alcanzables desde él. Una restricción puede consistir en incluir una arista o en excluirla. La raíz del espacio de búsqueda es el conjunto vacío de restricciones. Para procesar un nodo debemos considerar todas las aristas aceptables (una arista es aceptable si no viola ninguna de las restricciones y además existe al menos un camino tras elegir dicha arista), y calcular las cotas inferiores de los problemas resultantes de excluir cada una de dichas aristas, eligiéndose la arista que incremente más la cota inferior. Se generan dos nuevas tareas: la que tiene la restricción de excluir la arista, y la que la incluye. Dado que el conjunto de aristas aceptables puede variar entre $O(1)$ y $O(n^2)$, y el cómputo por cada una es $O(n^2)$, el cómputo de cada nodo variará entre $O(n^2)$ y $O(n^4)$. Esto proporciona una buena granularidad para la versión paralela. Además, este algoritmo permite podar muchos más nodos que el primer algoritmo expuesto. La implementación en Eden utilizará el esqueleto `rwBB`:

```
type Edge    = (Int,Int)                -- Origin and destination
type Tour    = (Int,[Edge])             -- The Int is the tour cost
newtype Task = MkT ([Edge],[Edge],Int) -- Included and excluded edges,
                                        -- and lower bound
salesman :: Int -> Graph -> Tour
salesman np g = rwBB np 2 g worker iniUB lbf fst [MkT ([],[],c)]
  where iniUB = ... cost of a non optimal greedy solution
        lbf   = (\(MkT(_,_,lb))->lb)
        c     = lowerBound g [] []
        worker :: Graph -> (Task,Int) -> Either [Task] Tour
        worker g (MkT (included,excluded,lb),ub) = ...
```

Resultados obtenidos

A diferencia del resto de aplicaciones mostradas en este capítulo, los resultados no se han obtenido con una arquitectura de tipo Beowulf, sino con una estación de trabajo que dispone de cuatro procesadores UltraSparc II a 250MHz. La estación dispone de 512 MBytes de memoria compartida, y 1Mbyte de memoria *cache* por cada procesador. Aunque la arquitectura dispone de memoria compartida, la utilizaremos como si tuviera memoria distribuida, para obtener un comportamiento más similar al que se obtendría con una Beowulf. De hecho, utilizaremos una implementación de PVM sobre TCP/IP, en vez de sobre memoria compartida. La latencia de la arquitectura bajo estas condiciones será de 109 μsegundos.

La decisión de no emplear la arquitectura Beowulf se debe a problemas técnicos. El compilador actual de Eden no es completamente estable, sino

que aún necesita que se lleven a término tareas de depuración de su RTS, que aún presenta errores para algunos problemas concretos. Para el problema del viajante, las ejecuciones realizadas en la máquina Beowulf fallaban en tiempo de ejecución, debido fundamentalmente a problemas en la gestión de memoria del RTS. Afortunadamente, dicho fallo se produce sólo en la versión para arquitecturas Linux, pero no en el caso de máquinas SUN, por lo que hemos podido medir la aceleración en otra arquitectura, siendo el único problema que la nueva arquitectura sólo dispone de cuatro procesadores.

Como era de esperar, en la Figura 7.7 puede apreciarse que las aceleraciones obtenidas con la versión ingenua son bastante pobres (2.04 con 4 procesadores). El motivo no es un mal reparto de carga, sino que el principal problema radica en que el algoritmo paralelo computa muchos más nodos que el secuencial. Por su parte, la versión que utiliza `rwBB` obtiene unas aceleraciones bastante buenas (3.58 con 4 procesadores). El problema de entrada que se ha utilizado está formado por un grafo de 16 nodos que representa un caso "promedio", en el sentido de que la versión paralela no explora muchos más nodos que la secuencial, ni tampoco menos. Como es sabido, las aceleraciones en los problemas de ramificación y poda son muy sensibles al problema particular que haya que resolverse. Así, hemos encontrado casos en los que, debido a cómputos especulativos en la versión paralela, las aceleraciones eran peores que las mostradas en la figura, mientras que hemos encontrado otros casos patológicos en los que la aceleración era superlineal.

Los resultados obtenidos son satisfactorios para cuatro procesadores, pero será necesario esperar a poder utilizar máquinas con más procesadores para sacar conclusiones más relevantes acerca de la escalabilidad de nuestra solución.

7.2.6 Gradiente conjugado

Descripción del problema

El gradiente conjugado (véase e.g. [KGGK94, Capítulo 11]) es un método iterativo para encontrar soluciones aproximadas en sistemas de ecuaciones en los que la matriz de coeficientes sea definida positiva. En cada iteración del algoritmo se refina la solución actual x aplicando la función

$$x(t) \;=\; x(t-1) \;+\; s(t)d(t)$$

donde d es el vector de dirección hacia la solución, y s es el tamaño del paso a realizar. Cada iteración necesita realizar los siguientes cómputos:

$$g(t) \;=\; Ax(t-1) \;-\; b$$

$$d(t) \;=\; -g(t) \;+\; \frac{g(t)^T g(t)}{g(t-1)^T g(t-1)} d(t-1)$$

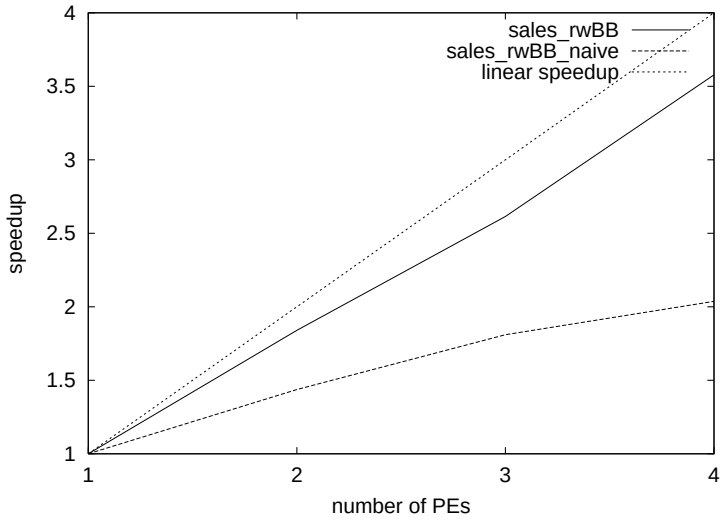


Figura 7.7: Aceleraciones del algoritmo del viajante

$$s(t) \;=\; -\frac{d(t)^T g(t)}{d(t)^T A d(t)}$$

$$x(t) \;=\; x(t-1) \;+\; s(t)d(t)$$

donde en la primera iteración se inicializan a cero los vectores $x(0)$, $d(0)$ y $g(0)$, mientras que $g(0)$ se inicializa con $-b(0)$.

El método del gradiente conjugado garantiza que x converge en a lo sumo n iteraciones, siendo n el número de ecuaciones del sistema. Dado que la complejidad de cada paso es $O(n^2)$, el coste total del algoritmo es $O(n^3)$.

Algoritmo paralelo

Este algoritmo es un ejemplo típico de uso del esqueleto `iterUntil`: está formado por varias etapas iguales; cada etapa puede paralelizarse; y cada etapa debe completarse totalmente antes de empezar la siguiente.

El algoritmo tiene una complicación adicional con respecto a los algoritmos típicos de iteración, y es que cada etapa realmente está formada por dos etapas paralelizables: el producto de A por x y el de A por d. Dichos productos son los más costosos, por lo que son los que deben paralelizarse. Desgraciadamente no puede hacerse en un único paso paralelo, pues es necesario conocer los resultados producidos por otros procesadores antes de poder realizar el segundo producto. Para poder diferenciar ambas etapas en el programa Eden, bastará utilizar el tipo `Either` para saber en qué subeta-

pa nos encontramos cada vez. A continuación se muestra el código fuente, en el que puede apreciarse cómo se distinguen ambas etapas:

```
type Input     = (Matrix,Vector,Vector,Vector,Vector)
type Task      = Either Vector Vector        -- d or x
type SubResult = Either Vector Vector        -- Ad or Ax
type LocalW    = (Matrix,Vector)             -- A_i and b_i
type LocalM    = (Vector,Vector,Vector,Double,Int)
                  -- d,g,x,gg,iterations left

gc :: Int -> Matrix -> Vector -> Vector
gc a b = gc' a b n0s b (map negate b)
  where n0s = replicate (length b) 0

gc' a b x d g = iterUntil split f_it comb (a,b,x,d,g)   where
  split :: Input -> Int -> ([LocalW],[Task],LocalM)
  split (a,b,x,d,g) np = (splitIntoN np (zip a b),
                          replicate np (Left d),
                          (d,g,x,prVV g g,length b))

  f_it :: LocalW -> Task -> (SubResult,LocalW)
  f_it l t = (f_it' l t,l)  -- The local data is always the same
  f_it' (ai,bi) (Right x) = Right (zipWith (-) (prMV ai x) bi) -- g
  f_it' (ai,bi) (Left d)  = Left (prMV ai d)                   -- Ad

  comb :: LocalM -> [SubResult] -> Either Vector ([Task],LocalM)
  comb (d,g,x,gg,cont) srs@(Left _:_)
    | cont <= 0 = Left newx
    | otherwise = Right (replicate np (Right newx),
                         (d,g,newx,gg,cont-1))
    where ad   = concat (map theLeft srs)
          s    = -(prVV d g) / (prVV d ad)
          newx = zipWith (+) x (prEV s d)
          np   = length srs

  comb (d,g,x,gg,cont) srs@(Right _:_)
    = Right (replicate np (Left newd),(newd,newg,x,num,cont))
    where newg = concat (map theRight srs)
          newd = zipWith (-) gds newg
          num  = prVV newg newg
          gds  = prEV (num/gg) d
          np   = length srs

prEV e v   = zipWith (e*) v
prVV v1 v2 = sum (zipWith (*) v1 v2)
prMV m v   = zipWith prVV m (repeat v)
```

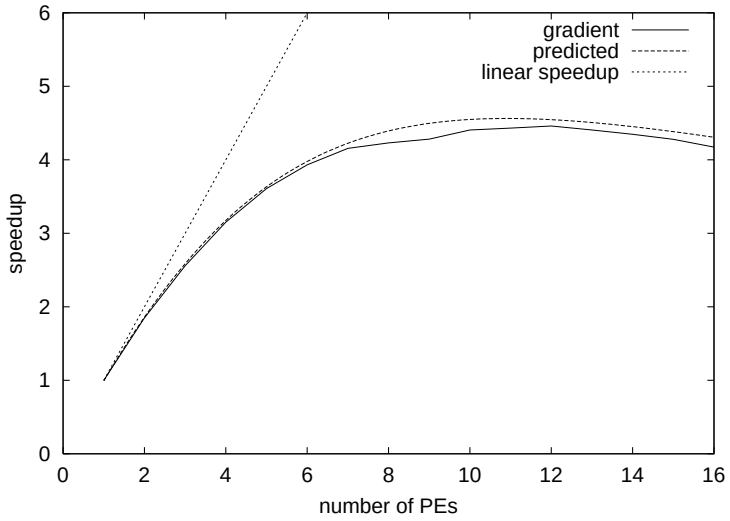


Figura 7.8: Aceleraciones del gradiente conjugado

Resultados obtenidos

La Figura 7.8 muestra las aceleraciones obtenidas para un sistema de 600 ecuaciones, cuyo tiempo de ejecución secuencial es de 684 segundos. El motivo de los malos resultados se debe a la baja proporción cómputo-comunicaciones. Siendo n el número de ecuaciones, y p el de procesadores, en cada iteración el proceso principal debe comunicar $\mathrm{O}(n{*}p)$ datos, mientras que el cómputo de cada trabajador es $\mathrm{O}(n^2/p)$. Así pues, sólo se obtendrán buenos resultados cuando se cumpla que $n \gg p^2$. Por dicho motivo, la aceleración para $p = 4$ es bastante buena, pero deja de serlo para valores alrededor de 10.

En términos de los modelos de coste, el principal cuello de botella es el coste t_{PackI} que aparece en t_{parent}. Dicho coste es proporcional al número de procesadores, pues el proceso principal debe empaquetar el mismo vector para cada uno de los trabajadores, y debe hacerlo en cada una de las I iteraciones. En nuestro ejemplo concreto, el número I de iteraciones es 1200, el tamaño de los vectores es de 600 elementos, y el sobrecoste por cada procesador es de 5.5 segundos. Un segundo cuello de botella menos importante es un cómputo secuencial de 30 segundos para generar el sistema de ecuaciones al comienzo de la ejecución.

La Figura 7.9 muestra un *zoom* del comportamiento paralelo habitual del programa. Puede apreciarse claramente que se alternan fases de cómputo secuencial en el padre, y fases paralelas en los hijos. Ahora bien, debido a que el envío de datos por parte del padre es muy costoso y lento, la reducción en la aceleración resulta más que notable.

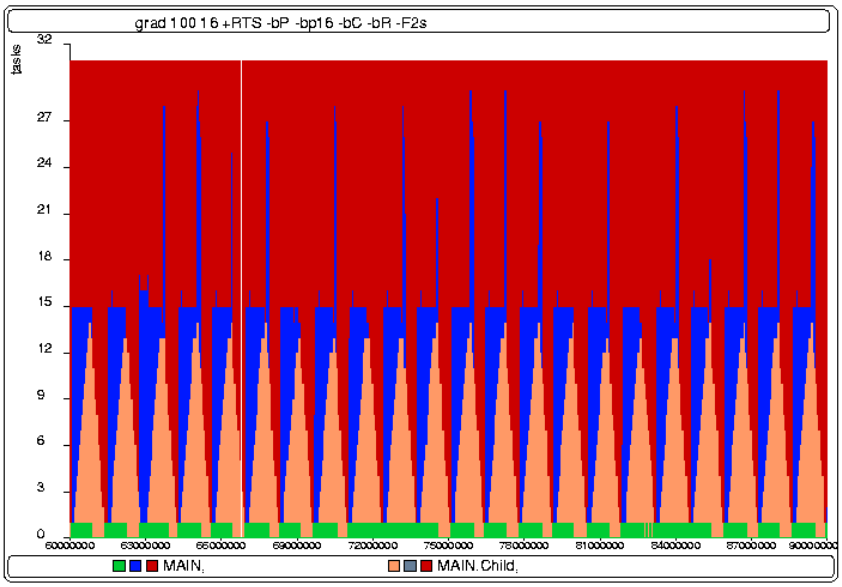


Figura 7.9: Simulación Paradise del gradiente conjugado

7.2.7 Producto de matrices

El producto de una matriz M1 de tamaño $m \times n$, y de otra matriz M2 de tamaño $n \times p$, produce como resultado una matriz M de tamaño $m \times p$, donde cada M(i,j) es el producto escalar de la fila i-ésima de M1 y la columna j-ésima de M2:

```
prMMseq m1 m2 = prMMTr m1 (transpose m2)
prMMTr m1 m2  = [[prVV row col| col <- m2 ] | row <- m1]
prVV row col  = sum (zipWith (*) row col)
```

Algoritmo paralelo

En principio, cada elemento de la matriz resultante puede calcularse en paralelo, pero esto implicaría una cantidad de comunicaciones excesiva. Para poder paralelizar el algoritmo adecuadamente será preciso obtener una buena proporción entre cómputo y comunicaciones.

Si las matrices son de tamaño $n \times n$ y disponemos de p procesadores, deberían crearse sólo p tareas mediante un `map`. Si cada tarea se responsabiliza del cómputo de n/p filas de la matriz resultante, entonces el cómputo de cada proceso sería O(n^3/p), mientras que las comunicaciones de cada proceso serían O(n^2) debido a que sería necesario comunicar a todos los procesos la segunda matriz completa. Aunque pueda parecer una buena proporción

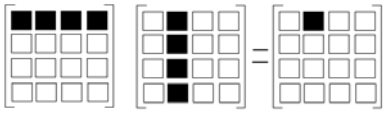


Figura 7.10: Producto de matrices por bloques

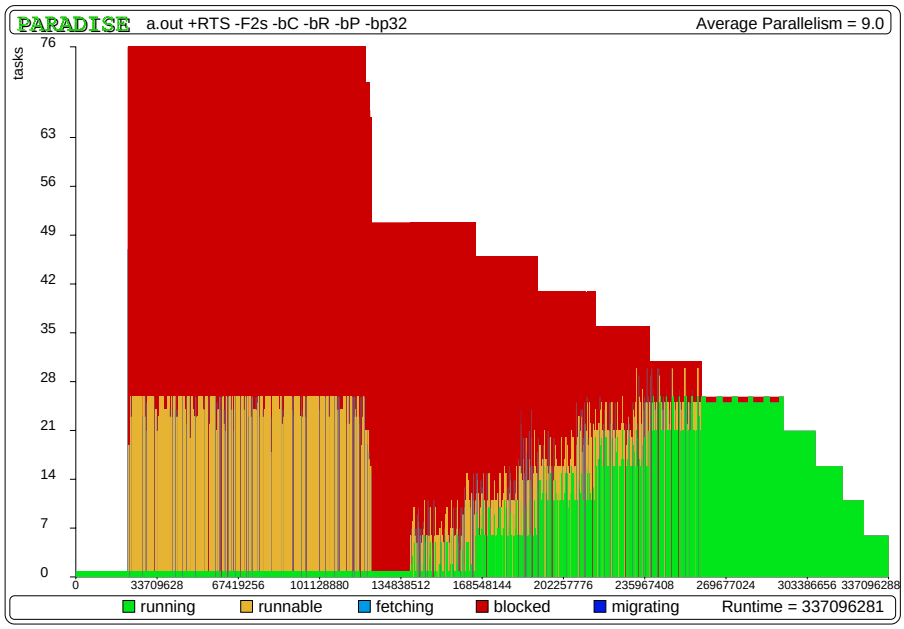


Figura 7.11: Producto de matrices con un `map` paralelo

entre cómputo y comunicaciones, debe tenerse en cuenta que el proceso principal tendrá que enviar una copia de la segunda matriz a cada uno de los procesos trabajadores, por lo que sus comunicaciones serán $O(n^2 * p)$. Por ello, sólo se obtendrán buenas aceleraciones si $n \gg p^2$. Nótese que la granularidad de las tareas es muy regular, por lo que en la implementación Eden utilizaremos `map_naive`:

```
prMM :: Matrix -> Matrix -> Matrix
prMM m1 m2 = concat out
  where out = map_naive (uncurry prMMTr)
                (zip (splitIntoN noPe m1)(repeat (transpose m2)))
```

donde `noPe` representa el número de procesadores disponibles, y `splitIntoN n xs` divide `xs` en `n` sublistas de tamaño casi-igual.

Para reducir las comunicaciones puede utilizarse el algoritmo de Gentleman [Gen78], en el que cada proceso calcula una bloque rectangular de la

matriz final, como se muestra en la Figura 7.10. Utilizando esta división del trabajo los cómputos de cada proceso siguen siendo $O(n^3/p)$, pero sus comunicaciones se reducen a $O(n^2/\sqrt{p})$, puesto que ahora no es necesario recibir la segunda matriz completa. La implementación de esta idea puede realizarse utilizando una versión paralela de `map`, pero eso implicaría que el proceso principal tendría que realizar $(O(n^2 * \sqrt{p}))$ comunicaciones al comienzo del cómputo, por lo que podría convertirse en un cuello de botella. De hecho, la simulación Paradise de la Figura 7.11 muestra un cuello de botella en el proceso principal, que no da abasto para suministrar datos a los 25 trabajadores existentes. Dicho cuello de botella representa la tercera parte del cómputo, por lo que no pueden esperarse buenas aceleraciones.

Una solución mejor pasa por emplear una topología en forma de toroide, de modo que inicialmente cada proceso sólo reciba del padre un bloque de cada matriz a multiplicar, y posteriormente reciba el resto de bloques de sus vecinos: los bloques de la primera matriz se transmitirán de izquierda a derecha en el toro, mientras que los de la segunda matriz lo harán de arriba a abajo. Con esta aproximación se consigue que el proceso principal sólo realice $O(n^2)$ comunicaciones. Nótese que hay una gran diferencia entre $O(n^2 * \sqrt{p})$ y $O(n^2)$, pues en el segundo caso las comunicaciones no dependen del número de procesadores, por lo que se mejora la escalabilidad. El único inconveniente de la solución que utiliza el toroide es que hace falta que el número de procesadores sea un cuadrado perfecto.

El programa Eden necesita especificar el tamaño del toroide (i.e. $\lfloor\sqrt{p}\rfloor$), suministrar las funciones que dividen las matrices en bloques y que combinan los bloques para obtener la matriz resultante, y definir la función a utilizar en cada proceso trabajador. Dicha función debe realizar una lista de productos de bloques (uno por cada par de bloques que reciba) y sumarlos todos. El número de productos realizados en cada nodo coincidirá con el tamaño del toroide.

```
mult :: Matrix -> Matrix -> Matrix
mult m1 m2 = torus torusSize split combine (mult' torusSize) (m1,m2)
  where torusSize = (floor . sqrt . fromInt) noPe
        combine   = concat . (map (foldr (zipWith (++)) (repeat [])))
        split     = ...

-- Function performed by each worker
mult' :: Int -> ((Matrix,Matrix),[Matrix],[Matrix]) ->
         (Matrix,[Matrix],[Matrix])
mult' size ((sm1,sm2),sm1s,sm2s) = (result,toRight,toBottom)
  where toRight  = take (size-1) (sm1:sm1s)
        toBottom = take (size-1) (sm2':sm2s)
        sm2'     = transpose sm2
        sms      = zipWith prMMTr (sm1:sm1s) (sm2':sm2s)
        result   = foldl1' addMatrices sms
```

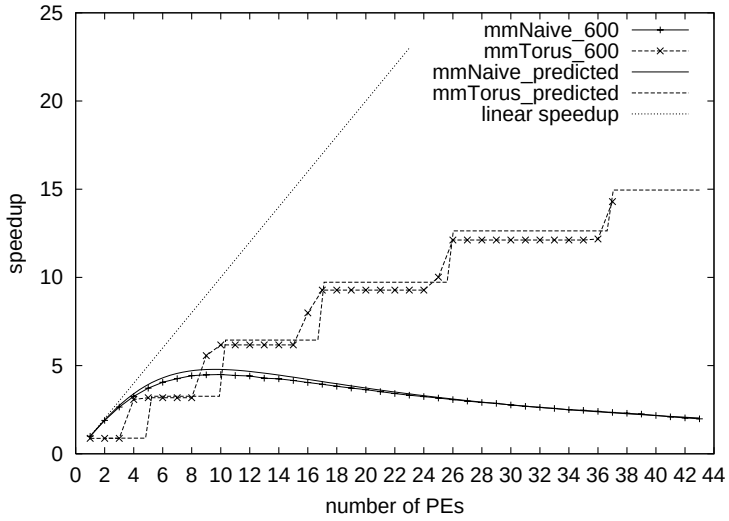


Figura 7.12: Aceleraciones del producto de matrices

donde `split` es una simple función Haskell que divide las matrices en bloques (véase la Figura 7.10) y los desplaza convenientemente para que encajen adecuadamente en el toroide.

Resultados obtenidos

La Figura 7.12 muestra las aceleraciones obtenidas. Las medidas se realizaron con matrices cuadradas de 600×600, siendo el tiempo de ejecución secuencial 221 segundos. Puede apreciarse que la escalabilidad de la versión ingenua no es satisfactoria, mientras que la del algoritmo que utiliza el toroide es mucho mejor, siendo el motivo de esta diferencia la relación cómputo-comunicaciones.

La predicción realizada para la versión ingenua se ajusta bastante bien a los resultados reales. En este caso, el parámetro dominante del modelo de coste es t_{packI}, pues se necesitan 2.3 segundos para empaquetar la segunda matriz completa. Dado que este parámetro se multiplica por p en L_{init}, a medida que incrementa el número de procesadores se incrementan también los sobrecostes. Es por ello que no sólo no se mejora el tiempo de ejecución con muchos procesadores, sino que incluso empeora a medida que añadimos procesadores.

Las predicciones para la versión que usa el toroide también son bastante precisas. Ahora bien, fallan cuando el número de procesadores es un cuadrado perfecto. El motivo es muy simple: el modelo de coste supone que el proceso principal no comparte procesador con ningún proceso trabajador, pero eso no sucede en nuestras medidas cuando el número de procesadores es un cuadrado perfecto. Si se quisiera podría modificarse trivialmente el

modelo de coste para distinguir este caso.

La principal razón por la que la escalabilidad de la versión del toroide es mejor nos la da el modelo de coste. En el toroide L_{init} no depende de forma relevante del número de procesadores, pues t_{packC} es proporcional a $1/p$. Por tanto, a medida que incrementa p disminuye el tamaño de los bloques, por lo que las comunicaciones totales iniciales siguen siendo las mismas.

7.2.8 Cálculo de fuerzas entre n partículas

Descripción del problema

Dado un conjunto de n partículas, queremos calcular la fuerza global que ejercen las demás partículas sobre cada una de ellas. El vector de fuerza f_i que actúa sobre cada partícula se define mediante

$$f_i = \sum_{j=1}^{n} F(x_i, x_j)$$

donde $F(x_i, x_j)$ denota la atracción o repulsión existente entre las partículas x_i y x_j. Se considerará que para todo x_i se cumple que $F(x_i, x_i) = 0$.

Algoritmo paralelo

El problema puede paralelizarse empleando n tareas independientes, cada una encargada de calcular la fuerza que actúa sobre cada partícula. Con el objetivo de obtener una granularidad más razonable, a cada tarea no se le asignará el cómputo de fuerzas sobre una única partícula, sino sobre n/p, siendo n el número de partículas y p el número de procesadores. Para reducir las comunicaciones del proceso padre, en lugar de emplear un `map` paralelo usaremos un anillo. Inicialmente, el padre enviará a cada trabajador su correspondiente conjunto de partículas, que los trabajadores conservarán durante todo el cómputo. Cada trabajador calculará las fuerzas que se ejercen entre sí las partículas que se le han asignado. Posteriormente, en cada iteración los trabajadores enviarán a su vecino el conjunto de partículas que hayan recibido en la iteración anterior, y calcularán las fuerzas que ejercen las nuevas partículas sobre su conjunto de partículas. Dichas fuerzas se sumarán a las ya calculadas en las iteraciones anteriores. El código fuente es el siguiente:

```
force :: [Atom] -> [ForceVec]
force xs = ring noPe splitIntoN concat (force' np) xs
force' :: Int -> ([Atom],[[Atom]]) -> ([ForceVec],[[Atom]])
force' np (local,ins) =  (total,outs)
  where outs         = take (np - 1) (local : ins)
        total        = foldl1' f forcess
        f acums news = zipWith addForces acums news
```

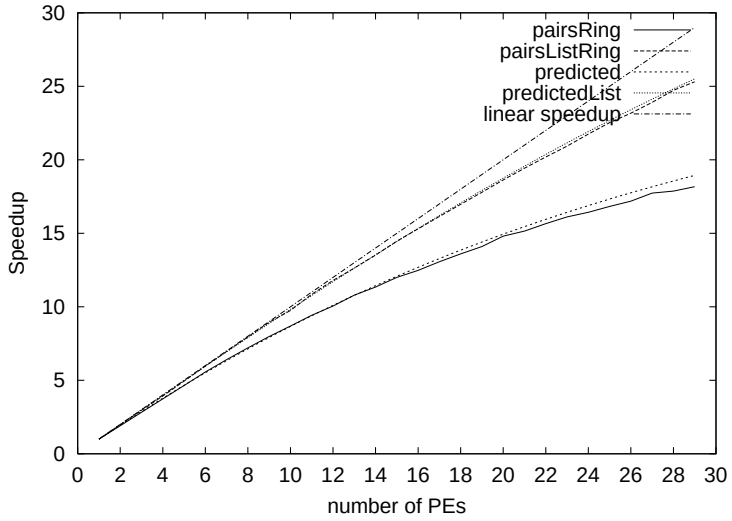


Figura 7.13: Aceleraciones para interacciones entre partículas

```
forcess      = [map (faux ats) local | ats <- (local:ins)]
faux xs y    = sumForces (map (forcebetween y) xs)
sumForces l  = foldl' addForces nullvector l
```

Resultados obtenidos

La Figura 7.13 muestra las aceleraciones obtenidas utilizando 7000 partículas, problema que requiere 194.86 segundos para su resolución secuencial. Las comunicaciones totales de cada proceso son $O(n)$, mientras que sus cómputos son $O(n^2/p)$, siendo n el número de partículas y p el de procesadores. Puede apreciarse que hemos incluido dos conjuntos de medidas. El primero de ellos se corresponde con la implementación exacta que hemos descrito en esta sección. Aunque sus resultados son bastante buenos, existe un cuello de botella secuencial de 2.7 segundos, debido a que el proceso principal debe enviar todas las partículas una a una a los procesos trabajadores, y las fuerzas calculadas por los trabajadores también deben recibirse una a una. El problema está en el tipo de los canales de comunicaciones con los hijos, que es `[Atom]`, por lo que debe emplearse un mensaje distinto por cada átomo que se transmita. El segundo conjunto de medidas se ha realizado tras una modificación trivial del código fuente para que los canales sean de tipo `[[Atom]]`, donde la lista más externa sólo contiene una única lista:

```
splitIntoNl n [xs] = map (:[]) (splitIntoN n xs)
concatl xsss = [concat (map head xsss)]
force np xs = head (ring np splitIntoNl concatl (force' np) [xs])
force' :: Int -> ([[Atom]],[[Atom]]) -> ([[ForceVec]],[[Atom]])
force' np ([local],ins) = ([total],outs)   where ...
```

De esta forma se consigue que el proceso principal envíe todas las partículas de cada hijo con un único mensaje, reduciéndose los tiempos de comunicaciones hasta hacerse despreciables con respecto a los tiempos de cómputo. Ahora la principal ineficiencia desde el punto de vista de la paralelización es el tiempo que necesita el proceso principal para crear un hijo y mandarle los datos correspondientes, que requiere tan sólo 0.03 segundos por cada procesador.

Como puede apreciarse en la Figura 7.13, la mejora obtenida con el uso de listas de listas es más que notable. Por dicho motivo, el programador Eden deberá tener cuidado para no utilizar canales de tipo `[a]` cuando no necesite realmente utilizar *streams*, pues de lo contrario incurrirá en graves sobrecostes por tener que realizar demasiados empaquetamientos, envíos y recepciones.

7.3 Comparación con otros lenguajes funcionales paralelos

Recientemente, en [**LRS$^+$01**] hemos tenido la oportunidad de realizar una comparación detallada entre tres de los lenguajes funcionales paralelos más representativos del estado del arte: PMLS, GpH y Eden. PMLS (véase la Sección 2.3.2) es una extensión con esqueletos `map` y `fold` del lenguaje estricto ML, mientras que GpH es una extensión paralela de Haskell que permite al programador introducir anotaciones sobre paralelismo (véase la Sección 2.4.3).

Para la comparación se han elegido tres ejemplos: un resolutor de sistemas de ecuaciones (`linsolv`), un trazador de rayos, y la suma de los números de Euler. Dichos ejemplos se han implementado en cada uno de los tres lenguajes, utilizando en todos ellos la misma estructura, para facilitar las comparaciones. De hecho, en el caso de GpH y Eden se ha reutilizado el código Haskell, y sólo se ha modificado el código necesario para la especificación del paralelismo. La equidad en la comparación se garantiza por la participación en el estudio de desarrolladores de los tres lenguajes. Más concretamente, los programas y medidas en PMLS han sido desarrollados por Norman Scaife, y los de GpH por Hans-Wolfgang Loidl.

El objetivo es comparar no sólo la eficiencia de los lenguajes, sino también su capacidad expresiva, facilidad de uso y portabilidad a distintas arquitecturas. Comenzaremos presentando los distintos ejemplos por separado, incluyendo para cada uno de ellos los resultados obtenidos en cada uno de los lenguajes, así como las dificultades encontradas en su desarrollo. Posteriormente presentaremos las conclusiones obtenidas tras el estudio comparativo.

En todas las medidas se ha utilizado la misma máquina paralela: un Beowulf de 32 procesadores del que dispone la universidad Heriot-Watt de Edimburgo. Cada uno de los nodos tiene un procesador Celeron a 533MHz,

128kB de memoria cache, 128MB de memoria RAM y un disco duro de 5.7GB. El sistema operativo de todos los nodos es Linux RedHat 6.2, y la interconexión de los procesadores se realiza por medio de un *switch fast Ethernet* a 100Mb/s, siendo 142μs la latencia del sistema. Aunque la máquina paralela dispone de 32 procesadores, las medidas se han realizado con sólo 16, para no entrar en conflictos con otros usuarios.

7.3.1 LinSolv

Descripción del problema

El algoritmo `linsolv` encuentra una solución exacta a un sistema de ecuaciones de la forma $Ax = b$ donde $A \in \mathbf{Z}^{n \times n}, b \in \mathbf{Z}^n$. A diferencia de otros algoritmos numéricos que suelen producir una solución aproximada utilizando números en coma flotante con una cierta precisión, este algoritmo encuentra la solución exacta, trabajando con enteros de longitud ilimitada.

Para calcular la solución a un sistema de ecuaciones, el algoritmo emplea un enfoque de *múltiples imágenes homomórficas*. Este enfoque, habitual en el área de álgebra computacional, realiza los siguientes pasos:

1. Proyectar los datos de entrada sobre varias imágenes.
2. Calcular la solución en cada una de ellas.
3. Combinar los resultados de las imágenes para obtener un resultado en el dominio original.

En el caso de `linsolv` la estructura de la implementación es la que se aprecia en la Figura 7.14.[1] Los datos de entrada se proyectan módulo distintos números primos, se resuelve el sistema para cada primo, y finalmente se combinan los resultados aplicando el teorema chino de los restos (CRA, véase [Lip71] para una descripción matemática detallada del algoritmo). La única restricción sobre cada primo es que el determinante de la matriz de coeficientes no debe ser cero módulo el primo. Además, el número de imágenes necesarias depende del tamaño de los datos de entrada. La estructura principal del programa Haskell, originalmente presentado en [Loi97], es la siguiente:

```
linSolv :: SqMatrix Integer ->              -- nxn matrix A
           Vector Integer ->                -- n vector b
           (Vector Integer,Integer,Integer) -- n vector x | A*x=b
linSolv a b = x     where
   {- Step1: forward mapping -}
    ...
```

[1] El autor tanto de esta gráfica como la descripción del algoritmo, es Hans-Wolfgang Loidl, que amablemente me ha autorizado para reutilizarlas.

```
{- Step2: Computation of solutions in Z/p -}
 ...
 -- Infinite list of hom. solutions of a*x=b in Z_p
 xList = map get_homSol primes

 get_homSol :: Integer -> [Integer]
 get_homSol p = p : modDet : pmx
   where b0 = toHom p b
         a0 = toHom p a
         modDet = toHom p (determinant a0)
         pmx = [ toHom p (determinant (replaceColumn j a0 b0 ))
               | j <- [jLo..jHi] ]
         ((iLo,jLo),(iHi,jHi)) = matBounds a

{- Step3: lifting via list-based CRA -}
 ...
 primeList = projection 0 xList -- primes (bases for the hom ims)
 detList   = projection 1 xList -- dets in all hom ims
 det       = snd (list_cra pBound primeList detList detList)
 x_i i     = snd (list_cra pBound primeList x_i_List detList)
             where x_i_List = projection (i+2) xList
 -- overall solution:
 x = vector (map x_i [0..n-1])
 ...
```

La función `toHom` realiza la proyección sobre las imágenes, mientras que `xList` es una lista infinita de los resultados en todas las imágenes. La función `list_cra` combina los resultados de las imágenes aplicando el algoritmo chino de los restos.

Este enfoque es apropiado para operaciones sobre enteros de longitud ilimitada debido a que el dominio original es el conjunto de todos los posibles enteros $\mathbf{Z}$, mientras que las imágenes homomórficas están en $\mathbf{Z}$ módulo p, denotado $\mathbf{Z}_p$, donde p es un número primo. La ventaja se aprecia cuando los números de entrada son muy grandes y cada número primo cabe en una única palabra física. En ese caso la aritmética en las imágenes se reduce a simples operaciones con enteros cuyo resultado siempre cabe en una palabra máquina, por lo que no hay que pagar costes adicionales para mantener una precisión ilimitada. Dichos costes sólo volverán a aparecer cuando se aplique CRA para combinar los resultados de las imágenes.

Algoritmo paralelo

Dado que tenemos que resolver el mismo problema en distintas imágenes homomórficas, ello puede hacerse en paralelo. Ahora bien, existen dos dificultades en la paralelización: (1) debe garantizarse que se calculan nuevos resultados si alguno de los primos no es exitoso, es decir, si el determinante de la matriz A módulo dicho primo es cero; y (2) deben evitarse cuellos de

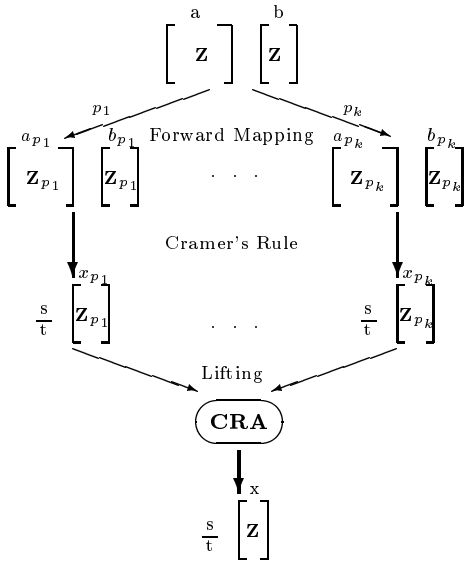


Figura 7.14: Estructura del algoritmo `linsolv`

botella secuenciales en la fase final de combinación de resultados.

GpH La versión escrita en GpH simplemente asocia una estrategia a la expresión principal del código secuencial. Dicha estrategia es la siguiente:

```
strat =
  \ res ->
    rnf noOfPrimes                                          `seq`
    parListN noOfPrimes par_sol_strat xList                 `par`
    parList rnf xs
    where par_sol_strat :: Strategy [Integer]
          par_sol_strat = \ (p:modDet:pmx) -> rnf modDet `seq`
                                              if modDet /= 0
                                                then parList rnf pmx
                                                else ()
```

donde `xList` es una lista infinita de tuplas que representan los resultados de todas las imágenes homomórficas, asociándolas con el primo correspondiente. La estrategia trata de adivinar el número de primos `noOfPrimes` que será necesario, y utiliza la estrategia `parListN` para generar paralelismo sobre un segmento de la lista `xList`. Cuando se detecta que uno de los primos no ha sido exitoso por ser el determinante correspondiente cero, `list_cra` evaluará más resultados demandando un elemento más de la lista. La estrategia final `parList rnf x` especifica que todos los elementos deben combinarse en paralelo.

Eden La versión Eden inicial de este programa no fue desarrollada por el autor de esta tesis, sino por Ulrike Klusik, si bien el doctorando modificó la versión Eden, simplificándola notablemente, y mejorando la obtención de paralelismo. El único cambio necesario en el código fuente es sustituir un `map` por una versión paralela del mismo, más concretamente por una versión basada en trabajadores replicados:

```
xList_all = map_rwND get_homSol primes

xList = filter lucky xList_all
```

donde `map_rwND` es una variante no-determinista de `map_rw` ideada por Ulrike Klusik. La única diferencia está en que la lista de resultados no se ordena mediante `sortMerge`, sino que se devuelve en el orden en el que se computara. En este caso es irrelevante qué resultados se han calculado antes, pues es suficiente con saber que el número de resultados parciales es suficientemente alto, por lo que no necesitamos determinismo. Nótese que esta versión es especulativa, se realizan cómputos que no sean relevantes para el resultado final. A cada proceso se le envían nuevos primos en cuanto termina su cómputo anterior, y una vez recibido un primo puede comenzar a resolver el sistema de ecuaciones módulo dicho primo. En cuanto el gestor principal recibe la suficiente cantidad de soluciones, no necesita más, por lo que pueden descartarse los trabajos que estén realizando todos los trabajadores en ese momento, que habrán resultado inútiles. Nótese que sólo se realiza trabajo especulativo al final del cómputo, en la última tarea asignada a cada procesador. Así pues, dichos cómputos no hacen perder tiempo de proceso, pues sólo se realizan cuando no había disponible ninguna otra tarea no-especulativa.

PMLS: La principal dificultad en PMLS está en el manejo de los primos no válidos. Dado que SML es un lenguaje estricto, no es posible demandar nuevos primos durante la evaluación del esqueleto `map`. La solución empleada en PMLS consiste en iterar la fase de generación de imágenes hasta que el número de primos válido sea correcto. La estructura básica del programa PMLS es la misma, salvo por el hecho de que se añade dicha fase de iteración:

```
(* Solve ax = b modulo p *)
fun gen_xList a b p =
  let
    val (a0,b0) = (matHom p a,vecHom p b)
    val modDet = modHom p (determinant a0)
    val ((iLo,jLo),(iHi,jHi)) = matBounds a
    val pmx =
      fxlist jLo (jHi-jLo+L1)
        (fn j => modHom p (determinant (replaceColumn j a0 b0)))
  in
```

```
    p::modDet::pmx
  end

(* Iterative forward mapping phase *)
fun getSols xList [] = xList
  | getSols xList primes =
  let
    val xList' = map (gen_xList aN bN) primes
    val noUnlucky = countUnlucky xList'
    val xList' = filter (not o isUnlucky) xList'
    val primes' = additionalprimes primes noUnlucky
  in
    getSols (xList@xList') primes'
  end
val xList = getSols [] (primesuptomaxprod pBound)

(* Combination via CRA *)
val detList = projection 1 xList
val det = list_cra pBound primes detList detList
fun x_i i =
  let
    val x_i_List = projection (i+2) xList
  in
    list_cra pBound primes x_i_List detList
  end
val x = seqmap x_i (fxlist 0 n (fn x => x))
```

Resultados obtenidos

Como problema de entrada hemos usado una matriz densa de tamaño 20×20, con enteros de precisión arbitraria y una densidad del 90%. El tiempo de ejecución en Eden fue de 528.6 segundos, mientras que en GpH fue 771.7 y en PMLS 940.9 segundos. La razón por la que GpH es más lento que Eden se debe a que, aunque utiliza una versión más moderna y eficiente del compilador GHC, aún no permite utilizar el recolector de memoria generacional de GHC, lo cual ralentiza las ejecuciones de aplicaciones intensivas en memoria, como es el caso en este ejemplo. La Figura 7.15 muestra las aceleraciones relativas obtenidas en el Beowulf de Heriot-Watt. Puede apreciarse que Eden obtiene la mejor aceleración (13.1 con 16 procesadores). Los resultados tanto de GpH como de PMLS son también bastante buenos, pero algo inferiores (12.74 para GpH y 12.3 para PMLS con 16 procesadores). Nótese que el tiempo de ejecución en Eden era mucho mejor que en GpH y PMLS, por lo que desde el punto de vista de la comparación, la aceleración de Eden es bastante mejor que la de los otros lenguajes, pues tiene más mérito paralelizar adecuadamente una versión cuya parte de cómputo está más optimizada, pues el efecto de las comunicaciones es más significativo en dicho caso.

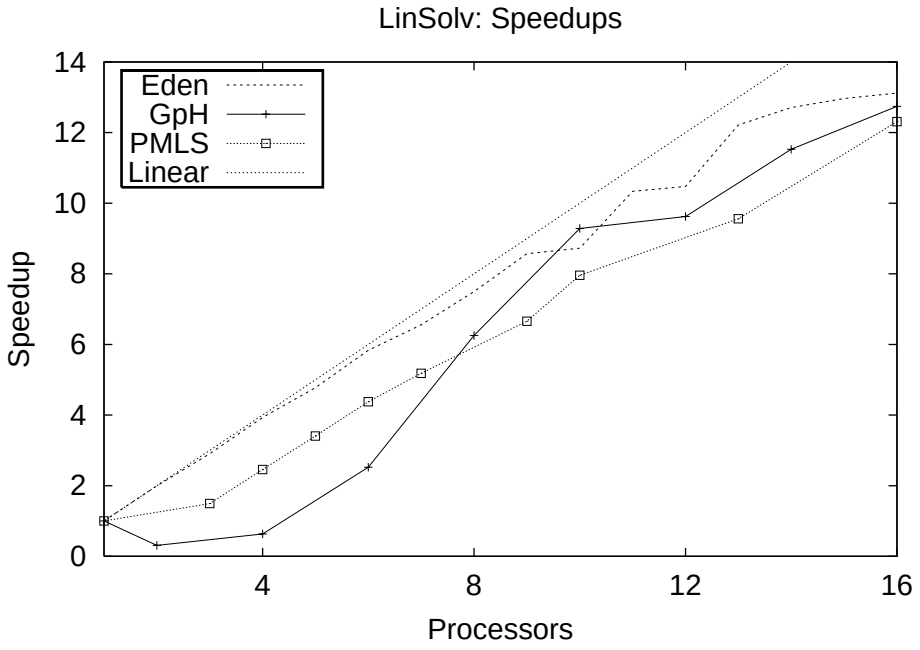


Figura 7.15: Aceleraciones de `linsolv`

Por lo que respecta a GpH, la principal anomalía es una ralentización del cómputo con menos de cinco procesadores, que se debe a problemas en la distribución de los datos y tareas entre procesadores. Por su parte, PMLS tiene un comportamiento bastante regular.

Nótese que la gráfica de Eden no es completamente regular, como sucedía en las aplicaciones que se han presentado hasta el momento. La razón está en que el número de tareas disponibles no es suficientemente alto, por lo que los trabajadores replicados no siempre consiguen un reparto de carga óptimo. Por dicho motivo, para determinados números de procesadores no se mejora significativamente la eficiencia. Por ejemplo, con 16 procesadores se obtiene una aceleración de 13.1, mientras que con 15 se obtiene 12.9, pero esto no significa que la aceleración no mejore con más procesadores, sino sólo que ese valor concreto no mejora particularmente los resultados. De hecho, con 30 procesadores la aceleración es de 23.8.

7.3.2 Trazador de rayos

Descripción del problema

El problema es exactamente el mismo que el descrito en la Sección 7.2.2.

Algoritmo paralelo

Como ya se vio en la Sección 7.2.2, este algoritmo puede paralelizarse explotando un simple `map` sobre las líneas de la ventana a visualizar.

GpH La implementación en GpH usa una estrategia de paralelización de `map`. Para incrementar la granularidad de las tareas, la estrategia dispone de un parámetro `chunk` para especificar cuántas líneas agrupar en una única tarea:

```
ray :: Int -> Int -> Int -> [Sphere] -> [[((Int, Int), Vector)]]
ray chunk x y world = map (do_line world) sizes_y
                      `using` parListChunk chunk rnf
```

PMLS El programa en PMLS está formado por dos esqueletos `map` anidados, pero se ha deshabilitado el anidamiento de esqueletos, de modo que sólo se explote paralelismo en el más externo, pues de lo contrario la granularidad sería muy baja. Además, para el `map` externo se agruparán varias líneas para formar cada una de las tareas. Nótese que este problema es ideal para PMLS, pues es un `map` puro, y el problema puede paralelizarse en tiempo de compilación. El código fuente es el siguiente:

```
fun ray x y world =
  let val (firstray, scrnx, scrny) = camparams x y
      fun do_pixel ij =
        let  val (i,j) = (ij div 1000,ij mod 1000)
        in ((i,j),tracepixel world lights (real i) (real j)
                              firstray scrnx scrny)
        end
      val ind = indxs 0 (x - 1) 0 (y - 1)
  in map (map do_pixel) ind
  end
```

Eden El programa Eden que se utilizó en las comparaciones era ligeramente distinto al mostrado en la Sección 7.2.2, aunque las aceleraciones son prácticamente iguales. La diferencia radica en que en lugar de utilizar la técnica de los trabajadores replicados hemos utilizado la del autoservicio (véase la Sección 6.3.2). Esta decisión se tomó principalmente para mostrar la versatilidad del lenguaje Eden a la hora de especificar algoritmos paralelos, y no para mejorar la eficiencia. El programa es el siguiente:

```
ray :: Int -> Int -> Int -> [Sphere] -> [[String]]
ray np x y world = shuffle outps
  where outps     = [ (process i -> f_dm i) # () | i <- [0..np-1]]
                      `using` spine
        f_dm n _  = map (do_line world) (takeEach np (drop n sizes_y))
```

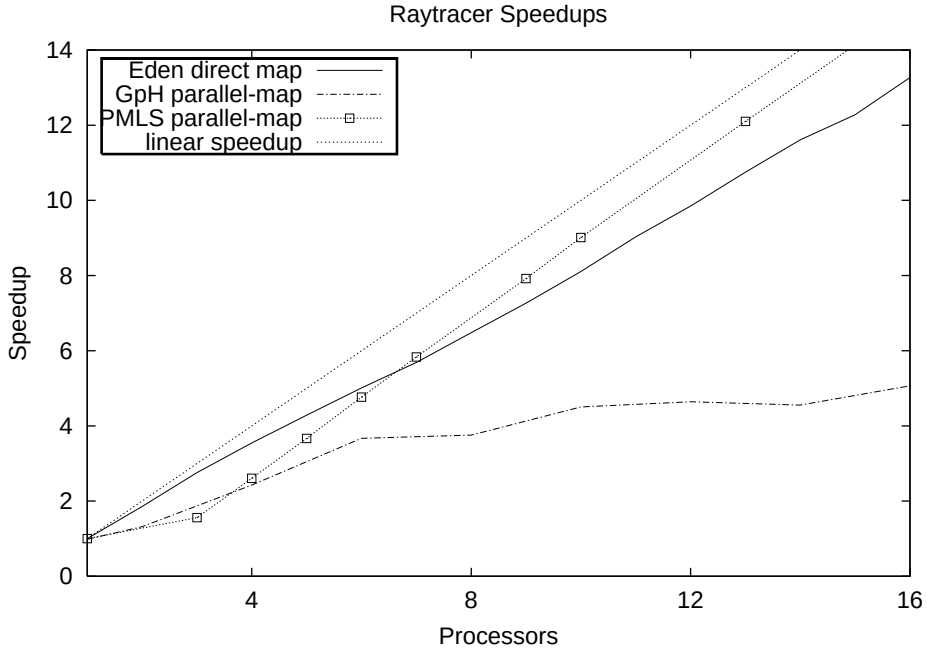


Figura 7.16: Aceleraciones del trazador de rayos

donde la función `f_dm` representa el trabajo a realizar en cada procesador. La función `takeEach` reparte el trabajo entre los procesos mediante un método *round-robin*, mientras que `shuffle` combina los resultados obtenidos.

Resultados obtenidos

La Figura 7.16 recoge las aceleraciones obtenidas con el mismo problema que se usó en la Sección 7.2.2: la escena está formada por 640 esferas, y el tamaño de la ventana es de 350×350. El tiempo de ejecución secuencial en los tres lenguajes es muy parecido: 176.99 segundos en Eden, 163.31 en GpH y 172.10 en PMLS. GpH es ligeramente mejor que Eden debido a que utiliza la versión más reciente del compilador GHC, mientras que la eficiencia de PMLS en este ejemplo está entre las de GpH y Eden.

Para mejorar la granularidad, las tareas creadas tanto por GpH como PMLS constaban de 10 líneas cada una. Dado que el problema es muy regular, se adapta muy bien al esquema de paralelización estática de PMLS, como se refleja en la gráfica. Las aceleraciones de Eden (13.3 con 16 procesadores) son casi tan buenas como las de PMLS (15.1 con 16 procesadores), pero Eden tiene que pagar algunos sobrecostes debidos a utilizar una paralelización dinámica, mientras que PMLS distribuye las tareas en tiempo de compilación. Por su parte, GpH queda a gran distancia de Eden y PMLS, con una aceleración de sólo 6.77. Según los desarrolladores de GpH, una de

las posibles razones del mal comportamiento de GpH es un mal reparto de carga debido a que todas las tareas las genera el mismo procesador, y el resto de procesadores no *roban* suficientes tareas para equilibrar la carga.

A pesar de que PMLS obtiene las mejores aceleraciones en este ejemplo, hay que mencionar que la primera versión de PMLS que se probó generaba aceleraciones mucho peores que las de Eden y GpH. Debido a una limitación[2] en el compilador, fue necesario una intervención manual durante el proceso de compilación para poder sacar provecho al paralelismo presente en el programa.

7.3.3 Suma de números de Euler

Descripción del problema

El problema es el mismo que se describió en la Sección 7.2.3.

Algoritmo paralelo

Como se vio en la Sección 7.2.3, este algoritmo es un ejemplo típico del esquema *map & reduce*, por lo que su paralelización es trivial.

GpH La estructura de la paralelización en GpH es la de un `map`, donde cada hebra realiza otro `map` de la función `euler` y suma todos los resultados que calcula. La lista de tareas se genera dividiendo en bloques de tamaño `c` la lista de entrada:

```
sumEuler :: Int -> Int -> Int
sumEuler c n = sum ([(sum . map euler) x | x <- splitAtN c [n,n-1..0]]
                    `using` parList rnf)
```

Nótese que la lista de entrada se genera de mayor a menor. Este hecho es muy relevante en GpH, pues permite que las tareas de grano más grueso se creen al principio. Esto es importante porque a la hora de *robar* tareas siempre se eligen primero las que primero se crearon. Por tanto, el programa consigue que se distribuyan apropiadamente las tareas de grano más grueso, por lo que se obtiene un buen reparto de carga gracias a este detalle.

PMLS En principio, este problema debería paralelizarse trivialmente en PMLS, pues es una combinación de los dos esqueletos que suministra el lenguaje: `map` y `fold`. Ahora bien, las versiones más simples que se emplearon no consiguieron buenos resultados, debido fundamentalmente a cuellos de botella en la aplicación del paso `fold`. Tras varias aproximaciones, la versión definitiva utilizada fue la que se muestra a continuación, que requiere añadir tipos de datos para distinguir distintas fases del cómputo:

[2] La limitación está relacionada con la currificación y descurrificación de funciones durante el proceso de compilación.

```
(* Argument list for folded versions *)
datatype EI = E of I.int | I of I.int
val nListEI = seqmap (fn n => I n) nList
fun eulerEI (I n) = euler n
  | eulerEI (E n) = n
fun eiplus (I i1,I i2) = E ((euler i1) + (euler i2))
  | eiplus (I i,E e) = E ((euler i) + e)
  | eiplus (E e,I i) = E (e + (euler i))
  | eiplus (E e1,E e2) = E (e1 + e2)

(* Nested parallel fold and parallel map *)
val nListEIS = split c nListEI
fun sumeulerEI (eil1,eil2) =
  let val resultList = map eulerEI (eil1 @ eil2)
  in [seqfold eiplus (E L0) resultList]
  end
val [E result] = fold sumeulerEI [E L0] nListEIS
```

Eden El programa Eden es el mismo que el usado en la Sección 7.2.3, y también se utilizó 10 como valor del parámetro `th`:

```
sumEuler     :: Int -> Int -> Int
sumEuler th n = map_reduce_as th euler (+) 0 [n,n-1..1]
```

Resultados obtenidos

La Figura 7.17 muestra las aceleraciones obtenidas con los tres lenguajes, donde tanto GpH como PMLS utilizaron tareas de tamaño 100. Como era de esperar, los tiempos de ejecución secuenciales fueron similares en Eden (70.48 segundos) y GpH (71.93 segundos), ya que ambos utilizan básicamente el mismo programa, y compiladores muy similares. La ejecución del programa PMLS fue bastante más lenta: 165.9 segundos.

A diferencia de los resultados obtenidos en la aplicación anterior, ahora GpH muestra aceleraciones comparables a las de Eden y PMLS. De hecho, PMLS no supera a GpH hasta los 12 procesadores. A partir de ahí, la aceleración de GpH se estabiliza, mientras que la escalabilidad de Eden y PMLS sigue siendo buena. La razón de la mejoría de GpH se debe a que en este caso sí que saca partido de su distribución dinámica de recursos, pues las tareas de grano más grueso se crean al principio, y además las comunicaciones necesarias son bastante bajas. De todas formas, los mejores resultados los consigue Eden, y lo hace empleando una paralelización trivial. Nótese que aunque la diferencia en aceleraciones no es muy grande entre Eden y PMLS (13.9 en Eden y 12.7 en PMLS con 16 procesadores), realmente el tiempo de ejecución secuencial en Eden es menos de la mitad del de PMLS, por lo que las aceleraciones de Eden son aún más meritorias.

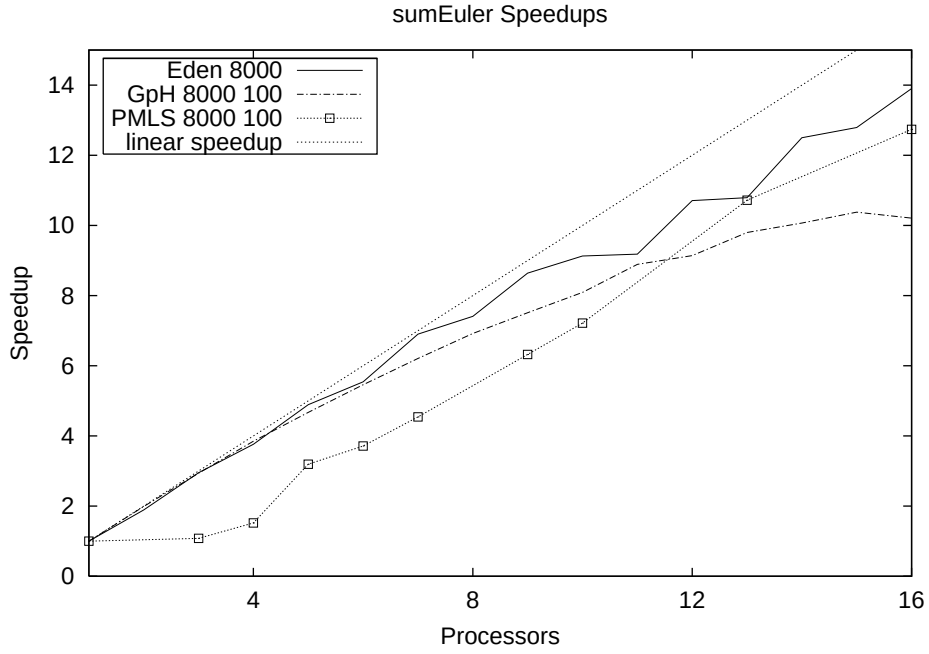


Figura 7.17: Aceleraciones de la suma de números de Euler

7.3.4 Comparación de los tres lenguajes

Aceleraciones y tiempos de ejecución En términos de aceleraciones, cuando el programa es altamente regular (trazador de rayos) PMLS obtiene los mejores resultados, seguido de cerca por Eden. En el resto de casos Eden es ligeramente mejor que PMLS. Puede decirse que, en general, ambos lenguajes obtienen resultados similares, mientras que GpH está siempre por detrás de ambos, estando incluso muy por detrás de ellos en el trazador de rayos.

En lo que se refiere a tiempos de ejecución secuenciales, GpH y Eden son muy similares, debido a que ambos se basan en el compilador GHC. La diferencia entre ellos está en que GpH utiliza la versión más reciente de GHC, mientras que Eden usa una versión más antigua, por lo que la eficiencia de GpH debería ser ligeramente mejor. Ahora bien, GpH todavía no puede utilizar el recolector de basura generacional de GHC, sino que sólo dispone del recolector de dos espacios, por lo que pierde eficiencia comparado con Eden. Así pues, en aplicaciones intensivas en memoria Eden es mejor, mientras que en el resto GpH es ligeramente más eficiente. Con respecto a PMLS, su tiempo secuencial sólo era comparable al de GpH y Eden en el caso del trazador de rayos, siendo bastante más lento en los otros dos ejemplos.

Capacidad expresiva La capacidad expresiva de Eden es muy superior a la de GpH y PMLS: PMLS sólo puede explotar paralelismo a partir de `map` y `fold`, mientras que GpH no tiene capacidad para expresar topologías de comunicaciones debido a la ausencia del concepto de proceso.

Tiempo de desarrollo de programas paralelos Debido a su alto grado de abstracción, los tres lenguajes permiten rápidas paralelizaciones. En GpH suele ser suficiente con anotar los programas con determinadas estrategias de evaluación. Idealmente, en PMLS no hace falta modificar absolutamente nada el código secuencial, pues el paralelismo se extrae directamente de las aplicaciones de `map` y `fold`. Desgraciadamente, en la práctica suele ser necesario modificar sustancialmente el código fuente para obtener buenos resultados. En lo que respecta a Eden, cuando se utilizan esqueletos simples los cambios son mínimos (por ejemplo reemplazar `map` por `map_rw`), mientras que hacen falta cambios más sustanciales cuando se quieren aplicar topologías más elaboradas como el toroide.

Disponibilidad de los lenguajes GpH se distribuye como una parte más del compilador GHC, y está disponible incluso para las versiones más modernas de GHC, gracias a lo cual dispone de más usuarios que Eden y PMLS. En la página web del proyecto Eden puede obtenerse una versión del correspondiente compilador, aunque para obtener las versiones más actualizadas es necesario contactar con los desarrolladores. El proceso de instalación de Eden es básicamente el mismo que el de GHC, con algunas pequeñas complicaciones añadidas. Por último, PMLS no está disponible para su uso general, debido a su complejo sistema de instalación y uso, y no está previsto que se distribuya ni a corto ni a medio plazo.

Comparación general El lector es libre de decidir qué lenguaje es mejor, dependiendo de cuáles sean los criterios que considere más importantes. Desde el punto de vista del autor de esta tesis, Eden es el lenguaje que sale mejor parado del estudio, pues es el único que conjuga buenas aceleraciones con facilidad de uso y disponibilidad del lenguaje. Aunque GpH es fácil de usar, y su disponibilidad es absoluta, sus resultados suelen ser bastante más pobres que los de Eden. Por su parte, aunque PMLS tiene buenas aceleraciones, conseguirlas suele requerir un profundo conocimiento del sistema, y un gran esfuerzo por parte del programador. Además, el compilador aún no es completamente automático.

Merece la pena resaltar que Eden es la solución adecuada gracias al uso de esqueletos. Si no se emplearan esqueletos, un programador no experto en el lenguaje necesitaría emplear mucho tiempo en desarrollar sus aplicaciones empleando directamente abstracciones y concreciones de procesos, y normalmente obtendría peores aceleraciones, pues no sería capaz de diseñar

soluciones tan óptimas como las suministradas en los esqueletos.

Otra ventaja competitiva que merece la pena resaltar es la existencia de modelos de coste en Eden, que permite predecir las aceleraciones que se obtendrán. Nótese que esta predecibilidad también se debe al empleo de esqueletos, y que no podría conseguirse utilizando directamente las construcciones básicas del lenguaje. Al estar basado en esqueletos, PMLS también dispone de modelos de coste, pero no son visibles al programador, sino que sólo los usa el compilador internamente. Por último, resulta impensable predecir el comportamiento de los programas GpH, debido a que el RTS es quien decide dinámicamente qué debe paralelizarse y cómo hacerlo. De hecho, dos ejecuciones consecutivas de un mismo programa GpH pueden tener tiempos de ejecución muy diferentes.

7.4 Conclusiones

En el presente capítulo se ha presentado un conjunto de aplicaciones que pueden considerarse representantes de una amplia gama de esquemas paralelos. Algunas de ellas, como el producto de matrices, el gradiente conjugado o los conjuntos de Mandelbrot, son problemas clásicos en programación paralela. Otras como la suma de los números de Euler, el trazador de rayos o `linsolv` son ejemplos que se están convirtiendo en un vehículo de comparación entre distintos lenguajes funcionales paralelos. El resto de ejemplos (fuerzas entre partículas, algoritmo de Karatsuba, y problema del viajante) son menos clásicos en programación paralela, si bien formarán parte de un futuro banco de pruebas común a distintos lenguajes funcionales paralelos. De hecho, actualmente el grupo de desarrollo de GpH está traduciéndolas a su lenguaje, y probablemente también sean traducidas en breve a PMLS, y potencialmente a otros lenguajes funcionales paralelos.

Las conclusiones básicas que se pueden obtener a partir de los ejemplos mostrados en esta sección son las siguientes:

1. Los algoritmos se implementan de forma clara y concisa en Eden.
2. Las aceleraciones obtenidas son casi siempre aceptables, y en muchos casos muy buenas.
3. Los resultados obtenidos son competitivos, estando a la altura de los de otros lenguajes funcionales paralelos que representan el estado del arte del área.
4. Las aceleraciones son predecibles.

El motivo por el que se cumplen las afirmaciones anteriores no se debe únicamente al lenguaje Eden, sino que resulta fundamental la metodología de programación basada en esqueletos. Los algoritmos se pueden expresar claramente gracias a la existencia de esqueletos en el lenguaje. Si no fuera

así, resultaría tedioso el desarrollo de toda aplicación no trivial. Además, la eficiencia de dichos programas sería peor, a no ser que los programas los desarrollara un programador experto, e invirtiera mayores cantidades de tiempo en el desarrollo. Por último, la predecibilidad también se obtiene gracias al empleo de esqueletos que incorporan modelos de coste.

Resumiendo, podemos asegurar que la programación en Eden cumple la divisa de "aceleración aceptable con poco esfuerzo". Es decir, no se aprovecha al máximo el paralelismo disponible, pero es fácil obtener resultados suficientemente buenos. Así pues, la relación "calidad/precio" es muy buena.

Además de los resultados positivos obtenidos, a partir de las experiencias negativas de este capítulo puede aprenderse también una lección a tener en cuenta cuando se desarrolle cualquier programa en Eden: no deben utilizarse *streams* si no es necesario. En la Sección 7.2.8 quedó patente el hecho de que es mucho más eficiente comunicar muchos datos juntos que hacerlo de uno en uno. Así pues, cuando no necesitemos *streams*, pero el tipo de una de las componentes de entrada o salida de un proceso sea una lista, habrá que convertirla en otro tipo, como por ejemplo una lista de listas, o simplemente añadir un constructor de un nuevo tipo `newtype D a = D a`.

La reflexión anterior puede aplicarse no sólo a las listas, sino también a las tuplas, que también tienen un tratamiento especial en Eden. El programador sólo deberá utilizar tuplas como entrada o salida de un proceso cuando realmente le interese calcular y/o enviar de forma independiente cada una de las componentes. En caso contrario será preferible utilizar una redefinición del tipo de las tuplas, para garantizar que se crea una única hebra y que los datos se envían de forma compacta.

Otra lección que puede aprenderse se refiere a la optimización del propio compilador de Eden. Como era de esperar, los ejemplos en los que se han obtenido peores resultados son aquellos en los que la relación cómputo v.s. comunicaciones era más baja. Dichos malos resultados eran esperables, pero son peores de los que suelen obtenerse con lenguajes de más bajo nivel. Aunque no es posible mejorar las diferencias en órdenes de complejidad, sí que es posible y deseable mejorar las constantes involucradas en los costes de las comunicaciones. En la versión actual de Eden no se realiza ninguna optimización en las rutinas de empaquetamiento y desempaquetamiento, que de optimizarse podrían reducir los tiempos necesarios para comunicar grandes estructuras de datos. Por ejemplo, cuando se transmite un paquete que contiene una lista, se envían no sólo los elementos de la lista, sino también la estructura de la lista propiamente dicha, es decir, todos los constructores `(:)` intermedios. Dado que sabemos que los datos que se envían están en forma normal, no es preciso enviar dicha estructura, sino que basta con enviar la lista de elementos, y la estructura podrá reconstruirse en el receptor del mensaje. Así pues, una de las principales tareas que deberá llevarse a cabo para mejorar el compilador será optimizar las rutinas de empaquetamiento y desempaquetamiento de clausuras.

Capítulo 8

Conclusiones y trabajo futuro

El principal eslogan que defendemos en esta tesis es que el lenguaje Eden permite obtener una eficiencia aceptable con poco esfuerzo. Esta máxima no se cumpliría simplemente por la definición del lenguaje, sino que resultan fundamentales las tres líneas de trabajo que se han expuesto en esta tesis:

1. El desarrollo de perfiladores para detectar y corregir ineficiencias tanto en cómputos secuenciales como paralelos.
2. El desarrollo de un compilador que realice optimizaciones automáticas para mejorar tanto el código secuencial como el paralelo.
3. El desarrollo de una metodología de programación paralela basada en la utilización de esqueletos.

En lo referente a perfiladores, la principal contribución de la tesis es el simulador Paradise. Paradise ha demostrado ser útil no sólo para detectar y corregir ineficiencias en programas concretos, sino que también ha sido una pieza importante en el proceso de mejora del compilador de Eden, ya que su uso apuntó hacia la introducción en el compilador del lanzamiento impaciente de procesos.

La realimentación gráfica que proporciona Paradise es especialmente útil para programadores inexpertos, a los que les suele resultar más complejo razonar acerca del comportamiento real de sus programas. Por su parte, el programador más experimentado encontrará menos atrayente el uso del simulador, salvo en aplicaciones especialmente complejas. Esta distinción entre programadores inexpertos y expertos ha sido vivida en primera persona por el autor de la tesis, quien inicialmente recurría con mucha frecuencia a simulaciones para comprender el comportamiento real de los programas, mientras que en la actualidad no suele emplear la herramienta durante sus desarrollos. De hecho, la principal facilidad de Paradise que el autor usa actualmente es su vertiente didáctica, ya que es una herramienta muy útil para enseñar a los nuevos programadores de Eden cómo utilizar adecuadamente el lenguaje.

Evidentemente, para que un programa paralelo sea eficiente es necesario que su correspondiente versión secuencial también lo sea. Por dicho motivo era fundamental reutilizar la mayor parte de las optimizaciones automáticas que realiza GHC en tiempo de compilación. En esta tesis no sólo hemos conseguido reutilizar las optimizaciones que realiza GHC sobre el código secuencial, sino que también hemos planteado un esquema de compilación que permite introducir optimizaciones sobre el código paralelo. Utilizando dicho esquema, hemos especificado e implementado una transformación automática que lanza impacientemente procesos, de modo que se incremente el grado de paralelismo de los programas, y por tanto se mejore la aceleración obtenida. Asimismo, hemos especificado completamente el conjunto de transformaciones que serían precisas para poder realizar un análisis de *bypassing* que permita minimizar el número de mensajes que se envíen entre procesos. La implementación del lanzamiento impaciente de procesos no sólo ha mejorado muy notablemente el rendimiento de los programas Eden, sino que también nos ha proporcionado conocimientos de bajo nivel acerca de cómo introducir nuevas transformaciones, de modo que resulte sencillo implementar cualquier otro análisis o transformación automática.

Las dos líneas de trabajo expuestas anteriormente (perfiladores y transformaciones automáticas) son muy importantes para mejorar la eficiencia de los programas, pero resultan insuficientes para mejorar el tiempo de desarrollo de las aplicaciones paralelas. Aunque los perfiladores ayudan a detectar ineficiencias rápidamente, y por tanto reducen el tiempo de desarrollo, no sirven para implementar aplicaciones rápidamente. Para ello hemos empleado una metodología basada en esqueletos. La programación con esqueletos proporciona estructuras paralelas de muy alto nivel, de modo que el programador sólo debe especificar el esqueleto concreto a utilizar, sin necesidad de proporcionar detalles de bajo nivel sobre su implementación.

En esta tesis hemos mostrado cómo puede utilizarse Eden tanto como lenguaje de sistema para implementar esqueletos, como lenguaje de aplicaciones para utilizar los esqueletos en programas concretos. Gracias a su segunda vertiente, obtenemos los mismos beneficios que los lenguajes de esqueletos: facilidad de desarrollo de aplicaciones, predictibilidad de las aceleraciones, y eficiencia de las soluciones paralelas debido a utilizar esqueletos previamente optimizados. Ahora bien, a diferencia de los lenguajes de esqueletos, en los que el conjunto de esqueletos disponibles es fijo, Eden es también un lenguaje de sistema, por lo que no impone ninguna restricción al tipo de aplicaciones que pueden resolverse, y de hecho hemos demostrado que es sencillo desarrollar nuevos esqueletos. Así pues, el programador de Eden podrá desarrollar sus propios esqueletos específicos de sus áreas de aplicación. Por dicho motivo, la capacidad expresiva de nuestro enfoque es muy superior a la capacidad expresiva de los lenguajes basados en esqueletos, puesto que no restringimos en ningún sentido el tipo de paralelismo que puede explotarse.

Merece la pena resaltar que ésta es la primera vez que se implementa una librería de esqueletos en un lenguaje funcional paralelo. La razón por la que Eden permite el desarrollo de esta librería radica en la forma en la que conjuga estructuras de alto nivel y de bajo nivel: el proceso `merge` y los canales dinámicos confieren al lenguaje el suficiente bajo nivel como para permitir su uso como lenguaje de sistema, pero sin necesidad de perder el alto nivel de abstracción propio de los lenguajes funcionales.

Gracias a las tres vertientes en las que hemos mejorado la eficiencia del lenguaje, hemos comprobado que las aplicaciones escritas en Eden suelen obtener aceleraciones aceptables, y en muchos de los casos incluso muy buenas. Más importante todavía, el estudio comparativo con los lenguajes GpH y PMLS demuestra que nuestra aproximación es, cuando menos, competitiva con dichos lenguajes, que son representativos del estado del arte en programación funcional paralela y programación con esqueletos respectivamente, y que han estado desarrollándose durante toda una década. De los tres lenguajes, Eden suele ser el que mejores resultados obtiene, tanto en términos de aceleraciones como de tiempos paralelos, y todo ello con unos tiempos de desarrollo muy cortos. Además, es el que tiene una mayor capacidad expresiva, pues PMLS sólo puede explotar sus esqueletos predefinidos, y GpH carece del concepto de proceso, por lo que no permite controlar de forma adecuada ni la distribución de los datos ni la de las tareas.

Tras el trabajo realizado a lo largo de la tesis, la metodología de desarrollo de aplicaciones en Eden puede resumirse de la forma siguiente:

1. Desarrollar una versión secuencial eficiente, utilizando para ello las facilidades que incorpora GHC (perfiladores, interfaces con otros lenguajes, etc.).

2. (a) Si existe un esqueleto adecuado para nuestro problema, utilizarlo con las funciones secuenciales oportunas, utilizando los modelos de coste para elegir la implementación del esqueleto que mejor se adapte al problema concreto.
   (b) Si no existe un esqueleto apropiado:
      i. Si el esquema paralelo puede reutilizarse en otras aplicaciones, se escribirá un nuevo esqueleto.
      ii. En caso contrario se implementará la aplicación utilizando directamente procesos, sin usar esqueletos auxiliares.

3. Compilar utilizando todas las optimizaciones posibles (incluyendo el lanzamiento impaciente de procesos).

4. Si la aceleración es satisfactoria habremos terminado, en caso contrario:
   (a) Utilizar Paradise para detectar las fuentes de las ineficiencias.
   (b) Volver al punto 2.

Parte del trabajo futuro a realizar ya se ha planteado a lo largo de la tesis. Cabe recordar que las aplicaciones que se presentaron en el Capítulo 7 mostraron que los tiempos de empaquetamiento y desempaquetamiento suelen ser los que imponen las mayores limitaciones al grado de paralelismo de las aplicaciones en Eden. Por dicho motivo, una de las principales líneas de trabajo para el futuro cercano será la mejora de las rutinas encargadas de dichos empaquetamientos y desempaquetamientos. La mejora se basará en minimizar la cantidad de datos a empaquetar, aprovechando el conocimiento de que disponemos acerca de las clausuras que se transmiten: sabemos su tipo y sabemos que las estructuras de datos se transmiten en forma normal. Por ejemplo, en el caso de transmitir una lista en forma normal no necesitaremos enviar las clausuras correspondientes a los constructores, sino sólo la lista de elementos, reduciéndose así la cantidad de información enviada.

Otra limitación del lenguaje que se ha apreciado al desarrollar aplicaciones en Eden es la falta de facilidades de "multidifusión". Por ejemplo, en la implementación del producto de matrices con un `map` paralelo, la segunda matriz debía transmitirse a todos los procesos trabajadores, por lo que el proceso padre debía empaquetarla y enviarla tantas veces como procesos hijo hubiera. El mismo problema sucedía en el problema del gradiente conjugado, en el que en cada iteración era preciso enviar a todos los procesos el mismo vector. Este comportamiento es claramente ineficiente, como puede apreciarse en las aceleraciones que se obtuvieron con ambos ejemplos. Con las herramientas de que dispone Eden en la actualidad, la única solución pasa por utilizar topologías como el toroide en el caso del producto de matrices, o un anillo en el caso del gradiente conjugado (inicialmente podría enviarse a cada proceso trabajador sólo una parte del vector, y los pedazos del vector circularían por el anillo hasta que todos los procesos tuvieran el vector completo), pero son soluciones que requieren modificar notablemente el código fuente. Como trabajo futuro se realizará un análisis de multidifusión, que en tiempo de compilación detectará datos que deban transmitirse a varios procesos por igual. La información obtenida en tiempo de compilación se utilizará en tiempo de ejecución, de modo que se minimice el número de mensajes enviados.

Los resultados mostrados en esta tesis nos hacen pensar que el lenguaje Eden y su entorno de programación han alcanzado la madurez necesaria para comenzar a cruzar la frontera que separa los lenguajes de investigación de los lenguajes "reales". Por dicho motivo, la principal línea de trabajo futura se centra en el uso de Eden para desarrollar aplicaciones reales. El objetivo a medio y largo plazo es utilizar Eden como lenguaje de programación de aplicaciones reales que se lleven a cabo en la industria. Ahora bien, antes de dar ese gran salto, tenemos previsto iniciar la "venta" del lenguaje en entornos menos reacios a la programación no imperativa, como es el caso de los departamentos universitarios de otras áreas de conocimiento. De hecho, ya hemos comenzado nuestros contactos con algunos departamentos, y em-

pezaremos en breve a trabajar con miembros del departamento de Álgebra de la Universidad Complutense para paralelizar un problema concreto de monodromía de trenzas en el que están interesados. Actualmente disponen de una solución desarrollada en Maple [CGG+91, NW96], y nuestro objetivo es utilizar Eden como lenguaje de coordinación, reutilizando para las partes secuenciales el código Maple ya existente. Será preciso implementar una interfaz Eden-Maple, para lo cual se reutilizará un interfaz existente entre GHC y Maple, que ha sido desarrollado recientemente por Hans-Wolfgang Loidl y Wolfgang Schreiner (véase [LS00]). El desarrollo de dicho interfaz permitirá no sólo resolver el problema concreto de monodromía de trenzas, sino que será una herramienta muy valiosa para muchos tipos de aplicaciones, ya que Maple es uno de los entornos matemáticos más utilizados en la actualidad.

Bibliografía

[ADG+96] P. Au, J. Darlington, M. Ghanem, Y.-K Guo, H. W. To y J. Yang. Co-ordinating Heterogeneous Parallel Computations. tomo 1123 de "LNCS", páginas 601–614. Springer-Verlag (1996).

[AJ89] L. Augustsson y T. Johnson. The Chalmers Lazy-ML Compiler. *Computer Journal* **32**(2), 127–141 (1989).

[Amd67] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. En "AFIPS Conference", tomo 30, páginas 483–485. AFIPS Press (Abril 1967).

[AWV93] J. Armstrong, M. Williams y R. Virding. "Concurrent Programming in Erlang". Prentice-Hall, Englewood Cliffs, NJ (1993).

[BDO+95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti y M. Vanneschi. P^3L: A Structured High Level Programming Language and its Structured Support. *Concurrency: Practice and Experience* **7**(3), 225–255 (1995).

[BG96] G. E. Blelloch y J. Greiner. A Provable Time and Space Efficient Implementation of NESL. En "ACM SIGPLAN International Conference on Functional Programming, ICFP'96", páginas 213–225, New York (Mayo 24–26 1996). ACM Press.

[BK95] G. H. Botorog y H. Kuchen. Algorithmic Skeletons for Adaptive Multigrid Methods. En A. Ferreira y J. Rolim, editores, "Irregular'95", tomo 980 de "LNCS", páginas 27–41. Springer-Verlag (1995).

[BK98] G. H. Botorog y H. Kuchen. Efficient High-Level Parallel Programming. *Theoretical Computer Science* **196**(1–2), 71–107 (6 Abril 1998).

[BKL98] S. Breitinger, U. Klusik y R. Loogen. From (Sequential) Haskell to (Parallel) Eden: An Implementation Point of View. En "International Symposium on Programming Languages, Implementations, Logics and Programs, PLILP'98", tomo 1490 de "LNCS", páginas 318–334. Springer-Verlag (1998).

[Ble93] G. E. Blelloch. NESL: A Nested Data-Parallel Language. Informe técnico CMU-CS-93-129, Carnegie Mellon University (Abril 1993).

[Ble96] G. E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM* **39**(3), 85–97 (Marzo 1996).

[BLOP96] S. Breitinger, R. Loogen, Y. Ortega-Mallén y R. Peña. Eden, the Paradise of Functional-Concurrent Programming. En "European Conference on Parallel Processing, EUROPAR'96", tomo 1123 de "LNCS", páginas 710–713. Springer-Verlag (1996).

[BLOP97] S. Breitinger, R. Loogen, Y. Ortega-Mallén y R. Peña. The Eden Coordination Model for Distributed Memory Systems. En "Workshop on High-level Parallel Programming Models, HIPS'97", páginas 120–124. IEEE Computer Science Press (1997).

[BLOP98] S. Breitinger, R. Loogen, Y. Ortega-Mallén y R. Peña. Eden: Language Definition and Operational Semantics. Informe técnico Bericht 96-10, Philipps-Universität Marburg (Alemania) (1998). Versión revisada en 1.998.

[Bot98] G. H. Botorog. "High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms". Tesis Doctoral, RWTH-Aachen (Enero 1998). También: Informe técnico 97-15.

[Bra89] B. Branner. The Mandelbrot Set. En "Chaos and Fractals: The Mathematics Behind the Computer Graphics", páginas 75–106 (1989).

[Bra92] T. A. Bratvold. Determining Useful Parallelism in Higher Order Functions. En H. Kuchen y R. Loogen, editores, "4th International Workshop on Parallel Implementation of Functional Languages". RWTH Aachen (1992). Informe técnico No. 92-19.

[Bra93] T. A. Bratvold. A Skeleton-Based Parallelising Compiler for ML. En "Draft Proceedings of Implementation of Functio-

nal Languages, IFL'93", páginas 23–33, Nijmegen, The Netherlands, September (1993).

[Bra94a] T. A. Bratvold. Parallelising a Functional Program Using a List-Homomorphism Skeleton. En H. Hong, editor, "First International Symposium on Parallel Symbolic Computation, PASCO'94", páginas 44–53, Hagenberg/Linz (Austria) (Septiembre 1994). World Scientific Publishing Company.

[Bra94b] T. A. Bratvold. "Skeleton-Based Parallelisation of Functional Programs". Tesis Doctoral, Heriot-Watt University (Noviembre 1994).

[Can92] D. Cann. Retire Fortran? A Debate Rekindled. *Communications of the ACM* **35**(8), 81–89 (Agosto 1992).

[CF90] D. Cann y J. Feo. SISAL versus Fortran: a Comparison Using the Livermore Loops. En "Supercomputing'90", páginas 626–636. IEEE Computer Society Press (Noviembre 1990).

[CG89] N. Carriero y D. Gelernter. Linda in Context. *Communications of the ACM* **32**(4) (Abril 1989).

[CGG+91] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan y S. M. Watt. "The Maple V Language Reference Manual". Springer-Verlag (1991).

[Col88] M. I. Cole. "Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation". Tesis Doctoral, University of Edinburgh, Computer Science Department (1988).

[Col89] M. Cole. "Algorithmic Skeletons: Structured Management of Parallel Computations". MIT Press (1989). Research Monographs in Parallel and Distributed Computing.

[CRW01] O. Chitil, C. Runciman y M. Wallace. Freja, Hat and Hood — A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. En M. Mohnen y P. Koopman, editores, "Implementation of Functional Languages, IFL'00", tomo 2011 de "LNCS", páginas 176–193. Springer-Verlag (2001).

[DFH+93] J. Darlington, A. Field, P. Harrison, P. Kelly, R. While y Q. Wu. Parallel Programming using Skeleton Functions. En "PARLE'93", tomo 694 de "LNCS", páginas 146–160. Springer-Verlag (Junio 1993).

[DGT93] J. Darlington, M. Ghanem y H. W. To. Structured Parallel Programming. En "Massively Parallel Programming Models Conference", páginas 160–169. IEEE Computer Society Press (Septiembre 1993).

[DGTY95a] J. Darlington, Y.-K. Guo, H. W. To y J. Yang. Functional Skeletons for Parallel Coordination. tomo 966 de "LNCS", páginas 55–69. Springer-Verlag (1995).

[DGTY95b] J. Darlington, Y.-K. Guo, H. W. To y J. Yang. Parallel Skeletons for Structured Composition. En "5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95", páginas 19–28, Santa Barbara, California (EEUU) (Julio 1995).

[Fly72] M. J. Flynn. Some Computer Organizations and their Effectiveness. *IEEE Transactions on Computers* **21**(9), 948–960 (Septiembre 1972).

[For93] High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming* **2**(1) (1993).

[Fos95] I. Foster. "Designing and Building Parallel Programs". Addison-Wesley (1995).

[GBDJ94] A. Geist, A. Beguelin, J. Dongarra y W. Jiang. "PVM: Parallel Virtual Machine". MIT Press (1994).

[GC92] D. Gelernter y N. Carriero. Coordination Languages and their Significance. *Communications of the ACM* **35**(2), 97–107 (Febrero 1992).

[Gen78] W. M. Gentleman. Some Complexity Results for Matrix Computations on Parallel Computers. *Journal of the ACM* **25**(1), 112–115 (Enero 1978).

[Gil00] A. Gill. Debugging Haskell by Observing Intermediate Data Structures. En "4th Haskell Workshop". University of Nottingham (2000).

[GLS94] W. Gropp, E. Lusk y A. Skjellum. "Using MPI". MIT Press (1994).

[GMP89] A. Giacalone, P. Mishra y S. Prasad. FACILE: A Symmetric Integration of Concurrent and Functional Programming. *International Journal of Parallel Programming* **18**(2), 121–160 (Abril 1989).

[Ham00] M. Hamdan. "A Combinational Framework for Parallel Programming Using Algorithmic Skeletons". Tesis Doctoral, Department of Computing and Electrical Engineering. Heriot-Watt University (2000).

[HHLT97] K. Hammond, C. V. Hall, H. W. Loidl y P. W. Trinder. Parallel Cost Centre Profiling. En "Glasgow Functional Programming Workshop" (Septiembre 1997).

[HJ92] K. Hammond y S. L. Peyton Jones. Profiling Scheduling Strategies on the GRIP Multiprocessor. En "4th International Workshop on the Parallel Implementation of Functional Languages", páginas 73–98 (Septiembre 1992).

[HKL+00] K. Hammond, D.J. King, H-W. Loidl, Á.J. Rebón Portillo y P.W. Trinder. The HasPar Performance Evaluation Suite for GpH: a Parallel Non-Strict Functional Language. *Software — Practice and Experience* (Febrero 2000). Sometido a revisores.

[HL00] C. A. Herrmann y C. Lengauer. HDC: A Higher-Order Language for Divide-and-Conquer. *Parallel Processing Letters* **10**(2/3), 239–250 (Septiembre 2000).

[HLG+99] C. A. Herrmann, C. Lengauer, R. Günz, J. Laitenberger y C. Schaller. A Compiler for HDC. Informe técnico MIP-9907, Fakultät für Mathematik und Informatik, Universität Passau (Mayo 1999).

[HLP94] K. Hammond, H. W. Loidl y A. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. Department of Computing Science. University of Glasgow (1994).

[HM99] K. Hammond y G. Michaelson, editores. "Research Directions in Parallel Functional Programming". Springer-Verlag (1999).

[HN01] F. Huch y U. Norbisrath. Distributed Programming in Haskell with Ports. En M. Mohnen y P. Koopman, editores, "Implementation of Functional Languages, IFL00", tomo 2011 de "LNCS", páginas 107–121. Springer-Verlag (2001).

[HO00] M. Hidalgo Herrero y Y. Ortega Mallén. A Distributed Operational Semantics for a Parallel Functional Language. En "Trends in Functional Programming 2 (Selected Papers of the 2nd Scottish Functional Programming Workshop)", páginas 89–102. Intellect (2000).

[HPR00] F. Hernández, R. Peña y F. Rubio. From GranSim to Paradise. En "Trends in Functional Programming (Selected papers of the 1st Scottish Functional Programming Workshop)", páginas 11–19. Intellect (2000).

[HS78] E. Horowitz y S. Sahni. "Fundamentals of Computer Algorithms". Pitman (1978).

[Hud86] P. Hudak. Para-Functional Programming. *IEEE Computer* **19**(8), 60–70 (1986).

[Imp] Impala. Impala – (IMplicitly PArallel LAnguage Application Suite). `http://www.csg.lcs.mit.edu/impala`.

[Inm88] Inmos Ltd. "Occam2 Reference Manual". Prentice-Hall (1988).

[JHHP92] S. L. Peyton Jones, C. V. Hall, K. Hammond y W. Partain. The Glasgow Haskell Compiler: a Technical Overview. Department of Computer Science, University of Glasgow (Diciembre 1992).

[JM99] S. L. Peyton Jones y S. Marlow. Secrets of the Glasgow Haskell Compiler Inliner. En "International Workshop on Implementation of Declarative Languages, IDL'99" (Septiembre 1999).

[JS98] S. L. Peyton Jones y A. L. M. Santos. A Transformation-based Optimiser for Haskell. *Science of Computer Programming 32(1-3):3-47* (Septiembre 1998).

[JvdBvdH93] S. Joosten, K. van den Berg y G. van der Hoeven. Teaching Functional Programming to First-Year Students. *Journal of Functional Programming* **3**, 49–65 (1993).

[Kel87] P. Kelly. "Functional Programming for Loosely-Coupled Multiprocessors". Tesis Doctoral, Department of Computer Science, Westfield College (1987).

[Kel89] P. Kelly. "Functional Programming for Loosely-Coupled Multiprocessors". Pitman (1989).

[Ker95] E. T. Keravnou. Introducing Computer Science Undergraduates to Principles of Programming Through a Functional Language. En "Functional Programming Languages in Education, FPLE'95", tomo 1022 de "LNCS", páginas 15–34. Springer-Verlag (1995).

[Kes95] M. Kesseler. Constructing Skeletons in Clean: The Bare Bones. En A. P. Wim Bohm y John T. Feo, editores, "High Performance Functional Computing", páginas 182–192 (Abril 1995).

[KGGK94] V. Kumar, A. Grama, A. Gupta y G. Karypis. "Introduction to Parallel Computing. Design and Analysis of Algorithms". Benjamin/Cummings (1994).

[KHT98] D. J. King, J. Hall y P. W. Trinder. A Strategic Profiler for Glasgow Parallel Haskell. En "Draft Proceedings of Implementation of Functional Languages, IFL'98" (Septiembre 1998).

[KLPR01] U. Klusik, R. Loogen, S. Priebe y F. Rubio. Implementation Skeletons in Eden: Low-Effort Parallel Programming. En M. Mohnen y P. Koopman, editores, "Implementation of Functional Languages, IFL'00", tomo 2011 de "LNCS", páginas 71–88. Springer-Verlag (2001).

[Klu98] U. Klusik. Implementors Handbook for the Eden Compiler 0.04. Disponible bajo petición a la autora (Diciembre 1998).

[Knu71] D. E. Knuth. An Empirical Study of FORTRAN Programs. *Software – Practice and Experience* **1**, 105–133 (1971).

[KO62] A. Karatsuba y Y. Ofman. Multiplication of Multidigit Numbers on Automata. *Doklady Akademii Nauk SSSR* **145**(2), 293–294 (1962).

[KOP99] U. Klusik, Y. Ortega-Mallén y R. Peña. Implementing Eden - or: Dreams Become Reality. En "Implementation of Functional Languages, IFL'98", tomo 1595 de "LNCS", páginas 103–119. Springer-Verlag (Septiembre 1999).

[KPR01] U. Klusik, R. Peña y F. Rubio. Replicated Workers in Eden. En "Constructive Methods for Parallel Programming, CMPP'2000". Nova Science (2001). En proceso de publicación.

[KPS00] U. Klusik, R. Peña y C. Segura. Bypassing of Channels in Eden. En "Trends in Functional Programming (Selected papers of the 1st Scottish Functional Programming Workshop)", páginas 2–10. Intellect (2000).

[KT99] P. Kelly y F. Taylor. Coordination Languages. En "[HM99]", páginas 305–321 (1999).

[Les93] B. P. Lester. "The Art of Parallel Programing". Prentice-Hall International Inc. (1993).

[Lip71] J. D. Lipson. Chinese Remainder and Interpolation Algorithms. En "Symposium on Symbolic and Algebraic Manipulation, SYMSAM'71", páginas 372–391. Academic Press (1971).

[LLR93] T. Lambert, P. Lindsay y K. Robinson. Using Miranda as a First Programming Language. *Journal of Functional Programming* **3**, 5–34 (1993).

[LMSK63] J. D. Little, K. G. Murty, D. W. Sweeney y C. Karel. An Algorithm for the Traveling Salesman Problem. *Operations Research* **11**(6), 972–989 (1963).

[Loi96] H. W. Loidl. GranSim User's Guide. Department of Computing Science. University of Glasgow (1996).

[Loi97] H. W Loidl. LinSolv: a Case Study in Strategic Parallelism. En "Glasgow Workshop on Functional Programming" (Septiembre 1997).

[Loi98] H. W. Loidl. "Granularity in Large-Scale Parallel Functional Programming". Tesis Doctoral, Department of Computing Science. University of Glasgow (1998).

[**LOP+02**] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe y F. Rubio. Parallelism Abstractions in Eden. En F. A. Rabhi y S. Gorlatch, editores, "Patterns and Skeletons for Parallel and Distributed Computing". Springer-Verlag (2002). En proceso de publicación.

[**LRS+01**] H. W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, Á. J. Rebón Portillo, S. Priebe y P. W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher-Order and Symbolic Computation* (2001). Sometido a revisores.

[LS00] H. W. Loidl y W. Schreiner. GHC–Maple Interface. `http://www.risc.uni-linz.ac.at/software/ghc-maple` (Noviembre 2000).

[Mes94] Message Passing Interface Forum. MPI: A Message-passing Interface Standard. *International Journal of Supercomputer Applications* **8**((3/4)) (1994).

[MH95] R. Mirani y P. Hudak. First-Class Schedules and Virtual Maps. En "7th International Conference on Functional Programming Languages and Computer Architecture, FPCA'95", páginas 78–85, La Jolla, California (EEUU) (Junio 25–28, 1995). ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[MJ95] R. G. Morgan y S. A. Jarvis. Profiling Large-Scale Lazy Functional Programs. En A. P. W. Bohm y J. T. Feo, editores, "High Performance Functional Computing", páginas 222–234 (Abril 1995).

[MJ98] R. G. Morgan y S. A. Jarvis. Profiling Large-Scale Lazy Functional Programs. *Journal of Functional Programming* (1998).

[MSBK01] G. Michaelson, N. Scaife, P. Bristow y P. King. Nested Algorithmic Skeletons from Higher Order Functions. *Journal of Parallel Algorithms and Applications* (2001). En proceso de publicación.

[MSK86] D. May, R. Shepherd y C. Keane. Transputers and Occam. En "Future Parallel Computers", tomo 272 de "LNCS", páginas 35–81 (1986).

[NPP95] M. Núñez, P. Palao y R. Peña. A Second Year Course on Data Structures based on Functional Programming. En "Functional Programming Languages in Education, FPLE'95", tomo 1022 de "LNCS", páginas 65–84. Springer-Verlag (1995).

[NS97] H. Nilsson y J. Sparud. The Evaluation Dependence Tree as a Basis for Lazy Functional Debugging. *Automated Software Engineering: An International Journal* (Abril 1997).

[NSvP91] E. G. J. M. H. Nocker, J. E. W. Smetsers, M. C. J. D. van Eekelen y M. J. Plasmeijer. Concurrent Clean. En "Parallel Architectures and Languages Europe, PARLE'91", tomo 505 de "LNCS", páginas 202–219. Springer-Verlag (1991).

[NW96] R. A. Nicolaides y N. Walkington. "Maple: a Comprehensive Introduction". Cambridge University Press, Cambridge, UK (1996).

[Pel93] S. Pelagatti. "A Methodology for the Devolopment and the Support of Massively Parallel Programs". Tesis Doctoral, Dipartamento di Informatica, University of Pisa (1993).

[Pel98] S. Pelagatti. "Structured Development of Parallel Programs". Taylor and Francis (1998).

[Pel02] S. Pelagatti. Task and Data Parallelism in P3L. En F. A. Rabhi y S. Gorlatch, editores, "Patterns and Skeletons for Parallel and Distributed Computing". Springer-Verlag (2002). En proceso de publicación.

[Pey96] S. L. Peyton Jones. Compiling Haskell by Program Transformation: A Report from the Trenches. En "European Symposium on Programming, ESOP'96", tomo 1058 de "LNCS", páginas 18–44, Linköping, Sweden, April 22–24 (1996). Springer-Verlag.

[PGF96] S. L. Peyton Jones, A. Gordon y S. Finne. Concurrent Haskell. En "23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'96", páginas 295–308, St. Petersburg Beach, Florida (21–24 Enero 1996).

[PH99] S. L. Peyton Jones y J. Hughes. Report on the Programming Language Haskell 98. Informe técnico, (Febrero 1999).

[POR99] R. Peña, Y. Ortega-Mallén y F. Rubio. Teaching Monadic Algorithms to First-Year Students. En "Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL'99, parte de PLI'99", páginas 33–45 (1999). `http://www.cs.columbia.edu/~cdo/waaapl.html`.

[PPP99] C. Pareja, I. Pita y P. Palao. El Efecto de las Optimizaciones Secuenciales sobre Eden. Informe técnico 92.99, Dpto. Sistemas Informáticos y Programación, Univ. Complutense de Madrid (1999).

[PPRS00] C. Pareja, R. Peña, F. Rubio y C. Segura. Optimizing Eden by Transformation. En "Trends In Functional Programming 2 (Selected papers of the 2nd Scottish Functional Programming Workshop)", páginas 13–26. Intellect (2000).

[PPRS01] C. Pareja, R. Peña, F. Rubio y C. Segura. Adding Traces to a Lazy Monadic Evaluator. En "Eurocast 2001", tomo 2178 de "LNCS". Springer-Verlag (2001). En proceso de publicación.

[PR01] R. Peña y F. Rubio. Parallel Functional Programming at Two Levels of Abstraction. En "Principles and Practice of Declarative Programming, PPDP'01", páginas 187–198. ACM Press (Septiembre 2001).

[PRS01] R. Peña, F. Rubio y C. Segura. Deriving Non-Hierarchical Process Topologies. En "Draft Proceedings of the 3rd Scottish Functional Programming Workshop", páginas 157–168 (2001).

[PS00] C. Pareja y C. Segura. Efecto de las Transformaciones de GHC sobre Edén. Informe técnico 101-00, Dpto. Sistemas Informáticos y Programación, Univ. Complutense de Madrid (2000).

[PS01a] R. Peña y C. Segura. Non-Determinism Analysis in a Parallel-Functional Language. En "Implementation of Functional Languages, IFL'00.", tomo 2011 de "LNCS", páginas 1–18. Springer-Verlag (2001).

[PS01b] R. Peña y C. Segura. Sized Types for Typing Eden Skeletons. En "Draft Proceedings of Implementation of Functional Languages, IFL'01.", páginas 33–51 (2001).

[PTL00] R. F. Pointon, P. W. Trinder y H. W. Loidl. Runtime System Level Fault Tolerance for a Distributed Functional Language. En "Trends in Functional Programming 2 (Selected papers of the 2nd Scottish Functional Programming Workshop)", páginas 103–114. Intellect (2000).

[PvE98] R. Plasmeijer y M. van Eekelen. Concurrent Clean Language Report - version 1.3. Informe técnico CSI-R9816, Computing Science Institute, University of Nijmegen (Junio 1998).

[PVM93] Parallel Virtual Machine Reference Manual, Version 3.2. Oak Ridge National Laboratory, University of Tennessee (Agosto 1993).

[Qui94] Michael J. Quinn. "Parallel Computing". McGraw-Hill (1994).

[RBMS97] D. Ridge, D. Becker, P. Merkey y T. Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. En "IEEE Aerospace" (1997).

[Rep91] J. H. Reppy. CML: a Higher-Order Concurrent Language. *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.* **26**(6), 293–305 (Junio 1991).

[RR96a] N. Röjemo y C. Runciman. Lag, Drag, Void and Use – Heap Profiling and Space-Efficient Compilation Revisited. En "International Conference on Functional Programming, ICFP'96", páginas 34–41. ACM Press (1996).

[RR96b] C. Runciman y N. Röjemo. New Dimensions in Heap Profiling. *Journal of Functional Programming* (1996).

[RR96c] C. Runciman y N. Röjemo. Two-pass Heap Profiling: a Matter of Life and Death. En "Implementation of Functional Languages, IFL'96" (1996).

[RS94] F. A. Rabhi y J. Schwarz. POPE: A Paradigm Oriented Parallel Programming Environment for SIT Algorithms. En J. Glauert, editor, "Draft Proceedings of Implementation of Functional Languages, IFL'94", UEA Norwich, UK (Septiembre 1994).

[RTHM97] Milner R, M. Tofte, R. Harper y D. MacQueen. "The Definition of Standard ML (Revised)". MIT Press (1997).

[Rub99] F. Rubio. Programación Funcional Paralela Eficiente. Trabajo de Tercer Ciclo. Dep. Sistemas Informáticos y Programación. Univ. Complutense de Madrid (1999).

[RW91] C. Runciman y D. Wakeling. Problems & Proposals for Time & Space Profiling of Functional Programs. En "1990 Glasgow Workshop on Functional Programming". Springer-Verlag (1991).

[RW93a] C. Runciman y D. Wakeling. Heap Profiling of a Lazy Functional Compiler. En "1992 Glasgow Workshop on Functional Programming". Springer-Verlag (1993).

[RW93b] C. Runciman y D. Wakeling. Heap Profiling of Lazy Functional Programs. *Journal of Functional Programming* (1993).

[San94] P. M. Sansom. "Execution Profiling for Non-strict Functional Languages". Tesis Doctoral, Department of Computing Science. University of Glasgow (1994).

[San95] A. L. M. Santos. "Compilation by Transformation in Non-strict Functional Languages". Tesis Doctoral, Department of Computing Science. University of Glasgow (1995).

[Sch94] S. Scholz. Single Assignment C — Functional Programming Using Imperative Style. En "Implementation of Functional Languages, IFL'94", University of East Anglia, Norwich (Reino Unido) (1994).

[Sch96] S. Scholz. "Single Assignment C — Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen". Tesis Doctoral, Institute of Computer Science and Applied Mathematics, University of Kiel (1996).

[Sch98] S. Scholz. On Defining Application-Specific High-Level Array Operations by Means of Shape-Invariant Programming Facilities. En "ACM-SIGAPL International Conference on Array Processing Languages, APL'98", páginas 40–45, Roma (Italia) (1998).

[Sij89] B. A. Sijtsma. On the Productivity of Recursive List Definitions. *ACM TOPLAS* **11**(4), 633–649 (Octubre 1989).

[SJ95] P. M. Sansom y S. L. Peyton Jones. Time and Space Profiling for Non-Strict, Higher-Order Functional Languages. En "22nd ACM Symposium on Principles of Programming Languages, POPL'95" (Enero 1995).

[SJ97] P. M. Sansom y S. L. Peyton Jones. Formally Based Profiling for Higher-Order Functional Languages. En "ACM Transactions on Programming Languages and Systems" (1997).

[SMH01] N. Scaife, G. Michaelson y S. Horiguchi. Comparative Cross-Platform Results from a Parallelizing SML Compiler. En "Draft Proceedings of Implementation of Functional Languages, IFL'01", Stockholm (Sweden) (2001).

[SR96] J. Schwarz y F. A. Rabhi. A Skeleton Based Implementation of Iterative Transformation Algorithms Using Functional Languages. En "Abstract Machine Models for Parallel and Distributed Computing". IOS Press (Abril 1996).

[SR97] J. Sparud y C. Runciman. Tracing Lazy Functional Computations Using Redex Trails. En H. Glaser, P. Hartel y H. Kuchen, editores, "International Symposium on Programming Languages, Implementations, Logics and Programs, PLILP'97", tomo 1292 de "LNCS", páginas 291–308. Springer-Verlag (1997).

[Tay97] F. Taylor. "Parallel Functional Programming by Partitioning". Tesis Doctoral, Department of Computing, Imperial College of Science, Technology and Medicine, University of London (Enero 1997).

[THJ+96] P. W. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge y S. L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. En "ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'96". ACM Press (1996).

[THLP98] P. W. Trinder, K. Hammond, H-W. Loidl y S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming* **8**(1), 23–60 (Enero 1998).

[TLP01] P. W. Trinder, H. W. Loidl y R. F. Pointon. Parallel and Distributed Haskells. *Journal of Functional Programming* (2001). En proceso de publicación.