Universidad de Murcia

Facultad de Informática

Técnicas de autooptimización en recorridos de árboles por medio de *backtracking*

Proyecto Fin de Carrera
Ingeniero en Informática

Autor:
Manuel Quesada Martínez

Tutor:
Domingo Giménez Cánovas

Murcia, 13 de febrero de 2009

Índice general

Capítulo 1

Introducción

El objetivo que se persigue con este proyecto es proponer una metodología para la autooptimización de recorridos de árboles por medio de *backtracking*. Se estudiarán diferentes esquemas de programación identificando parámetros que influyen en el tiempo de ejecución, y se modelará su comportamiento. Se diseñará una metodología para la toma de decisiones en problemas secuenciales: estimar el valor de parámetros que modelen el comportamiento del algoritmo y seleccionar entre diferentes esquemas el que mejor se comporte. Se propondrá una metodología para tomar decisiones en soluciones paralelas, donde el número de parámetros a considerar aumenta. Se estudiarán diferentes esquemas identificando parámetros y modelando su comportamiento. Se diseñarán estrategias que seleccionen valores adecuados para dichos parámetros. Tanto en el caso secuencial como en el paralelo se identificarán esquemas básicos de programación. En el caso paralelo se intentará abstraer al usuario de la implementación paralela proponiendo un esquema secuencial que internamente se paralelizará.

1.1 Introducción

Cada vez es más habitual la aparición de máquinas multiprocesadores capaces de acelerar la resolución de problemas computacionales de alto coste. Para ello, los programadores utilizan las capacidades que les proporciona el paralelismo, definiendo paralelismo como la posibilidad de división de un determinado problema computacional en partes que se pueden resolver de forma independiente y simultánea.

Aunque posiblemente el modelo de computador que más éxito ha tenido y que más ha aportado al desarrollo de la informática es el modelo secuencial, muy pronto se intentó utilizar el paralelismo para explotar los recursos disponibles. Es vital su utilización para conseguir procesadores (actualmente multiprocesadores) cada vez más potentes, salvando los problemas físicos que suponen un estancamiento en el aumento de la frecuencia de reloj. Desde su aparición, el paralelismo ha venido utilizándose a diversos niveles [3]. El paralelismo a nivel hardware planifica las instrucciones en el *pipeline* del procesador para aprovechar la ejecución paralela en las distintas unidades funcionales que lo forman, de manera que se obtiene un mayor número de instrucciones de salida por ciclo de reloj [21]. Por encima tendríamos el nivel de software básico y el de software medio, donde se

encuentra, por ejemplo, la programación basada en hilos, que intenta explotar el paralelismo entre los distintos procesos, o incluso la ejecución en los diferentes núcleos (librerías como OpenMP o MPI). Por encima se diferencian dos niveles más, el nivel de software, con librerías como BLAS o PBLAS, y un último nivel de aplicación donde encontramos librerías numéricas como LAPACK y su versión paralela ScaLAPACK. Sin embargo, hasta hace unos años se ha trabajado en los primeros niveles estando a un nivel demasiado bajo para que los usuarios finales tuvieran idea de lo que está sucediendo realmente. Incluso entre los propios programadores, muchos de ellos siguen viendo los desarrollos como ejecuciones secuenciales y son pocos los que se interesan en explotar directamente los beneficios que este tipo de técnicas brindan.

Sin embargo, en los últimos dos o tres años, las máquinas multiprocesador están adquiriendo una importancia cada vez mayor. Este campo ya no solo queda reservado a grupos reducidos de usuarios, sino que paulatinamente cualquiera que utilice un ordenador (a nivel medio) está tomando conciencia de esta nueva forma de abordar los problemas y su funcionamiento paralelo. Desde el 2005 varios fabricantes (Intel, IBM, SUN, AMD...) empezaron a presentar diseños en los que varios procesadores están implementados sobre un solo chip, dando lugar a la tecnología *multicore* o multinúcleo. Cualquier usuario que adquiera hoy en día un ordenador personal dispondrá de hardware paralelo y tendrá la necesidad o curiosidad de explotar su rendimiento. Actualmente, es esta tecnología la que se propone para hacer más veloces los ordenadores y mejorar su funcionamiento general. Las previsiones son incluir decenas e incluso centenares de núcleos en un chip, con miles de threads en ejecución. En la actualidad SUN comercializa procesadores con 8 núcleos (SUN ULtraSparc T1 Niagara); IBM, Sony y Toshiba han presentado un procesador heterogéneo de uno mas ocho núcleos (CELL Broadband Engine); Intel ha comercializado sus procesadores Itanium 2 Montecito (con dos núcleos) e Intel Quad Core (con cuatro núcleos); por último AMD tiene en el mercado un procesador binúcleo (AMD Athlon 64 X2 Dual-Core) y en breve ofrecerá uno con cuatro núcleos.

Especial interés tiene aplicar paralelismo en el ámbito científico. Prueba de ello es la cantidad de dinero que se está invirtiendo también en supercomputadores para utilizar en la resolución de problemas con un alto coste computacional (Roadrunner, BlueGene/L...). Estas máquinas están formadas por varios multiprocesadores que en total pueden alcanzan cifras de miles de núcleos. Incluso el sector empresarial, aunque reacio, se decanta con prestar un mayor esfuerzo en este sector en desarrollo. Numerosas son las empresas que se están encargando de reprogramar sus programas a códigos paralelos.

Este proyecto está destinado a los usuarios potenciales de paralelismo, como ingenieros y científicos que se deben enfrentar a problemas con un coste computacional elevado. Una correcta solución paralela a un problema hasta ahora resuelto de manera secuencial, puede suponer una reducción significativa en el tiempo empleado para resolverlo.

Sin embargo, estas nuevas rutinas paralelas traen consigo nuevos parámetros de ajuste que hasta ahora no existían como tales: número de procesadores a utilizar en la ejecución, número de hilos, geometría de la malla de procesos, tamaños de los bloques de comunicación... y otros tipos de parámetros específicos de los algoritmos que se utilicen para resolver el problema [2]. Un profesional, acostumbrado a este tipo de soluciones, cono-

ce esta problemática. No obstante, incluso un usuario experto sería incapaz, a primera vista, de dar con los parámetros de ejecución óptimos o simplemente adecuados para ejecutar cada una de las rutinas en máquinas totalmente distintas. La elección de estos parámetros es un proceso decisivo en el tiempo de ejecución total de nuestras rutinas. Deberemos seleccionarlos correctamente para hacer que el tiempo de ejecución sea mínimo, pues dependiendo de su valor este podrá aumentar y disminuir de manera considerable.

Aunque lo ideal sería diseñar programas paralelos que a medida que aumentamos el número de procesadores resolvieran el problema cada vez más rápido, esto no siempre es así. Se puede observar en los dos peores casos de la Figura 1.1: a menudo, las propias características del problema limitan el grado de paralelismo, y las comunicaciones pueden generar sobrecarga. Imaginemos una rutina donde implementamos paralelismo mediante el paradigma de paso de mensajes (el código se ejecutará en nodos diferentes que no tienen una memoria común y deben intercambiarse la información a través de la red de comunicaciones). En este caso puede ser que a partir de un cierto número de procesadores la ganancia de nuestro algoritmo comience a disminuir, y un posible motivo es que se esté perdiendo más tiempo en la comunicación entre los distintos procesos que en cálculo real para solucionar el problema. Es labor del usuario de la rutina saber cual es el número de procesadores exacto que debe utilizar dependiendo del sistema donde lo vaya a ejecutar. Este es el parámetro más obvio, pero a lo largo del proyecto veremos que existen otros muchos parámetros cruciales en la ejecución óptima o aceptable de una rutina paralela.

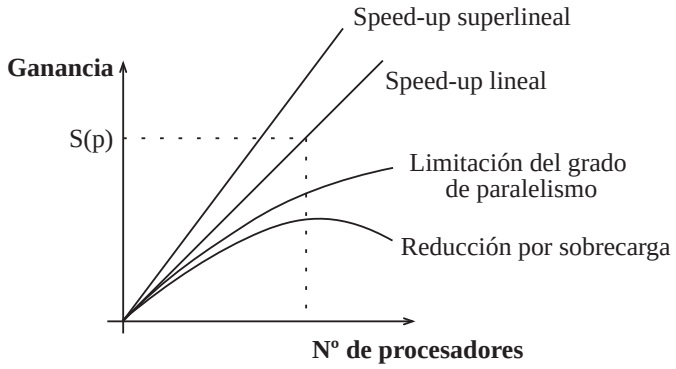


Figura 1.1. Ganancias en programas paralelos.

Como se verá más adelante, existen parámetros que dependen del sistema donde ejecutemos la rutina y otros que dependerán explícitamente del tipo de algoritmo que estemos utilizando para la resolver el problema. A su vez existirán dependencias entre ellos (parámetros del sistema y parámetros del algoritmo) que influirán significativamente en el tiempo de ejecución.

En este apartado es donde adquiere una importancia relevante este proyecto. Se centrará en problemas que necesiten para resolverlos algoritmos de recorrido de un árbol de soluciones. Concretamente se utilizará para recorrerlos la técnica de *Backtracking* o vuelta a atrás. Se intentará diseñar un proceso de autoconfiguración de los parámetros con un problema académico reducido para detectar los principales problemas que nos plantean este tipo de técnicas. En primer lugar se diseñará una estrategia de decisión para una

solución secuencial del problema. Finalmente se propondrán técnicas de autooptimización para esquemas paralelos de este tipo de recorridos de árboles.

Como se verá en posteriores capítulos, buscaremos una forma de abordar el problema mediante una solución paralela y extraeremos los principales parámetros que entran en juego para su ejecución óptima. Será objetivo de nuestras librerías obtener unos parámetros de configuración adecuados para una correcta ejecución de la rutina, con independencia de las características del sistema donde se esté ejecutando. Se identificarán aspectos interesantes a tener en cuenta para construir un esqueleto algorítmico paralelo para la técnica del *backtracking*. El esquema proporcionaría a los programadores huecos para completar las funciones básicas secuenciales que lo forman. El esquema los abstraerá de cualquier detalle del paralelismo que funcionará por debajo.

1.2 Principios de autooptimización

Como ya hemos comentado, pretendemos estimar cuales son los parámetros que determinan una correcta ejecución de una rutina paralela de *backtracking*. Es obvio que esta estimación deberá de realizarse antes de ejecutar nuestra rutina y sin que la decisión que debemos tomar nos lleve un tiempo total excesivo, entendiendo por tiempo total a la suma del dedicado a la decisión de los parámetros y el dedicado a la ejecución de la rutina.

En la Figura 1.2 se puede observar de una manera bastante sencilla cual es la problemática a la que nos enfrentamos. Cada uno de los diagramas representa el tiempo de ejecución total. Se pretende ilustrar las distintas situaciones que se nos podrían presentar.

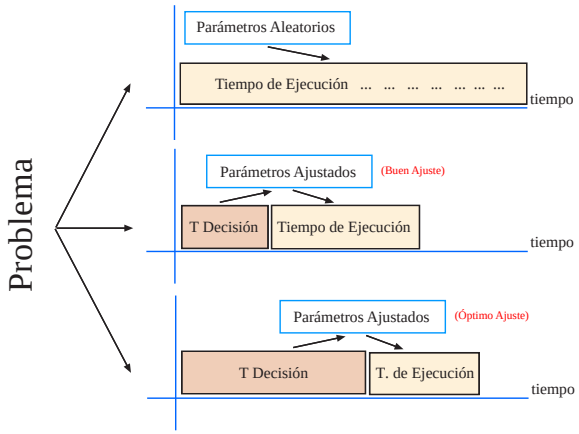


Figura 1.2. Ejemplos de tiempos de ejecución de un mismo problema con y sin autooptimización de parámetros.

- En el primer caso se supone que no utilizamos ningún mecanismo de autoconfiguración. Es el propio usuario el que se encarga de introducir los parámetros que considere oportunos. Como no tiene unos conocimientos avanzados de paralelismo

lo normal sería que eligiese parámetros que le llevasen a un tiempo de ejecución elevado.

- El segundo caso representa lo que se pretende conseguir al final de este proyecto. El objetivo es que dedicando poco tiempo a la autoconfiguración de los parámetros para cada una de las entradas, el tiempo de ejecución real de la rutina se vea reducido de manera significativa.

- El último caso ilustra la situación donde la toma de decisión fuese muy buena. Es decir, se consigue ajustar los parámetros de forma óptima y esto llevaría a conseguir el menor tiempo de ejecución posible de la rutina. Sin embargo, el tiempo que dedicamos a ajustar los parámetros es excesivo y la suma de tiempo de decisión mas tiempo de ejecución supera al que conseguiríamos con un ajuste peor.

Se pretende en este proyecto continuar la labor iniciada en el trabajo [4]. En este trabajo se modeló un esquema paralelo de *backtracking* proponiendo un mecanismo de autooptimización para un ejemplo académico, la mochila 0/1. En este proyecto pretendemos contrastar los resultados obtenidos en dicho trabajo modelando recorridos en árboles. Evaluaremos los parámetros en los problemas secuenciales con más detalle que en el trabajo citado. Consideramos que merece la pena estudiar su comportamiento de forma más extensa para afinar más en la paralelización. Para ello vamos a intentar modelar, a priori, el tiempo de ejecución de este tipo de algoritmos en su versión secuencial; de esta forma se podrán ver los distintos valores de tiempo teórico que obtendríamos variando únicamente los parámetros ajustables de nuestro modelo. Para utilizarlos en nuestras pruebas, diseñaremos seis esquemas distintos de backtracking que resuelvan el problema de la mochila 0/1 y contrastaremos los resultados del trabajo [4] con los nuestros. Una vez estudiado el caso secuencial, propondremos dos esquemas paralelos para estudiar nuevos parámetros de configuración y su estimación. Se implementará una adaptación de la metodología para estos dos esquemas y se validarán los métodos de autoptimización en ellos.

Una primera aproximación del tiempo de ejecución de una rutina paralela es [2]:

$$T.\ Ejecución(n) = f(n, Parámetros\ del\ Sistema, Parámetros\ del\ Algoritmo) \tag{1.1}$$

El primer parámetro de esta fórmula es n, y representa el tamaño del problema.

El segundo parámetro corresponde a lo que hemos denominado parámetros del sistema (SP). Estos serán específicos de cada uno de los sistemas computacionales donde deseamos ejecutar nuestra rutina. A modo de introducción, destacamos que estos parámetros deberán calcularse para cada nuevo sistema donde ejecutemos la rutina. Puede ser necesario calcularlos solo una vez para cada sistema, por lo que sería interesante obtenerlos en el momento de instalación de nuestras librerías. Ejemplos de parámetros de sistema son: coste de una operación aritmética, o de operaciones de distintos tipos o niveles (BLAS 1, 2 y 3 en rutinas de álgebra lineal), los tiempos de inicio de las comunicaciones (start-up

time, t_s) y de envío de un dato (word-sending time, t_w) en operaciones de comunicación... Con la utilización de estos parámetros se reflejan las características del sistema de cómputo y de comunicaciones, tanto físico (hardware) como lógico (las librerías que se utilizan para llevar a cabo las comunicaciones y las operaciones de cómputo básicas).

El tercer y último parámetro corresponde a los parámetros del algoritmo (AP). Como hemos comentado, en este proyecto vamos a centrarnos en un esquema de *backtracking* que tendrá sus AP particulares. Ejemplos de este tipo de parámetros son: número de procesadores a utilizar de entre todos los disponibles, qué procesadores utilizar si el sistema es heterogéneo, el número de procesos a poner en marcha y su mapeo en el sistema físico, parámetros que identifiquen la topología lógica de los procesos (como puede ser el número de filas y columnas de procesos en un algoritmo para malla lógica 2D), el tamaño de los bloques de comunicación o de particionado de los datos entre los procesos, el tamaño de los bloques de computación en algoritmos de álgebra lineal que trabajan por bloques...

Son estos últimos parámetros (parámetros del algoritmo) los que sería labor del usuario de nuestra rutina ajustar para cada una de las máquinas donde se ejecute y cada una de las entradas a resolver. Sería muy interesante poder proporcionar un mecanismo de autooptimización que le proponga a nuestro usuario los parámetros adecuados para una ejecución eficiente de la rutina, sin necesidad de que él tenga conocimientos avanzados de paralelismo. Lo ideal sería ofrecerle los parámetros que consigan tiempos de ejecución óptimos, pero en algunos casos el modelo de tiempo teórico depende de las entradas del problema. Es lo que sucede con las técnicas de *backtracking*, donde las entradas determinan la forma en que se recorre el árbol de búsqueda y esto influye en el tiempo de ejecución total. En estos casos se produce un rango muy grande de valores donde puede oscilar el tiempo de ejecución en función de los parámetros que le propongamos, por lo que es interesante que aunque no se le ofrezcan al usuarios parámetros que consigan tiempos de ejecución óptimos, estos estén cercanos al óptimo y alejados de los peores casos que un usuario sin conocimientos de paralelismo podría elegir.

Por último, cabe destacar que normalmente los valores de los parámetros del sistema estarán influenciados por los parámetros del algoritmo. Por ejemplo, los tiempos de inicio de comunicación y de envío de un dato pueden depender del tamaño de los bloques de comunicación, o de la topología y el número de procesadores; mientras que en algoritmos de álgebra lineal por bloques el coste de las operaciones aritméticas depende del tamaño del bloque de computación. Por esto, los parámetros del sistema se pueden expresar como una función del tamaño de la entrada y de los parámetros AP:

$$\textit{Parámetros del Sistema} = g(n, \textit{Parámetros del Algoritmo}) \tag{1.2}$$

Así la ecuación 1.1 quedaría como:

$$\textit{T. Ejecución}(n) = f(n, g(n, AP), AP) \tag{1.3}$$

En resumen, los parámetros del sistema se pueden determinar en el momento en el

que se instala la librería en cada nuevo sistema. Previamente habremos identificado los parámetros de sistema que intervienen en el modelo, y habremos diseñado una estrategia de instalación, que incluye para cada rutina los experimentos a realizar para la estimación de sus parámetros. En estos experimentos se determinarán los valores de los *SPs* y se guardarán en la máquina junto con el modelo de tiempo. Los parámetros de sistema serán calculados una única vez, y serán los parámetros del algoritmo los que sí que habrá que determinar por cada ejecución que realicemos. Podemos dividir el tiempo de decisión que veíamos antes en un tiempo de decisión para los parámetros del sistema y otro tiempo de decisión para los parámetros del algoritmo.

Ahora podremos dedicar más tiempo al ajuste de los parámetros del sistema en tiempo de instalación, reduciendo también el tiempo que dedicamos a la decisión de los parámetros del algoritmo, que es el que verdaderamente percibe el usuario. Se observa en la Figura 1.3 como quedan perfectamente delimitados el tiempo de instalación y el tiempo de ejecución. De esta forma conseguimos que el tiempo de decisión de parámetros en la ejecución sea menor.

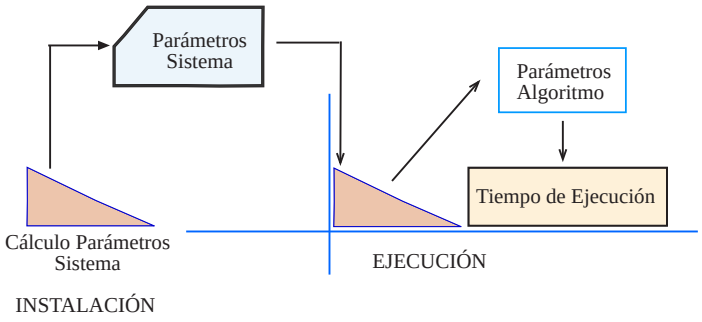


Figura 1.3. Esquema general de la estimación de los parámetros de una rutina en tiempo de instalación y de ejecución.

Los parámetros del sistema y el modelo teórico del tiempo de ejecución de la rutina quedarán guardados para cada sistema como parte de las librerías. Siempre que se produzca una variación en la arquitectura de la máquina donde deseemos ejecutar nuestras rutinas deberemos recompilar las librerías para que se adapten a la nueva configuración.

A continuación se describe como un usuario final debería utilizar estas rutinas autooptimizables:

- El usuario instalará la rutina en aquella máquina en que desee ejecutarla.
- En tiempo de instalación se calcularán los parámetros de sistema oportunos y se guardarán para su uso posterior. Este tiempo de instalación puede ser relativamente alto siempre y cuando suponga una interesante reducción del tiempo a la hora de estimar los parámetros del algoritmo, que recordemos que dependen de cada una de las entradas.

- El usuario ejecutará la rutina en aquella máquina donde la haya instalado previamente. En este preciso instante y a partir de la entrada concreta que se esté utilizando, se estimará cuales son los tiempos de ejecución teóricos sustituyendo los parámetros evaluados en el modelo. Con esto deberá de estimar cuales serían los parámetros del algoritmo que llevarían a una ejecución aceptable. El tiempo de tomar esta decisión debe ser mínimo, de manera que en ningún momento las pruebas de extracción de los parámetros adecuados superen una ejecución aceptable del algoritmo con unos parámetros predeterminados.

1.3 Trabajos relacionados

Desde hace algunos años se han venido aplicando técnicas de autooptimización de rutinas paralelas con el fin de conseguir rutinas que se adapten automáticamente a las características del sistema de cómputo, reduciendo el periodo de tiempo necesario para tener rutinas optimizadas para un nuevo sistema, y que sean capaces de ejecutarse de manera eficiente independientemente de los conocimientos que tenga el usuario.

Entre otros campos, las técnicas de autooptimización se han venido aplicando en problemas de álgebra lineal [8], trabajos sobre las transformadas de Fourier [16], sistemas de altas prestaciones [6]... Concretamente los problemas de álgebra lineal suponen el elemento básico de cómputo en muchos problemas científicos con alto coste computacional. Es importante tener las rutinas de álgebra lineal autooptimizadas para cada uno de los sistemas. Así se reducirá el tiempo empleado al utilizar estas rutinas en cualquier sistema donde se desee [8], [11] y [18].

De entre las distintas técnicas para el desarrollo de rutinas con capacidad de autooptimización, nuestro trabajo se centra en la técnica basada en la parametrización del modelo de tiempo de ejecución. En el Grupo de Computación Científica se trabaja con este tipo de técnicas, estudiando su aplicación a rutinas de álgebra lineal con paso de mensajes [10], a la mejora de una jerarquía de librerías de álgebra lineal con autooptimización [11] al diseño de polilibrerías para acelerar la computación en álgebra linear [1]. Todos estos trabajos se centran básicamente en sistemas homogéneos, pero se ha analizado también la posibilidad de adecuar la técnica a entornos heterogéneos [9], [13], [12] y [17].

Por otro lado, es importante resaltar que en la propuesta de desarrollo futuro de ScaLAPACK se incluye la posibilidad de utilizar parametrización para obtener rutinas que sean más fáciles de usar y que se ejecuten de manera más eficiente [14].

Más recientemente se ha empezado a trabajar en la aplicación de este tipo de técnicas a esquemas algorítmicos paralelos. Se trabaja sobre heurísticas y mapeo de tareas sobre sistemas heterogéneos [13] y [20].

En el tema de esquemas algorítmicos destacan los trabajos sobre introducción de métodos de autooptimización en algoritmos divide y vencerás [5], en algoritmos de programación dinámica [19] y en algoritmos de recorrido de árboles de soluciones [4].

El objetivo es poder establecer entre todos los proyectos de autooptimización que se están realizando en el Grupo de Paralelismo de la Universidad de Murcia una metodología global para abordar problemas similares.

Es también interesante incorporar estás técnicas de autooptimización a esquemas algorítmicos paralelos. De esta manera se simplificará al usuario la programación, abstrayéndole del paralelismo y la configuración de los parámetros. Solo tendrá que programar en secuencial los fragmentos del esquema necesarios. Automáticamente se paralelizaría el problema eligiendo los mejores parámetros para su ejecución. Sobre esqueletos de esquemas algorítmicos existen trabajos realizados por el Grupo de la Universidad de La Laguna. Especial interés tiene para nuestro proyecto el que trata sobre esqueletos paralelos para las técnicas de ramificación y acotación [15].

Se pretende con este proyecto estudiar los mecanismos de autooptimización en recorridos de árboles de soluciones. Se considera como punto de partida el trabajo de Juan Manuel Beltrán [4]. Se intentarán validar los resultados obtenidos en dicho trabajo con los que nosotros obtengamos. Sin embargo, en el trabajo citado no se profundizó en exceso en el comportamiento secuencial de estas técnicas. Lo que pretendemos es hacer un análisis más detallado de los recorridos secuenciales y proponer un método de selección y evaluación entre distintos esquemas. Una vez sacadas conclusiones de los métodos secuenciales se pasará al estudio de la paralelización de la técnica, realizando de nuevo un estudio de la misma.

1.4 Metodología

Para el desarrollo de este proyecto partiremos de una metodología similar a la utilizada en [4] donde: se identificaron dos esquemas paralelos para resolver problemas de *backtracking*, se modeló su comportamiento identificando parámetros proponiendo mecanismos para estimarlos, y por último se evaluaron los resultados prácticos de ambos esquemas resolviendo un problema académico.

En nuestro caso se utilizará una variante de esta metodología intentando granular más su nivel de detalle. No se trabajará directamente con los esquemas paralelos sino que se intentará estudiar primero los modelos secuenciales. Además se intentará introducir información de la instalación y combinarla con la de la ejecución para mejorar la toma de decisiones. Una vez se haya estudiado en profundidad el caso secuencial, se propondrán diferentes esquemas paralelos con diferentes parámetros y características cada uno. Intentaremos extraer los principales problemas que nos plantea la paralelización de la técnica y que no se daban en el caso secuencial. Además identificaremos cualidades que debe tener el código para poder construir un esquema algorítmico secuencial, que internamente realice la paralelización abstrayendo de ella a los usuarios finales. Para concluir se evaluarán nuestras técnicas con el problema académico de la Mochila 0/1, y se contrastarán los resultados con los obtenidos en [4].

Se explica con más detalle la metodología que se seguirá en cada uno de los pasos

indicados.

1.4.1 Parte Secuencial

- Nos centraremos en problemas que se resuelven con algoritmos de *backtracking*. Lo primero que se deberá hacer para poder autooptimizar estas técnicas será estudiar la técnica y su comportamiento, de manera que nos permita analizarla teóricamente.

- Una vez estudiada la técnica secuencial, se extraerán los primeros *APs* y obtendremos la primera fórmula que modele el comportamiento del algoritmo. Se propondrán varias formas de estimar los valores de los *APs*. Se propondrán varios tipos de tomas de decisión en función de los parámetros identificados. Se diseñará una metodología general de instalación de las rutinas, registro de la información de la instalación, procesamiento de la información en la ejecución y toma de decisión final.

- Una vez diseñada la estrategia de toma de decisión (modelo teórico, estrategia de instalación/ejecución y estrategia de decisión) implementaremos cinco esquemas para resolver el problema académico de la mochila 0/1. Se analizará como se comporta en la práctica la estimación de los parámetros teóricos que se han extraído.

- Se propondrá una estrategia para que a partir de las estimaciones hechas con nuestro modelo, podamos decidir cual de los diferentes esquemas que tenemos para un mismo problema es mejor.

1.4.2 Parte Paralela

- Una vez estudiada la técnica secuencial pasaremos a proponer un conjunto de esquemas para paralelizar un recorrido de este tipo. Se pretende detectar por medio de este análisis los principales problemas que se plantean al intentar paralelizar este tipo de técnicas. Tras ello, se identificarán cuales son los nuevos *APs* y *SPs* que intervienen significativamente en el tiempo de ejecución del algoritmo paralelo. Finalmente se intentará modelar estos comportamientos, de manera que se pueda predecir cual será el comportamiento de la rutina (tiempo de ejecución) antes de ejecutarla. En teoría una vez obtenida la ecuación que modela el tiempo de nuestras rutinas tendremos el mecanismo para predecir los parámetros que mejor se ajusten a la máquina donde vayamos a ejecutarlos. El problema es que las técnicas que se van a estudiar tienen en su comportamiento alta dependencia de las entradas, lo que hace que el modelo varíe respecto a los resultados prácticos.

- Una vez realizado el estudio teórico se diseñará una metodología general para estimar los valores de los parámetros del algoritmo de cada esquema paralelo. Entre otras acciones, deberemos decidir como podremos calcular los valores de los *SPs* que se calculan en tiempo de compilación y donde guardarlos. También decidiremos

qué técnicas utilizaremos para extraer los *APs* en tiempos que no sean excesivamente largos para no aumentar en exceso el tiempo de ejecución de la rutina. La metodología diseñada será general e independiente de los esquemas paralelos que utilicemos, y servirá como base para futuras extensiones de este trabajo.

- Tras tener definida la metodología y analizados los diferentes esquemas paralelos, se procederá a su implementación y estudio experimental. Estudiaremos el comportamiento de nuestras rutinas para dos esquemas paralelos distintos. Se utilizará para su implementación tanto memoria compartida como memoria distribuida. Al igual que en el modelo secuencial, se utilizará como caso de prueba el problema académico de la mochila 0/1. A partir de los resultados obtenidos podremos volver a refinar la metodología general, los modelos o la forma de calcular algunos de los parámetros.

1.5 Herramientas

Para implementar las rutinas autooptimizables deberemos utilizar el lenguaje de programación que más nos convenga, así como utilizar librerías que nos faciliten la programación paralela. Una vez implementadas se probarán en una máquina paralela. Los recursos y herramientas que vamos a utilizar son:

- Se programarán rutinas autooptimizables que propongan algunos valores de configuración antes de su ejecución. Estas librerías estarán programadas en C++. Programaremos sus versiones paralelas en memoria compartida (utilizando el estándar OpenMP) y en paso de mensajes (utilizando la librería MPI).
- La evaluación de las librerías diseñadas se llevará a cabo en la máquina del Grupo de Computación Científica de la Universidad de Murcia (Línea de Computación Paralela): SOL. El cluster está compuesto por 16 núcleos con procesadores Intel Xeon 3.00GHz de 64 bits, con 1.5 GB de RAM. Los núcleos se reparten de la siguiente manera: 4 núcleos en sol, nodo1 y nodo2 y 2 núcleos en nodo3 y nodo4.

1.6 Objetivos

Como ya se ha comentado, el objetivo principal del proyecto es diseñar técnicas de autooptimación para los parámetros que surgen al intentar aplicar paralelismo a técnicas de recorrido de árboles por medio de *backtracking*. Se persigue intentar liberar a los usuarios finales de conocimientos avanzados en paralelismo, detectando también un esquema algorítmico secuencial que se paralelice internamente. Además de este objetivo general, los principales objetivos que se pretenden conseguir son:

- Contrastar los resultados obtenidos con los del trabajo de investigación "Autooptimización en esquemas paralelos de recorrido de árboles de soluciones" [4].

- Estudiar con más detalle la estimación de los parámetros en los problemas secuenciales antes de pasar a los esquemas paralelos. Ampliar el número de esquemas secuenciales del problema de la mochila 0/1 con el que se está trabajando. En este caso ampliaremos a cinco versiones secuenciales que representarán mejor la variedad de entradas y comportamientos que se dan en los problemas reales.
- Introducir información de la instalación en la estimación de los parámetros influyentes en el tiempo de ejecución.
- Proponer un método de selección entre diferentes esquemas algorítmicos secuenciales que resuelven un mismo problema.
- Ampliar el número de esquemas paralelos estudiados para detectar el mayor número de problemas que se plantean cuando intentamos paralelizarlos.
- Estudiar como este tipo de técnicas de autooptimización se comportan al utilizarlas en problemas paralelos de memoria compartida. Este tipo de soluciones se podrán utilizar en los procesadores multicore que se comercializan actualmente en los ordenadores personales.
- Estudiar como este tipo de técnicas de autooptimización se comportan al utilizarlas en problemas paralelos de paso de mensajes.
- Suponer un avance en lo que a técnicas de autooptimización de esquemas algorítmicos de recorrido de árboles se refiere.

1.7 Contenido de la memoria

En esta sección se resume el contenido de los capítulos que componen este documento:

- **Capítulo 2: Recorrido de árboles de soluciones**: se repasarán los algoritmos de recorrido de árboles de soluciones, concretamente el de *backtracking*. Se comentarán cuales son los principales problemas que se dan en estos recorridos y los diferentes tipos de árboles que se pueden generar. Se enunciarán varios esquemas algorítmicos de *bactracking* comentando las diferencias entre ellos. Se pretende dar una idea general de la técnica y detectar los primeros parámetros que intervienen en los recorridos de *backtracking*.
- **Capítulo 3: Modelado de la técnica secuencial**: se estudiará el comportamiento de los recorridos secuenciales por medio de *backtracking*. Se extraerán parámetros que intervienen en su comportamiento y se modelará su tiempo de ejecución. Se propondrán diferentes mecanismos para estimar los parámetros, tanto en tiempo de instalación como en tiempo de ejecución. Se explicará un método general de toma de decisiones recabando información tanto en tiempo de instalación como en tiempo de ejecución.

- **Capítulo 4: Resultados experimentales secuencial. Ejemplo Mochila 0/1**: se analizan los resultados obtenidos al aplicar los conceptos del capítulo anterior al problema de la mochila 0/1. Pretende evaluar la calidad de las soluciones propuestas en el Capítulo 3.

- **Capítulo 5: Paralelización de los recorridos de *backtracking***: se estudian diferentes esquemas para paralelizar este tipo de técnicas. Se detectan los problemas generales que se nos pueden plantear en la paralelización. Se discutirán los beneficios de cada uno de ellos y sus diferencias.

- **Capítulo 6: Modelado de la técnica paralela**: se diseñará una metodología general para la toma de decisiones en esquemas paralelos de *backtracking*. Se estudiarán dos esquemas paralelos de *backtracking* detectando parámetros que intervienen en el tiempo de ejecución. Se comentarán las principales características que debe tener un esquema algorítmico de *backtracking* para realizar una paralelización implícita de él. Se adaptará la metodología general a los dos esquemas paralelos estudiados para su posterior estudio experimental.

- **Capítulo 7: Resultados experimentales paralelos. Ejemplo Mochila 0/1**: se explicará la implementación paralela de los esquemas anteriores en sus versiones de memoria compartida y memoria distribuida. Se analizarán los resultados experimentales de aplicar la metodología a dos esquemas paralelos que resuelven el problema de la Mochila 0/1.

- **Capítulo 8: Conclusiones y trabajos futuros**.

Capítulo 2

Recorrido de árboles de soluciones

En este capítulo vamos a comentar las características de la técnica de *backtracking* ejecutado secuencialmente y los principales problemas que nos encontramos en este tipo de recorridos. Una vez hecho esto, intentaremos modelar el tiempo de ejecución del algoritmo y se intentará obtener una fórmula teórica que ayude a estimar el tiempo de ejecución para entradas concretas y a realizar diferentes tipos de tomas de decisiones.

2.1 El espacio de búsqueda

Uno de los aspectos más importantes que debemos conocer a la hora de estudiar una determinada técnica algorítmica es el espacio de soluciones (árbol de búsqueda) por donde se va a mover. En el caso de la técnica que se va a estudiar, nos moveremos por un escenario con estructura de árbol.

Si representamos nuestras soluciones como una tupla de n elementos donde cada uno de ellos puede tomar un total de h valores, el árbol por el que nos moveremos tendrá una estructura similar a la de la Figura 2.1.

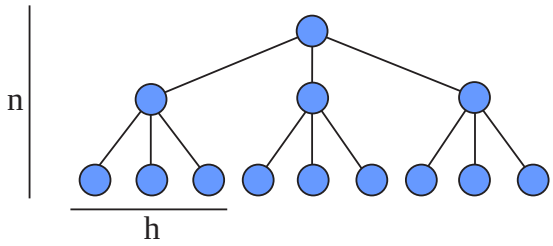


Figura 2.1. Espacio de estados generado por técnicas de recorrido de árboles.

Analíticamente nos moveremos en un árbol de búsqueda formado por un número de nodos total de:

$$\text{Nodos Totales} = \frac{h^{n+1} - 1}{h - 1} \in \theta(h^n) \tag{2.1}$$

Para hacernos una idea del espacio de estados en el que nos estaríamos moviendo:

- Ejemplo	n=4	h=4	nodos =	341
- Ejemplo	n=8	h=8	nodos =	19.173.961
- Ejemplo	n=13	h=13	nodos =	9.726.655.034.461

Recorrer todo este espacio de búsqueda supondrá un coste demasiado elevado que no se podrá permitir en todos los problemas. Por ejemplo, en un problema de investigación operativa donde se pretende encontrar el mejor camino desde una dirección origen a un destino en función del tráfico, no podríamos permitirnos que el tiempo de decisión fuesen varias horas, y sería conveniente resolverlo lo antes posible aunque el resultado encontrado no sea el óptimo. Sin embargo, en ocasiones la necesidad de aplicar este tipo de técnicas se debe a que se busca resolver problemas de optimización donde se deberán recorrer todas las posibles soluciones para poder obtener nuestra solución final, que además queremos que sea la óptima.

Por este motivo son interesantes técnicas que nos permitan acotar el árbol de búsqueda, recorriendo solo aquellos nodos que lleven con mucha probabilidad a las mejores soluciones. Incluso podríamos aplicar técnicas metaheurísticas que dejasen fuera soluciones óptimas, siempre y cuando la solución obtenida no esté muy alejada de la óptima.

2.2 Introducción a la técnica de *backtracking*

El *backtracking* (método de retroceso o vuelta atrás) es una técnica general de resolución de problemas, aplicable a problemas con diferentes objetivos: problemas donde se pretende encontrar **una solución** cualquiera a un problema, problemas de **optimización**, problemas en los que se desea encontrar **todas las soluciones** y problemas donde se **desconoce la existencia de una solución** y debemos por lo tanto recorrer todo el posible espacio de soluciones para saber si existe o no.

La técnica presenta más problemas de computación en problemas donde se persigue optimizar una función objetivo o encontrar todas las soluciones a un problema. En estos casos se necesitará realizar una búsqueda exhaustiva y sistemática en el espacio de soluciones por lo que suele resultar ineficiente.

Generalmente se expresará la solución como una tupla de n elementos $s(t_1, t_2, ..., t_n)$, satisfaciendo unas restricciones $P(t_1, t_2, ..., t_n)$ y tal vez optimizando una cierta función objetivo.

En cada momento, el algoritmo se encuentra en un cierto nivel l, con una solución parcial $(x_1, ..., x_l)$. Si se puede añadir un nuevo elemento a la solución (x_{l+1}), se genera y se avanza al nivel $l + 1$. En otro caso, se prueban otros valores de x_l que corresponderán con los nodos hermanos del nodo actual. Si no existe ningún valor posible por probar, se retrocede al nivel anterior, $l - 1$.

De esta manera recorreríamos todo el espacio de búsqueda y, dependiendo del tipo

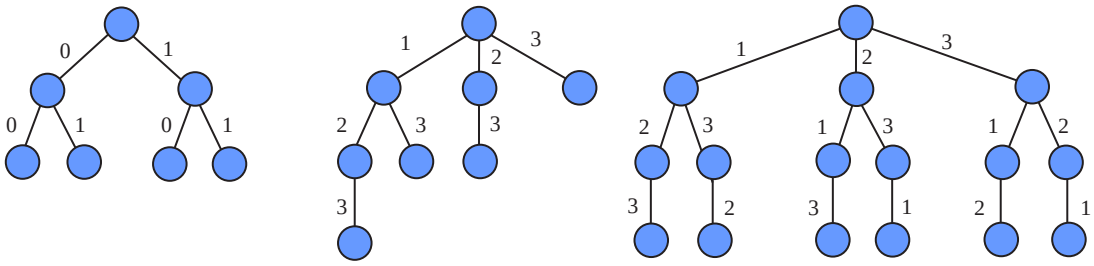
de problema que se esté resolviendo, el proceso continuaría hasta que se obtiene una solución del problema, o hasta que no queden más posibles soluciones que evaluar (se haya recorrido el árbol de búsqueda completo).

En su versión más simplificada el resultado es equivalente a hacer un recorrido en profundidad en el árbol de soluciones. Las secuencias de decisiones corresponden a árboles de soluciones, donde cada nodo representa una solución parcial, y en cada paso del algoritmo se está en un nodo del árbol diferente. Aunque el algoritmo tiene un coste computacional elevado, el coste en memoria es reducido ya que el árbol de búsqueda que estamos recorriendo es un árbol implícito; es decir, no se construye completo en memoria sino que únicamente se expande aquel nodo por el que nos estaremos moviendo.

Hasta ahora hemos hablado de varios tipos de esquemas de *backtracking*. Lo que se persigue en este proyecto es que los mecanismos de autooptimización que se diseñen sean independientes del esquema que se utilice. Se deberán predecir correctamente los parámetros ya estemos utilizando un esquema de optimización o uno de búsqueda de la primera solución.

2.3 Tipos de árboles de soluciones

En función de como se vayan generando las expansiones de los nodos del árbol y como sean las soluciones parciales de cada uno de los niveles, la estructura del árbol de búsqueda puede cambiar. Por este motivo, no todos los árboles serán simétricos como el que hemos visto en la Figura 2.1, ni tendrán que seguir ninguna estructura común. Dependiendo del problema se podrán generar distintos tipos de árboles de soluciones que harán a nuestra técnica comportarse de una manera concreta. Entre todos los tipos de árboles que se podrían generar hemos seleccionado los más representativos (representación en la Figura 2.2).



a) Árbol Binario b) Árbol Combinatorio c) Árbol Permutacional

Figura 2.2. Tipos de árboles de soluciones con diferentes estructuras.

- **Árboles n-arios**: en estos árboles se decide para cada nivel que valor incluir entre un total de h posibles valores. En este caso, el número de niveles coincide con la

cantidad de decisiones a tomar. Si se relaciona con la Figura 2.2.a, donde tenemos un árbol binario, la variable n representará el número de decisiones a tomar y la h tendrá valor dos. Los nodos terminales se encuentran en el último nivel.

$$\text{Número de nodos} = \frac{h^{n+1} - 1}{h - 1} \in \theta(h^n) \tag{2.2}$$

- **Árboles combinatorios**: en estos árboles en cada nivel se decide que valor incluir en la solución, y no importa el orden en que se incluyan los valores. Al igual que en el caso anterior, si se toma como referencia la Figura 2.2.b, el número máximo de niveles n coincide con la cantidad máxima de decisiones a tomar pero ahora el valor de h depende de cual sea el nivel en el que nos encontremos. Los nodos terminales se pueden encontrar en todos los niveles.

$$\text{Número de nodos} = 2^n \in \theta(2^n) \tag{2.3}$$

- **Árboles permutacionales**: en cada nivel se decide que valor incluir en la solución, e importa el orden en que se incluyen. El número de niveles n coincide con la cantidad de decisiones a tomar y el valor de h variará dependiendo de la profundidad del nodo.

$$\text{Número de nodos} = n + n(n-1) + n(n-1)(n-2) + ... + n! \in \theta(n!) \tag{2.4}$$

Generalizando estos tres tipos de árboles, es posible obtener otras estructuras donde para distintos niveles o desde distintos nodos la cantidad de decisiones a tomar sea distinta, con lo que nodos distintos pueden tener número de descendientes diferentes.

Es importante tener una idea de la estructura del árbol de búsqueda que utilizaremos en cada uno de los problemas para modelar mejor el comportamiento de este tipo de técnicas.

Destacar que en este proyecto no se pretende optimizar los *backtracking* que un usuario programe para conseguir recorridos más eficientes. Lo que se pretende es proponer un método que estime el comportamiento de un determinado esquema, con independencia del tipo de árboles que se empleen para resolverlo.

2.4 Esquema básico de *backtracking*

Una vez se han comentado las principales generalidades del *backtracking*, vamos a ver un primer esquema en su versión más simple (Código 2.1). Este es el esquema más simple que se puede encontrar. Lo que se conseguiría con él sería encontrar la primera solución válida a un determinado problema. En el momento en el que se encuentre se abortaría la búsqueda devolviendo la solución encontrada.

```
1  s = (0,0, ...0); nivel = 0; fin = false;
2
3  do {
4      s = Generar(s,nivel);
5
6      if ( ( Solucion(s,nivel) ) ) {
7          fin = true;
8      }
9
10     if ( (! fin) and (Criterio(s,nivel) ) ) {
11         // Si cumple el criterio no se poda.
12         nivel ++;
13     } else {
14         while ( (! fin) and (! MasHermanos(s,nivel)) ) {
15             Retroceder (s, nivel);
16         };
17     }
18 } while (! fin);
```

Código 2.1. Esquema básico de *backtracking*.

Para comprender con más detalle que hace realmente el algoritmo se va a explicar para que sirven y que es lo que hacen en su versión más básica cada una de estas funciones y variables:

Variables:

- s: almacena la solución parcial encontrada hasta un determinado momento. Representará en cada momento la parte del árbol (nodo) por donde se encuentra la búsqueda.
- *nivel*: es la variable que contiene el nivel l en que se encuentra el algoritmo.
- fin: es la variable booleana que controla si se ha encontrado la solución o no.

Funciones:

- *Generar*: esta función genera un nodo del árbol por el que va a continuar la búsqueda en el espacio de estados. Determinará la forma en la que se genera y se recorre el árbol.
- *Solucion*: devuelve a partir de s que se le pasa como parámetro si es solución al problema o no. Un nodo se considerará solución al problema cuando sea solución y cumpla las restricciones del enunciado. Los nodos solución pueden localizarse en nodos finales o intermedios.
- *Criterio*: es la encargada de devolver si el nodo que se le pasa como parámetro puede generar más nodos hijos. En su versión más simple se considera que un nodo podrá tener descendencia cuando no sea un nodo hoja.

- *MasHermanos*: si como consecuencia de haber recorrido todos los descendientes de un nodo se vuelve al mismo nodo, hay que generar un hermano de dicho nodo si lo hay. En caso contrario se volvería al nivel superior. Esta función es la que se encarga de determinar esta situación e informará si a partir de un nodo nos quedan más hermanos por recorrer.

- *Retroceder*: esta función se utiliza para retroceder de un nodo a su nodo padre.

Si se traza el algoritmo, se puede ver que el recorrido que se realiza en la versión más simple de un *backtracking* es un recorrido sistemático y en profundidad. En la Figura 2.3 se muestra como se comportaría este esquema básico para resolver un problema donde el número de decisiones a tomar es dos y solo podrían darse los casos de elegir o no elegir. Los descendientes de un nodo se generan de izquierda a derecha, pero el orden en que se generan puede variar siempre y cuando cambiemos la función *Generar* (por ejemplo, se podrían generar los hijos de derecha a izquierda o generar primero los que se considera que tienen más probabilidad de llevar a la solución que buscamos). Esto son modificaciones del algoritmo donde se buscan recorridos más eficientes del árbol, como explicaremos en el apartado siguiente.

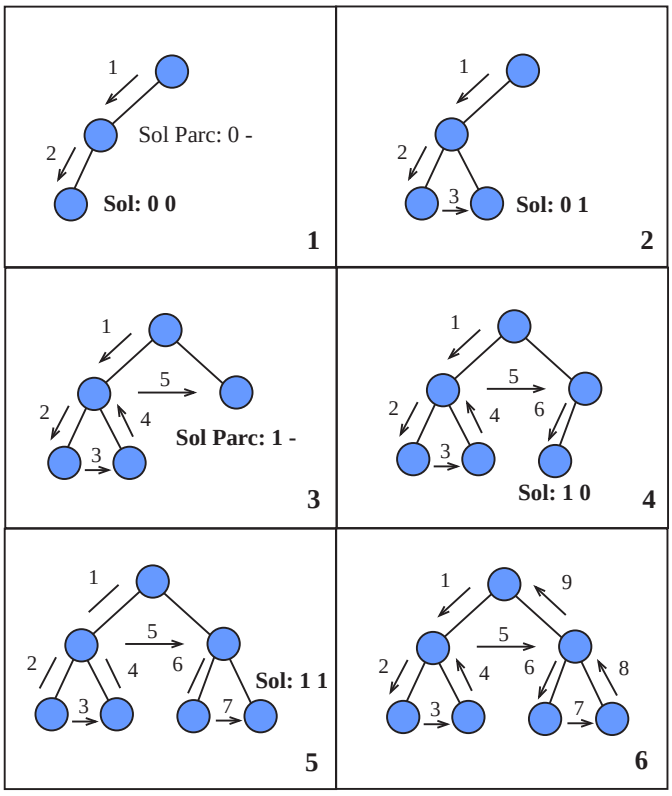


Figura 2.3. Recorrido paso a paso de un árbol binario por medio de un esquema básico de *backtracking*.

2.5 Esquema para problemas de optimización

Hasta ahora se ha visto un esquema básico de *backtracking* que realiza una búsqueda recorriendo todos los nodos del espacio de búsqueda. A medida que aumentamos el tamaño del problema, el tiempo de ejecución de estas rutinas puede pasar de segundos a minutos, horas, días, meses... Por este motivo se deben buscar técnicas que nos permitan optimizar nuestras búsqueda evitando recorrer y generar nodos que sabemos que no nos llevan a ninguna solución, o que simplemente no nos llevarían a soluciones prometedoras.

En el Código 2.2 se muestra un esquema de *backtracking* para problemas de optimización. Los algoritmos de *backtracking* se suelen aplicar en la resolución de un gran número de problemas, y muy especialmente en los de optimización. Para resolverlos, debemos recorrer todo el árbol de búsqueda para dar con aquella solución que optimice una función objetivo.

```
1  s = (0,0,...0), nivel = 0;
2
3  do {
4      Generar (s, nivel);
5
6      // Si se ha llegado a un nodo solucion y su valor mejora el valor optimo
7      // actual lo actualizamos.
8
9      if ((Solucion (s, nivel)) and (valor (s, nivel) < VOA)) {
10
11         VOA = valor (s, nivel);
12     }
13
14     // Determinamos si podemos podar el nodo actual.
15
16     if ( Criterio (s, nivel, VOA) ) {
17         // Si cumple el criterio no se poda.
18         nivel ++;
19     } else {
20         while (  ( nivel >= -1 ) and ( ! ( MasHermanos (s, nivel) ) )) {
21             Retroceder (s, nivel);
22             nivel --;
23         };
24     }
25 } while ( nivel != -1);
```

Código 2.2. Esquema modificado de *backtracking* para problemas de optimización.

A estos problemas donde es necesario aplicar podas nos vamos a referir a partir de este punto del proyecto. Será en ellos donde se plantea un reto para predecir su comportamiento. Es interesante identificar parámetros que intervengan en su ejecución, para poder ajustarlos o estimarlos previamente en un modelo teórico.

Para agilizar las búsquedas, no recorreremos el árbol de búsqueda completo. Intentaremos eliminar nodos cuando consideremos que estos no nos llevarán a soluciones mejores que la última encontrada. Cuando se decide no expandir un nodo se está realizando su poda. Estas podas disminuyen significativamente el espacio de búsqueda a recorrer. Nos moveremos en árboles donde se podrá saber a priori su estructura, pero no la parte del espacio que se recorre para cada una de las entradas. Solo seremos capaces de saber este

valor una vez resuelto el problema. Ya no nos movemos por el espacio completo de la Figura 2.1 sino por uno algo más parecido al de la Figura 2.4.

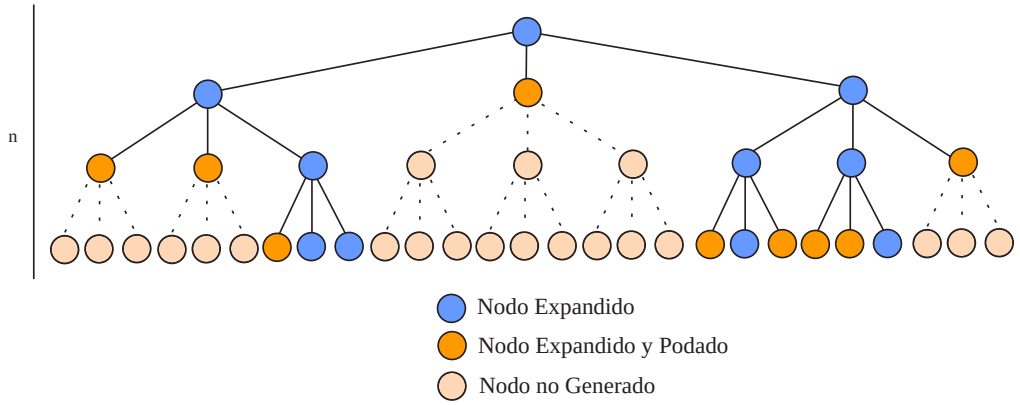


Figura 2.4. Espacio de estados donde se realizan podas.

Para realizar las podas se introduce en el Código 2.2 el *VOA* a la función criterio. Una posible forma de podar sería: si el mejor valor esperado asociado a un nodo no mejora el mejor valor encontrado hasta el momento (*VOA*) se poda dicho nodo. El tipo de podas empleadas puede ser tan dispar como el usuario encargado de programar la rutina decida.

En los árboles n-arios, el crecimiento del espacio de búsqueda es exponencial. Por ello cobran especial importancia cualquier tipo de podas cuando nos enfrentamos a tamaños de entrada grandes. Para estos tamaños una poda en un nivel superior del árbol tendría una reducción considerable en el número total de nodos a recorrer. La expresión analítica que nos da el total de nodos que nos ahorramos en una poda a nivel *Nivel de Poda* es:

$$Nodos\ Podados = \frac{h^{n-Nivel\ de\ Poda+1} - 1}{h - 1} - 1 \tag{2.5}$$

A modo de ejemplo se incluye en la Tabla 2.1 el número de nodos eliminados en un determinado espacio de búsqueda, haciendo las podas en distintos niveles. Se puede apreciar cuales serían los pesos de cada una de esas podas.

Es tal la importancia de realizar buenas podas para acotar las búsquedas con técnicas de *backtracking*, que a veces se eliminan nodos utilizando heurísticas, de manera que aunque queden fuera soluciones que podrían ser óptimas se reduzca el tiempo de ejecución consiguiendo soluciones aceptables.

n	**h**	**N.Tot**	**Nv.Poda**	**Nd.Podados**	**Nd.Pod / Nd.Tot**
2	3	13	1	4	30,76 %
			2	1	7,69 %
5	3	364	1	121	33,24 %
			2	40	10,98 %
			3	13	3,57 %
			4	4	1,09 %
			5	1	0,27 %

Tabla 2.1. Ejemplos del número de nodos podados a distintos niveles del espacio de búsqueda.

Capítulo 3

Modelado de la técnica secuencial

Una vez conocemos las características y el comportamiento de la técnica de *backtracking*, vamos a extraer los primeros parámetros que intervienen en su comportamiento. El objetivo que se persigue en este proyecto es intentar modelar la técnica para ser capaces de poder estimar parámetros de configuración, clasificar diferentes esquemas de *backtracking* según diferentes criterios... Se pretende diseñar y construir una metodología capaz de inferir estas decisiones.

A partir del modelo matemático, podremos obtener aproximaciones que nos permitan saber cuales serían los valores adecuados de los parámetros configurables e influyentes en el tiempo de ejecución. Veremos así los primeros problemas que van a surgir a la hora de modelar el comportamiento de las técnicas de *backtracking*.

3.1 Modelo del esquema básico

Consideramos el esquema más básico en el que no se realizan podas. Es interesante modelar su comportamiento para ir paulatinamente consiguiendo modelos de esquemas más complejos. En este esquema se generaría un espacio de búsqueda simétrico (todos los nodos tienen el mismo número de hijos), similar al de la Figura 2.1. Se recorre todo el espacio de búsqueda evaluando cada nodo para comprobar si es solución o no. Por lo tanto, una primera aproximación para sacar un modelo del tiempo de ejecución de la rutina podría ser:

$$TE(n) = NNG \cdot TCN \tag{3.1}$$

donde TE representa el "Tiempo de Ejecución". La variable NNG equivale al "Número de Nodos Generados". Este término dependerá del tipo de árbol que genere el problema al que nos enfrentamos, nosotros lo vamos a estudiar con árboles n-arios y extraemos su valor de la Ecuación 2.2. TCN es el "Tiempo de Cómputo de un Nodo".

De esta manera sabemos cuanto se tarda en recorrer todo el espacio de búsqueda. El problema es que no se sabe exactamente cuando se activaría la condición de parada,

es decir, cuando se encuentra la solución. Esta condición de parada dependerá de los problemas y de las entradas, y no se podrá conocer a priori.

3.2 Modelo del esquema de optimización

El segundo esquema estudiado (esquema de optimización) incluye un concepto interesante para un buen modelado de la técnica: las podas de nodos que no son necesarios expandir. Aunque en el esquema propuesto solamente hemos considerado podas en función del valor óptimo actual encontrado, estas podrían ser tan diversas como se quisiera.

3.2.1 Modelo 1 - Sin considerar niveles

En una primera aproximación se podría pensar que el modelo de tiempo para estos esquemas es similar al de la Ecuación 3.1. Ahora no tendríamos como condición de parada encontrar una solución, sino recorrer todo el espacio de búsqueda. El problema es conocer que porcentaje exacto de nodos se recorre. Para poder predecir correctamente el comportamiento de nuestros algoritmos, deberemos diseñar mecanismos que nos permitan estimarlos reflejando la realidad.

Surge la pregunta de como podríamos modelar este comportamiento del algoritmo. El número de podas es dependiente de un gran número de factores: la forma del árbol, el valor de las entradas, como se recorre el árbol, el orden de generación de los nodos...

Una posible solución a este problema es introducir un parámetro probabilista que represente cual es el porcentaje de nodos no podados para una determinada entrada. A este porcentaje lo llamaremos k, con $\{k \, \epsilon \, \mathbb{R} \mid 0 \leq k \leq 1\}$.

De este modo podremos saber cual es el porcentaje de nodos que se han podado, y por tanto el número total de nodos recorridos. A través de una variación de la Ecuación 1.1 podemos obtener cual es el número total de nodos que evitamos expandir cuando realizamos una poda a determinado nivel, es decir, cuantos hijos generaría este nodo. Se puede representar el valor de k como:

$$k = 1 - \frac{\text{Nodos Podados}}{\text{Nodos Totales}} \tag{3.2}$$

La ecuación final que modela el comportamiento del esquema de optimización realizando podas en los nodos quedaría:

$$TE(n) = k \cdot NNG \cdot TCN \tag{3.3}$$

Veámoslo con un ejemplo donde se realizan dos podas de nodos del árbol (Figura 3.1). Calculando estos valores en la Tabla 3.1 , para la “Poda 1” hecha a nivel dos se recorren

Poda	Nivel	NumPodados	Totales
Poda 1	2	2	
Poda 2	1	6	
		8	15

$k = 0.466666667$
T.Ej = 7 T.Comp

Tabla 3.1. Cálculo del parámetro coeficiente de poda para el ejemplo de la Figura 3.1.

dos nodos menos y para la "Poda 2" a nivel uno se evita recorrer un total de seis nodos.

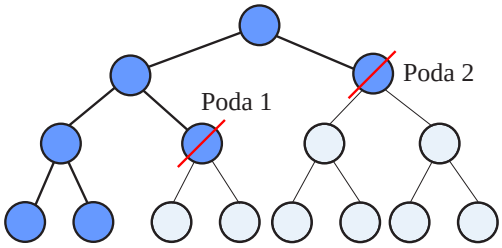


Figura 3.1. Ejemplo de recorrido donde se podan nodos.

Este es un ejemplo ilustrativo de que introduciendo el coeficiente de poda k en el modelo, la técnica de estimación funcionaría bien. El problema es la dificultad de calcular el valor de k. En este ejemplo, se obtiene una vez resuelto el problema para una entrada concreta. El valor de k es dependiente de cada entrada y necesitamos diseñar una estrategia que nos permita estimar su valor para un esquema determinado, con independencia de la entrada a resolver. Este cálculo representa un verdadero problema por enfrentamos a un rango de entradas grande. Dos entradas diferentes podrían llevarnos a valores de k completamente distintos.

3.2.2 Modelo 2 - Considerando niveles

El modelo visto hasta ahora presentará problemas cuando las operaciones que realizamos a distintos niveles sean diferentes. Por ejemplo si hacemos un avance rápido para calcular el valor asociado a un nodo, este tardará más en los primeros niveles y menos en los niveles más próximos a los nodos hojas. Podemos por tanto proponer un nuevo modelo más preciso que aproxime mejor el comportamiento de nuestras rutinas.

$$TE = k \cdot \left(\sum_{i=1}^{NumNiveles} NNV_i \cdot TCN_i \right) \tag{3.4}$$

La variable NNV_i se corresponde con el número de nodos visitados a nivel i. De la misma manera, TCN_i equivale al tiempo de cómputo de un nodo a nivel i.

3.3 Metodología para la toma de decisiones

Dependiendo del tipo de problemas que queramos resolver y la naturaleza de los mismos, podríamos considerar diferentes enfoques a la hora de desarrollar nuestro método de trabajo:

- **Caja Negra**: los usuarios de nuestras librerías disponen de esquemas algorítmicos de *backtracking* ya compilados donde no podemos modificar el código. Por las propiedades que tienen este tipo de técnicas intentar extraer información de ellos sería complejo. Nos encontramos con problemas donde no es fácil modelar su comportamiento simplemente observando ejecuciones de los mismos sin poder obtener otra información más detallada de su evolución.
- **Caja Negra + Modelo de Tiempo**: otra posibilidad es que los usuarios proporcionasen las rutinas ya compiladas y además un modelo de tiempo que represente su comportamiento. La rutinas ofrecerían además de la solución al problema, la información de los parámetros que se definan en el modelo. Este enfoque tampoco sería interesante al obligar a los usuarios de nuestras rutinas a identificar un modelo de tiempo para cada uno de los esquemas que queramos estudiar. Lo que perseguimos es abstraerlos lo máximo posible de detalles fuera de la implementación para poder extender nuestras tomas de decisiones tanto como podamos.
- **Esquema algorítmico**: por último podríamos proporcionar a los usuarios de nuestras librerías un esquema algorítmico donde únicamente tendrían que completar las funciones propias de *backtracking* [15]. De esta manera nosotros, como dueños de las librerías, podríamos añadir programación adicional que nos proporcionarse información interesante para trabajar con el modelo teórico propuesto: tiempo total empleado, tiempo medio por nodo, número de nodos generados, número de nodos podados, tiempo medio por nodo/nivel... Así podríamos modificar e incluso utilizar varios modelos de tiempo de manera transparente al usuario. Simplemente tendríamos que añadir al esquema instrucciones para conseguir la información que necesitamos.

Al enfrentarnos a problemas dependientes de las entradas sin tener acotados sus valores, vamos a elegir una estrategia del tercer tipo al ser la que más información y posibilidades nos dará. En una primera versión no vamos a proporcionar ningún esquema algorítmico. Nosotros nos consideraremos los responsables de las rutinas que el usuario quiera utilizar. Podremos añadir funcionalidad que complete la información que se necesite siempre y cuando no varíe la funcionalidad que el programador implementó.

Independientemente de los modelos utilizados y los parámetros o decisiones a tomar, deberemos diseñar una **herramienta** y una **metodología** que permita obtener la

información necesaria para tomar decisiones. Tal y como se comentó en el capítulo de introducción vamos a dividir nuestra metodología en dos partes:

- Durante la **instalación** de la rutina se realizarán diferentes pruebas que nos den información general para poder estimar los parámetros. Los resultados obtenidos en esta parte deben agilizarnos el tiempo de toma de decisión en la ejecución para una entrada concreta. El tiempo dedicado y el grado de detalle en esta parte podrá ser configurado por los usuarios.
- A menudo la información obtenida en la instalación deberá ser completada con otra extraída en cada una de las **ejecuciones**. Esta información de la ejecución será relativa a cada entrada particular que deseamos resolver, por lo que nos puede dar información más precisa para complementar la información genérica que obtenemos en la instalación.

Nosotros nos consideramos los **diseñadores** de todo el sistema. Además vamos a establecer diferentes roles para los usuarios que utilizarán nuestras librerías. Destacamos la importancia de estos roles por tener cada uno de ellos diferentes responsabilidades. En algunos casos una misma persona puede jugar más de un rol:

- **Usuarios:** son los usuarios finales que utilizarán las rutinas de *backtracking*. No conocen las máquinas donde se ejecutarán los programas y quedan exentos de configurar cualquier parámetro.
- **Programadores:** son los encargados de programar las rutinas que desean utilizar los usuarios. Conocen los detalles de implementación pero tampoco es su responsabilidad configurar los parámetros necesarios para autooptimizarlas.
- ***Manager:*** es el administrador de la máquina donde se van a ejecutar las rutinas. Deberá conocer los detalles de su configuración. Será el encargado de la instalación de las rutinas y la correcta configuración de los parámetros necesarios. Conocerá que tipo de problemas desean resolver los usuarios finales para configurar correctamente las rutinas, obteniendo los usuarios finales resultados adecuados.

3.3.1 Herramienta de instalación

Las pruebas que realicemos durante la instalación deben ser de calidad, es decir, las deberemos realizar generando entradas aleatorias tan variadas como nos sea posible para obtener datos realistas. No serviría de nada realizar pruebas durante la instalación en un rango pequeño de valores si las entradas que después se van a resolver no representan al mismo tipo de problemas.

En este proyecto proponemos una estrategia de instalación genérica que consiste en generar una serie de problemas aleatorios, resolverlos con la rutina que se está instalando y

obtener resultados que nos ofrezcan la posibilidad de prever futuros valores. El problema es, ¿cuanto tiempo dedicamos a estas pruebas? Al estar trabajando con la técnica de *backtracking* sabemos que los tiempos de ejecución para valores relativamente grandes de las entradas puede ser elevados. Por este motivo se diseñará un mecanismo que ofrezca una serie de parámetros configurables por el *manager* para instalar la rutina. Por medio de estos parámetros el *manager* podrán decidir cuanto tiempo decide dedicar a la instalación, así como la precisión de las pruebas que quiere realizar. El tipo de entradas aleatorias generadas deberá ser similar a las entradas finales que los usuarios resolverán.

Los parámetros identificados para nuestra rutina de instalación son:

- **Tiempo máximo de instalación**: es el tiempo máximo que queremos dedicar en la instalación a realizar pruebas para obtener datos.
- **Tamaño mínino del problema**: es el tamaño mínimo del problema que queremos resolver.
- **Tamaño máximo del problema**: es el tamaño máximo del problema que queremos resolver.
- **Incremento**: es el incremento en el tamaño de los problemas generados que queremos resolver.
- **Número de tests**: es el número total de pruebas que se van a hacer para cada uno de los tamaños.

Una vez explicados los parámetros, lo que se hará será generar un total de "Número de tests" pruebas aleatorias para cada valor desde "Tamaño mínimo del problema" hasta "Tamaño máximo del problema" incrementando según la variable "Incremento". Aunque estos parámetros tendrán unos valores por defecto el *manager* los podrá modificar. Tenemos por tanto dos condiciones de parada para esta estrategia de instalación: que se hayan generado todos los subproblemas que nosotros hemos establecido o que haya pasado un tiempo máximo. Una vez finalizada la instalación dispondremos de información genérica del esquema que nos permitirá poder tomar nuestras primeras decisiones.

Dependiendo del tiempo que queramos dedicarle a la instalación podremos obtener más y mejores valores. Será decisión del *manager* que instala nuestras rutinas elegir el nivel de detalle que quiere emplear. Lo podrá configurar ajustando los parámetros que hemos indicado anteriormente.

En la Figura 3.2 tenemos un diagrama de la metodología propuesta. En una primera **fase de programación**, los programadores deberán implementar las funciones propias del esquema de *backtracking* y una rutina capaz de generar entradas aleatorias para el problema que queremos resolver. Esta rutina podrá ser también parametrizada para generar entradas tan diferentes como se desee. Durante la **fase de instalación**, la "Rutina Instalación" será configurada por el *manager* del sistema para que realice las pruebas adecuadas. La configuración se hará por medio de los "Param. Config.". Esta rutina

utilizará la rutina de *backtracking* que deseamos instalar, generando entradas aleatorias por medio de la función "generarEntrada()". Se almacenarán los resultados en un fichero ("Fichero Información Esquema") que será utilizado en la ejecución. En **fase de ejecución** conoceremos la entrada específica que el usuario desea resolver. El modelo de tiempo interpretará la información extraída en "Fichero Información Esquema" para pasársela a la "Rutina Ejecución". La "Rutina Ejecución" se encargará de inferir información a partir de la información específica de la entrada y la información que le proporcione el modelo. Los objetos en color verde y azul representan binarios de nuestros programas, el color rojo ficheros programados por los usuarios y los amarillos ficheros de texto que almacenan información.

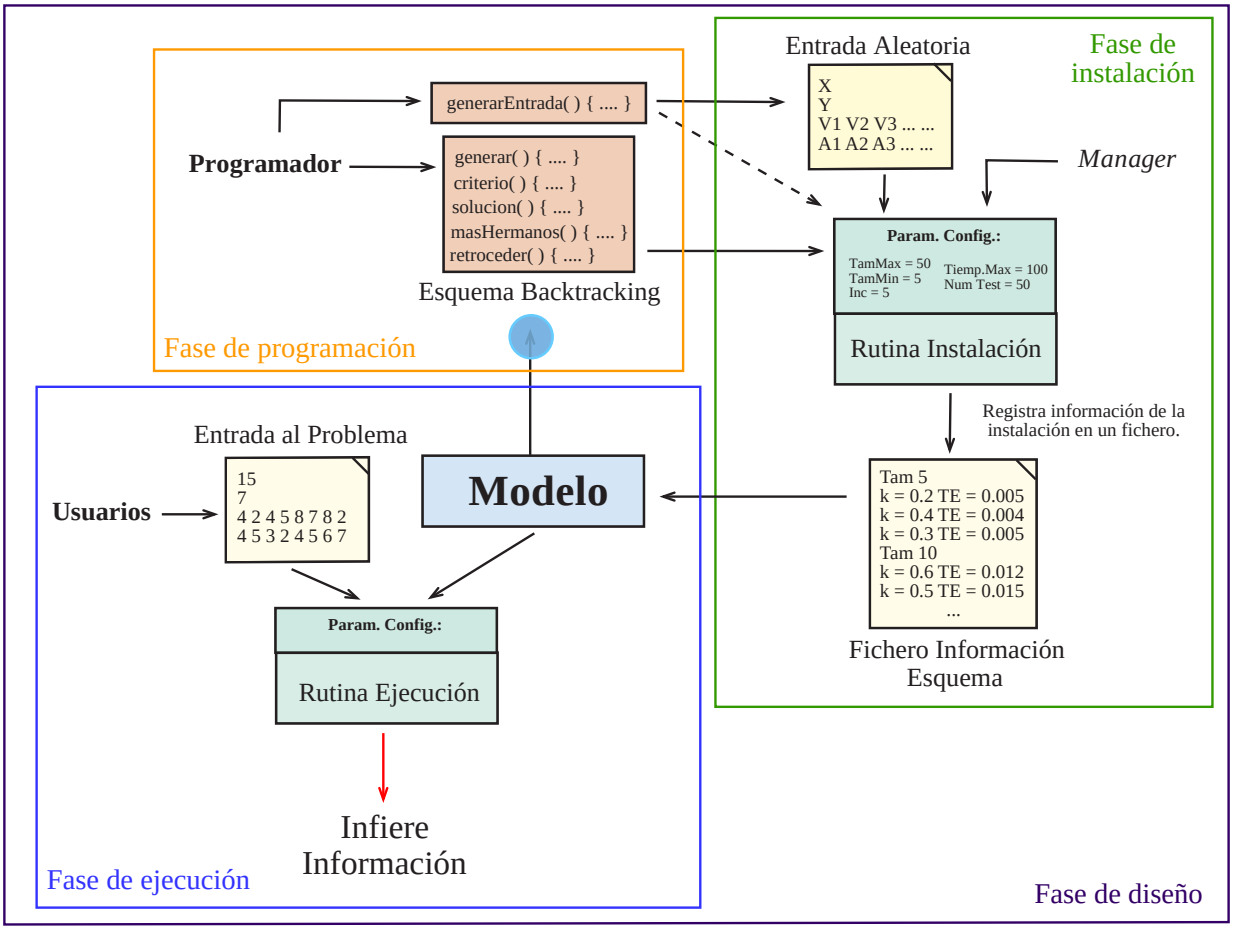


Figura 3.2. Esquema general de la metodología propuesta.

Como ya discutimos en la sección 3.3 (Estrategias de estimación), el modelo de tiempo debe ser establecido por los programadores o bien impuesto por los creadores de las rutinas autooptimizables añadiéndolo en el esquema a completar. Por este motivo se considera que la presencia del modelo está tanto en la fase de diseño con en la fase de ejecución. En la fase de diseño se podría encontrar directamente definido por el programador o de forma implícita en el esquema que debe de programar; en este segundo caso es el que hemos elegido para nuestro proyecto. En la fase de ejecución estará presente para interpretar la

información extraída previamente en la instalación.

El tipo de información inferida dependerá del tipo de problema que estemos resolviendo. En caso de un problema secuencial el usuario podría darnos un conjunto de esquemas donde podremos considerar como parámetro de decisión cual es el mejor a utilizar. Si consideramos una implementación paralela de la técnica, podemos identificar otros parámetros como número de procesadores, nivel de profundidad del proceso maestro... El valor de estos parámetros puede ser la decisión que queremos tomar, y siguiendo un enfoque similar al secuencial se puede también decidir el mejor de entre varios esquemas.

El método general propuesto anteriormente es válido para instalar un solo esquema y decidir los mejores valores de parámetros que hayamos identificado. Para decidir entre diferentes esquemas de un problema secuencial podemos generalizar la metodología. La instalación será realizada para cada uno de los esquemas que queramos instalar. Se guardará la información de cada uno de ellos. En tiempo de ejecución la "Rutina de Decisión" utilizará el modelo y opcionalmente la entrada para decidir cual es el mejor esquema para su resolución.

3.3.2 Recursos implementados por el programador

La metodología propuesta necesita que el programador de las rutinas implemente una serie de recursos. Los recursos que se le solicitan a los programadores son:

Generador de entradas

Al trabajar con una técnica altamente dependiente de las entradas y sin conocer con precisión que problemas nos presentarán los usuarios, debemos diseñar el método de estimación de parámetros lo más general posible.

Durante el periodo de instalación deberemos sacar pruebas sobre datos que representen el conjunto de problemas que el usuario va a resolver. El programador será el encargado de establecer cual es el formato de los ficheros de entrada, por este motivo le solicitamos que adicionalmente programe un generador de problemas aleatorios. Sería interesante que este generador de problemas esté parametrizado para configurar los diferentes tipos de problemas que queremos generar. Será obligación del *manager* conocer cual es el rango de problemas que querrán resolver los usuarios finales, por ello configurará durante la fase de instalación el generador de entradas para que se obtengan valores similares a los que los usuarios finales se enfrentarán. La calidad de los datos obtenidos irá en función de la configuración establecida. Seguramente generar problemas dentro de un intervalo mayor nos ampliará el tiempo necesario para obtener resultados durante la instalación a cambio de un mayor realismo de los resultados.

Esquema de *backtracking*

Es obvio que el programador deberá completar el esquema de *backtracking* que quiere utilizar para resolver los problemas. Si decide utilizar nuestras rutinas de elección del mejor esquema proporcionará más de un esquema para que las rutinas utilicen el que crea más conveniente en tiempo de ejecución.

El generador de entradas y el esquema de *backtracking* serán lo único que los usuarios de nuestras librerías deban programar. A partir de este punto las librerías serán las encargadas de inferir la información necesaria.

3.4 Estrategias de estimación de los parámetros

Una vez explicada la metodología, debemos centrarnos en elegir diferentes modelos y proponer mecanismos para estimar los parámetros que se nos presentan. Ya propuestos dos modelos matemáticos que podrían representar el comportamiento de nuestra técnica, se deben proponer una estrategias y técnicas para estimar los parámetros que hemos identificado en ellos (TCN y k). Cada uno de estos parámetros se deberán analizar de forma independiente. Aspectos importantes de ellos serían:

- k: como hemos visto este coeficiente depende altamente de las entradas del problema y no va a ser fácil encontrar una forma de estimar su valor de forma general.
- TCN: se deberá estimar el tiempo de cómputo empleado para cada nodo generado. En ocasiones el tiempo de cómputo será el mismo para todos los nodos del árbol, independientemente del nivel. Sin embargo, vimos que en ocasiones puede ser más complejo de estimar, lo que ha dado lugar al segundo modelo de tiempo (Ecuación 3.4).

Pasemos a estudiar y proponer diferentes formas de estimarlos comentando las ventajas y los inconvenientes.

3.5 Estimación del porcentaje de nodos generados (k)

Estudiaremos mecanismos para estimar el parámetro k en tiempo de instalación, proponiendo modificaciones para actualizar esta información con la obtenida en tiempo de ejecución. Vamos a proponer diferentes esquemas que creamos que pueden ser buenos para aproximar nuestras decisiones. Los contrastaremos con los resultados prácticos del siguiente capítulo.

3.5.1 En tiempo de instalación

En un primer enfoque propondremos mecanismos que solo tienen en cuenta la información extraída en tiempo de ejecución para estimar el valor de k.

Valor constante

Una primera forma de obtener información sobre el parámetro k es realizar pruebas durante la instalación y obtener una media del índice de poda de todas las ejecuciones. Podremos evaluar cuando un esquema poda más o menos según esta media que hemos obtenido:

$$k_{Media} = \frac{\sum_{i=1}^{NumTest} k_{Ejecucion(EntradaAleatoria_i)}}{NumTest} \tag{3.5}$$

El principal problema lo encontramos en que dos valores de entrada que se encuentren en extremos opuestos del rango pueden darnos valores de k completamente diferentes. Sin embargo, la media de todos los valores resulta interesante para obtener información general de los esquemas, que podrá ser complementada con otra información que la refine.

Al generar esta técnica una medida general, es muy importante que las "EntradaAleatoria" cubran el mayor rango posible de valores de las entradas. De esta manera obtendremos valores genéricos, dispares y realistas. En caso de que el usuario vaya a utilizar la rutina para resolver problemas encuadrados dentro de un rango bien definido, la estimación de esta k_{Media} podrá hacerse para valores de dicho rango.

Extrapolando valores ya obtenidos

El enfoque anterior plantea una serie de problemas. Generalmente el índice de poda $(1 - k)$ no será el mismo para todos los tamaños de entrada. Podría ser interesante aumentar el nivel de detalle a la hora de estimar el valor de k y obtener diferentes índices en función del tamaño de problema que vamos a resolver. Esta forma de obtener k sería más realista que la anterior.

Normalmente el índice de nodos podados seguirá una tendencia similar a medida que aumentamos el tamaño del problema. Proponemos como posible forma de estimar los valores de k buscar dentro de los datos que hemos calculado en la fase de instalación. Podemos almacenar los valores que obtenemos en una tabla que contiene los valores medios de k para cada uno de los diferentes tamaños de problema que hemos resuelto. Por ejemplo podríamos tener los valores de índice de poda $(1 - k)$:

	tam=10	**tam=20**	**tam=30**	**tam=40**
k_{media}	0.59	0.76	0.86	0.93

Para un valor de entrada *tamEntrada* podríamos buscar dentro de los valores que tenemos en nuestra tabla y aproximar al que más cerca se encuentre. Esta opción sería buena cuando el banco de datos sea muy amplio y cubra un amplio rango de los tamaños de las entradas que vamos a resolver.

Si no tenemos información para un tamaño de problema determinado podríamos extrapolar los valores que tengamos para estimar los que desconocemos. Si representamos estos valores podríamos tener un gráfico similar al de la Figura 3.3. En esta figura el eje x representa los diferentes tamaños de problema que hemos tenido que resolver. El eje y representa los valores del coeficiente de nodos totales generados, k.

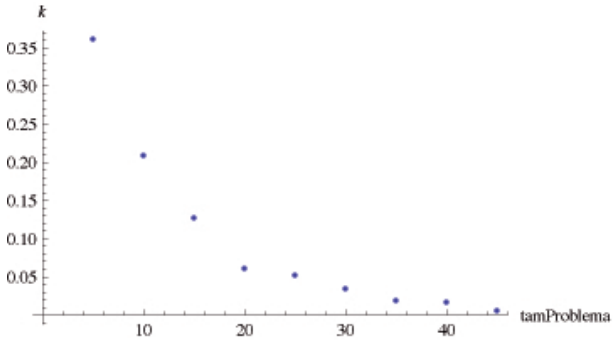


Figura 3.3. Valores de k medios para diferentes tamaños de entrada.

Vamos a intentar extrapolar por medio de una recta aquellos valores que desconozcamos. En la Figura 3.4 se muestra el procedimiento que se sigue. Cuando deseemos conocer el valor de k para un tamaño del que no tenemos registrada información buscaremos entre nuestros valores de medias el último punto anterior (a) y el primer punto siguiente (b). A partir de estos dos puntos se calculará la ecuación de la recta que pasa por ellos $f(x)$. A partir de esta recta estimaríamos el valor de k que obtendríamos con $x = \textit{Tamaño a Estimar}$.

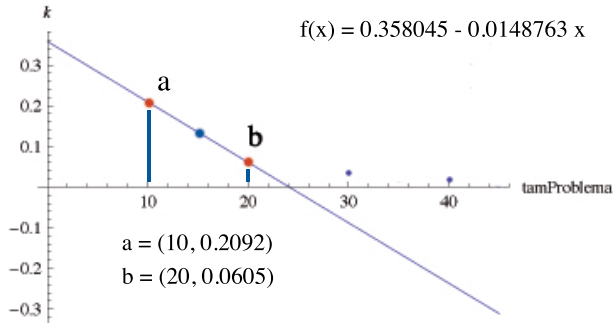


Figura 3.4. Aproximación de valor de k por medio de una recta que pasa por dos puntos conocidos.

Sin embargo esta solución genera algunos problemas. Si el valor que queremos calcular es anterior o posterior a todos los puntos almacenados durante la instalación no podemos obtener el punto anterior x_0 y el posterior y_0. En estos casos podríamos calcular la ecuación de la recta a partir de los dos primeros o los dos últimos puntos respectivamente. En

esta situación se nos plantea el problema de que la pendiente de la recta sea demasiado elevada y la función que describe la recta podría dar valores inconsistentes para nuestro modelo (valores menores que cero o mayores que uno). En estos casos aproximaremos la solución a un valor cercano a cero (0.00001) o a un valor cercano a uno (0.99999) según corresponda.

Variable por medio de una función

En problemas donde la tendencia de los resultados es fácil de predecir, podría ser más adecuado intentar aproximar a una función conocida. De esta manera la información que predecimos podría abarcar un rango mayor, con una mejor aproximación.

En este tipo de problemas el crecimiento del árbol de búsqueda será exponencial respecto al tamaño del problema. El índice de poda tendrá una tendencia similar. A partir de diferentes pruebas realizadas obtuvimos los valores de la Figura 3.3. En este caso parece que sería lógico aproximar a una función $\frac{a}{x}$, donde a es el parámetro a ajustar. Podríamos realizar un ajuste por mínimos cuadrados a esta función. Podemos apreciar el resultado en la Figura 3.5.

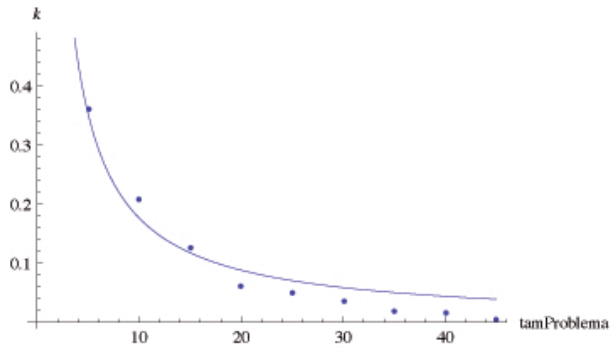


Figura 3.5. Aproximación del valor de k por medio de la función $\frac{a}{x}$.

El problema es que para otros problemas el ajuste realizado con $\frac{a}{x}$ no sea del todo bueno. Sería interesante ofrecer la posibilidad de elegir entre varias funciones de aproximación. Las funciones que se utilicen dependerán de la tendencia que el *manager* observe en los datos obtenidos durante la instalación. Por ello el papel del *manager* adquiere mayor importancia, al ser el encargado de instalar nuestras rutinas podría supervisar la calidad del ajuste y ser el encargado de elegir la función a ajustar.

Si comparamos la propuesta de extrapolar valores con la de estimar por medio de una función, el ajuste a una función es una solución mucho más genérica. Cuando no sabemos que tipo de problemas son los que vamos a resolver, sería más conveniente utilizar la información de la instalación y extrapolar valores. La información de la instalación representará mejor el comportamiento que fijando a una función concreta.

3.5.2 En tiempo de ejecución

Al trabajar con problemas con alta dependencia de las entradas sería interesante completar la información general de la instalación con otra más refinada y adecuada al tamaño y los valores del problema que obtenemos en la ejecución.

La información obtenida en la ejecución modificará la información $k_{TiempoInstalacion}$ en un porcentaje determinado previamente. Los dos tipos de informaciones obtenidas se consideran interesantes para estimar la información final. Daremos un peso del 50 por ciento a cada una, pero este valor podrá ser configurado por el *manager*. La k_{media} estimada con información conjunta del tiempo de instalación y el tiempo de ejecución quedaría como:

$$k_{media} = 0.5 \cdot k_{TiempoInstalacion} + 0.5 \cdot k_{TiempoEjecucion} \tag{3.6}$$

Vamos a proponer dos posibles métodos para obtener información durante la ejecución:

Agrupando las entradas y generando problemas más pequeños.

Podemos reducir la entrada a tamaños del problema que sabemos que se han resuelto en tiempo aceptables. De las pruebas realizadas en tiempo de instalación tenemos los tiempos medios que hemos tardado para cada uno de los diferentes tamaños de los problemas. A partir de esta información se seleccionará un valor al que reducir el tamaño del problema que no consuma un tiempo excesivo.

Para generar la entrada de menor tamaño utilizaremos la entrada a resolver. En la Figura 3.6 se muestra gráficamente como se generaría una entrada de menor tamaño para el problema de la Mochila 0/1. La entrada reducida que se genera debe de guardar las mismas propiedades que la entrada original para que los datos obtenidos representen la realidad. Un forma de guardar estas relaciones es no modificar los valores de la entrada sino agruparlos. La capacidad de la mochila seguirá siendo la misma. Se reducirá el tamaño del problema reduciendo el número de objetos. Agruparemos valores contiguos sumando sus valores. Se agruparán de manera que se consiga tener una entrada del nuevo tamaño reducido que nos hemos propuesto. La nueva entrada reducida estará formada por los valores agrupados de pesos y beneficios.

Seleccionando valores aleatoriamente

Una variación de la reducción anterior sería no realizar ningún tipo de partición y seleccionar aleatoriamente entre los diferentes valores que componen la entrada original. Esta solución daría lugar a soluciones reducidas más dispares que podrían llegar a darnos valores aleatorios. Si realizamos varias reducciones aleatorias podríamos obtener medidas más interesantes.

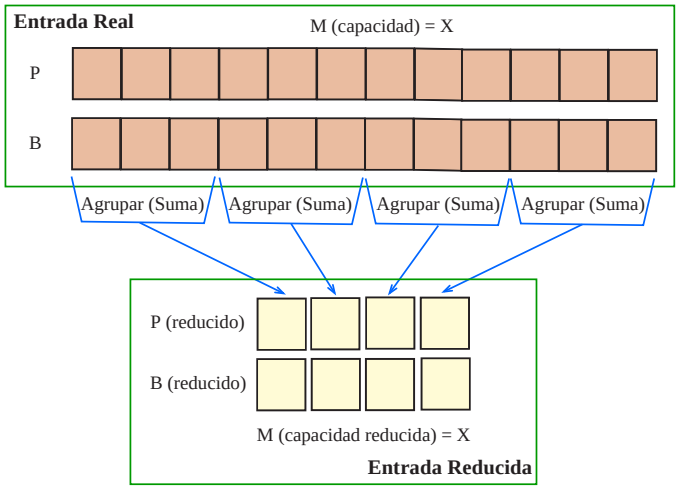


Figura 3.6. Reducir el tamaño del problema agrupando valores contiguos.

3.6 Estimación del tiempo de cómputo de un nodo (*TCN*)

Al igual que hemos propuesto diferentes métricas para obtener el parámetro k vamos a hacer lo mismo con el *TCN*. Cuando definimos el modelo ya propusimos una forma de calcular su valor que consistía en asignar el valor de una iteración del bucle central. En este apartado vamos a proponer otras formas de calcularlo.

3.6.1 Asignación general

Independientemente de estar trabajando con un esquema donde el tiempo de expansión de un nodo es homogéneo o heterogéneo, vamos a proponer una primera aproximación que se comporte de forma razonable para cualquiera de los casos. Lo que haremos será considerar como tiempo de cómputo de un nodo una media entre lo que ha tardado en ejecutarse la rutina y el número de nodos generados:

$$TCN = \frac{\text{Tiempo de Cómputo Total}}{\text{Número de Nodos Generados}} \tag{3.7}$$

Esta medida incluye mucho tiempo adicional que no es el tiempo que se dedica a generar un nodo. Como este tiempo adicional sería el mismo para todos los esquemas en el que lo apliquemos, la sobrecarga será un valor similar en todos ellos. Para obtener resultados más fiables en la instalación volveremos a tomar como TCN de un esquema una media de todas las ejecuciones:

$$TCN_{Medio} = \frac{\sum_{i=1}^{NumTest} TCN_i}{NumTest} \tag{3.8}$$

Donde TCN_i es el tiempo de cómputo de un nodo para la entrada número i y TCN_{Medio} es el tiempo medio de cómputo de un nodo en las distintas ejecuciones.

3.6.2 Equivalente al tiempo de cómputo de las funciones del esquema

La estimación anterior es demasiado general e incluye tiempo adicional que sobrecargará los valores obtenidos con tiempo que no es exclusivo de la computación de un nodo. Generalmente la expansión de un nodo es equivalente al total de operaciones que se hacen por cada una de las iteraciones del bucle central del esquema de *backtracking* (Código 2.1). El *TCN* sería realmente el tiempo que tardamos en expandir un nodo:

$$\begin{array}{rcll} TCN & = & TG & \text{(Tiempo Generar)} \\ & + & TS & \text{(Tiempo Solución)} \\ & + & TC & \text{(Tiempo Criterio)} \\ & + & TMH & \text{(Tiempo MasHermanos)} \\ & + & TR & \text{(Tiempo Retroceder)} \end{array} \tag{3.9}$$

Para calcular el tiempo de cómputo de esta manera, se añadiría a nuestro esquema de *backtracking* una función capaz de calcular el tiempo que se dedica a cada una de estas funciones, obteniéndose así el valor con mayor precisión. Estaríamos eliminando la sobrecarga de la que se hablaba en el apartado anterior.

En nuestra aproximación consideramos que el tiempo de expansión es igual para todos los nodos del árbol. Sin embargo, este tiempo podría variar en futuros esquemas dependiendo del nivel en el que se encuentre cada nodo.

3.6.3 Asignación dependiente del nivel

Los problemas que nos plantea la solución anterior podríamos solucionarlos realizando el estudio del tiempo de expansión de un nodo para cada uno de los niveles. Si extendemos nuestro esquema de manera que en tiempo de instalación se calcule el tiempo que tardan las funciones anteriores para cada uno de los niveles, tendremos cuanto tardamos en expandir un nodo para cada uno de ellos.

Capítulo 4

Resultados experimentales secuenciales. Mochila 0/1

Para ver la validez de la estrategia propuesta vamos a programar el ejemplo académico de la mochila 0/1 [7]. Vamos a programar un total de cinco esquemas diferentes y veremos los resultados de nuestra rutina con cada uno de ellos. Los esquemas se diferencian en la forma de generar los nodos y de realizar las podas. Todos los esquemas realizan algún tipo de poda:

- **Esquema 1**: eliminando nodos cuyo peso acumulado exceda del de la mochila.
- **Esquema 2**: eliminando nodos cuyo peso exceda del de la mochila o que sumando el beneficio de los objetos que quedan no se pueda alcanzar el mejor beneficio actual.
- **Esquema 3**: intentando primero incluir objetos en la mochila, eliminando nodos cuyo peso exceda del de la mochila o que sumando el beneficio de los objetos que quedan no se pueda alcanzar el mejor beneficio actual.
- **Esquema 4**: ordenando los objetos de mayor a menor beneficio/peso. Intentando primero incluir objetos en la mochila. Eliminando nodos cuyo peso exceda del de la mochila o que sumando el beneficio de los objetos que quedan no se pueda mejorar el mejor beneficio actual.
- **Esquema 5**: ordenando los objetos de mayor a menor beneficio/peso, intentando primero incluir objetos en la mochila, eliminando nodos cuyo peso exceda del de la mochila o que sumando al beneficio el beneficio obtenido con un avance rápido no se pueda mejorar el beneficio actual.

Además de los cinco esquemas que ha programado el supuesto usuario de las librerías, hemos programado la rutina encargada de generar las entradas aleatorias. Como este es un problema general donde las entradas a resolver pueden variar mucho, hemos parametrizado la rutina de generación para poder variar las pruebas que realizamos. Entre los distintos parámetros que podemos configurar tenemos:

- Elegiremos un valor n para generar el tamaño de la mochila.

- Definiremos un valor mínimo y máximo para los beneficios (B_{min} y B_{max}), y unos valores mínimo y máximo de variación de los beneficios (B_{vmin} y B_{vmax}).
- Definiremos un valor mínimo y máximo para los pesos (P_{min} y P_{max}), y unos valores mínimo y máximo de variación de los pesos (P_{vmin} y P_{vmax}).

Generaremos aleatoriamente el valor de la capacidad de mochila que oscilará entre n y $10n$. Posteriormente generaremos un valor de peso central (P_c) y un valor de variación del peso (P_v). Haremos lo mismo con los beneficios generando un beneficio central (B_c) y una variación de beneficio (B_v). A parir de estos valores tendremos definido un intervalo de generación de pesos $[P_c - P_v, P_c + P_v]$ y un intervalo de generación de beneficios $[B_c - B_v, B_c + B_v]$. Podemos ver en la Tabla 4.1 un ejemplo donde los valores configurados los tenemos en la tercera columna: $n = 10$, $P_{min} = 5$, $P_{max} = 25$, $B_{min} = 5$, $B_{max} = 25$, $P_{vmin} = 1$, $P_{vmax} = 10$, $B_{vmin} = 1$ y $B_{vmax} = 10$.

Tamaño Mochila (M)	n – $10n$	10 – 100	$M = 50$
P_c	P_{min} – P_{max}	5 – 25	$P_c = 12$
B_c	B_{min} – B_{max}	5 – 25	$B_c = 20$
P_v	P_{vmin} – P_{vmax}	1 – 10	$P_v = 3$
B_v	B_{vmin} – B_{vmax}	1 – 10	$B_v = 7$
			$P : [9, 15]$
			$B : [13, 27]$

Tabla 4.1. Ejemplo de configuración del generador de entradas para el problema de la mochila 0/1

En la última columna tenemos un ejemplo de posibles valores que pueden obtenerse en la generación de una entrada aleatoria para el problema de la mochila. El valor aleatorio de peso central es 12 y el de beneficio central es 20. El valor aleatorio de variación de peso es 3 y el de variación de beneficio es 7. Aplicando lo explicado en el párrafo anterior generaremos valores para los beneficios aleatoriamente en el intervalo $[9, 15]$ y para los pesos aleatoriamente en el intervalo $[13, 27]$.

4.1 Evaluación del método de estimación de *k*

Antes de comparar como se comporta nuestra rutina de selección entre los diferentes esquemas, vamos a evaluar como se comportan los diferentes métodos propuestos para estimar el valor de k. En este experimento hemos decidido extraer y estimar valores de k para el esquema número cinco.

Lo que nos interesa sería predecir tiempos de ejecución para tamaños de problema que no hemos podido resolver en tiempo de instalación. En este tipo de recorridos donde el orden de complejidad es exponencial encontrar soluciones para ciertos tamaños de entradas puede tardar varias horas, días... Debido a las limitaciones de tiempo de realización del

proyecto no podemos permitirnos estudiar estos tamaños. Por este motivo los tamaños de problemas configurados en la instalación no serán excesivamente grandes.

Lo que haremos será configurar nuestro generador de entradas con los parámetros de la Tabla 4.1. Configuraremos la rutina de instalación con: Tiempo máximo de instalación = 5 horas, Tamaño mínimo del problema = 5, Tamaño máximo del problema = 25, Incremento = 5, Número de tests = 50. Generaremos por tanto un total de 50 entradas aleatorias de cada tamaño. Los tamaños oscilarán desde cinco hasta 20 incrementando de cinco en cinco. Esta información será la que utiliza el modelo para inferir información del tiempo de instalación.

Lo que se hará en este experimento será generar entradas aleatorias de tamaños 30, 35 y 40. Estos tamaños de entrada son mayores que los que hemos utilizado para extraer información en la instalación. Para cada uno de estos tamaños generaremos un total de tres entradas para obtener información más realista.

En la Tabla 4.2 se muestran los valores obtenidos en las pruebas. Se generan tres entradas aleatorias de tamaños 30, 35 y 40. Estos tamaños de entrada son mayores que los que hemos utilizado para extraer información en la instalación. Vamos a comparar los valores de k utilizando el esquema número 5. Este esquema es el más realista al ser el que realiza podas más extremas, algo que suele ser lo más común en los problemas reales. Para cada entrada vamos a obtener el valor de k real, el valor de k estimado por el método de la media, el valor de k estimado por la función de una recta y añadiendo información de la ejecución (por el método de agrupación de la entrada) a esta última técnica. Como última columna añadiremos el valor de k utilizando únicamente información de la ejecución.

	k_{Real}	k_{Media}	k_{Recta}	$k_{Recta(0.5)+Ejecucion(0.5)}$	$k_{Ejecucion}$
30	0.999974	0.945759	0.987712	0.992170	0.996628
	0.999970	—	—	0.991163	0.994614
	0.999999	—	—	0.993497	0.999283
35	0.999999	0.945759	0.988091	0.993941	0.999790
	0.999999	—	—	0.992636	0.997181
	0.999999	—	—	0.993639	0.999187
40	0.999976	0.945759	0.98847	0.986691	0.984911
	0.999999	—	—	0.994147	0.999823
	0.997213	—	—	0.978668	0.968866

Tabla 4.2. Comparativa de los valores de k real y los estimados con los diferentes métodos.

Las conclusiones extraídas de la Tabla 4.2 son:

- Al tratarse de un esquema donde se produce un número de podas muy elevado, los valores del parámetro k resultan muy similares para los tamaños de problemas estudiados. La diferencia en los índices de poda reales comienza a apreciarse en la quinta cifra decimal en la mayoría de los casos.

- Al trabajar con problemas altamente dependientes de las entradas, no incluir información procedente de la ejecución lleva a resultados no del todo correctos. Estimando únicamente con información de la instalación (k_{Real} y k_{Media}), entradas con comportamiento completamente distinto se están aproximando con un mismo valor.
- Incluir información de la ejecución aproxima mejor el valor de k para cada una de las entradas.

En la Tabla 4.3 estudiamos la media del error de los cuatro métodos propuestos. La media de error la calcularemos como $|k_{Real} - k_{Estimada}|/k_{Real}$.

	k_{Media}	k_{Recta}	$k_{Recta(0.5)+Ejecucion(0.5)}$	$k_{Ejecucion}$
30	5.4 %	1.2 %	0.7 %	0.3 %
35	5.4 %	1.1 %	0.6 %	0.1 %
40	5.3 %	1 %	1.3 %	1.5 %

Tabla 4.3. Tabla comparativa de porcentaje de error con cada uno de los métodos.

Obtenemos una aproximación del error mejor cuando introducimos información de la ejecución. Sin embargo para la entrada de tamaño 40 se equiparan los métodos dos y tres. La razón es que una de las entradas de tamaño 40 es una entrada totalmente desfavorable, y en este caso las medidas que estamos utilizando no nos resultan válidas.

Las conclusiones que sacamos de este experimento es que el método se está comportando adecuadamente para estimar valores de k con los métodos que utilizan información de la ejecución. Existen diferencias poco significativas entre los dos. En problemas donde las entradas vayan a tener un comportamiento parecido será interesante completar la información de la ejecución con otra adicional de la instalación. En casos generales donde desconocemos las entradas, lo más apropiado sería utilizar un método que únicamente se base en la información de la ejecución; la información obtenida en la instalación podría perjudicar más que beneficiar.

Nos quedaremos con el tercer método estudiado para realizar los experimentos posteriores.

4.2 Evaluación de la selección entre diferentes esquemas

En este experimento vamos a ver como se comporta la rutina de selección entre diferentes esquemas. Además aprovecharemos los experimentos para evaluar como es capaz de predecir el tiempo de ejecución nuestro modelo.

Vamos a utilizar la estimación de k en tiempo de instalación (aproximando por una recta) añadiendo información en tiempo de ejecución (agrupando las entradas). Vamos a

estimar el valor de TCN a través de la media. Para generar la información de la instalación utilizaremos la misma configuración que en el apartado anterior.

En la Tabla 4.4 se obtienen los valores de k y TCN medios al ejecutar el experimento anterior con cada uno de los diferentes esquemas de la Mochila 0/1 que tenemos implementados. La información que obtenemos durante la instalación es una información realista. Los esquemas que hemos programado van haciendo podas de nodos cada vez más agresivas. El incremento de las podas se refleja en los valores de k cada vez mayores a medida que utilizamos un esquemas más agresivo en las podas. Además el TCN aumenta de manera muy brusca en el último caso, se debe a que estamos realizando un avance rápido para predecir el coste de los nodos.

	Esquema 1	**Esquema 2**	**Esquema 3**	**Esquema 4**	**Esquema 5**
k_{Medio}	0.66	0.78	0.80	0.81	0.93
TCN_{Medio}	2.43	1.44	1.24	1.89	78.93

Tabla 4.4. Valores medios de k y de TCN realizando la batería de pruebas con los cinco esquemas propuestos. (TCN en nano-segundos)

Si utilizamos la estrategia de decisión sobre distintos esquemas obtenemos los resultados de la Tabla 4.5. En esta tabla vemos que para cada uno de los esquemas tenemos la información del tiempo de ejecución estimado (T_{EX}) y del tiempo de ejecución real (T_{RX}). La variable Tam representa el tamaño de los problemas a resolver. Hemos resuelto tres problemas diferentes para cada uno de los tamaños de ejemplo. El tiempo de la tabla está medido en segundos. Lo que haría nuestra rutina sería inferir cual de los cinco esquemas es el más adecuado para resolver un problema utilizando como información el tiempo estimado. En negrita queda señalado cual es el esquema que nuestras rutinas decidirían y cual es el que realmente tarda menos tiempo en ejecutarse.

	Esquema 1		**Esquema 2**		**Esquema 3**		**Esquema 4**		**Esquema 5**	
Tam	T_{E1}	T_{R1}	T_{E2}	T_{R2}	T_{E3}	T_{R3}	T_{E4}	T_{R4}	T_{E5}	T_{R5}
30	–	27.03	0.164067	9.23e-05	0.127421	3.69e-05	0.156003	**2.97e-05**	**0.0194118**	0.0035
	–	1.5952	0.299966	0.3951	0.247095	0.4066	0.513488	0.1857	**0.0278335**	**0.0035**
	–	0.0189	0.179676	0.0104	0.157265	0.0106	0.247628	0.0089	**0.0217373**	**9.885e-5**
35	–	0.1427	5.07818	0.04262	4.45849	0.0358	5.31344	0.03671	**0.594301**	**0.0001**
	–	5.8902	8.29062	1.4499	7.97455	1.84043	6.96553	0.434884	**0.701118**	**0.004458**
	–	2.2591	8.43733	0.8799	7.41706	0.79329	9.92612	1.31357	**0.678421**	**0.004631**
40	–	557.367	233.665	188.13	210.833	169.829	472.009	157.484	**31.2062**	**2.59603**
	–	27927.2	100.542	0,0001607	96.9454	**0.0001317**	90.1806	0.001218	**18.0325**	0.00139
	–	3757.99	236.053	1512.37	212.31	1212.15	752.587	1287.94	**61.3567**	**264.16**

Tabla 4.5. Tabla de decisión de nuestra rutina entre diferentes esquemas de resolución de problemas para tamaños mayores de los resueltos en la instalación. Aproximación de k (Recta+Agrupación). Aproximación de TCN (Media). Tiempo expresado en segundos.

Las conclusiones que extraemos de la Tabla 4.5 respecto al criterio de decisión de la rutina son:

- El método propuesto elige correctamente cual es el esquema que resuelve en un tiempo de ejecución menor para cada una de las entradas en una amplia mayoría.
- Los casos donde falla se corresponden con entradas extremas.

Las conclusiones que extraemos de la Tabla 4.5 respecto a la estimación del tiempo de ejecución son:

- La aproximación de los tiempo de ejecución va alejándose cada vez más a medida que aumentamos el tamaño de los problemas a resolver. Esto se podría deber a dos motivos:
  - Aunque el porcentaje de error en el coeficiente k se mantenga constante para los diferentes tamaños de problema, a medida que aumenta el número de nodos generados es mayor (crece exponencialmente) y este error se manifiesta en mayor medida en el tiempo de ejecución estimado.
  - En el caso de estas entradas, la estimación de k para problemas de tamaño 40 aumentaba en error medio por tratarse de tres situaciones de entrada completamente distintas.
- A pesar de no aproximar bien el tiempo, el orden de ejecución de los problemas ordenando por el tiempo estimado (T_{EX}) es el mismo que si ordenamos por el tiempo real (T_{RX}).

El ejemplo anterior está comparando entre los cinco esquemas que hemos programado. Estos esquemas son excesivamente distintos entre sí. Es razonable que el esquema que nuestro modelo elige como el más adecuado sea siempre el esquema cinco al ser el más radical en las podas. Para estudiar con más detalle el comportamiento de nuestra técnica vamos a repetir el experimento en la Tabla 4.6, pero únicamente se contemplan los esquemas dos, tres y cuatro que son realmente muy similares entre sí.

	Esquema 2		**Esquema 3**		**Esquema 4**	
Tam	T_{E2}	T_{R2}	T_{E3}	T_{R3}	T_{E4}	T_{R4}
30	0.164067	9.23e-05	**0.127421**	3.69e-05	0.156003	**2.97e-05**
	0.299966	0.3951	**0.247095**	0.4066	0.513488	**0.1857**
	0.179676	0.0104	**0.157265**	0.0106	0.247628	**0.0089**
35	5.07818	0.04262	**4.45849**	**0.0358**	5.31344	0.03671
	8.29062	1.4499	7.97455	1.84043	**6.96553**	**0.434884**
	8.43733	0.8799	**7.41706**	**0.79329**	9.92612	1.31357
40	233.665	188.13	**210.833**	169.829	472.009	**157.484**
	100.542	0,0001607	96.9454	**0.0001317**	**90.1806**	0.001218
	236.053	1512.37	**212.31**	**1212.15**	752.587	1287.94

Tabla 4.6. Tabla de decisión de nuestra entre esquemas muy similares entre ellos. Aproximación de k (Recta+Agrupación). Aproximación de *TCN* (Media). Tiempo expresado en segundos.

En este caso vemos que se comporta bien, eligiendo siempre entre los dos esquemas que teóricamente son más eficientes. Sin embargo no es capaz de acertar siempre eligiendo

el que da un menor valor de tiempo de ejecución. Estudiemos cual es el coeficiente de error medio al hacer caso al modelo y en caso de elegir siempre cualquiera de los demás esquemas (Tabla 4.7).

$\frac{TR_{Modelo}}{TR_{Optimo}}$	$\frac{TR_{Esquema2}}{TR_{Optimo}}$	$\frac{TR_{Esquema3}}{TR_{Optimo}}$	$\frac{TR_{Esquema4}}{TR_{Optimo}}$
1.21	1.80	1.61	1.09

Tabla 4.7. Tabla de valores medios de coeficientes entre los tiempos reales y los que obtenemos con el modelo. Comparativa con los tiempo entre los diferentes esquemas.

Como se observa en la Tabla 4.7, la elección del esquema dos supondría un error medio de 1.80 respecto al error medio de 1.21 que obtenemos utilizando el modelo teórico; sin embargo si elegimos el esquema cuatro, el error medio obtenido sería de 1.09 y mejoraría respecto a utilizar el modelo. En cualquier caso, merece la pena utilizar el modelo porque el error cometido ante una mala elección (esquema dos o esquema tres), es mayor que la ganancia que obtendríamos si sabemos elegir el óptimo (esquema cuatro). Hay que tener en cuenta, que en la práctica no sería fácil detectar que el esquema cuatro nos lleva a mejores soluciones que los otros dos.

4.3 Influencia del parámetro *k* en la estimación del tiempo de ejecución

Desde un primer momento se ha identificado el parámetro k como un punto decisivo para estimar el tiempo de ejecución de nuestras técnicas. Vamos a realizar un experimento que nos permita ver como influye en la práctica su estimación.

El experimento consiste en ver la estimación del tiempo de ejecución que conseguimos, para el esquema número cinco, cuando utilizamos los diferentes métodos de estimar k. En la Tabla 4.8 se muestran los resultados obtenidos. TE representa el tiempo de ejecución estimado por el modelo para las diferentes formas de estimar k. Utilizaremos las mismas entradas y las mismas configuraciones que en los experimentos anteriores.

De los resultados de la Tabla 4.8 extraemos las siguientes conclusiones:

- Debido a la alta dependencia de las entradas estimar el valor de k por medio de un valor constante nos lleva a resultados muy alejados de la realidad.
- La estimación por medio de una función nos da resultados que se acercan más al tiempo de ejecución real. Sin embargo los resultados aproximados siguen siendo constantes para un mismo tamaño de la entrada. No representa realmente la dependencia de las entradas que se da en la ejecución.

Tam	TE k_{Media}	TE k_{Recta}	TE $k_{Recta+Ejecucion}$	Tiempo Real
30	0.1654	0.037473	0.0194118	0.0035
30	—	—	0.0278335	0.0035
30	—	—	0.0217373	0.0000098
35	5.29	1.1621	0.594301	0.0001
35	—	—	0.701118	0.004458
35	—	—	0.678421	0.004631
40	169.382	36.0059	31.2062	2.59603
40	—	—	18.0325	0.00139
40	—	—	61.3567	264.16

Tabla 4.8. Tabla comparativa de los diferentes tiempos de ejecución estimados por el modelo utilizando diferentes formas de estimar el *k*. (Tiempo en segundos)

- Al añadir información de la ejecución los valores se aproximan a cada una de las entradas. Se reduce el error medio.
- Los coeficientes de error de tiempo de ejecución son elevados para cualquiera de los casos. No se aproxima el tiempo de forma adecuada.
- Para la técnica de *backtracking* predecir el tiempo exacto de ejecución es muy complicado. La alta dependencia de las entradas junto al orden de complejidad elevados impiden conseguir aproximaciones válidas para el método en general.
- Para aproximar mejor los tiempos de ejecución deberíamos restringir el ámbito de las entradas que vamos a resolver en nuestros problemas, de forma que podamos encuadrar mejor las ejecuciones.

Capítulo 5

Paralelización de los recorridos de *backtracking*

Hasta ahora hemos visto técnicas de autooptimización para recorridos secuenciales de *bactracking*. En este capítulo se pretende estudiar diferentes formas de paralelizar este tipo de recorridos. Se analizarán varios esquemas paralelos y se identificarán los nuevos parámetros configurables que aparecen.

5.1 Introducción a los recorridos paralelos

Con la aparición generalizada de máquinas paralelas abordar estos problemas con soluciones paralelas suele ser lo más razonable. Se persigue utilizar la programación paralela para intentar reducir el tiempo empleado en resolver los problemas. La paralelización de las rutinas nos permitirá reducir el tiempo de ejecución o aumentar el nivel de detalle de las búsquedas en el mismo tiempo.

En este capítulo vamos a proponer diferentes esquemas de paralelización de los algoritmos de *backtracking*. La idea general será distribuir el espacio de búsqueda entre los distintos procesadores de los que se disponen, de forma que cada uno busque la solución del problema en un subespacio de soluciones distinto. De esta manera se explorarán varias ramas del árbol de soluciones al mismo tiempo en distintos procesadores. En problemas de búsqueda de solución aumentan las posibilidades de encontrar la solución en menos tiempo. En la Figura 5.1 tenemos un ejemplo donde disponemos de tres procesadores ($P0$, $P1$ y $P2$) y se asigna a cada uno de ellos un subespacio del árbol de soluciones.

Al estar trabajando con problemas donde se realizan podas durante la ejecución, los subespacios de búsqueda no tienen por que ser iguales. En la Figura 5.1 podemos observar que los subespacios asignados a cada uno de los procesadores contienen un número de nodos diferente. El tamaño de estos espacios no se podrá conocer hasta que no resolvamos el problema para una entrada determinada. Por este motivo realizar la distribución del espacio de búsqueda entre los distintos procesadores es un factor muy importante a considerar a la hora de diseñar las soluciones paralelas.

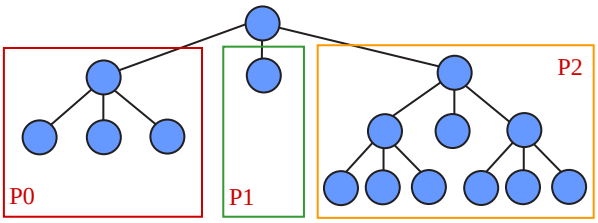


Figura 5.1. Distribución paralela del espacio de búsqueda.

Otro aspecto interesante son las comunicaciones entre todos los procesadores para poder realizar el trabajo de forma independiente pero a la vez conjunta. Existen una gran variedad de esquemas paralelos que adoptan diferentes enfoques. A lo largo del capítulo vamos a ver dos esquemas paralelos diferentes: un esquema maestro-esclavo con asignación estática de tareas y un esquema maestro esclavo con asignación dinámica de tareas. Nos centraremos más en el primer esquema, y se realizará el estudio práctico solo para él.

También sería importante identificar para cada esquema paralelo un esquema algorítmico. Realmente cuando se aplica el paralelismo estamos resolviendo problemas secuenciales más pequeños de manera simultánea. Sería interesante identificar estas funciones secuenciales que son utilizadas por el esquema paralelo para construir la solución paralela. Se podría construir un esquema algorítmico que ofrezca a los programadores la posibilidad de completar las funciones secuenciales que resuelven sus problemas. Internamente nuestras rutinas utilizarían las funciones del esquema secuencial para elaborar la solución paralela identificada en cada uno de los esquemas.

5.2 Esquema maestro esclavo (M/E)

Una posible configuración sería considerar un proceso maestro que coordine a un conjunto de procesos esclavos [3]. A estos esquemas los denominaremos, desde este punto y a lo largo del proyecto, esquemas *M/E*. En la Figura 5.2 se refleja como el proceso maestro es el encargado de asignar el trabajo a cada uno de los procesos esclavos. Los procesos esclavos enviarán información al maestro cuando lo consideren necesario. Las comunicaciones entre dos procesos esclavos se realizarán a través del proceso maestro.

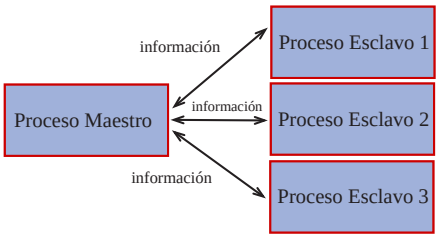


Figura 5.2. Representación de un esquema maestro esclavo con comunicaciones.

Se ha hablado en general de la distribución del espacio de búsqueda entre los distintos procesadores pero, ¿como decide el proceso maestro los diferentes espacios que tiene que asignar? En el caso de la técnica de *backtracking* podríamos realizar el recorrido en varias etapas. Se recorre primero el árbol de búsqueda hasta un determinado nivel (l). Una vez alcanzado este nivel, se guarda la información de los nodos encontrados a nivel l. Posteriormente se van haciendo nuevos recorridos desde los nodos almacenados hasta el último nivel. Cada uno de estos recorridos se puede considerar un *backtracking* independiente. En la Figura 5.1 se ha realizado un *bactracking* inicial hasta el nivel l=*1*.

De este modo tenemos una división del espacio de búsqueda tal y como se desea. En un esquema paralelo maestro esclavo se considera que el primer *backtracking* hasta nivel l lo realiza el proceso maestro. El número de nodos que se encuentran a nivel l corresponderá con el total de trabajos que puede asignar a los procesos esclavos, también conocidos como *backtracking* secundarios. Una vez finalizado el *backtracking* inicial se considerará al proceso maestro como un esclavo más. Son estos últimos *backtrackings* secundarios los que realmente se realizan en paralelo, no podrán empezar hasta que el proceso maestro no haya terminado el primer recorrido. Existen dos alternativas en la asignación de los procesos por parte del proceso maestro: asignación estática y asignación dinámica.

5.2.1 Asignación estática de tareas (*EMEAE*)

Podemos asignar de manera estática los trabajos entre los distintos procesadores. A cada procesador se le asignan un número de estos trabajos. Cada procesador comienza a buscar una solución en el espacio de búsqueda que le corresponda. No se necesitan comunicaciones entre ellos. Cada procesador únicamente puede dedicarse a resolver aquellos problemas que le han sido asignados en un principio. A partir de este punto nos vamos a referir a estos esquemas como *EMEAE*.

Se pueden adoptar diferentes criterios para realizar la asignación de tareas:

- **Distribución por bloques**: podemos dividir el total de tareas a asignar entre el número de procesadores de que disponemos y asignar a cada uno el número de tareas contiguas que le correspondan. En ocasiones las podas se encuentran localizadas en un cierto área del espacio de estados. Si realizamos la asignación por bloques puede que a algunos procesadores se le asignen trabajos donde el porcentaje de podas sea muy superior al de otros puntos del espacio de búsqueda. Esto desequilibraría la distribución de los trabajos y no se estaría aprovechando completamente el paralelismo.

- **Distribución cíclica**: se asignarán las tareas cíclicamente entre todos los procesadores de los que disponemos. En este caso no se produce el problema que hemos explicado para la distribución por bloques. La asignación de trabajos es más equitativa respecto a la posición en el espacio de estados donde se encuentre.

- **Distribución aleatoria**: se asignarán los subtrabajos de forma aleatoria entre todos los procesadores. Esta asignación se podría hacer de varias formas, en nuestro caso optamos por proponer una solución donde se fuerza asignar a cada procesador el mismo número de subtrabajos. En caso contrario, las diferencias en los tiempos de ejecución podrían ser muy diferentes entre varias asignaciones aleatorias. El resultado obtenido con este tipo de asignaciones dependerá de cada una de las ejecuciones. Este tipo de distribución se utilizará en las pruebas como caso extremo para comprobar si la asignación realmente influye en el tiempo de ejecución total.

En la Figura 5.3 disponemos de tres procesadores para la asignación de los subtrabajos. Los colores representan las asignaciones a los distintos procesadores: el color rojo para los subespacios asignados al procesador cero, el color verde para los asignados al procesador uno y el azul para los asignados al procesados dos. En la Figura 5.3 a) se realiza una asignación estática siguiendo una distribución por bloques contiguos. En la figura 5.3 b) se realiza una asignación estática siguiendo una distribución cíclica.

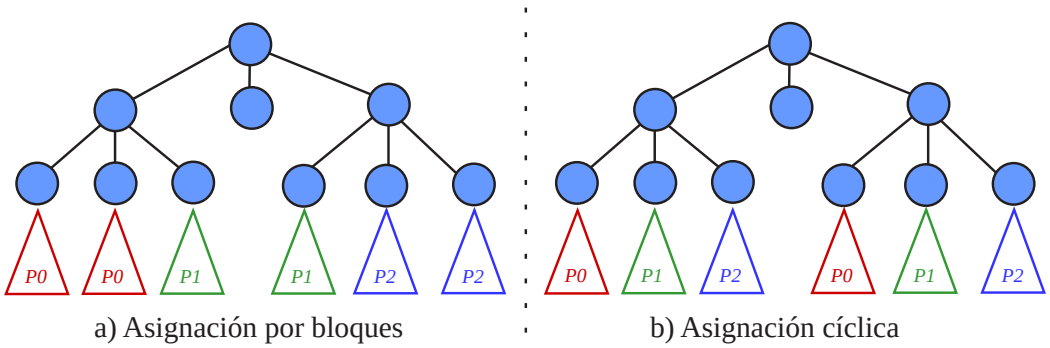


Figura 5.3. Asignación estática de tareas entre tres procesadores (*P0*, *P1* y *P2*).

El problema de esta asignación estática es que no se conoce a priori si se divide el espacio de búsqueda equitativamente entre los distintos procesadores. A un procesador puede que se le asigne poco trabajo y esté la mayor parte del tiempo inactivo. Esta situación se puede dar con cualquiera de los tres tipos de asignaciones. En la primera parte de este proyecto hemos estudiado mecanismos para predecir el comportamiento de *bactracking* secuenciales. Se pueden utilizar estas técnicas para estimar el tiempo de ejecución de cada uno de los subtrabajos. Una vez estimados podremos realizar una asignación estática que distribuya los subtrabajos homogéneamente utilizando la información obtenida.

A continuación vamos a ver el Código 5.1 correspondiente con un esquema paralelo maestro esclavo con asignación estática de tareas (*EMEAE*). En primer lugar el proceso maestro invoca al *bactracking* primario hasta nivel l. Se pasará como parámetro la lista que contendrá todos los subtrabajos generados en este primer recorrido (*listaDeSubtrabajos*). A continuación calculamos el número de subtrabajos total (TS) y realizamos una asignación (A) sobre los procesadores de los que disponemos (p). Por último cada uno de los procesadores (incluido el maestro) recorrerá el array de asignaciones y realizará los trabajos que le han sido asignados. Estos trabajos los consulta de la *listaDeSubtrabajos* que se rellenó durante el *backtracking* inicial.

```
1   ... ... ... ... ...
2
3   esquemaBacktracking( ---- , l, ---- , listaDeSubtrabajos);
4
5   int TS = length(listaDeSubtrabajos);
6
7   int * A = (int *) malloc (sizeof(int)*TS);
8   for ( int i = 0; i < TS; i++ ) A[i] = i mod p;
9
10  Cada Procesador idProc, para 0 < idProc < p {
11     for ( int j = 0 ; j < TS; j++ ) {
12        if ( A[j] == idProc ) {
13           struct nodoBacktraking * nodo;
14           nodo = getNodo(listaDeSubtrabajos, j);
15
16           esquemaBactracking( nodo , n-l, ---- , NULL);
17
18           free(nodo);
19        }
20     }
21  }
22
23  ... ... ... ... ...
```

Código 5.1. Esquema paralelo maestro-esclavo con asignación estática de tareas y sin intercambio de información. Se hace hasta un nivel l el *backtracking* primario de un total de n niveles.

5.2.2 Asignación dinámica de tareas (*EMEAD*)

En la asignación dinámica de tareas cada procesador sigue trabajando en una parte del espacio de búsqueda. El proceso maestro gestiona los subtrabajos pendientes y se encarga de distribuirlos a los esclavos bajo demanda, es decir, cuando un procesador esclavo termina todos los subtrabajos que le fueron asignados, solicita más trabajo al maestro. No existe una distribución de tareas fija. El proceso maestro actuará como coordinador de los demás procesos que están resolviendo problemas. En este caso el proceso maestro no actúa como lo esclavos, y no se dedica a resolver problemas sino a coordinarlos. Los procesos esclavos se dedican a resolver problemas y comunicar las soluciones encontradas al proceso maestro. A partir de este punto nos vamos a referir a estos esquemas como *EMEAD*.

Se propone una asignación dinámica por medio de una bolsa de tareas. El proceso maestro generará el *backtracking* inicial hasta un nivel l y construirá una bolsa de tareas con todos los nodos que estén en ese nivel. Posteriormente se irán asignando a los procesadores esclavos las tareas que tenemos en dicha bolsa.

En la Figura 5.4 el proceso maestro realiza un *backtracking* inicial hasta el nivel tres generando los nodos que se encuentran numerados (t=*0*). Los nodos numerados son los que constituirían la bolsa de tareas. Los subárboles con la pirámide pequeña tardan una unidad de tiempo en resolverse. Los subárboles con la pirámide grande tardan dos uni-

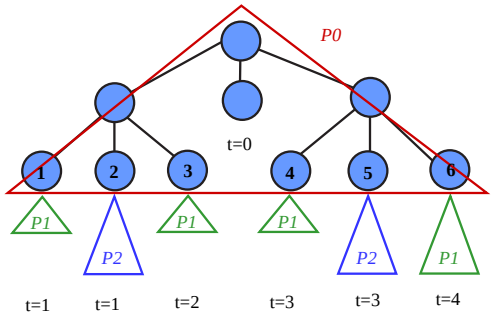


Figura 5.4. Asignación dinámica de tareas entre tres procesadores disponibles (*P0*, *P1* y *P2*).

dades de tiempo en resolverse. En el instante (*t=1*) el proceso maestro asigna las dos primeras tareas a los dos procesadores que tiene disponibles. En el instante *t=2* el procesador uno termina la tarea que tenía asignada y solicita más trabajo al procesador maestro. El procesador maestro asigna el siguiente trabajo que tiene disponible, el trabajo tres. En el instante *t=3* los dos procesadores esclavos terminan y solicitan más trabajo al maestro. El proceso maestro va asignando las tareas a los procesadores que vayan terminando hasta resolverse todos los problemas.

```
1  ... ... ... ... ...
2  esquemaBacktracking( ––– , l , ––– , bolsaDeSubtrabajos);
3
4  int TS = length(bolsaDeSubtrabajos);
5
6  Cada Procesador idProc, para 0 < idProc < p {
7     if ( idProc = 0 ) {
8        while (  TS > 0 ) {
9           recibeSolicitudDeTrabajo ( procDestino );
10          struct nodoBacktracking * nodo;
11          nodo = getNodo(bolsaDeSubtrabajos, TS);
12          enviaTrabajoARealizar ( procDestino, nodo, TS );
13          free(nodo); TS––;
14       }
15       for ( TS = 0; TS < p; TS++ ) {
16          enviaTrabajoARealizar ( TS, NULL, 0 );
17       }
18    }
19    else {
20      while ( trabajosPendientes != 0 ) {
21         enviaSolicitudDeTrabajo ( 0, idProc );
22         struct nodoBacktracking * nodo;
23         nodo = recibeTrabajoARealizar(procOrigen=0,trabajosPendientes);
24         if ( trabajosPendientes != 0 ) {
25
26            esquemaBacktracking( nodo , n–l , ––– );
27
28         }
29         fee(nodo);
30       }
31    }
32 }
33 ... ... ... ... ...
```

Código 5.2. Esquema paralelo maestro-esclavo con asignación dinámica de tareas y sin intercambio de información. Se hace hasta un nivel *l* el *backtracking* primario de un total de *n* niveles.

En el Código 5.2 tenemos el esquema de como sería la asignación dinámica de tareas. Se puede observar que en este caso el comportamiento del proceso maestro (*idProc=0*) es completamente diferente al de los procesos esclavos (*idProc!=0*). Una vez realizado el *backtracking* hasta nivel l el proceso maestro se limita a recibir solicitudes de trabajo por parte de los procesos esclavos y enviar los trabajos que deben realizar. El proceso maestro estará esperando solicitudes de trabajo mientras queden subtrabajos por asignar en la bolsa de tareas ($TS > 0$). Por su parte los procesos esclavos enviarán solicitudes de trabajos al proceso maestro indicando quienes son para que este les pueda responder. Mientras los procesos esclavos no reciban del proceso maestro que no quedan más trabajos por asignar (*trabajosPendientes != 0)*, solicitarán subtrabajos para resolverlos.

5.2.3 Esquemas con intercambio de información entre los procesadores (*+I*)

Además de solicitar trabajo al proceso maestro, los procesos esclavos pueden requerir comunicaciones entre ellos. En los recorridos por *backtracking* se realizan podas para reducir los espacios de búsqueda. Un tipo de podas consistía en podar comparando con el mejor valor encontrado hasta el momento. Cada cierto intervalo de tiempo o de nodos generados (e), los procesos esclavos pueden necesitar intercambiar información que sea utilizada para resolver los problemas localmente. El intervalo de tiempo o los nodos generados (e) será fijado por el programador. Es interesante que conociesen en todo momento cual ha sido el mejor valor encontrado por cualquiera de ellos, para aumentar su porcentaje de poda.

A partir de este punto si los esquemas anteriores realizan intercambio de información nos vamos a referir a ellos utilizando su abreviatura y añadiendo *+I*, por ejemplo, un esquema maestro esclavo con asignación estática de tareas e intercambio de información lo denominaremos *EMEAE+I*.

Como ya se comentó en la introducción existen dos paradigmas de programación paralela: memoria compartida y memoria distribuida. Si trabajamos con este tipo de esquemas en memoria compartida no supone mucho problema que todos los procesos conozcan el mejor valor encontrado. Simplemente dejando este valor en una variable compartida conocida por todos, los cambios serán conocidos automáticamente por todos los procesos.

Cuando trabajamos en memoria distribuida cada uno de los procesos trabaja en una máquina con memoria local. No existen partes de memoria que sean conocidas por todos los procesos esclavos. En estos casos el proceso maestro juega un papel importante. Será el encargado de recibir la información que un proceso esclavo quiera compartir con los demás, y distribuirla por medio de un mensaje.

5.3 Parámetros que intervienen en el tiempo de ejecución

Una vez explicados dos posibles esquemas paralelos aplicables a las técnicas de *backtracking* vamos a destacar cuales son los parámetros configurables que intervienen en el tiempo de ejecución.

5.3.1 Esquema maestro esclavo

Para aprovechar la exploración paralela de los diferentes subárboles de búsqueda nos interesa reducir al máximo el *backtracking* inicial que realiza el proceso maestro para generar los distintos subproblemas. Si el **nivel del *backtracking* inicial** (l) es demasiado pequeño generaremos pocos problemas para asignar. Como la carga de trabajo asociada a los subproblemas estará desbalanceada puede ser que al final tampoco estemos aprovechando correctamente los beneficios del paralelismo. Sin embargo, si el nivel l hasta el que se realiza el *backtracking* inicial es demasiado grande, el tiempo que tardamos en resolverlo sería elevado y podría llegar a no compensarnos, ya que no estaríamos aprovechando al máximo el paralelismo.

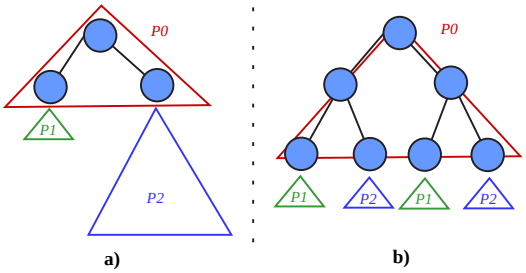


Figura 5.5. Árboles de búsqueda de un mismo problema generando el *backtracking* inicial a diferentes alturas.

En la Figura 5.5 se muestra la distribución del espacio de búsqueda de un mismo problema generando hasta diferentes niveles el *backtracking* inicial. En la Figura 5.5 a) el *backtracking* inicial no consume apenas tiempo pero el desbalanceo de la carga de los subproblemas es demasiado grande y penalizará el tiempo de ejecución final. En la Figura 5.5 b) se emplea más tiempo en el primer recorrido, pero el balanceo de los subproblemas mejora, aprovechándose mejor el paralelismo.

Debemos de intentar elegir el valor del parámetro l para el que consigamos obtener el menor tiempo de ejecución total. El valor óptimo para este parámetro dependerá de cada uno de los problemas, de la entrada a resolver y de la máquina donde se esté ejecutando la rutina. En ocasiones cesar antes en este *backtracking* inicial nos llevaría a que la información intercambiada por los procesos esclavos redujese los espacios de búsqueda reduciendo así el tiempo de ejecución.

Otro parámetro decisivo a elegir es el **número de procesadores** a utilizar (p). No siempre elegir mayor valor para p nos llevará a menores tiempo de ejecución. Deberemos seleccionar el número de procesadores que nos lleve a minimizar el tiempo de ejecución.

Un punto decisivo en la ejecución era que los procesos esclavos intercambiasen y actualizasen información que les permitiese disminuir al máximo los espacios de búsqueda que recorren. Cuando se trabaja en memoria distribuida la comunicación entre los procesos esclavos requiere de paso de mensajes entre ellos. Estos mensajes deberán enviarse a través de la red incrementando el tiempo de ejecución en lo que tarden en realizarse las comunicaciones.

Lo más deseable sería que los procesos esclavos intercambien la información necesaria cada vez que se consigue mejorar la información local que tienen. Realizar tantas comunicaciones puede ser demasiado costoso en tiempo (sobre todo si la red de comunicaciones es lenta) y no compensar con el tiempo de ejecución total de realizar los intercambios con menos frecuencia.

Se puede definir el parámetro de **porcentaje de intercambio de información** entre los procesos esclavos (e). Este parámetro dependerá en gran medida de las propiedades de la máquina donde se esté ejecutando la rutina. Será dependiente también del problema y las entradas que se estén resolviendo.

5.3.2 Asignación estática

En este caso un posible parámetro de configuración es elegir que **tipo de asignación estática** a utilizar. Al no conocer como se balancea la carga en los subproblemas que debemos asignar sería complicado elegir un tipo de asignación que funcione bien para todos los casos. Incluso para dos entradas distintas para un mismo problema una misma asignación nos puede dar resultados completamente diferentes.

Si se consigue conocer a priori como se balancea la carga de trabajo en los nuevos subproblemas se podrían asignar de manera que minimice el tiempo de ejecución total. Para ello podríamos utilizar las técnicas que se han estudiado para predecir el comportamiento de los problemas de *backtracking* secuenciales.

5.3.3 Asignación dinámica

Dependiendo del paradigma de programación que utilicemos, el envío de los trabajos a los procesos esclavos puede acarrearnos tiempos de comunicación. El proceso maestro no siempre tendría que distribuir los trabajos de uno en uno. Para no consumir excesivo tiempo en las comunicaciones podría enviar diferente número de subtrabajos a cada uno de ellos. El **número de subtrabajos a enviar** sería un parámetro del algoritmo sensible de configurar, lo denominaremos t.

Capítulo 6

Modelado de la técnica paralela

En el estudio secuencial de la técnica se intentó modelar su comportamiento para poder inferir parámetros que representaban el comportamiento de los recorridos secuenciales de *backtracking*. Para los recorridos paralelos de la técnica, el modelar el comportamiento se complica al incrementarse el número de variables a tener en cuenta. Se persigue en este capítulo identificar una metodología general de trabajo para estimar el comportamiento de estos recorridos. En la primera parte del capítulo se proponen técnicas generales para abordar problemas de este tipo. En la segunda parte se particularizará la metodología general para dos esquemas paralelos diferentes. Serán estos esquemas para los que se realizará un estudio experimental en el próximo capítulo, que nos ayude a mejorar la metodología.

6.1 Metodología para la toma de decisión en esquemas paralelos

En el Capítulo 3 diseñamos una metodología para los recorridos secuenciales. En un principio pareció que se podría aplicar esta misma metodología con pequeñas modificaciones para ajustar los parámetros que surgen con los esquemas paralelos. Al realizar un estudio conjunto del análisis teórico de la técnica y los resultados experimentales (entre ellos los del próximo capítulo), surge la necesidad de abordar la metodología desde otro punto de vista. Partimos de una metodología similar a la del caso secuencial, y se ha ido refinando a medida que obteníamos datos experimentales que nos daban más detalles sobre el comportamiento de la técnica paralela.

En los esquemas paralelos el número de parámetros del algoritmo aumenta y es más complicado elegir valores apropiados, por ello es interesante disponer de técnicas que estimen posibles valores para ellos. Por ejemplo, en un esquema maestro esclavo con asignación estática de tareas por medio de una distribución cíclica ($EMEAE$) debemos elegir los valores para configurar (l,p). Si introducimos como parámetro el tipo de asignación a realizar deberemos configurar los valores (l,p,A). Debemos ser capaces de diseñar una metodología capaz de inferir valores para estos parámetros. Estos valores deberán estar lo más cerca posible de la configuración que lleva al tiempo de ejecución óptimo, liberando

al usuario de conocimientos de paralelismo.

El principal problema al paralelizar la técnica es que dependiendo del tipo de esquema paralelo utilizado, el número y la naturaleza de los parámetros podrá ser muy diferente. Nuestro objetivo será proponer una metodología general para los problemas resueltos con esquemas paralelos de *backtracking*, y debemos conseguir que sea adaptable a los diferentes tipos de entradas y diferentes tipos de esquemas utilizados.

6.1.1 Diferentes modos de inferir información (Rutina de decisión)

En el caso de la metodología para los recorridos secuenciales (Figura 3.2), obteníamos los valores de los parámetros a inferir modelando el comportamiento de la técnica con una función analítica y sustituyendo valores estimados de sus parámetros para obtener el valor de la función. En ocasiones vimos que la dificultad de estimar los parámetros nos llevaba a obtener resultados alejados de los tiempos reales.

En esta nueva metodología podemos adoptar dos enfoques diferentes para inferir la información necesaria (Figura 6.1):

- Modelar el comportamiento de la técnica con una expresión analítica (Figura 6.1 a)): deberemos tener una expresión analítica para cada modelo, y los parámetros de entrada serán específicos para cada uno de ellos.
- Inferir resultados a partir de un banco de datos que represente los problemas que queremos resolver (Figura 6.1 b)): el banco de datos de entrada deberá contener la información suficiente para poder extraer conclusiones.

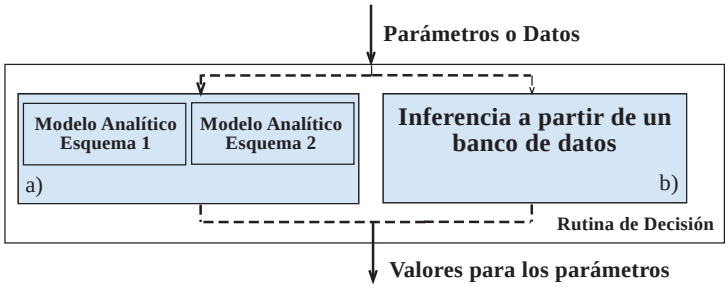


Figura 6.1. Esquema de los distintos enfoques que se pueden presentar en la rutina de decisión de la metodología para esquemas paralelos.

Si optamos por identificar un modelo para cada esquema (Figura 6.1 a)), el modelo tendrá como parámetros los que se detecten durante la fase de modelado de ese esquema particular. Entre los parámetros detectados estarán aquellos configurables que se deben estimar. En la mayoría de las ocasiones existen dependencias entre los parámetros que

dificultan su cálculo. Por estos motivos, no es sencillo identificar una expresión analítica que nos permita modelar el comportamiento del algoritmo como ocurría en el caso secuencial. En el caso de encontrar la expresión tendremos muchas dificultades para aproximar valores realistas de los parámetros que la componen. Por ejemplo, para un esquema secuencial puede resultar sencillo modelar el comportamiento de la técnica como hicimos en el Capítulo 3, y a pesar de las dificultades para estimar el valor del parámetro k se pueden conseguir valores aproximados. Vamos a considerar un esquema paralelo maestro esclavo con asignación estática de tareas y con intercambio de información (*EMEAE+I*). En este caso hemos detectado un total de cinco parámetros que intervienen en el tiempo de ejecución de la técnica. El valor de k será función de cuatro parámetros, el coeficiente de intercambio de información (e), el nivel de profundidad del *backtracking* inicial (l), el número de procesadores a utilizar (p) y el tipo de asignación utilizada (A). Además, la entrada que estamos resolviendo también influirá en el valor de k, que por tanto sería función de ella.

Para los esquemas donde tenemos dificultad de representar el comportamiento del algoritmo mediante una expresión analítica, debemos proponer otras soluciones. Se puede optar por un enfoque diferente, podemos intentar inferir el valor de los parámetros a partir de un banco de datos resultante de ejecuciones para diferentes entradas al problema.

Por ejemplo, imaginemos el esquema maestro esclavo estático con intercambio de información (*EMEAE+I*). Para este esquema hemos identificado como parámetros (l,p,e,A). Disponemos de suficiente información en el banco de datos para obtener los resultados de la Tabla 6.1, donde se representan los valores de los parámetros óptimos medios para diferentes tamaños de entrada (n). Cuando debamos resolver una entrada de un tamaño conocido consultaríamos en la tabla devolviendo como valor el de los parámetros almacenados. En caso de una entrada de tamaño 30 propondríamos utilizar (l, p, e, A)=(*8*, *6*, *18500*, *cíclico*). En este caso no se utiliza ningún modelo para tomar la decisión, únicamente se tiene en cuenta la información empírica de la que se dispone.

n	l	p	e	A
15	3	4	10000	bloques
20	4	6	10000	cíclico
25	7	6	10600	cíclico
30	8	6	18500	cíclico
35	9	6	19030	cíclico

Tabla 6.1. Tabla con información sobre los valores de los parámetros óptimos medidos para diferentes tamaños de entrada. (EMEAE+I)

Se presenta el problema de como obtener datos fiables que nos permitan inferir esta información con los valores adecuados. La alta dependencia de las entradas dificulta esta labor.

6.1.2 Procesamiento de la información de entrada en la rutina de decisión

Para inferir información realista, los datos de los que disponemos ("Parámetros o datos" de la Figura 6.1) deberán: en caso de los parámetros del modelo ser valores realistas y en caso del banco de datos tener un número de muestras suficientes. En ambos casos deben representar las entradas que se resolverán.

En este tipo de técnicas (*backtracking*) resolver problemas de tamaños medios suele llevarnos mucho tiempo, y este tiempo no podremos permitírnoslo como parte del tiempo de ejecución de la rutina, por lo que deberemos diseñar la metodología para conseguir un banco de datos suficientemente rico en la instalación. Al aumentar el número de parámetros el número de pruebas que se deben realizar en la instalación crece respecto al caso secuencial. Ofreceremos tres posibilidades en la metodología para conseguir la información necesaria:

- Utilizar información de la instalación: para problemas que tengamos localizadas las entradas a resolver y sigan un patrón similar podremos permitirnos utilizar solo información de la instalación para inferir los distintos valores de los parámetros que necesitamos ajustar.

- Utilizar información de la ejecución: si el rango de valores de las entradas es tan amplio que pueden llegar a tener comportamientos distintos, lo mejor sería intentar inferir la información una vez conocida la entrada a resolver.

- Utilizar información híbrida que combine la información de la instalación y ejecución: el problema de utilizar solamente información de la ejecución son las limitaciones de tiempo que nos impiden obtener buenas aproximaciones. En estas ocasiones es interesante combinar la información de la ejecución con la que tenemos previamente de la instalación para agilizar el tiempo de la toma de decisión.

La metodología deberá permitir tomar decisiones sobre cualquier tipo de problema que se plantee. En caso de que el *manager* encargado de instalar las rutinas tenga conocimientos avanzados del tipo de problema concreto y las entradas que se van a resolver, la metodología deberá dar la posibilidad de adaptarla a ese caso concreto. Para poder abarcar los casos generales y los casos particulares, la metodología diseñada debe permitir que la toma de decisión sea configurable: utilizar información de la instalación, de la ejecución o ambas.

En la Figura 6.2 se presenta un diagrama de esta parte de la metodología. Dispondremos del "Fichero de Datos procedente de la instalación" para amoldarlo al formato de salida y también dispondremos de la entrada concreta del problema que deseamos resolver. Podremos diseñar distintas estrategias que a partir de estas dos entradas, generen la información en "Parámetros o Datos" que será utilizada por las rutinas de decisión estudiadas en la sección anterior. Nosotros como diseñadores de la metodología debere-

mos encargarnos de que cada estrategia de decisión de la Figura 6.1 tenga al menos una estrategia de procesamiento de la información de entrada.

Vamos a utilizar un ejemplo concreto para explicar la Figura 6.2. Estamos trabajando con dos esquemas paralelos: un esquema *EMEAE+I* y un esquema *EMEAD*. Durante la fase de diseño de las rutinas, decidimos modelar el esquema *EMEAE+I* con dos modelos diferentes, los denominaremos "Modelo 1" y "Modelo 2". Para el esquema *EMEAD* no vamos a modelar su comportamiento y estimaremos los parámetros del algoritmo a partir de los datos almacenados en la instalación.

- Para el "Modelo 1" se podrán estimar los parámetros utilizados en la "Rutina de Decisión" (Figura 6.1) de dos formas: una podría utilizar únicamente información de la ejecución ("Procesa Datos Ejecución Modelo 1", Figura 6.2) y la otra solo información de la instalación ("Procesa Datos Instalación Modelo 1", Figura 6.2).
- Para el "Modelo 2" solamente ofrecemos la posibilidad de estimar sus parámetros por medio de información combinada de la ejecución y la instalación ("Procesa Datos Ejec. + Inst. Modelo 2", Figura 6.2).

Para el esquema *EMEAD*, solamente necesitaremos datos a partir de los que inferir los valores para los *APs* para las nuevas entradas. La rutina "Prepara datos instalación para inferir" (Figura 6.2) se encargará de manipular los datos que se registraron durante la instalación para adaptarlos a este esquema concreto.

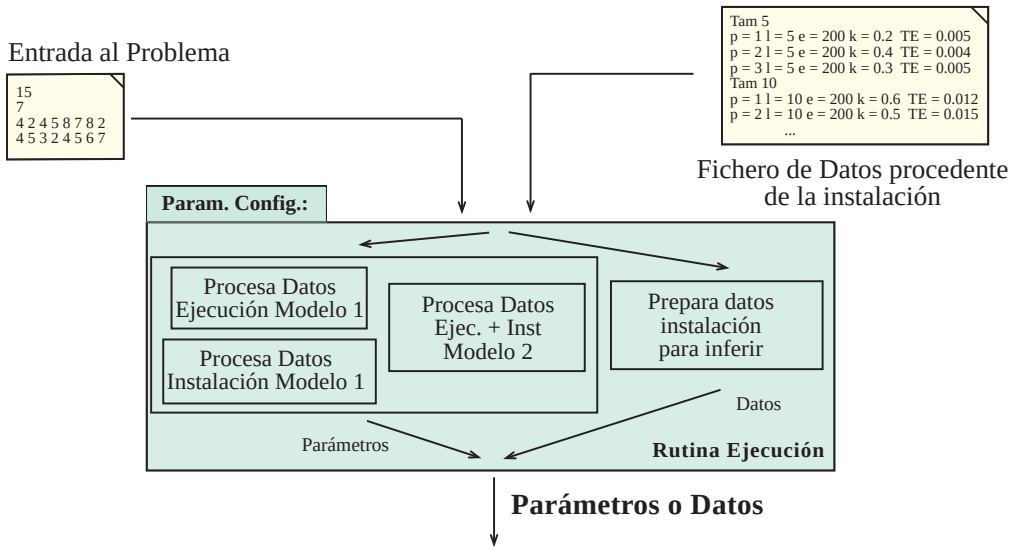


Figura 6.2. Esquema de los distintos enfoques que se pueden presentar en la rutina de decisión de la metodología para esquemas paralelos.

El *manager* configurará la "Rutina Ejecución" (Figura 6.2) para que utilice aquella solución que considere más adecuada. En este caso del ejemplo anterior decidirá si utilizar el esquema *EMEAE+I* o el esquema *EMEAD*. En caso de utilizar el *EMEAE+I* también configurará qué modelo quiere utilizar ("Modelo 1" o "Modelo 2"). Por último

si decide utilizar el "Modelo 1" configurará si estimar los parámetros con información de la instalación o de la ejecución.

A continuación vamos a proponer diferentes mecanismos para procesar la información de la que disponemos.

Enfoques para utilizar la información de la instalación

De manera muy similar a la que se estudió para el caso secuencial, podemos proponer diferentes enfoques para trabajar con la información de la instalación:

- Utilizar únicamente la información de los tamaños con los que hemos hecho pruebas para tomar las decisiones.
- Utilizar la información disponible e intentar extrapolar valores para los tamaños que no conocemos.
- Intentar realizar ajustes a funciones a partir de la nube de puntos formada por el conjunto de observaciones que tenemos.

Para problemas donde tenemos en cierta medida acotada la naturaleza de las entradas y el comportamiento de las mismas va a ser similar: podemos utilizar la información de la instalación para inferir el comportamiento de nuestra rutina. Se podrían realizar pruebas en la instalación que nos den información representativa de los valores de los parámetros que desconocemos. Almacenaríamos en una tabla los resultados obtenidos que nos permitan inferir la información que deseamos. Podemos optar por dos enfoques diferentes:

- Un primer enfoque consistiría en guardar información que nos permita obtener los valores de ciertos parámetros que utilizaremos en un modelo analítico del esquema. Por ejemplo, para un recorrido secuencial identificábamos un modelo teórico y en la instalación almacenábamos información de los diferentes valores que alcanzaban los parámetros del sistema al ejecutarlo con diferentes entradas. Utilizábamos esta información para estimar los parámetros del modelo y sustituirlos en él para las nuevas entradas. Concretamente almacenábamos valores del valor de k y TCN en la instalación, que servían para estimar los valores de estos mismos parámetros para las nuevas entradas.
- Un segundo enfoque consiste en guardar información directamente de los tiempos de ejecución que se consiguen para diferentes entradas y diferentes configuraciones de los parámetros identificados. Por ejemplo, recordemos un *EMEAE+I* donde identificábamos los parámetros (l,p,e,A). En tiempo de instalación podríamos ejecutar pruebas variando las entrada y los valores de los parámetros del algoritmo, para cada una de las ejecuciones almacenaríamos la configuración de los parámetros y el tiempo de ejecución obtenido; a partir de esta información registrada, podríamos

construir una tabla similar a la Tabla 6.1 a partir de la que seleccionar los valores para los parámetros.

En lugar de generar una tabla con valores, podríamos intentar aproximar a una función como se hacía en la metodología secuencial. Por ejemplo, en el caso del esquema *EMEAE+I* con un único tipo de asignación disponible, deberemos de ser capaces de decidir sobre los parámetros (l,p,e). Podemos intentar aproximar la superficie que formaría la variación de estos parámetros por medio de una función. El problema es que el número de factores a tener en cuenta en una función de estas características podría ser excesivamente elevado. Podríamos estudiar la influencia de cada uno de estos factores y descartar aquellos que no suponen suficiente variación. Aun así abordar problemas de este tipo donde existe tanta dependencia de las entradas puede que no sea la mejor solución. Solamente en casos donde las variación de las entradas siga un comportamiento bien definido merecería la pena dedicar esfuerzo a estudiar una función que aproxime sus valores. Será responsabilidad del *manager* que instala las rutinas, identificar la función a aproximar. Para añadir estas nuevas funciones, la metodología ofrecerá la posibilidad de definir nuevas funciones de aproximación durante la fase de programación.

Enfoques para utilizar la información de la ejecución

Incorporar simplemente información de la instalación acarrea los mismos problemas que ya se vieron para el caso secuencial. Podríamos generar versiones reducidas de las entradas de las que disponemos, resolverlas e inferir información. El problema en el caso paralelo es que el número de parámetros a considerar crece de forma notable y el tiempo empleado por la rutina se reduce (pudiendo dedicar menos tiempo a la toma de decisión durante la ejecución). Realizar pruebas para cada una de las combinaciones de los parámetros, supondría demasiado tiempo para la toma de decisión. Por ejemplo, supongamos el esquema *EMEAE+I* donde debemos ajustar los parámetros (l,p,e,A). ¿Para qué valores de l probamos?, ¿y de e?, ¿y de p?, ¿merece la pena probar para todos los tipos de asignaciones identificadas? Estas preguntas serán difíciles de resolver sin información adicional sobre el problema a resolver. Si el *manager* que instala nuestras rutinas conoce el tipo de problemas y los parámetros del esquema que estamos utilizando podría configurar los valores adecuados para estas rutinas, en caso contrario no podríamos estableces unos rangos fiables por defecto de manera general.

Enfoque híbrido, información de instalación más ejecución

Cada uno de los enfoques anteriores nos proporciona unas ventajas e inconvenientes. Las ventajas que obtenemos con uno son las desventajas que obtenemos con el otro. ¿Por qué no combinarlos? Esto es lo que vamos a tratar de explicar en este apartado.

El utilizar solamente información de la instalación plantea problemas cuando las entradas a resolver tienen comportamientos muy distintos. En estos casos la información de

la ejecución es crucial, pero no tenemos referencias sobre para que valores de parámetros explorar ni el número de pruebas a realizar. La solución propuesta en nuestra metodología es intentar buscar alrededor de unos valores razonables de los parámetros y realizar pruebas en la ejecución (con versiones reducidas de la entrada) mientras vayamos mejorando la función objetivo (tiempo de ejecución).

Por ejemplo, imaginemos el problema donde debemos de configurar los valores para (l,p,e). Si conocemos un valor para los parámetros a partir del cual empezar, podríamos realizar una búsqueda alrededor de ese punto variando los parámetros individualmente. Si al variar alguno de los parámetros conseguimos un valor menor que el anterior continuamos la búsqueda en esa dirección. En caso de empeorar, volveríamos al paso anterior y seguiríamos buscando en la dirección de otro parámetro. En la Figura 6.3 tenemos un ejemplo: partimos de una configuración para los parámetros (l,p,e)= (4, 3, 500) en el Paso 1. Decidimos buscar variando el valor de e y obtenemos un valor mejor del tiempo de ejecución (Paso 2). Seguimos buscando en esa dirección y para valores de e más pequeños el tiempo de ejecución aumenta, nos quedamos con e=400 y proseguimos la búsqueda variando l (Paso 4).

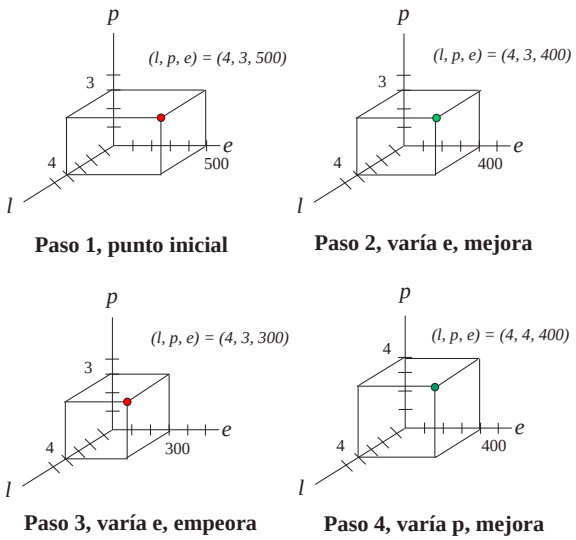


Figura 6.3. Búsqueda local de los parámetros del algoritmo alrededor de un punto inicial.

El problema es conocer el valor inicial de los parámetros a partir del cual empezar a buscar. Se proponen dos posibilidades en esta metodología:

- Utilizar un valor aproximado razonable para un tamaño de entrada o un tipo de problema. Por ejemplo, (l, p, e)= ($n/2$, $p/2$,$NodosTotales(n)/100$). Estos valores para los parámetros del algoritmo serán establecidos a partir de experimentos realizados por los diseñadores de la metodología, y podrán ser configurados por el *manager*. El problema es que este valor podría ir bien en algunos problemas pero no de forma general.

- Podríamos utilizar como valor inicial el que se haya aproximado durante la fase de instalación. Por ejemplo, se pueden utilizar valores similares a los de la Tabla 6.1 para obtener los valores iniciales para la búsqueda.

6.1.3 Diagrama general de la metodología

Una vez explicados todos los pasos que componen la nueva metodología adaptada a los esquema paralelos, se muestra en la Figura 6.4 un diagrama de la metodología completa. Las principales diferencias con el esquema de la metodología secuencial de la Figura 3.2 son: la decisión a tomar en la Figura 6.4 son los valores de los parámetros configurables. Añadimos la "Rutina Ejecución" y la "Rutina Decisión" que nos permiten una mayor variedad de configuraciones a la hora de inferir los valores de los parámetros. La rutina de instalación no solamente realiza pruebas secuenciales sino que puede realizar pruebas paralelas variando los diferentes parámetros configurables del esquema paralelo. Los datos almacenados en la fase de instalación pueden ser para ejecuciones paralelas o secuenciales. Es necesaria la presencia de los modelos tanto en la "Rutina Ejecución" para calcular los parámetros de los modelos teóricos, como en el esquema de *backtracking* para que registre la información sensible de estudiar.

El último aspecto a destacar en la Figura 6.4 es que se mantiene en la fase de programación un "Esquema Bactracking Secuencial" en lugar de un esquema algorítmico paralelo. Lo ideal sería abstraer al programador incluso de programar una solución paralela al problema. Si diseñamos un esqueleto secuencial amoldado a nuestras necesidades, nuestras rutinas por debajo se encargarían de manipular las funciones secuenciales del esquema y componer una solución paralela que resuelva el problema. El esquema algorítmico paralelo se encontraría por debajo del esquema secuencial, y quedaría oculto a los programadores. La rutina paralela podría configurar los parámetros del algoritmo con los inferidos por ella misma.

A continuación vamos a proponer dos esquemas algorítmicos diferentes y amoldaremos la metodología para estimar los valores de los parámetros.

6.2 Modelo de asignación estática simplificado (*MAES*)

El primer esquema que vamos a modelar va a ser un esquema de asignación estática de tareas. En este esquema no se va a producir intercambio de información entre los procesos esclavos. El proceso maestro una vez terminado el *backtracking* inicial, se va a comportar como un esclavo más resolviendo los subtrabajos que le sean asignados. Vamos a proponer un modelo muy general para obtener, en la sección de experimentos, aspectos significativos de este tipo de recorridos.

En este esquema maestro esclavo se realiza primero un *backtracking* inicial hasta nivel l para empezar a distribuir posteriormente el trabajo. Una primera aproximación del tiempo de ejecución tendrá una parte secuencial y una parte paralela:

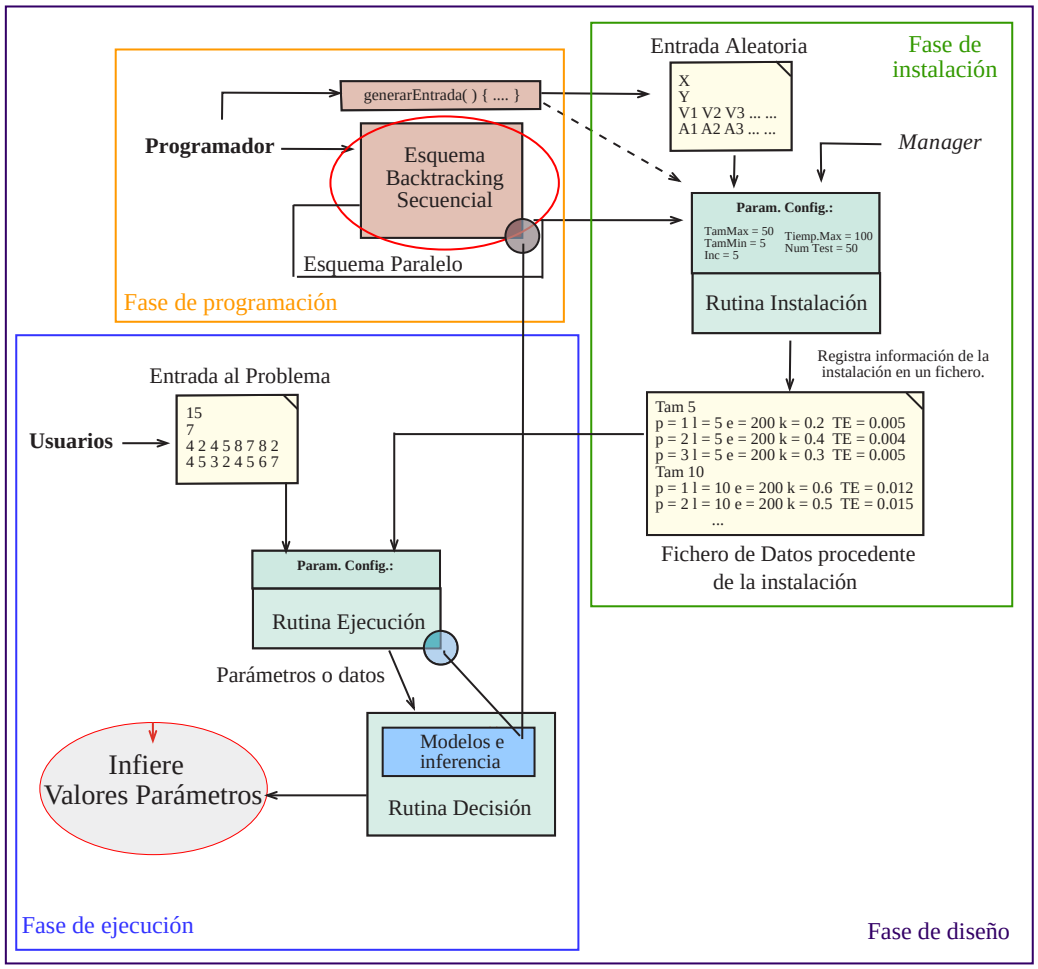


Figura 6.4. Esquema de la metodología para inferir información de autooptimización para una solución paralela en esquemas de *backtracking*.

$$TE(n,p) = TE_{Secuencial}(l) + TE_{Paralelo}(n-l,p) \tag{6.1}$$

La parte secuencial podremos aproximarla utilizando cualquiera de las ecuaciones del Capítulo 3. Tras haber sido respaldada con los resultados prácticos se propone utilizar la Ecuación 3.3.

Para modelar la parte paralela de este primer modelo, vamos a suponer que:

- El tiempo de ejecución de los subtrabajos es el mismo para todos. A todos ellos se asociará el mismo coeficiente de nodos recorridos (k) y se considera el mismo tiempo de cómputo (TCN).

- El trabajo se distribuirá entre todos los procesadores de forma homogénea: a todos los procesadores le será asignada la misma carga de trabajo.

A partir de estas premisas, el tiempo empleado en la parte paralela del algoritmo será equivalente al tiempo que tarde cualquiera de los procesadores que se encuentran resolviendo subtrabajos. Al suponer una distribución de la carga homogénea, podemos modelar el $TE_{Paralelo}(n-l,p)$ como:

$$TE_{Paralelo}(n-l,p) = \left\lceil \frac{TS(n-l)}{p} \right\rceil * TE_{Secuencial}(n-l) \tag{6.2}$$

donde TS es el número total de subtrabajos a asignar.

Con este modelo estamos suponiendo que los recorridos de *backtracking* son perfectamente paralelizables. El grado de paralelización crecería linealmente a medida que aumentamos el número de procesadores.

Los parámetros identificados en este esquema son: k, p y l. Para obtener el tiempo de ejecución mínimo deberíamos minimizar la Ecuación 6.1 sobre los parámetros del algoritmo a decidir: l y p.

6.3 Modelo de asignación estática complejo (*MAEC*)

Para tener un modelo más realista hay que tener en cuenta que el balanceo de la carga asignada a los subtrabajos no se encuentra balanceado. Además, el tiempo empleado en la parte paralela de este algoritmo dependerá directamente de que tipo de asignación se esté utilizando. Realmente, al disponer de varios procesadores trabajando simultáneamente, el tiempo a considerar en el modelo será el de aquel procesador que tarde más tiempo en resolver los problemas que le han sido asignados. En problemas donde realizamos podas es muy poco probable que exista un balanceo real de la carga entre los procesadores.

Se supone una asignación de subtrabajos asignados a los procesadores disponibles. Esta asignación se define como una estructura de "Número de subtrabajos" (NS) elementos donde cada uno de sus componentes adquirirá un valor en el intervalo de enteros $[0..p-1]$ (donde p es el número de procesadores disponibles). Se define esta asignación como:

$$\begin{array}{c} A(i) \quad , \quad \forall i = 0..NS-1 \\ A(i) \epsilon [0..p-1] \end{array} \tag{6.3}$$

Por tanto se puede identificar el tiempo que se emplea en la parte paralela del esquema como:

$$TE_{Paralelo}(n-l,p,A) = max_{j=0}^{p-1} \left\{ \sum_{i=0,A(i)=j}^{NS-1} TE_{Secuencial}(n-l) \right\} \tag{6.4}$$

Como se comentó en el capítulo de la introducción deberemos detectar los parámetros que intervengan en el tiempo de ejecución de las rutinas. Por un lado estarán los parámetros del sistema que no son configurables por los usuarios, entendiendo por sistema el conjunto del sistema físico más sistema lógico. Por otro lado se encuentran los parámetros del algoritmo, que sí serán configurables por los usuario. La clasificación para este esquema es:

- **Parámetros del sistema**: tiempo de cómputo de un nodo (TCN) y el índice de nodos generados (k). Realmente el último parámetro no es un parámetro que se deba de configurar para cada sistema o del que el usuario pueda elegir su valor. Consideramos k como un tipo especial de SP, ya que su valor no es configurable pero sí interviene de forma decisiva en una buena estimación del tiempo de la rutina. Se podría considerar un parámetro del sistema lógico, pero a su vez también es un parámetro de la entrada ya que depende de ella.
- **Parámetros del algoritmo**: número de procesadores a utilizar (p), nivel del *backtracking* inicial (l) y asignación de los subtrabajos a los procesadores (A).

6.3.1 Minimización del tiempo del modelo

Se va a diseñar una estrategia de estimación que nos permita aproximar que valores de los parámetros comentados nos llevan a los tiempos de ejecución mínimos. Los usuarios no expertos pueden no tener conocimientos suficientes para conocer los valores apropiados a asignar a estos parámetros. Por este motivo es interesante que nuestras rutinas le propongan valores que aunque no sean los óptimos no estén muy alejados de ellos.

Entre los diferentes parámetros que hemos identificado el k no es un parámetro configurable. La influencia de k en el tiempo de ejecución es decisiva y por este motivo hay que saber estimarla. Aunque ya se estudiaron las diferentes versiones en el capítulo secuencial, vamos a suponer que TCN también se mantendrá constante para cada una de las entradas.

Los parámetros algorítmicos sí son parámetros configurables en tiempo de ejecución. Se deberá minimizar la Ecuación 6.1, para lo que se deberá minimizar la suma de la parte secuencial y la parte paralela. Debemos ser capaces de diseñar un método capaz de estimar su valor. Para estimar los recorridos secuenciales podemos aprovechar las técnicas del capítulo tres: utilizando información conjunta de la instalación y de la ejecución.

Si mantenemos constantes los valores de l y p, deberemos minimizar la ecuación:

$$min_{(l,p,A)} \{TE_{secuencial}(n,l) + TE_{Paralelo}(n-l,p,A)\} \tag{6.5}$$

El valor de esta ecuación depende directamente de la asignación de subtrabajos que hagamos a los procesadores (suponiendo l y p fijos). Al minimizarla se estará consiguiendo la distribución de subtrabajos a los procesadores que haga mínimo el tiempo de ejecución

teórico. El valor de A encontrado corresponde con la distribución estática que se debe hacer entre los distintos procesadores.

En la Figura 6.5 tenemos un ejemplo de como deberíamos realizar la estimación del tiempo de ejecución del modelo paralelo. Se decide utilizar como nivel del *backtracking* inicial un valor de dos, se obtiene un tiempo de ejecución estimado de este *backtracking* secuencial de 2.1 segundos. A continuación se supone que se aplica la asignación estática por bloques entre dos procesadores establecida en la estructura *A=(0,0,0,0,1,1)*. Calculamos el valor de la Ecuación 6.4 y obtenemos como valor 2.75. Se obtiene como tiempo de ejecución final la suma de los dos tiempos estimados para valores de l=2 y p=2.

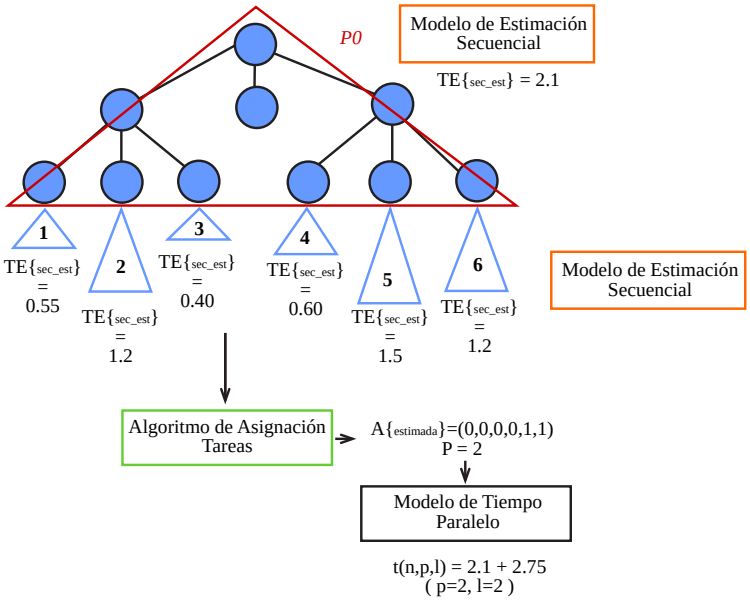


Figura 6.5. Esquema de estimación del tiempo de cómputo de un esquema maestro esclavo con asignación estática de tareas.

Se debe conocer para cada subtrabajo un valor de peso, que en este caso equivaldría al tiempo de ejecución. Se propone un método donde a cada subtrabajo generado durante el *backtracking* inicial se le asignará una estimación del tiempo de ejecución (en la Figura 6.5 está representado por TE_{sec_est}). En nuestra técnica cada subtrabajo corresponde a resolver un nuevo *backtracking* de tamaño $n - l$. Cada uno de estos *backtracking* se comportará como un nuevo *backtracking* secuencial. Se tratará de resolver el mismo problema solo que con entradas diferentes. Podemos utilizar la técnica de estimación secuencial de aproximación por recta añadiendo información en tiempo de ejecución (en la Figura 6.5 está representado en los diagramas "Modelo de Estimación Secuencial"). Las demás técnicas estudiadas para el modelo secuencial no serían válidas. Estas técnicas inadecuadas devolvían estimaciones de tiempo iguales para un mismo tamaño del problema con independencia de las entradas. En este caso necesitamos una estimación diferente para cada uno de los subproblemas.

Una vez estimados los tiempos deberemos aplicar el algoritmo que minimice la asignación de los subtrabajos entre los distintos procesadores que tenemos disponibles. El

tiempo dedicado a la parte paralela será el máximo de todas las asignaciones.

En este proyecto vamos a suponer que el método se aplicará en entornos de memoria compartida o en entornos homogéneos de memoria distribuida. Supondremos que la estimación de *TCN* será la misma para todos los subtrabajos del problema. Sin embargo, sería interesante aplicar la técnica a entornos heterogéneos de memoria distribuida. En este caso estimaríamos los pesos asociados a cada subtrabajo y suponiendo que los resolvemos en cada procesador con una capacidad de cálculo distinta. El valor de la variable *TCN* ya no sería el mismo para todos los procesadores. Se complicaría la búsqueda de la distribución de los subtrabajos entre los distintos procesadores.

6.3.2 Algoritmo de mapeo de tareas en los procesadores

Encontrar la asignación a los procesadores que minimice la Ecuación 6.4 no es un problema trivial. El problema es conocido en la literatura como asignación de tareas a procesadores [13]. Es un problema *NP-completo* por lo que resolverlo plantea los mismos problemas que resolver el problema de *backtracking* que estamos intentando autooptimizar. En este caso se deberá buscar una asignación de subtrabajos válida para cada uno de los diferentes valores de l que se prueben en el modelo de estimación de parámetros. Estos cálculos deberemos hacerlos en tiempo de ejecución.

Lo ideal para encontrar la solución óptima al problema sería buscar cual es la asignación mínima. El marco donde se nos plantea el problema nos impide dedicar un tiempo excesivo a esta búsqueda. No podemos dedicar más tiempo a encontrar la asignación que a resolver el propio problema. Por este motivo aplicaremos métodos voraces o técnicas metaheurísticas para conseguir las asignaciones.

Algoritmo voraz

En este caso se podría diseñar cualquier algoritmo voraz capaz de asignar las tareas entre los diferentes procesadores para minimizar la función de tiempo de ejecución. Cuanto más se acerque al valor óptimo y menos tiempo tarde mejor será el algoritmo. Como no es objetivo de este proyecto conseguir un algoritmo voraz óptimo, vamos a proponer uno sencillo para analizarlo en la sección de experimentos. Lo compararemos con las asignaciones estáticas por bloques, cíclicas y aleatorias.

Lo que se hará será recorrer todos los subtrabajos y asignar cada uno de ellos al procesador que menos incremente, según la distribución parcial hasta ese momento, el tiempo de ejecución del modelo.

Métodos metaheurísticos

También se podrían aplicar los métodos metaheurísticos, que dan valores aceptables en tiempo de ejecución pequeños. Como no es el objetivo de este proyecto no se va a profundizar más en este tema. Se puede consultar más acerca de estas técnicas en los trabajos [13] y [20].

6.4 Modelo de asignación estática con intercambio de información (MAEII)

El último esquema que vamos a abordar es el maestro esclavo con asignación estática de tareas pero añadiendo intercambio de información. Para no complicar el modelo, vamos a utilizar una única asignación cíclica de las tareas. Este esquema es más complicado de modelar que los anteriores. Tendríamos como parámetros a decidir (l,p,e). Al igual que en el esquema anterior, utilizamos para modelar el esquema la Ecuación 6.1 añadiendo como parámetro el intervalo de nodos generados (e) para el intercambio de información:

$$TE(n,p,e) = TE_{Secuencial}(l) + TE_{Paralelo}(n-l,p,e) \tag{6.6}$$

Podemos utilizar las técnicas secuenciales para estimar el tiempo secuencial. La parte paralela del modelo la podríamos modelar como:

$$\begin{aligned} TE_{Paralelo}(n-l,p,e) &= k_{MedioParalelo} \cdot \left(\frac{NNG(n-l)\cdot TS(n-l))}{p-1}\right) \cdot TCN \\ &+ TC \cdot \frac{NNG(n-l)\cdot k_{MedioParalelo}}{e} \end{aligned} \tag{6.7}$$

Vamos a analizar los dos términos de la ecuación por separado:

- El primer término corresponde con el tiempo dedicado por los diferentes procesadores a resolver los subproblemas. Como se produce intercambio de información entre los procesos esclavos, vamos a suponer que la carga ahora sí se distribuye de manera homogénea. Consideraremos que el índice de poda se vuelve a repartir entre todos los procesadores, y por eso consideramos un valor medio estimado $k_{MedioParalelo}$. El segundo factor del primer término, equivale al número de nodos que generaría cada uno de los distintos procesadores, está dividido por *p-1* porque consideramos que el proceso maestro no resuelve tareas. Por último multiplicamos por el tiempo de cómputo de un nodo.

- El segundo término representa el coste de las comunicaciones. TC es el tiempo que se tardaría en cada intercambio de información, va multiplicado por el número total de comunicaciones que haría cada uno de los procesos esclavos. Consideramos

solo las comunicaciones de un proceso esclavo porque vamos a suponer que no hay solapamiento entre todas ellas.

Aunque hemos encontrado un modelo razonable, es muy complicado estimar el valor de $k_{MedioParalelo}$ ya que depende de todos los demás parámetros (l,p,e) y además de la propia entrada.

6.5 Esquema algorítmico secuencial con paralelización implícita

Cuando paralelizamos un código los diferentes procesos hijos compartirán el código fuente del programa. En el propio código fuente tendremos marcas que identifican las partes asignadas a unos procesadores u otros. Sin embargo existirán partes del programa que deberán ser realizadas por todos los procesadores. Una buena costumbre es programar nuestras rutinas de manera que todos los datos que se procesen dentro de funciones sean privados (ya sea añadiéndolos como parámetros a las funciones o definiéndolos dentro de cada función).

Pretendemos identificar las partes de un esquema algorítmico secuencial del que se pueda realizar internamente su paralelización por medio de un esquema maestro esclavo. Partiremos de un esquema algorítmico secuencial similar al Código 2.2. Los programadores solamente tendrán que completar los fragmentos de las funciones secuenciales, internamente se programaría el esquema paralelo utilizando estas funciones secuenciales. A continuación vamos a mostrar como deberíamos ampliar un esquema secuencial similar al del Código 2.2 para poder paralelizarlo implícitamente:

- En la función principal de *bactracking* deberíamos añadir como parámetros el nivel hasta donde queremos llegar en el recorrido de búsqueda (l) y la lista de tareas donde vamos a añadir los subtrabajos (*list*). Además, aunque no tiene nada que ver con el paralelismo, podremos extender el esquema básico con información que es necesaria calcular en el modelo de tiempo teórico. En este caso añadimos la variable k y *nodosGenerados*, y añadimos los fragmentos de código para calcularlos. Al final de la función principal de *backtracking* se calcularía el parámetro k y se devuelve a la función que lo invocó para que lo utilice (Código 6.1).

```
1  int m01back( ..., int l, struct lista * list,
2                   double * k, double nodosGenerados)
3  {
4     nodosGenerados = 0;
5     ... ... ...
6     k[0] = valorCalculadoDeK();
7  }
```

Código 6.1. Extensión de la función principal de *backtracking* para permitir un esquema *M/E*.

- En este caso, *nodosGenerados* también sería pasada a la función generar, que es donde se incrementa (Código 6.2).

```
1   void generar( ..., double * nodosGenerados)
2   {
3       ... ... ...
4       nodosGenerados[0]++;
5   }
```

Código 6.2. Extensión de la función generar.

- El valor del nivel final del *backtracking* inicial (l) y la lista de tareas (*list*) se deberán pasar a la función criterio. Se añadirá funcionalidad para determinar cuando un nodo está a nivel l, y en ese caso se añadirá a la bolsa de tareas y se devolverá el criterio como falso (Código 6.3).

```
1   int criterio( ... ... ... ... ..., int l, struct lista * list)
2   {
3       ... ... ...
4       if ( ( nivel == l ) & cumpleCriterio( ) ) {
5           nodo = generamosNodoSubproblema();
6           addLista ( list, nodo);
7           return (0);
8       }
9       ... ... ...
10  }
```

Código 6.3. Extensión de la función criterio.

- Deberemos tener programada una lista de tareas donde se van a ir añadiendo los subproblemas. La funcionalidad básica de esta lista estará programada, pero no podremos definir de antemano cual será la estructura de sus nodos. El programador del esquema definirá la estructura de un nodo, que contendrá toda la información asociada a un subproblema (Código 6.4).

```
1   struct nodoBacktraking {
2       // Debe definirlo el programador con los datos que
3       // identifican a cada instanacia del problema.
4   };
5
6   struct nodoLista {
7       struct nodoBacktraking * info;
8       struct nodoLista * sig;
9   };
10
11  struct lista {
12      struct nodoLista * inicio;
13      struct nodoLista * fin;
14  };
15
16  void addLista ( ... ) { ... }
17  struct nodoBacktraking * getNodo ( ... ) { ... }
18  struct nodoBacktraking * removeLista ( ... ) { ... }
19  int length ( ... ) { ... }
```

Código 6.4. Estructura genérica de la lista que almacena los subproblemas.

Si tenemos definido de esta manera nuestro esquema nos resultará sencillo componer las funciones para construir un esquema paralelo maestro esclavo similar al del Código 5.1, Código 5.2 o cualquier otro de este tipo que se nos ocurra.

A la hora de realizar esta paralelización interna deberemos tener en cuenta que paradigma de programación paralela estamos utilizando:

- Memoria compartida: deberemos proteger todas las variables globales definidas por el usuario con accesos en exclusión mutua.
- Memoria distribuida: entre otras tareas, el proceso maestro deberá enviar todas las variables globales definidas por el usuario al resto de procesos esclavos. La implementación de este esquema resultará más complicada.

6.6 Adaptación de la metodología y casos de prueba propuestos

A lo largo del capítulo hemos hablado de una metodología general y hemos analizado diferentes esquemas paralelos para la técnica del *backtracking*. En esta sección explicamos como vamos a realizar los experimentos, para que esquemas y como se adapta la metodología para cada uno de ellos.

- Metodología particular para *MAES*:
  - Implementación de una solución *M/E* con asignación estática de memoria y tres tipos de asignaciones (cíclica, bloques y aleatoria). Se utilizará el paradigma de memoria compartida.
  - Configuración de la instalación para generar problemas aleatorios generales y resolverlos secuencialmente registrando información útil para inferir valores de los parámetros de entrada al modelo.
  - Procesamiento de los datos de entrada de la instalación
  - Modelado del esquema mediante una expresión analítica.
- Metodología particular para *MAEC*:
  - Implementación de una solución *M/E* con asignación estática de memoria y tres tipos de asignaciones (cíclica, bloques y aleatoria). Implementación del algoritmo voraz utilizado para minimizar el tiempo de ejecución paralelo. Se utilizará el paradigma de memoria compartida.
  - Configuración de la instalación para generar problemas aleatorios generales y resolverlos secuencialmente registrando información útil para inferir valores de los parámetros de entrada al modelo.

  - Procesamiento de los datos de entrada de la instalación añadiendo información de la ejecución.
  - Modelado del esquema mediante una expresión analítica.

- Metodología particular para *MAEII*:
  - Implementación de una solución *M/E* con asignación estática de memoria y tres tipos de asignaciones (cíclica, bloques y aleatoria). Se utilizará el paradigma de memoria distribuida.
  - Configuración de la instalación para generar problemas aleatorios generales y resolverlos de forma paralela. Se registrará información sobre las ejecuciones paralelas de los problemas.
  - Procesamiento de los datos de entrada de la instalación para establecer un primer punto a partir del que realizar la búsqueda local. Búsqueda local en ejecución para una versión reducida de la entrada concreta.
  - Inferencia sobre los datos de entrada y la información que obtengamos en ejecución.

Capítulo 7

Resultados experimentales paralelos. Mochila 0/1

En este capítulo vamos a implementar los esquemas paralelos para resolver el problema de la Mochila 0/1. Utilizaremos los esquemas paralelos explicados en el Capítulo 5 e implementaremos como modelos paralelos para la toma de decisión los estudiados en el último apartado del Capítulo 6. Estudiaremos los problemas que plantean los modelos propuestos y que modificaciones sería conveniente realizar sobre ellos.

7.1 Esquema *EMEAE* (Memoria Compartida)

Vamos a implementar un esquema maestro esclavo con asignación estática de tareas. Se implementarán las asignaciones estudiadas en el Capítulo 6: las tres asignaciones básicas (bloques, cíclica y aleatoria) y la asignación voraz para el modelo complejo. Al igual que en el esquema secuencial utilizaremos para programarlo C++. La implementación la haremos en memoria compartida, y utilizaremos el estándar OpenMP para paralelizar nuestros códigos.

No se van a estudiar las cinco versiones que se vieron en el estudio práctico secuencial, sino que vamos a paralelizar las versiones dos y cinco utilizando un esquema similar al que se vio en la Sección 6.5.

De forma resumida utilizaremos la primitiva **"#pragma omp parallel for"** para generar los hilos correspondientes con todos los hijos (Código 7.1).

```
1  ... ... ... ... ...
2  #pragma omp parallel for
3  for ( int i=0; i<numProc; i++ ) {
4     --- --- ---
5  }
6  ... ... ... ... ...
```

Código 7.1. Primitiva en OpenMP con la que conseguimos crear un hilo por cada iteración del bucle. Si el total de iteraciones es el número de hilos que queremos generar crearemos el número de hijos deseado.

Como vimos en el Código 5.1, cada uno de los hijos generados recorrerá el array de asignaciones y resolverá aquellos problemas que le hayan sido asignados. El acceso al array de asignaciones, pese a ser una variable compartida, se podrá realizar de forma concurrente por realizarse lecturas a zonas diferentes. Como no intercambian información, cada uno de los procesos tendrá definida una variable local que almacena la mejor solución encontrada. Cuando cada proceso esclavo termine con todos los subtrabajos que le han sido asignados, actualizará la variable global que almacena la mejor solución encontrada por todos ellos. El acceso a esta variable global compartida deberá realizarse en exclusión mutua. Para ello utilizamos la primitiva **"#pragma omp critical (section)"**:

```
1  ... ... ... ... ...
2  #pragma omp critical (section1)
3  {
4     if ( VOA_privado > VOA ) {
5        VOA = VOA_privado;
6     }
7  }
8  ... ... ... ... ...
```

Código 7.2. Acceso a una variable en exclusión mutua mediante la directiva #pragma omp critical de OpenMP.

La ejecución de memoria compartida se llevará a cabo en la máquina del Grupo de Computación Científica de la Universidad de Murcia (Línea de Computación Paralela): SOL. Se utilizará uno de los nodos compuesto por 4 núcleos con Intel Xeon 3.00GHz de 64 bits, con 1.5 GB de RAM.

7.1.1 Utilizando el modelo *MAES*

A continuación vamos a realizar el estudio de los resultados prácticos obtenidos utilizando el modelo de asignación estática de tareas en su versión más simple. Al disponer de varios parámetros a ajustar, el número de resultados obtenido es grande. Un análisis global de todos los parámetros sería difícil de acotar, por lo que vamos a estudiar la influencia de cada uno de los parámetros en el tiempo de ejecución fijando los demás. Una vez estudiados independientemente procederemos a realizar el estudio general del modelo de autooptimización estudiado. En las próximas subsecciones vamos a resumir los resultados, mostrando y explicando aquellos que merece la pena destacar.

Particularizando la metodología general del capítulo anterior a este modelo tendríamos que: durante la fase de instalación se resolverían problemas secuenciales para tener información que nos permita estimar el tiempo de los subtrabajos. Durante la fase de ejecución aplicaremos el modelo *MAES* para estimar los valores de los parámetros configurables. La rutina de ejecución utilizará los datos obtenidos en la instalación para estimar los valores de los parámetros del modelo, que lo necesitaremos en la rutina de decisión. La rutina de decisión buscará valores en el modelo variando l desde uno hasta el número de niveles del *backtracking*, y variando p desde uno hasta el máximo número de procesadores que se pueden utilizar. El valor máximo de p deberá ser configurado por el *manager* que instala las rutinas.

De aquí en adelante, cuando se hable de estimaciones secuenciales de *backtracking* en tiempo de instalación, se utilizará una configuración de la rutina de instalación similar a la que se

utilizó en los experimentos secuenciales del capítulo 4.

Para obtener resultados más fiables vamos a realizar las pruebas para dos tipos de esquemas completamente diferentes: el esquema dos (donde los índices de podas eran pequeños) y el esquema cinco (las podas que se hacían eran mucho más agresivas al utilizarse heurísticas que acotaban significativamente las búsquedas). También probaremos diferentes asignaciones estáticas para ver la influencia real que tienen en el esquema.

Comportamiento del parámetro p para l fijo

Vamos a fijar la profundidad del nivel de *backtracking* inicial para ver como influye el variar el número de procesadores. La máquina donde se van a realizar las pruebas cuenta con un máximo de 4 núcleos para utilizarlos en memoria compartida. Por este motivo variaremos p entre uno y cuatro.

En la Tabla 7.1 tenemos los resultados obtenidos por el modelo (TM) para cada uno de los dos esquemas comentados. Se utilizará para estimar los valores de k los valores de la instalación extrapolando mediante una recta. Además se incluyen los tiempos reales para cada uno de los esquemas y utilizando asignación cíclica y por bloques, $TR_{TipoAsignacion}$.

			Esquema 2			Equema 5		
Tam	l	p	TM	$TR_{\text{cíclico}}$	$TR_{bloques}$	TM	$TR_{\text{cíclico}}$	$TR_{bloques}$
35	10	1	357.81	2.74	2.74	2.54	0.017	0.017
		2	178.90	1.95	2.01	1.27	0.015	0.018
		3	119.44	**1.01**	1.81	0.84	0.014	0.019
		4	**89.45**	1.16	**1.42**	**0.63**	**0.013**	**0.014**
40	10	1	596.01	346.05	345.05	76.74	3.38	3.37
		2	298.00	230.21	223.85	38.37	3.17	3.12
		3	198.92	**118.39**	193.26	25.61	**1.56**	12.06
		4	**149.00**	158.88	**155.19**	**19.18**	1.83	**2.85**

Tabla 7.1. Variación de los tiempo de ejecución manteniendo l fija y variando el parámetro p.

Aunque solo se muestran en la Tabla 7.1 los resultados para problemas de tamaño 35 y 40, se han realizado más experimentos con tamaños de problema menores (20, 25 y 30) y diferentes entradas para cada uno. Los valores de esta tabla representan los valores obtenidos para estos diferentes tamaños. Las conclusiones que podemos extraer de esta tabla son:

- El modelo supone una distribución homogénea de los subtrabajos y estima que a medida que aumentemos el número de procesadores se divide proporcionalmente el tiempo de ejecución para todos los casos. En los tiempos reales el descenso en tiempo real no es proporcional al aumentar el número de procesadores. En algunos casos, añadir más procesadores supone incrementar el tiempo de ejecución.
- Si evaluamos el modelo respecto a la decisión del número de procesadores a elegir, este modelo siempre elegirá utilizar el máximo número de procesadores que tengamos. Vamos

a comparar el cociente del error medio de la decisión del modelo con la máximo error que se podría cometer. En este caso los errores cometidos cuando falla el modelo son del 1.08, frente al 1.94 que podríamos cometer si elegimos la peor opción.

- El tipo de asignación que utilicemos sí influye directamente en el tiempo total de ejecución. El modelo no es capaz de diferenciar entre ellas. En el caso de los errores cometidos, una asignación por bloques se aproxima más a este modelo de distribución homogénea que estamos utilizando.
- El tiempo estimado del modelo es superior al tiempo real de la ejecución. El modelo utiliza para estimar los valores solamente información procedente de la instalación. Cuando generamos nuevas entradas, estas tienen también un comportamiento diferente para cada una de ellas. La información media de la instalación no aproxima bien los valores de los parámetros por no considerar la entrada concreta.

De los resultados obtenidos en la Tabla 7.1 llama la atención que al añadir más procesadores se incremente el tiempo de ejecución en lugar de reducirlo o mantenerlo constante en un caso desfavorable (asignación cíclica esquema dos, tamaño 35 y 40; asignación cíclica y por bloques, esquema cinco, tamaño 40). Una posible explicación es el desbalanceo real de los subtrabajos asignados a cada procesador. Vamos a buscar una explicación empírica a este comportamiento particular. Estudiaremos un problema de tamaño de entrada 40, con estimación de k en instalación con recta, *TCN* general, l igual a doce, p igual a cuatro y asignación estática cíclica de tareas. En la Tabla 7.2 se compara el k medio estimado por el modelo ($k_{MedioEstimado}$) con los k reales medios para distinto número procesadores utilizados ($k_{RMedio}(p = numProc)$).

$k_{MedioEstimado}$	$k_{RMedio}(p=1)$	$k_{RMedio}(p=2)$	$k_{RMedio}(p=3)$	$k_{RMedio}(p=4)$
0.987561	0.999976	0.999978	0.999974	0.999973

Tabla 7.2. Valor estimado de k y valores de k_{RMedio} reales para distinto número de procesadores utilizados.

De los resultados de la Tabla 7.2 podemos extraer las siguientes conclusiones:

- El valor de porcentaje de poda estimado ($k_{MedioEstimado}$) es mucho menor que el que se consigue en realidad. Esta situación explicaría que el tiempo del modelo sea superior al tiempo real del problema. Este problema es el mismo que se planteaba en el análisis secuencial de la técnica: la dificultad de aproximar el valor de k.
- Los valores de los coeficientes medios de poda no son válidos para representar el comportamiento real del problema. Cuando utilizamos tres procesadores el índice de poda es menor que utilizando dos, pero el tiempo de ejecución se reduce. Por contra, al aumentar a cuatro procesadores, el índice de poda también baja respecto a utilizar tres pero el tiempo de ejecución total es mayor.

El problema es que observando los coeficientes de poda medios de los procesadores no estamos representando el comportamiento de la rutina paralela. Vamos a aumentar el grado de detalle de recogida de datos. En la Tabla 7.3 se muestran los valores de poda locales de cada

procesador ($k_{LocalProcX}$). Al no intercambiar información entre los distintos procesadores, cada uno trabajará de forma independiente, pudiendo ser k completamente diferente en cada uno de ellos.

	p=1	**p=2**	**p=3**	**p=4**
$k_{LocalProc0}$	**0.999976**	0.999983	0.999976	0.999989
$k_{LocalProc1}$		**0.999973**	0.999980	0.999983
$k_{LocalProc2}$			**0.999967**	0.999968
$k_{LocalProc3}$				**0.999950**

Tabla 7.3. Valores locales de índice de poda para cada uno de los distintos procesadores que se utilizan para una entrada de tamaño 40 de la mochila 0/1.

Al resolver cada procesador problemas distintos, el tiempo de ejecución de cada uno de los procesadores será distinto. El tiempo de ejecución total, se corresponde con el de aquel que más tarde en acabar, que en este caso será el que tenga menor valor de k. Si mantenemos l fijo podemos despreciar la parte secuencial en la comparación. El tiempo empleado dependerá únicamente del número de nodos recorridos por el procesador que más tarda en terminar ($NRPML$).

En la Tabla 7.4 tenemos calculado el número de nodos recorridos por el procesador más lento para cada valor de p. Utilizamos para calcularlo una variación de la Ecuación 3.3 con TCN constante, el k_{Local} y multiplicando NNG por el factor $1/p$ (para considerar solo los nodos que le han sido asignados a un procesador).

	p=1	**p=2**	**p=3**	**p=4**
TR	3.7438	2.1510	**1.6848**	1.8623
NRPML	6442.5	3623.9	**2952.8**	3355.4

Tabla 7.4. Comparativa entre el tiempo de ejecución real TR y el número de nodos recorridos por el procesador que más tarda en realizar los subtrabajos asignados. (TR medido en segundos, $NRPML$ medido en nodos)

Los datos de la Tabla 7.4 dan explicación a porque a pesar de conseguir porcentajes de poda cada vez menores a medida que aumentamos p en la Tabla 7.3, el tiempo de ejecución no aumenta en todos los casos. Podemos concluir que si no se introduce intercambio de información entre los diferentes procesos, el k local de cada uno de ellos juega un papel decisivo en el comportamiento del algoritmo paralelo. El tiempo de ejecución total será función del grado de paralelismo conseguido y el valor k local más pequeño.

Estudio del comportamiento del parámetro l para p fijo

En este apartado vamos a estudiar como se comporta nuestro modelo al mantener fijo el número de procesadores y variar el parámetro l. En primer lugar vamos a mostrar como se comportaría el modelo según el tipo de esquemas que estamos utilizando.

En las tablas 7.5 y 7.6 tenemos los tiempos del modelo (TM) para los dos esquemas que hemos comentado. Variamos únicamente el valor de l, y mantenemos p fijo en cuatro. Se utilizará en todos los casos asignación estática de tareas. Mostramos solo los resultados de tamaño 40 para no sobrecargar este documento, se han probado con diferentes tamaños más pequeños y la tendencia era similar.

Esquema 2						
l	**TM**	$In40_1$	$In40_2$	$In40_3$	$In40_4$	$In40_5$
1	298.23	27.33	0.112	**0.000367**	128.24	227.63
2	**104.37**	21.23	0.095	0.000383	90.793	150.81
3	109.32	19.30	0.092	0.000381	93.200	154.15
4	114.28	18.18	0.090	0.000478	93.341	153.26
5	119.24	19.60	0.089	0.000422	95.581	149.53
6	124.20	19.19	0.090	0.000479	88.544	148.89
7	129.16	18.53	0.093	0.000426	81.341	147.50
8	134.12	18.09	0.082	0.000491	81.323	152.65
9	139.08	18.33	0.085	0.000481	88.450	152.06
10	144.04	19.63	0.076	0.000534	88.622	149.74
11	149.00	18.78	**0.069**	0.000423	81.448	153.29
12	153.96	19.34	0.071	0.000478	86.203	151.86
13	158.92	17.24	0.070	0.000422	95.635	147.84
14	163.88	17.63	0.075	0.000445	81.649	147.74
15	168.84	15.13	0.081	0.000542	82.674	145.11
16	172.21	13.86	0.093	0.000485	80.256	120.63
17	178.75	**13.18**	0.100	0.000528	69.701	121.17
18	183.71	14.90	0.117	0.000492	71.441	123.68
19	188.67	14.38	0.149	0.000443	74.040	**121.45**
20	193.63	15.85	0.194	0.000506	**67.332**	122.04
21	198.60	17.17	0.254	0.000497	68.539	121.55

Tabla 7.5. Tiempos de ejecución del modelo (TM y reales $In40_x$ para cinco entradas diferentes de tamaño 40. Se mantiene p fijo en cuatro con asignación cíclica de tareas. (Tiempo en segundos)

La interpretación que le damos a los datos del esquema dos, representados en la Tabla 7.5, es:

- En este esquema la búsqueda se realiza más a ciegas; es decir, aunque se realicen podas propias del problema no se guiará radicalmente el recorrido hacia la mejor solución. En este tipo de esquemas aplicar paralelismo sí es más apropiado, como muestran los tiempos de ejecución de la Tabla 7.5. En estos casos, estimar correctamente el valor de l que nos lleva a tiempos de ejecución más pequeños se complica por las diferencias de pesos en los subtrabajos asignados.

- Para este esquema el modelo no acierta en sus estimaciones. Los motivos por los que falla son:

Esquema 5						
l	**TM**	$In40_1$	$In40_2$	$In40_3$	$In40_4$	$In40_5$
1	54.00	0.521	0.000523	**0.00027**	19.1858	11.6363
2	**18.12**	**0.519**	**0.000455**	0.00029	**19.1762**	**11.6133**
3	18.23	0.526	0.000546	0.00030	19.1891	11.6400
4	18.35	0.521	0.000589	0.00031	19.1878	11.6186
5	18.47	0.524	0.000697	0.00033	19.2085	11.6217
6	18.59	0.521	0.000861	0.00037	19.2115	11.6120
...	...	...	...	...	...	...
16	19.71	0.940	0.071401	0.00050	19.6552	12.0836
17	19.89	1.264	0.096758	0.00052	20.0824	12.4917
18	20.01	1.855	0.142370	0.00053	20.8750	13.3813
19	20.13	2.664	0.160155	0.00051	22.3417	14.8241
20	20.25	4.194	0.220279	0.00054	25.2875	17.6664
21	20.37	6.815	0.305690	0.00053	30.5954	22.8753

Tabla 7.6. Tiempos de ejecución del modelo (TM y reales $In40_x$ para cinco entradas diferentes de tamaño 40. Se mantiene p fijo en cuatro con asignación cíclica de tareas. (Tiempo en segundos)

- Al tratarse de un modelo que supone asignaciones homogéneas de los subtrabajos, no tiene en cuenta un punto principal en este tipo de esquemas: la importancia de las podas entre los diferentes procesadores que cooperan.
- Se estiman los tiempos de los *backtracking* únicamente a partir de información de la instalación. Esta información no representa completamente la realidad. Según esta información el índice de poda es mayor a medida que aumenta el tamaño del problema. Sin embargo, al paralelizar la técnica esto no siempre es cierto. Al aumentar el l el modelo estima que los *backtracking* secundarios tienen todos un índice de poda menor y a esto se debe que deduzca como óptimo siempre l igual a dos (genera cuatro tareas).

La interpretación que le damos a los datos del esquema cinco, representados en la Tabla 7.6, es que para esquemas de *backtracking* donde se aproxime rápidamente a la solución a buscar, el modelado de la técnica no merece la pena con un esquema estático sin intercambio de información. Al introducir paralelismo estaríamos recorriendo un espacio de búsqueda mayor en lugar de acotarlo. Con una implementación de este tipo, lo único que se consigue a medida que aumentamos el nivel de profundidad es aumentar el tiempo de ejecución final. Este comportamiento se debe a que el esquema aproxima muy rápido la solución, y al aumentar el número de subtrabajos independientes generados, los coeficientes de poda locales a cada procesador (k) descienden y el tiempo de ejecución es mayor.

Aunque no aparece en los datos de las tablas 7.5 y 7.6, cuando el l supera la mitad de tamaño del *backtracking* se produce un aumento significativo en el tiempo de ejecución. Este comportamiento sí es identificado por el modelo.

A partir de este punto vamos a utilizar solamente el esquema dos para estudiar el comportamiento de los modelos. Este trabajo pretende estimar los parámetros y el comportamiento

general de la paralelización de las técnicas de *backtracking*, e incluir el estudio de esquemas particulares que tengan comportamiento extremo podría llevarnos a confusión y conclusiones inadecuadas.

7.1.2 Utilizando el modelo *MAEC*

En este apartado vamos a analizar los resultados que obtenemos con el modelo más complejo de la asignación estática. Utilizaremos las mismas entradas que se utilizaron en el estudio del modelo simplificado para comparar con él la estimación de este nuevo modelo. Al igual que en el modelo simplificado, vamos a estudiar el comportamiento de cada parámetro por separado fijando los demás. Por los motivos comentados en el párrafo anterior y para no sobrecargar este documento con excesivos datos, solo estudiaremos el comportamiento de los parámetros por separado para el esquema dos. El análisis de la elección conjunta de todos los parámetros sí lo realizaremos para los dos esquemas (dos y cinco).

Al igual que en el modelo anterior, deberemos adaptar la metodología general del capítulo seis a este esquema y modelo como se indica a continuación. Durante la fase de instalación se resolverían problemas secuenciales para tener información que nos permita estimar el tiempo de los subtrabajos. Durante la fase de ejecución aplicaremos el modelo *MAEC* para estimar los valores de los parámetros configurables. La rutina de ejecución utilizará los datos obtenidos en la instalación, complementándolos con los extraídos en la ejecución (resolviendo versiones reducidas de la entrada), para estimar los valores de los parámetros del modelo. Al recabar información de la ejecución no podemos inspeccionar todas las combinaciones de parámetros en tiempo de ejecución. Hemos decidido establecer un rango de valores por defecto donde buscar: l variará entre uno y el número de niveles del *backtracking* partido por dos; al igual que en el modelo anterior p variará entre uno y el máximo de procesadores a utilizar (configurado por el *manager*).

Estudio del comportamiento del parámetro p y tipo de asignación para l fijo

En la Tabla 7.7 están los datos referentes a la toma de decisión de nuestro modelo fijando el parámetro l. El modelo decidirá el mejor valor para p y cual sería el mejor tipo de asignación a utilizar. Se muestran solo los resultados para entradas de tamaño cuarenta, se han probado entradas de menor tamaño y se ha observado una tendencia similar.

Podemos observar cierta tendencia de los resultados de la Tabla 7.7. Este modelo ajusta mejor que el anterior:

- Las predicciones no las hace generales para cada tamaño de entrada, sino para cada una de las entradas que desea resolver.
- Al igual que ocurría en el modelo anterior, para casos extremos el modelo no predice bien (In_40_3). Generalmente para estos casos no merece la pena aplicar paralelismo. En estos casos extremos el tiempo de ejecución es muy pequeño, por lo que el tiempo que nos cuesta el fallo es inapreciable.

Entrada	l	p	$TM_{Ciclico}$	$TR_{Ciclico}$	$TM_{Bloques}$	$TR_{Bloques}$
In_40_1	10	1	245.72	36.24	245.72	36.24
	10	2	142.26	25.23	145.19	27.31
	10	3	**82.75**	**12.27**	112.69	25.25
	10	4	87.18	18.73	**84.44**	**21.85**
In_40_2	4	1	28.04	0.129	28.04	0.1281
	4	2	14.91	0.105	15.20	0.1095
	4	3	9.66	**0.063**	10.83	0.1087
	4	4	**7.89**	0.090	**7.85**	**0.0960**
In_40_3	6	1	0.1481	0.00014	0.1481	**0.00015**
	6	2	0.0740	0.00039	0.0740	0.00032
	6	3	0.0529	**0.00041**	**0.0529**	0.00042
	6	4	**0.0423**	0.00046	0.4234	0.00054
In_40_4	19	1	15.75	191.67	15.75	191.74
	19	2	11.86	130.87	11.96	129.45
	19	3	**6.24**	**65.24**	10.72	109.42
	19	4	6.74	67.34	**9.18**	**90.46**
In_40_5	18	1	1119.67	342.52	1119.67	342.25
	18	2	671.070	234.92	655.25	228.52
	18	3	400.613	137.13	467.90	192.34
	18	4	**336.28**	**121.45**	**356.46**	**151.33**

Tabla 7.7. Comparativa entre los tiempos del modelo y los tiempos de ejecución reales para l fijo y variando el parámetro p y el tipo de asignación. (Tiempo en segundos)

- Es capaz de discriminar correctamente que tipo de asignación es la más apropiada para resolver el problema.

- El modelo toma decisiones muy acertadas utilizando únicamente información de la ejecución. No hace falta utilizar información de la instalación para obtener buenas estimaciones. La alta variación de las entradas hace que sea muy difícil registrar información en la instalación válida para todas las entradas.

Estudio del comportamiento del parámetro l para p fijo

Vamos a repetir el experimento para ver como se comporta este modelo para predecir los tiempos de ejecución para diferentes valores de l. Al igual que en los otros ejemplos hemos realizado pruebas con diferentes tamaños pero solo vamos a mostrar los datos más significativos. En la Tabla 7.8 tenemos los resultados de esta comparación. Se representan los tiempos estimados, TM, frente a los tiempos reales para las entradas uno, dos y cuatro (para no sobrecargar eliminamos la entrada que tarda menos tiempo en ejecutarse y la que más).

El modelo no se comporta bien completamente para predecir el mejor valor de l. Pese a aproximar mejor que el modelo estático, para tiempos similares no estima de forma correcta. El

Esquema 2						
l	***TM***	$In40_1$	***TM***	$In40_2$	***TM***	$In40_4$
1	80.07	27.33	321.45	0.112	169.491	128.24
2	**43.07**	21.23	210.43	0.095	88.944	90.793
3	58.96	19.30	168.28	0.092	90.630	93.200
4	62.02	18.18	**162.43**	0.090	64.361	93.341
5	76.90	19.60	187.10	0.089	68.649	95.581
6	73.47	19.19	193.20	0.090	48.841	88.544
7	81.83	18.53	202.40	0.093	48.306	81.341
8	75.47	18.09	213.29	0.082	36.399	81.323
9	81.56	18.33	222.89	0.085	36.703	88.450
10	82.54	19.63	237.26	0.076	30.239	88.622
11	87.18	18.78	254.47	**0.069**	29.725	81.448
12	81.45	19.34	261.15	0.071	22.216	86.203
13	85.83	17.24	281.42	0.070	23.992	95.635
14	81.20	17.63	280.81	0.075	15.957	81.649
15	84.17	15.13	288.97	0.081	15.213	82.674
16	77.77	13.86	282.47	0.093	12.079	80.256
17	85.63	**13.18**	310.38	0.100	10.491	69.701
18	79.04	14.90	318.04	0.117	8.246	71.441
19	83.20	14.38	336.28	0.149	7.802	74.040
20	76.44	15.85	343.32	0.194	**6.742**	**67.332**
21	81.61	17.17	361.25	0.254	7.718	68.539

Tabla 7.8. Tiempos de ejecución del modelo (TM) y reales ($In40_x$) para tres entradas diferentes de tamaño 40. Se mantiene p fijo en cuatro con asignación cíclica de tareas. (Tiempo en segundos)

motivo es que las estimaciones de los *backtracking* secundarios no llegan a ser exactas. Realizando pruebas más significativas existen dos motivos por los que el modelo no termina de estimar bien:

- Al igual que en otras ocasiones, los valores de k estimados no representan su valor real de ejecución. Esto introduce mucho ruido en las estimaciones de los *backtracking* secundarios, lo que repercute negativamente en el tiempo total estimado.
- El modelo estima los *backtracking* secundarios como si fuesen completamente independientes. Si analizamos el esquema, estos *backtracking* secundarios no son totalmente independientes. Serán independientes todos los que no sean resueltos por el mismo procesador, en caso contrario habrá dependencias interesantes de poder representar en el modelo.

Estudio general de la rutina de autoconfiguración

Vamos a estudiar como funcionaría la rutina al elegir todos los parámetros disponibles sin fijar ninguno. En la Tabla 7.9 tenemos los tiempos que se consiguen con la configuración de parámetros que la rutina nos ofrece elegir (TR). Para evaluar la calidad de la configuración propuesta por el modelo, vamos a definir tres tipos de usuarios diferentes:

- **Usuario Óptimo** (*UserOpt*): es un caso hipotético donde se conociesen los parámetros que nos llevan a la ejecución óptima para cualquiera de las entradas. Los tiempos asignados a este usuario corresponden con los mejores tiempos experimentales que hemos encontrado en nuestras pruebas, sin utilizar la asignación voraz.

- **Usuario Medio** (*UserMed*): es un usuario con conocimientos medios de paralelismo. Elegirá como parámetros de configuración unos parámetros razonables, pero no tienen por que ser buenos. En nuestro caso este usuario elige un valor para l=5 porque se han generado suficientes subtrabajos como para distribuirlos, elige p=4 para intentar aprovechar toda la potencia de cálculo de la que dispone y decide utilizar una distribución cíclica.

- **Usuario Bajo** (*UserLow*): es un usuario sin conocimiento alguno de paralelismo. Elegirá valores al azar, por lo que sus decisiones pueden no ser acertadas. Para el caso concreto de nuestro ejemplo vamos a suponer que en una elección aleatoria configuró los valores: l=10, p=2 y una asignación por bloques.

	l	***p***	***A***	***TR***	***UserOpt***	***UserMed***	***UserLow***
In_40_1	2	4	*c*	21.26	**13.18**	21.26	27.27
In_40_2	2	4	*c*	0.095	**0.07057**	0.0949	0.1116
In_40_3	1	1	*c*	0.00031	**0.00029**	0.00043	0.00037
In_40_4	18	4	*v*	**49.13**	67.37	90.78	128.326
In_40_5	4	4	*v*	**117.56**	121.54	150.93	228.02

Tabla 7.9. Tiempos de ejecución con los parámetros seleccionados por el modelo, comparados con los tiempos que conseguirían tres roles de usuarios.

Se muestra el estudio realizado para las entradas de tamaño 40 estudiadas hasta el momento. Las conclusiones que se obtienen de la Tabla 7.9 son:

- Pese a no conseguir los tiempos óptimos con los parámetros inferidos por el modelo, la elección mejora los tiempos que conseguirían usuarios que configuren manualmente este esquema paralelo.

- El modelo detecta en problemas donde no se puede aprovechar el paralelismo por ser caos extremos (entradas dos y tres) que lo mejor es no utilizar un l mayor que dos. En la entrada tres (búsqueda directa a la solución) predice bien que no merece la pena utilizar más de un procesador.

- Para entradas con recorridos más pesados (entradas cuatro y cinco) utilizar el algoritmo voraz para realizar la asignación de tareas obtiene resultados mejores que los que obtendríamos utilizando asignaciones básicas (cíclicas y bloques). Para este tipo de entradas la configuración ofrecida por el modelo es la óptima.

- En entradas cuyo tiempo de ejecución es mayor que unos cuantos segundos, aplicar paralelismo sí es rentable. En estos casos elegir mal los parámetros acarrea un porcentaje de error mayor que el de los casos extremos. El cociente medio entre la elección del *UserLow* y el tiempo del modelo, para las tres primeras entradas, es de 1.21 y el tiempo adicional

medio que se emplea en esta mala elección es de 2.008 segundos. Para entradas con recorridos más pesados los valores serían: cociente de error 2.27 y el tiempo adicional medio de una mala elección 94.82 segundos. Para las entradas estudiadas, cuando resolvemos problemas con tiempo de ejecución elevado merece la pena utilizar nuestras rutinas de decisión.

- Si repetimos los cálculos de cocientes de error para el usuario con conocimientos medios de paralelismo (*UserMed*) obtenemos que: el cociente de error medio de *UserMed* respecto a nuestro modelo es de 1.27, y el tiempo adicional medio sería de 15 segundos. En las entradas que consumen más tiempo de ejecución merece la pena utilizar el modelo a otras configuraciones manuales razonables.

- Por la tendencia observada para las entradas que hemos probado, parece que para tamaños mayores de entrada, el modelo va a elegir parámetros óptimos o cercanos al óptimo.

Aunque no se muestran resultados del esquema cinco, los resultados obtenidos son similares a lo que se obtienen para entradas extremas del esquema dos. El modelo deduce que no merece la pena aplicar paralelismo y estima valores de l de uno o dos.

Influencia del tiempo de decisión respecto al tiempo total de ejecución

A diferencia del esquema simple, en este esquema la decisión del modelo se basa directamente en la información de la ejecución y consumirá parte de su tiempo. Como se comentó en la introducción nos interesa que este tiempo de toma de decisión sea lo menor posible. Vamos a estudiar en este apartado como influye este tiempo en nuestro problema.

De las experiencias adquiridas en el estudio del esquema simple, se observó la tendencia de que para valores de l mayores que la mitad del esquema global el tiempo de ejecución se disparaba al comenzar a penalizar la presencia del *backtracking* inicial. Para este modelo debemos resolver para cada uno de los l que queramos estimar el *backtracking* inicial. Por este motivo se deberá saber detectar cuando el tiempo de resolverlo superará el tiempo de ejecución final conseguido. En este caso, la solución adoptada es realizar solamente la estimación para valores de l menores o iguales que la mitad del tamaño del problema; para valores mayores de l el tiempo de decisión no compensaría con la reducción de tiempo de ejecución conseguida por el modelo.

En la Tabla 7.10 se muestran los tiempos dedicados a la estimación del modelo para cada una de las entradas de tamaño cuarenta resueltas anteriormente. Al igual que en otros apartados la tendencia observada para resolver otro tipo de entradas es similar. Se representa el tiempo real conseguido con los parámetros del modelo (TR) y el tiempo que hemos dedicado a la decisión (TD).

Considerando este tiempo de decisión como parte del tiempo de ejecución, los 15 segundos de tiempo adicional medio que teníamos al comparar el modelo con la configuración del *UserMed* bajarían a 7.6 segundos. Será razonable utilizar nuestra metodología para problemas donde el tiempo de ejecución sea elevado. En caso de problemas que se resuelvan en poco tiempo de ejecución, cualquier intento de toma de decisión penalizará más que beneficiará. Si comparamos los tiempos de penalización del modelo en los fallos frente a los aciertos (comparándolos con la

	$In40_1$	$In40_2$	$In40_3$	$In40_4$	$In40_5$
TR	21.26	0.095	0.00031	49.13	117.56
TD	8.11	0.9058	0.000538	5.37	23.41

Tabla 7.10. Tiempos de decisión frente a los tiempos de ejecución estimados. (Tiempo en segundos)

decisión del *UserMed*) tenemos que: el tiempo adicional del modelo frente al conseguido por el *UserMed* es de 3 segundos en los casos donde se falla, frente a los 23.12 segundos de beneficio cuando el modelo toma la decisión acertada.

Se puede concluir que sería interesante aplicar paralelismo en la rutina de toma de decisión para reducir todo lo posible su tiempo. El beneficio conseguido por nuestra metodología es mayor para problemas con tiempo de ejecución elevado, y donde aplicar paralelismo supone un descenso significativo en el tiempo de ejecución.

7.2 Esquema *EMEAE+I* (Memoria Distribuida)

Ahora vamos a implementar un esquema maestro esclavo con asignación estática de memorias pero vamos a añadir el intercambio de información. Cada número de nodos generados e, los procesos esclavos van a intercambiar la información del óptimo local que hayan encontrado. De esta forma evitaremos recorrer parte del espacio de búsqueda que sin intercambio de información era trabajo de más.

La implementación la realizaremos para un entorno de memoria distribuida. Se implementarán solamente las asignaciones básicas estudiadas en el Capítulo 6. Pretendemos con esta implementación estudiar lo que pasaría en entornos donde el coste de las comunicaciones es un factor a tener en cuenta, y el número de procesadores es mayor. Al igual que en el esquema secuencial utilizaremos para programarlo C++. Utilizaremos como estándar de paso de mensajes *MPI* [22].

La ejecución se llevará a cabo en la máquina del Grupo de Computación Científica de la Universidad de Murcia (Línea de Computación Paralela): SOL. El cluster está compuesto por 16 núcleos con procesadores Intel Xeon 3.00GHz de 64 bits, con 1.5 GB de RAM. Los núcleos se reparten de la siguiente manera: 4 núcleos en sol, nodo1 y nodo2 y 2 núcleos en nodo3 y nodo4.

7.2.1 Implementación del esquema con *MPI*

En escenarios de memoria distribuida no todos los hilos se ejecutan sobre un mismo chip que comparte memoria. El nuevo procedimiento es enviar copias de nuestro programa a todas las máquinas distribuidas donde se vaya a ejecutar el código y estas ejecutarán el código de forma independiente en cada una de ellas. Deberemos programar directamente en el código que procesos se van a encargar de ejecutar cada fragmento de código y las comunicaciones que se van a producir entre ellos. Puesto que no se comparte memoria deberemos programar los

intercambios de información que se producen entre ellos.

Lo primero que deberemos hacer en nuestros programas será inicializar el "entorno" distribuido, para ello utilizaremos las directivas del Código 7.3. La variable "name" almacenará que "id_Proceso" es el que va a ejecutar el resto de este código y podremos programar que debe hacer cada uno. El número de procesos que se están ejecutando los tendremos almacenado en la variable "numProc".

```
1  ... ... ... ... ...
2  MPI_Status status;
3  MPI_Init(&argc, &argv);
4  MPI_Comm_rank(MPI_COMM_WORLD, & name);
5  MPI_Comm_size(MPI_COMM_WORLD, & numProc);
6  ... ... ... ... ...
```

Código 7.3. Inicialización del entorno *MPI*.

Una vez inicializado el entorno, en el programa principal deberemos de diferenciar la labor de los procesos esclavos de la de los procesos maestros. Comparando con la variable "name" sabremos qué procesador es el que está ejecutando el programa. Si la variable "name" tiene valor 0 será el proceso maestro, en caso contrario será un proceso esclavo.

El Código 7.4 es la implementación del comportamiento del proceso maestro. El maestro estará recibiendo mensajes mientras queden procesos esclavos ejecutándose. Los procesos esclavos podrán enviar al maestro los valores locales de la mejor solución encontrada ("VoaParcial") en caso de mejorar la solución global que tienen registrada la almacenará. En caso de recibir la solicitud de actualizar el "VOA" por parte de un proceso esclavo ("actualizaVoa"), le enviará el valor global de la mejor solución encontrada por todos ellos. Por último los procesos esclavos le notificarán cuando finalicen sus ejecuciones utilizando un mensaje con "finTotal" activo.

```
1  ... ... ... ... ...
2  if ( name == 0 ) {
3     int procFuente;
4     double VoaParcial = -1;
5     int actualizaVoa = 0;
6     int finTotal = 0;
7     int procesosFinalizados = 0;
8
9     while ( procesosFinalizados < numProc ) {
10       recibirMensajeDelEsclavo
11          ( &procFuente, &VoaParcial, &actualizaVoa, &finTotal );
12
13       if ( ( VoaParcial != -1 ) & ( VOA < VoaParcial ) ) VOA = VoaParcial;
14
15       if ( actualizaVoa ) actualizaVoaAlEsclavo ( procFuente, VOA );
16
17       if ( finTotal ) procesosFinalizados++;
18    }
19 }
20 ... ... ... ... ...
```

Código 7.4. Implementación en *MPI* del proceso maestro.

El Código 7.5 es la implementación del comportamiento de los procesos esclavos. Al igual que en la solución de memoria compartida, los procesos esclavos recorrerán el espacio array de asignaciones resolviendo aquellas que le hayan sido asignadas. A la función de *backtracking* le pasará como parámetro el valor de e para que se intercambien los valores óptimos locales

con el proceso maestro. Cuando un proceso esclavo finaliza todas las tareas le envía un mensaje al proceso maestro indicándole que ha finalizado todos los trabajos (" FinTrabajos" igual a uno).

```
1   ... ... ... ... ...
2   if ( name == 0 ) {
3      ... ... ... ... ...
4   }
5   else {
6      int VOA_privado = -1;
7
8      for ( int j = 0 ; j < tamBolsa; j++ ) {
9         if ( arrayAsignaciones[j] == name ) {
10           struct nodoBacktraking * nodo;
11           nodo = getNodo(list , j);
12
13           int solP = m01back( nodo->M, nodo->n, nodo->b,
14                        nodo->p, nodo->s, n, nodo->ba, nodo->pa,
15                        VOA_privado , ... , e);
16
17           if (( solP > VOA_privado ) & (solP != -1 ) ) VOA_privado = solP;
18
19           free(nodo->b); free(nodo->p); free(nodo->s); free(nodo);
20        }
21     }
22
23     enviarMensajeAlMaestro ( /*Destino*/ 0, /*Fuente*/ name,
24              /*Voa*/ VOA_privado , /*ActualizaVoa*/ 0, /*FinTrabajos*/ 1 );
25  }
26  ... ... ... ... ...
```

Código 7.5. Implementación en *MPI* de un proceso esclavo.

Dentro de la función "m01back" añadiremos código que controle que cada vez que se recorran e nodos se envíe el valor óptimo local encontrado al proceso maestro, y que lo actualice con el óptimo global que se ha encontrado (Código 7.6).

```
1   ... ... ... ... ...
2   double m01back ( ... ... , voa, e ) {
3      ... ... ...
4      int contNodos = 0;
5      double VOA_local = voa;
6      ... ... ...
7
8      if  ( ( e != -1 ) & ( contador == e ) ) {
9         enviarMensajeAlMaestro ( /*Destino*/ 0, /*Fuente*/ name,
10                /*Voa*/ VOA_local , /*ActualizaVoa*/ 1, /*FinTrabajos*/ 0 );
11        actualizaVoaDelMaestro ( &VOA_local );
12        contador = 0;
13     }
14     ... ... ...
15  }
16  ... ... ... ... ...
```

Código 7.6. Fragmento de código donde se implementa el intercambio de información.

Como hemos visto en el Código 7.4 y Código 7.5, existen funciones que realizan las comunicaciones entre los diferentes procesos en ejecución: "enviarMensajeAlMaestro", "recibirMensajeDelEsclavo", "actualizaVoaAlEsclavo" y "actualizaVoaDelMaestro". Utilizaremos las primitivas **"MPI_Send"**, **"MPI_Recv"**. Utilizaremos las primitivas **"MPI_Pack"**, **"MPI_Unpack"** para empaquetar y desempaquetar datos, de manera que consuman menos tiempo de inicialización de las comunicaciones al enviar varios datos a la vez que si lo hacemos individualmente.

A modo de ejemplo mostraremos, en el Código 7.7, la función donde un procesador esclavo envía información al proceso maestro. Se reserva un array del tamaño del mensaje que deseamos enviar, se empaqueta la información utilizando **"MPI_Pack"** y finalmente se envía utilizando **"MPI_Send"**. Los parámetros más significativas de **"MPI_Send'** son: el primer parámetro indique la posición de memoria a partir de la que se va a comenzar a enviar, el segundo parámetro el número de datos del tipo definido en el tercer parámetro, el cuarto indica el procesador destino. Los últimos parámetros se pueden consultar en la documentación. La función de recibir los datos es similar pero utilizando las directivas **"MPI_Unpack"** y **"MPI_Recv"**.

```
1
2   void enviarMensajeAlMaestro ( int destino, int fuente, double Voa,
3                                 int actualizaVoa, int finTrabajos ) {
4      int tamReserva = 0;
5      tamReserva = tamReserva + ( sizeof(int));
6      tamReserva = tamReserva + ( sizeof(double));
7      tamReserva = tamReserva + ( sizeof(int));
8      tamReserva = tamReserva + ( sizeof(int));
9
10     int pos = 0;
11     char * buffer = (char * ) malloc ( tamReserva );
12
13     MPI_Pack( &fuente, 1, MPI_INT, buffer, tamReserva,
14               &pos, MPI_COMM_WORLD);
15     MPI_Pack( &Voa, 1, MPI_DOUBLE, buffer, tamReserva,
16               &pos, MPI_COMM_WORLD);
17     MPI_Pack( &actualizaVoa, 1, MPI_INT, buffer, tamReserva,
18               &pos, MPI_COMM_WORLD);
19         MPI_Pack( &finTrabajos, 1, MPI_INT, buffer, tamReserva,
20                   &pos, MPI_COMM_WORLD);
21
22     MPI_Send( buffer, tamReserva, MPI_PACKED, destino, 0, MPI_COMM_WORLD);
23
24     free(buffer);
25  }
```

Código 7.7. Función de comunicación implementada en *MPI*.

7.2.2 Estudio experimental

Al igual que en el esquema anterior, adaptaremos la metodología general del capítulo seis a este esquema. Durante la fase de instalación se resolverían problemas paralelos variando los parámetros del algoritmo (l,p,e). Durante la fase de ejecución se obtendrán los parámetros que consiguieron menores tiempos de ejecución medios entre todas las pruebas de la instalación. A partir de esta configuración se realizará una búsqueda local alrededor de ella resolviendo un problema reducido de la entrada. El modelo propondrá como configuración para los parámetros del algoritmo la mejor estimada tras la búsqueda local.

El *manager* deberá configurar los parámetros para representar la máquina donde se va a instalar la rutina. Por ejemplo, para nuestro estudio experimental, el parámetro p podrá variar entre 1 y 16 (porque disponemos de 16 núcleos), l irá en función del tamaño del problema y en principio e podrá tomar cualquier valor (siempre menor que el total de nodos generados). Si no disponemos de conocimientos previos, no sabemos que valores serían apropiados para configurar el parámetro e. A partir de los datos experimentales se propone al *manager* que configure valores de e menores que 100000 nodos. En nuestro experimento configuraremos la rutina de instalación

con un rango amplio de valores de e para estudiar su influencia sobre el tiempo de ejecución.

Al igual que en las secciones anteriores vamos a estudiar el comportamiento para entradas de tamaño cuarenta. Configuraremos la instalación para generar una batería de 10 entradas aleatorias para cada tamaño, y este variará entre 5 y 40 incrementándolo de 5 en 5. Los parámetros del generador de entradas son los mismos que los utilizados en el capítulo 4. Para cada entrada, la rutina de instalación realizará la ejecución configurando como parámetros del algoritmo las combinaciones de los que el *manager* ha utilizado para describir la máquina. En nuestro experimento, el parámetro p variará entre 1 y 16, l entre 1 y 25, y e tomará los valores 100, 500, 1000, 1500, 5000, 10000, 1000000. Si el *manager* dispone de información previa seleccionará los valores de estos parámetros para reducir el tiempo de ejecución.

A partir de los resultados obtenidos durante la instalación obtenemos que para entradas de tamaño 40, los parámetros que mejores tiempos medios de ejecución han obtenido son (l,p,e)=(13,12,10000).

Al igual que en el caso de memoria compartida vamos a comparar los resultados del modelo con los que obtendrían tres tipos de usuarios diferentes. Para este tipo de esquemas, el rol de un usuario medio sería complicado de encontrar. Para los parámetros del algoritmo identificados es difícil que se sepa dar con una combinación razonable; por este motivo vamos a sustituir el usuario medio por lo que denominamos usuario de la instalación. Este nuevo usuario configurará como parámetros directamente los que se obtienen a partir de los datos de la instalación, sin hacer la búsqueda local. La configuración de los parámetros que adopta cada uno de ellos es:

- **Usuario Óptimo** (*UserOpt*): los tiempos asignados a este usuario corresponden con los mejores tiempos experimentales que hemos encontrado durante nuestras pruebas.
- **Usuario Instalación** (*UserInst*): este usuario utilizará como parámetros los que han obtenido un mejor tiempo de ejecución durante la fase de instalación. En nuestro caso particular elegirá l=13, p=12 y e=10000.
- **Usuario Bajo** (*UserLow*): es un usuario sin conocimiento alguno de paralelismo. Elegirá valores al azar, por lo que sus decisiones pueden no ser acertadas. Para el caso concreto de nuestro ejemplo vamos a suponer que en una elección aleatoria configuró los valores: l=10, p=5 y e=100.

En la Tabla 7.11 tenemos los valores de tiempo de ejecución obtenidos. Se muestran para cuatro entradas diferentes de tamaño 40 la configuración que nos propone el modelo y el tiempo de ejecución obtenido para ella. Se completa la tabla con los tiempos de ejecución obtenidos por parte de los tres tipos de usuarios. En la Tabla 7.12 se comparan los cocientes de error medios de todas las ejecuciones respecto a los tiempos de ejecución óptimos. TMA representa el tiempo medio adicional que nos supone la elección respecto al que obtenemos con el óptimo.

Las conclusiones que se pueden extraer de este experimento son:

- Todas las decisiones que utilizan estimaciones para configurar los valores de los parámetros del algoritmo se quedan alejadas de los tiempos de ejecución óptimos.

	l	***p***	***e***	***TR***	***UserOpt***	***UserInst***	***UserLow***
In_40_1	12	14	1000	29.40	7.37	11.64	170.57
In_40_2	13	10	1000	0.135	0.030	0.033	0.4804
In_40_3	13	12	1000	0.00031	0.0002	0.0002	0.000234
In_40_4	16	16	1000	90.67	24.65	54.37	1233.31

Tabla 7.11. Tiempos de ejecución con los parámetros seleccionados por el modelo, comparados con los tiempos que conseguirían tres roles de usuarios. (Tiempo en segundos)

	Modelo	***UserInst***	***UserLow***
T_{Ejec} / $T_{EjecOpt}$	3.38	1.45	22.5
TMA	21.8	8.4	342.8

Tabla 7.12. Cociente de error medio y tiempos adicionales de las distintas configuraciones. (TMA medido en segundos)

- Los valores obtenidos por el *UserInst* se acercan más al óptimo que los propuestos por el modelo tras la búsqueda local. Realizar una búsqueda local a partir de la información de la instalación nos lleva a obtener peores resultados que si no la realizamos. Introducir información de la ejecución obtienen peores valores.

- Una mala elección de los parámetros (*UserLow*) podría llevarnos a tiempos de ejecución inaceptables, y podrían ser incluso mayores que las ejecuciones del algoritmo secuencial. Por este motivo es interesante utilizar nuestro modelo de autooptimización cuando paralelizamos la técnica. Fíjense que el cociente de error medio respecto al óptimo es de 3.38 frente al 22.5 que obtendría el *UserLow*.

Entre las conclusiones del experimento, nos llama la atención que al realizar la búsqueda local obtengamos peores configuraciones de los parámetros. En principio, esta búsqueda local utiliza información de la propia entrada (una entrada reducida agrupando valores) para ver como influyen los parámetros para esos valores de entrada. Lo que estamos haciendo es generar una entrada reducida a partir de la original y realizar ejecuciones paralelas variando cada uno de los parámetros por separado. A partir de los datos experimentales nos hemos dado cuenta de que al realizar intercambio de información los parámetro l y p adquieren una menor influencia en el tiempo total de ejecución cuando elegimos valores erróneos de e. Si estudiamos más detenidamente e, este parámetro es función del número total de nodos recorridos. Por ejemplo, supongamos un problema de tamaño 20 y un problema de tamaño 40; para un mismo valor de e, en el problema de tamaño 20 el coste acarreado de comunicaciones será menor que el que acarrearía en el de 40 siempre y cuando las comunicaciones no supongan un aumento significativo de las podas. Por este motivo añadir información de ejecución nos está llevando a valores incorrectos, al resolver entradas reducidas de tamaño 20, detecta que para un valor de e=1000 el tiempo se reduce (porque le restaría el tiempo de sobrecarga); sin embargo cuando resolvemos el mismo problema para tamaño 40 lo que estamos haciendo es incluir más sobrecarga que con el valor inicial de 10000 no se producía.

Se propone mejorar la búsqueda local para obtener valores más realistas, y que no cometan

el error comentado en el párrafo anterior. De los experimentos realizados en la instalación, podríamos estimar una tendencia del valor de e a medida que aumentamos el tamaño de la entrada, podríamos aproximar esa tendencia ajustando a una función o extrapolando los valores. La búsqueda local la realizaríamos para el tamaño reducido de la entrada. Una vez finalizada la búsqueda local y obtenida la configuración de los parámetros, podríamos extrapolar sus valores al tamaño real.

Cabe destacar que los resultados y las conclusiones obtenidas a lo largo de este capítulo son para los bancos de prueba que hemos configurado. Para este tipo de problemas los tamaños de los bancos de pruebas considerados pueden resultar escasos. Sería interesante repetir los experimentos aumentando el tamaño de todas las pruebas para ver si continúan teniendo la misma tendencia.

Capítulo 8

Conclusiones y trabajos futuros

El modelado de recorridos de árboles utilizando la técnica de *backtracking*, cuando se realizan podas en el espacio de búsqueda, ha presentado problemas causados por una alta dependencia de las entradas. Para un problema, dos entradas diferentes del mismo tamaño pueden generar comportamientos completamente distintos. Utilizando la metodología propuesta, no se consigue estimar de forma exacta los tiempo de ejecución, pero sí logramos aproximarnos al comportamiento que tendrá la rutina según la entrada. Detectar el comportamiento exacto, se complica al no poder dedicar mucho tiempo en la ejecución a la estimación de los parámetros. Utilizaremos la metodología para implementar rutinas capaces de seleccionar, entre diferentes esquemas, aquel que nos lleve a un menor tiempo de ejecución.

Vamos a resumir los principales resultados que se han conseguido para la técnica secuencial:

- Elaboración de una metodología de trabajo para estimar recorridos secuenciales de *backtracking*.
- Identificación de parámetros que influyen en el tiempo de ejecución para un esquema de optimización, y construcción de un modelo teórico que nos permite estimar el comportamiento para nuevas entradas.
- Implementación de cinco esquemas secuenciales para resolver el problema de la mochila 0/1 por medio de *backtracking*. Por último se han implementado las rutinas que permiten seleccionar entre ellos aquel que dedique menos tiempo de ejecución para resolver una entrada concreta.

En las soluciones paralelas de este tipo de técnicas, aparecen nuevos parámetros del algoritmo que hay que tener en cuenta si queremos aprovechar el paralelismo. Elegir valores inadecuados de estos parámetros, puede llevar a las técnicas paralelas a tardar incluso más tiempo de ejecución que las rutinas secuenciales. Un usuario sin conocimientos avanzados de paralelismo tendría complicado dar con una configuración adecuada. Modelar estos recorridos resulta complejo por el número de parámetros que aparecen. Además las dependencias que existen entre todos ellos y las entradas, dificultan la tarea de conocer de antemano su comportamiento. Con la metodología propuesta y las rutinas implementadas nos acercamos a configuraciones que, pese a no ser las óptimas, se acercan a ellas y nos permiten aprovechar el paralelismo. Los parámetros propuestos por el modelo alejan a los usuarios de las peores elecciones.

Al igual que en el caso secuencial, vamos a resumir los resultados obtenidos para las técnicas paralelas:

- Elaboración de una metodología general de trabajo para la estimación de los parámetros que aparecen en esquemas paralelos de *backtracking*.
- Estudio y modelado de dos esquemas paralelos. Implementación de rutinas que resuelven el problema de la mochila 0/1 con un esquema del tipo *EMEAE* (en memoria compartida) y otro del tipo *EAMEAE+I* (en memoria distribuida).
- Finalmente, se han implementado diferentes adaptaciones de la metodología general paralela, para inferir los valores configurables de los parámetros que aparecen en estos dos esquemas.

A partir de las conclusiones obtenidas a lo largo de este proyecto, se proponen diferentes líneas para continuar en el futuro:

- En una primera fase, se ha estudiado el esquema implementado para los sistemas de memoria compartida (*EMEAE*); elegimos esta opción por la expansión de este tipo de sistemas, debida a la utilización de *multicores* en los ordenadores personales. Una vez finalizado el estudio en memoria compartida, se pasó a modelar un nuevo esquema paralelo (*EMEAE+I*) que se ha implementado en memoria distribuida. A partir de las conclusiones obtenidas de los experimentos para este último esquema, sería interesante continuar su estudio para acercar las estimaciones a los parámetros óptimos. También se podría extender el proyecto estudiando nuevos esquemas paralelos.
- A lo largo del proyecto se han propuesto diferentes métodos para estimar los parámetros identificados en los modelos. Se ha utilizado tanto información de la instalación como de la ejecución. Se ha trabajado sobre diferentes alternativas, estudiando la influencia real que tendrían en la estimación del tiempo de ejecución. Sería interesante continuar este trabajo refinando dichas técnicas para lograr mejores aproximaciones a los tiempos de ejecución exactos.
- Para este tipo de técnicas, el tiempo de ejecución para tamaños grandes llega a ser realmente elevado. Hemos realizado los experimentos y las pruebas con entradas de tamaño inferior o igual a 40. Utilizar tamaños mayores podría llevarnos a tiempos de ejecución que no nos podemos permitir. Sería interesante continuar el proyecto aumentando el tamaño de los problemas y el número de pruebas realizadas.
- Se ha utilizado como caso de prueba la resolución del problema de la Mochila 0/1. Una continuación del proyecto sería aplicar la metodología para resolver problemas reales.
- En este proyecto se han estudiado las características que debería tener un esquema algorítmico secuencial, para paralelizarlo de manera transparente a los usuarios. Sería interesante realizar su implementación. Una vez implementado, se podría extender la herramienta web de esquemas algorítmicos con este nuevo esquema, y adaptar la metodología para que se autoconfiguren los parámetros paralelos.
- Aplicar y refinar la metodología para otras técnicas de recorrido de árboles como *Branch and bound*.

Bibliografía

[1] P. Alberti, P. Alonso, A. M. Vidal, J. Cuenca, and D. Giménez. Designing polylibraries to speed up linear algebra computations. *Int. J. High Perform. Comput. Netw.*, 1(1-3):75–84, 2004.

[2] F. Almeida, J. M. Beltrán, M. Boratto, D. Giménez, J. Cuenca, L. P. García, J. P. Martínez, and A. M. Vidal. Parametrización de esquemas algorítmicos paralelos para autooptimización. XVII Jornadas de Paralelismo, Septiembre 2006.

[3] F. Almeida, D. Giménez, J. M. Mantas, and A. M. Vidal. *Introducción a la programación paralela*. Paraninfo, 2008.

[4] J. M. Beltrán. Autooptimización en esquemas paralelos de recorrido de árboles de soluciones: Esquema del problema de la mochila 0/1. Trabajo de iniciación a la investigación, Universidad de Murcia, Julio 2006.

[5] M. Boratto. Parametrización en esquemas paralelos divide y vencerás. Trabajo de iniciación a la investigación, Universidad de Murcia, Septiembre 2006.

[6] E. A. Brewer. *Portable High-Performance Supercomputing: High-Level Platform-Dependent Optimization*. PhD thesis, MIT, 1994.

[7] E. Castillo, A. J. Conejo, P. Pedregal, R. García, and N. Alguacil. *Formulación y resolución de modelos de programación matemática en ingeniería y ciencia*. Universidad de Castilla-La Mancha, 2002.

[8] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self adapting software for numerical linear algebra and LAPACK for clusters. *Parallel Computing*, 29:1723–1743, 2003.

[9] J. Cuenca, L. P. García, D. Giménez, and J. Dongarra. Processes distribution of homogeneous parallel linear algebra routines on heterogeneous clusters. In *CLUSTER*, pages 1–10, 2005.

[10] J. Cuenca, D. Giménez, and J. González. Modeling the behaviour of linear algebra algorithms with message-passing. *EUROMICRO Workshop on Par. and Dist.*, pages 12–25, 2001.

[11] J. Cuenca, D. Giménez, and J. González. Architecture of an automatic tuned linear algebra library. *Parallel Computing*, 30:187–220, 2004.

[12] J. Cuenca, D. Giménez, J. González, J. Dongarra, and K. Roche. Automatic optimisation of parallel linear algebra routines in systems with variable load. In *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing (EUROMICRO-PDP 2003)*, pages 401–408, 2003.

[13] J. Cuenca, D. Giménez, and J. P. Martínez. Heuristics for work distribution of a homogeneous parallel dynamic programming scheme on heterogeneous systems. *Parallel Computing*, 31(7), 2005.

[14] J. Demmel, J. Dongarra, B. N. Parlett, W. Kahan, M. Gu, D. Bindel, Y. Hida, Xiaoye S. Li, O. Marques, E. Jason Riedy, C. Vömel, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, J. Langou, and S. Tomov. Prospectus for the next lapack and scalapack libraries. In *PARA*, pages 11–23, 2006.

[15] M. I. Dorta. *Esquemas paralelos para la técnica de ramificación y acotación.* PhD thesis, Universidad de La Laguna, 2004.

[16] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. pages 1381–1384. IEEE, 1998.

[17] L. P. García, J. Cuenca, and D. Giménez. Using experimental data to improve the performance modelling of parallel linear algebra routines. In *PPAM*, pages 1150–1159, 2007.

[18] T. Katagiri, K. Kise, H. Honda, and T. Yuba. Effect of auto-tuning with user's knowledge for numerical software. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, pages 12–25. ACM, 2004.

[19] J. P. Martínez. Autooptimización en esquemas paralelos de programación dinámica: esquema del problema de devolución de monedas. Trabajo de iniciación a la investigación, Universidad de Murcia, Septiembre 2003.

[20] J. P. Matínez, F. Almeida, and D. Giménez. Mapping in heterogeneous systems with heuristic methods. In *PARA*, pages 1084–1093, 2006.

[21] J. Ortega, M. Anguita, and A. Prieto. *Arquitectura de Computadores.* Thomson, 2005.

[22] P. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann, 1996.