TESIS DOCTORAL

AUTOOPTIMIZACIÓN EN ESQUEMAS ALGORÍTMICOS PARALELOS ITERATIVOS

Juan Pedro Martínez Gallar

UNIVERSIDAD DE MURCIA
DEPARTAMENTO DE INFORMÁTICA Y SISTEMAS

Diciembre 2009

Directores : Dr. Domingo Giménez Cánovas

Dr. Francisco Almeida Rodríguez

RESUMEN

Una de la razones que explican el rápido desarrollo experimentado en los sistemas informáticos de las últimas décadas, tanto en prestaciones hardware como software, es la necesidad de utilizarlos en el ámbito científico para realizar cálculos complejos y masivos. Ha sido necesario desarrollar algoritmos y librerías de códigos de muy diversa naturaleza (recursivos, iterativos, matriciales, recorrido de árboles, grafos...) que resuelven los problemas en cuestión en tiempos de ejecución razonables para problemas con alto coste computacional. El proceso de **optimización** del software ha estado presente a lo largo de este periodo mediante la modificación, adaptación y ajuste del código a las características particulares de la arquitectura destino, con el fin de realizar una gestión más eficiente de los recursos y/o de reducir los tiempos de ejecución. Cabe mencionar que este ajuste es fuertemente dependiente del software a optimizar y de la arquitectura hardware, y habitualmente no resulta una tarea sencilla. Las técnicas de optimización del software y la evolución de la capacidad computacional del hardware han permitido disponer de sistemas paralelos de altas prestaciones que facilitan la investigación científica y el cálculo aplicado.

Sin embargo, es frecuente también que los usuarios de estos sistemas, científicos e ingenieros, que muestran gran interés en obtener resultados óptimos y en un tiempo reducido de los problemas que resuelven, no dispongan de amplios conocimientos en paralelismo ni en las técnicas de optimización del código, y desean que el uso de las herramientas hardware y software no les suponga un esfuerzo suplementario a su labor investigadora. Sería deseable, por tanto, que el proceso de optimización y adaptación del software a las diferentes arquitecturas se pudiera realizar de forma transparente al usuario, o que, al menos, no supusiera finalmente un esfuerzo significativo para éste. Surge, de este modo, la necesidad y el interés en el desarrollo de software y de técnicas con capacidad de **autooptimización**. Es interesante contar con software adaptativo que sea capaz de conseguir que el código, previamente optimizado, pueda ser usado en un entorno ajeno al de su diseño original sin reducir las prestaciones iniciales. Esta capacidad de autooptimización debería incluir la adaptación a distintos sistemas y configuraciones, de manera que el código se ajuste de manera automática a los entornos en que se instala, reduciendo el coste del proceso de optimización del software sobre diferentes arquitecturas.

Situándonos en el contexto de los sistemas paralelos hay que destacar que gran

parte del software desarrollado ha sido diseñado inicialmente para su uso en sistemas de naturaleza homogénea, es decir, sistemas cuyos componentes tienen características físicas similares, y hay que realizar la adaptación hacia diferentes plataformas. Gracias a la evolución tecnológica y a la reducción constante de los precios de los componentes hardware, actualmente es usual disponer de sistemas heterogéneos compuestos de elementos con prestaciones computacionales posiblemente muy diferentes. Si se pretende la explotación de este software en entornos de naturaleza heterogénea, se hace necesario un nuevo proceso de adaptación hacia plataformas heterogéneas con el fin de que no se perjudique su optimalidad. Nuevamente, surge la necesidad de introducir mecanismos de optimización y autooptimización. La heterogeneidad introduce de manera natural un nivel de complejidad adicional al subyacente al caso homogéneo debido al rango de variabilidad de los recursos involucrados.

En este trabajo planteamos el estudio de la autooptimización en sistemas paralelos, tanto homogéneos como heterogéneos. En particular, nos centramos en una clase de algoritmos que aparece con mucha frecuencia en la resolución de problemas académicos y científicos, los **algoritmos iterativos**. Se caracterizan por ejecutar repetidamente un conjunto de instrucciones hasta que se cumple un criterio de convergencia. Nuestro objetivo es desarrollar técnicas que eviten (en la medida de lo posible) la intervención del usuario en el proceso de optimización del software para esta clase de algoritmos sobre el mayor conjunto posible de plataformas. Se trata de un objetivo muy ambicioso pues, en particular, permitiría evitar nuevos rediseños de software desarrollado, cuya eficacia ha sido previamente probada, para conseguir usarlo de forma eficiente en un entorno para el que no fue originalmente diseñado.

Aunque la autooptimización ha sido abordada desde diferentes perspectivas, nos concentramos en la propuesta que considera modelos matemáticos parametrizados del tiempo de ejecución de los algoritmos. Los modelos incluyen parámetros que reflejan las características del sistema y otros que pueden variarse para obtener distintas versiones del algoritmo en cuestión. La autooptimización consistirá en obtener valores de los parámetros del algoritmo con los que se consigue un tiempo de ejecución mínimo sobre la plataforma considerada. Para la adaptación de software homogéneo a un entorno heterogéneo, el modelo reasignará las tareas o procesos (que originariamente se repartían de forma equitativa entre los elementos del entorno homogéneo) sobre el entorno heterogéneo. Para realizar esta operación de minimización/asignación se pueden utilizar tanto técnicas exhaustivas de búsqueda como métodos aproximados y heurísticas. Como prueba de concepto de nuestras propuestas, las técnicas desarrolladas han sido aplicadas a la resolución de problemas de optimización mediante los esquemas iterativos de programación dinámica.

ABSTRACT

One of the reasons for the rapid development experienced in the computer systems the last decades, both in hardware and software, is the need to use them in scientific problems to perform complex and massive calculations. It has been necessary to develop algorithms and libraries of codes of diverse nature (recursive, iterative, trees, graphs...) that solve high computational cost problems in reasonable execution times. In order to achieve a more efficient management of the resources and to reduce the execution time, the **optimization** of the software has been done by modifying and adjusting the codes to the particular characteristics of the target architecture. This adjustment depends heavily on the software to optimize and the hardware architecture, and in general it is not a simple task.

The optimization techniques of the software and the evolution of the computational capacity of the hardware have allowed parallel systems of high capacities which facilitate scientific research. However, the users of these systems, scientists and engineers, are frequently interested in obtaining optimal results and in a short time, but they do not have a wide knowledge of either parallelism or code optimization techniques, and they would like the use of hardware and software tools not to suppose a supplementary effort to their research. Therefore, it would be desirable that the process of optimization and adjustment of the software to the different architectures be transparent to the user, or that, at least, its use should not suppose a significant effort. Thus, the development of software and techniques with capacity of **auto-optimization** is of interest. It is desirable to have adaptive software that should be capable of adapting previously optimized code to new environments, different to those for which it was originally designed. This capacity of auto-optimization should include the adjustment to different systems and configurations, and that the code adjusts automatically to the different environments, so reducing the cost of optimization of the software.

In the context of the parallel systems, it is necessary to emphasize that a large amount of software has been initially designed for use in homogeneous systems, that is, systems whose components have similar physical characteristics, and it is necessary to adapt this software to different platforms. Thanks to technological evolution and constant reduction of the prices of the components, today it is usual to have heterogeneous systems, which have elements whose computational capacities are possibly very

different. To use this software efficiently in heterogeneous systems, a new adaptation process is necessary. Again, it is important to introduce mechanisms of optimization and auto-optimization. The heterogeneity introduces a level of additional complexity to the homogeneous case, because more resources are involved in the process.

In this work, we present a study of auto-optimization in parallel systems, both homogeneous and heterogeneous. The study centres on **iterative algorithms**, a class of algorithms that appears frequently in the solution of academic and scientific problems. They work by repeatedly executing a set of instructions until a convergence criterion is achieved. Our goal is to develop techniques that avoid (as far as possible) the intervention of the user in the optimization process of the software for this class of algorithms on most of the platforms. It is a very ambitious task, because it would allow avoid the need to redesign previously developed software which has proved to be efficient, so using it efficiently in an environment it was not originally designed for.

Although the auto-optimization has been approached from different perspectives, we consider here mathematical parametrized models of the execution time of the algorithms. The models include parameters that reflect the characteristics of the system and others that can be changed to obtain different versions of the algorithm in question. The auto-optimization will consist of the calculation of the values of the parameters of the algorithm with which minimum execution times are obtained on the platform considered. For the adaptation of homogeneous software to a heterogeneous environment, an assignation method of the tasks or processes (that originally were equitably distributed on a homogeneous system) to the processors in the heterogeneous environment must be designed. Exhaustive search techniques, approximation methods and heuristics can be used to solve this minimization/assignment problem. Our proposal has been tested by applying it to dynamic programming problems with parallel iterative schemes.

Domingo Giménez Cánovas,

Francisco Almeida Rodríguez,

Facultad de Informática...

A mi madre:

Gracias.

Agradecimientos

Agradezco a mi director de tesis, Dr. Domingo Giménez Cánovas la guía, asesoría y la oportunidad que me dio al formar parte de su grupo de investigadores, con lo cual fue posible desarrollar el presente proyecto de tesis doctoral.

De la misma forma, también quiero hacer patente mi deuda y agradecimientos al Dr. Francisco Almeida Rodríguez, codirector de esta tesis, cuyas sugerencias e ideas enriquecieron el contenido del proyecto.

Juan Pedro Martínez Gallar

... Una de las mayores tentaciones del demonio es ponerle a un hombre en el entendimiento que puede componer e imprimir un libro.

Don Quijote de La Mancha

Índice general

Índice de figuras

Índice de Tablas

Capítulo 1

Introducción, Objetivos y Metodología

En este capítulo introductorio se presenta el problema que se pretende abordar en esta tesis situándolo en el contexto científico. En primer lugar, se introducen los conceptos de optimización y autooptimización indicando su interés y dificultad de desarrollo. Se analiza el estado del arte en relación con las propuestas y resultados obtenidos por la comunidad investigadora. A continuación, se establecen las hipótesis de investigación, los objetivos que se pretenden conseguir y la metodología de trabajo a emplear. Además, se comentan todos aquellos aspectos relacionados con la notación y la terminología que se usará a lo largo de la memoria.

1.1. Optimización de software

Los sistemas informáticos, desde su origen en los años 40 hasta la actualidad, han sido usados principalmente como una herramienta que facilita la realización de cálculos complejos para resolver problemas que difícilmente podían ser resueltos sólo con la intervención humana. La demanda por parte de la comunidad científica de sistemas con capacidad de dar respuesta a los problemas que a diario se plantean es continua. A día de hoy es frecuente encontrar sistemas paralelos de altas prestaciones con decenas de miles de elementos de cómputo y se espera que los futuros sistemas puedan contar con cientos de miles [124]. A medida que los sistemas de cómputo crecen en prestaciones se incrementa también la complejidad de su uso y explotación.

De modo simultáneo al crecimiento de los sistemas de cómputo han ido desarrollándose técnicas y métodos con los que garantizar que el software (algoritmos, códigos y librerías de códigos) hagan un uso eficiente de los mismos. Este conjunto de técnicas ha permitido optimizar el software en la plataforma destino de acuerdo al criterio de

interés en cada caso. Durante el proceso de **optimización**, el software es modificado o adaptado a la arquitectura destino para conseguir una explotación eficiente de los recursos. En ocasiones se trata de optimizar un recurso de forma individual como puede ser la memoria, o la distribución de los procesadores del sistema o el tiempo de ejecución del programa. Desde el punto de vista de la tecnología empleada, en la práctica es posible hablar de diferentes niveles de optimización: optimización a nivel de diseño, a nivel de código fuente, a nivel de compilación, a nivel de ensamblaje, a nivel de tiempo de ejecución, etc.

La gran evolución producida en la optimización de software se ha debido principalmente a la necesidad de obtener soluciones aceptables a los retos científicos en tiempos de ejecución asumibles por los sistemas informáticos disponibles en cada periodo histórico. Como consecuencia de esto, desde hace décadas han venido desarrollándose librerías de software de cálculo optimizadas para el uso en ingeniería e investigación, tales como BLAS [41], LAPACK [10], ScaLAPACK [21]... Se trata de grandes colecciones de funciones matemáticas de muy diversos ámbitos, que han sido diseñadas y probadas con el fin de emplear tiempos de ejecución reducidos de los algoritmos que implementan sobre las plataformas en que se ejecutan.

Las técnicas de optimización han mostrado una importante evolución a lo largo de décadas y su aplicación práctica ha sido enormemente útil. Gracias a las técnicas de optimización de software y a la capacidad computacional del hardware, hoy día es usual disponer de sistemas de altas prestaciones que facilitan la investigación científica.

El desarrollo de esta tesis se situará en el marco de la optimización a nivel de tiempo de ejecución y en menor medida en el nivel de diseño y de código fuente. Estamos particularmente interesados en desarrollar técnicas que permitan adaptar/ajustar el software a la arquitectura destino con el fin de reducir sus tiempos de ejecución y esperamos que este proceso de adaptación y ajuste pueda ser realizado de forma automática.

1.2. Autooptimización de software

Tradicionalmente, la optimización del software ha venido realizándose de forma manual, esto es, los programadores han sido responsables de analizar el código y de aplicarle diferentes técnicas para optimizarlo. La tarea no siempre es sencilla debido a la fuerte dependencia que hay entre el tipo de optimización a aplicar, el algoritmo y la arquitectura. Los usuarios de los sistemas paralelos, suelen ser científicos que conocen específicamente su ámbito de trabajo pero no tienen grandes conocimientos ni interés

en los aspectos relacionados con los elementos hardware ni software. Estos científicos desean que los códigos que están usando puedan seguir estando optimizados con independencia de la plataforma que estén empleando y, además, están interesados en que la adaptación de las librerías que emplean, que fueron desarrolladas pensando en un determinado tipo de hardware, se haga de forma que interfiera lo menos posible en su trabajo diario y sin que ello les suponga un tiempo y un esfuerzo presupuestario suplementario. Es decir, desean que todo lo que hasta ahora funcionaba adecuadamente siga valiendo sobre los nuevos sistemas informáticos que puedan emplear. Tiene interés, por tanto, que sean los propios sistemas informáticos quienes se encarguen del proceso de optimización del software. Se trata de que el propio software sea capaz de conseguir optimizar los códigos a ejecutar mediante mecanismos de **autooptimización**, evitando en la medida de lo posible la intervención humana. El beneficio es claro, puesto que se liberaría al usuario de su intervención directa en esta tarea.

Una característica común a una gran mayoría de sistemas de altas prestaciones es su naturaleza homogénea, esto es, se trata de sistemas cuyos elementos de proceso tienen características físicas similares. La mayor parte de las librerías desarrolladas han sido orientadas y optimizadas para este tipo de sistemas homogéneos. Actualmente, gracias a la constante reducción del precio del hardware y a su evolución, es habitual que los centros de cálculo e investigación dispongan de sistemas heterogéneos en que los elementos de proceso poseen características computacionales diferentes. Dado que las librerías han sido diseñadas y optimizadas para trabajar sobre entornos homogéneos, cuando se intenta su explotación en entornos heterogéneos (fuertemente condicionados por sus características físicas) se hace necesario un proceso de adaptación adicional. El elevado número de factores y su rango de variabilidad confieren al proceso de optimización en sistemas heterogéneos un nivel de dificultad adicional a la optimización en el caso homogéneo. Para conseguir este propósito cabe la posibilidad de reescribir todo el software desarrollado, lo que conlleva un elevado coste económico y temporal, pues supondría diseñar y validar de nuevo lo anteriormente optimizado.

Nuevamente, sería ideal que este proceso de adaptación del software a los sistemas heterogéneos, se hiciera de forma transparente al usuario, es decir, que sea el propio software quien sea capaz de adaptarse, de forma automática, al nuevo entorno de trabajo. Para ello se hace necesario desarrollar software adicional que se encargue de trasladar el código original optimizado a entornos cuyos elementos presentan características computacionales y de comunicaciones muy diversas. La autooptimización debe ser considerada, por tanto, en el proceso de optimización de cualquier tipo de código y en cualquier entorno tanto homogéneo como heterogéneo.

Con ello se conseguiría cumplir uno de los elementos importantes en el ciclo de vida del software, como es la reutilización, evitando rediseñar e implantar de nuevo lo que previamente ha costado un importante esfuerzo económico y temporal en su construcción y optimización. Se trata de un objetivo muy atractivo pues conllevaría el diseño y la construcción de software adaptativo que fuera capaz de hacer reutilizable el software optimizado previamente sin mermar sus prestaciones iniciales.

1.2.1. Trabajos relacionados

El trabajo que se está desarrollando forma parte de una línea de investigación de desarrollo de **software paralelo** con capacidad de autooptimización [25], de la que existen proyectos sobre **optimización** en distintos campos: ATLAS [131], LFC [27], FFTW [49, 50], ILIB [86]. Quizás el campo donde más se ha trabajado por su gran interés en aplicaciones científicas es el del álgebra lineal [27, 31, 34, 42, 43, 85].

Se está haciendo también un enorme esfuerzo para estudiar y comprender la autooptimización de esquemas algorítmicos paralelos. En nuestro caso la investigación se concentra en el diseño de esqueletos para ser incluidos en lenguajes paralelos de alto nivel [109], y el diseño de esquemas para ser usados en sistemas paralelos [72]. En algunos casos el estudio de los parámetros óptimos para obtener tiempos de ejecución reducidos ya ha sido abordado [70].

También es posible trabajar en la autooptimización de rutinas para distintos esquemas algorítmicos [24, 30]: programación dinámica, divide y vencerás, *backtracking*, *branch and bound*...

Referente a los esquemas algorítmicos paralelos, destacar que hay trabajos precedentes donde, entre otras, se usan técnicas de evaluación temporal de los tiempos de ejecución como forma de conseguir una buena paralelización y por lo tanto una reducción de los tiempos de ejecución. Algunos de ellos, agrupados en distintas técnicas son:

- Lenguajes de programación paralelos como P3L [110]. Estos lenguajes de programación paralela se caracterizan por evaluar durante el proceso de compilación el coste computacional del código con el fin de optimizar su paralelización.
- Técnicas pipeline [9, 16, 69]. Consisten en la ejecución ordenada, continua y simultánea de diferentes partes de un algoritmos de forma que se reduzca su tiempo de ejecución.
- Técnicas maestro-esclavo [8, 15, 81]. Se trata de establecer estrategias de secuenciación de tareas que permitan una sincronización de la ejecución simultánea de

diferentes partes de un código.

- Esqueletos [71, 111, 116]. Muchas aplicaciones paralelas comparten un conjunto común de patrones de interacción. Es decir, hay partes de los algoritmos que se pueden considerar como genéricos con independencia del código que implementen. Se trataría de distinguir estos elementos comunes (esqueleto) de tal forma que la construcción de código supusiera simplemente rellenar con el código específico de cada algoritmo los huecos del esqueleto.

En esta tesis nos centramos en el estudio de **técnicas de autooptimización de esquemas paralelos iterativos**.

Estos esquemas se utilizan en la resolución de gran cantidad de problemas de ámbito científico y académico: problema de la mochila, problema de las monedas, problemas de caminos de coste mínimo entre dos nodos de un grafo dirigido, métodos de Jacobi, resolución iterativa de sistemas de ecuaciones, multiplicación de matrices [7, 13, 24]... Usamos algunos de ellos pero tratamos de orientar el estudio de forma que sea fácil su extensión o aplicación a otros problemas que usan el mismo esquema [24, 30]. Más aún, pretendemos que se puedan aplicar las técnicas desarrolladas a otros tipos de esquemas, no sólo a los iterativos.

1.2.2. Optimizando a través del modelado del tiempo de ejecución con parámetros

El análisis de la literatura en relación con las diferentes aproximaciones hacia la autooptimización de código, nos permite considerar dos grandes alternativas:

- Realización de tests de ejecuciones durante la instalación del software: se trata de realizar diferentes pruebas del software a optimizar durante su instalación en el sistema de cómputo, de forma que se pueda ajustar el software a las nuevas características físicas del sistema mediante decisiones tomadas sobre las mejores ejecuciones obtenidas. Puesto que el software en cuestión se optimiza sobre arquitecturas concretas, la aproximación es a menudo criticada por carecer de los mecanismos de generalidad necesarios para introducir en metodologías y herramientas de autooptimización. Este enfoque ya ha sido propuesto en trabajos relacionados con software adaptativo [49], optimización automática de software [131] y optimización en sistemas de altas prestaciones [25].

- Modelado del tiempo de ejecución: se plantea la optimización mediante el estudio y construcción de un modelo matemático parametrizado del tiempo de

ejecución, donde aparecen reflejadas tanto las características del software a optimizar como las características del entorno para el que se va a optimizar (velocidades, capacidades, arquitecturas...). La optimización del software consiste en la búsqueda de los parámetros que minimizan la función que modela el tiempo de ejecución. Aunque la aproximación posee la ventaja de ser suficientemente genérica como para ser aplicada a un amplio conjunto de códigos y librerías, a menudo es criticada en relación con la dificultad inherente a la búsqueda del modelo matemático y su posterior minimización. Como veremos a lo largo de esta memoria, la técnica es adecuada para introducir en herramientas de autooptimización.

Antes de empezar a hablar sobre la metodología más adecuada a seguir para lograr la autooptimización, definiremos los conceptos de sistema homogéneo y heterogéneo, que serán sobre los que se realizarán los experimentos, desarrollando primero la investigación sobre sistemas homogéneos y a continuación adaptándola a sistemas heterogéneos:

- Un **sistema homogéneo** será aquel en que los elementos de proceso tienen las mismas características y la red de comunicación es idéntica vista desde los distintos nodos de cómputo. Esto querrá decir que los procesadores son idénticos, con la misma velocidad, memoria..., y que el coste de las comunicaciones entre dos procesadores cualesquiera es idéntico. Estas condiciones se pueden relajar, y podemos considerar un sistema homogéneo aunque no lo sea totalmente: puede que los procesadores no tengan la misma capacidad de memoria, que los nodos tengan distinto número de procesadores, que las velocidades de cómputo o de transmisión de los datos no sean exactamente las mismas aunque sí parecidas... En ocasiones, es posible abstraerse a detalles concretos de la arquitectura, y desde el punto de vista del análisis es el diseñador el que decide si en su metodología el sistema a considerar es homogéneo o no. En la categoría de sistema homogéneo entrarán normalmente los multicomputadores y los clusters de ordenadores del mismo tipo que utilizan una red común.

- Un **sistema heterogéneo** es aquel en que los elementos de proceso son distintos o en la red de comunicación se ven los procesadores de manera asimétrica (con distintas velocidades de transmisión entre pares de procesadores distintos). Como hemos comentado, no nos interesará si el sistema es heterogéneo o no, sino si para nuestra metodología lo consideramos como tal. Algunas veces se considera heterogeneidad en la computación pero no en la red de comunicación

(aunque ésta sea heterogénea) y otras veces se considera heterogeneidad tanto en la computación como en la red. En la categoría de los sistemas heterogéneos encontramos: los clusters de ordenadores, que pueden estar compuestos por procesadores de características distintas aunque compartan la red de comunicación; los sistemas distribuidos en los que se conectan varios componentes que pueden ser a su vez clusters, supercomputadores, monoprocesadores...; e incluso en un mismo nodo es posible encontrar en la actualidad heterogeneidad, con procesadores de distinto tipo en un mismo chip.

Nos interesará desarrollar una metodología lo más general posible, que abarque la heterogeneidad tanto en la computación como en las comunicaciones, y que se pueda extender a los distintos tipos de sistemas mencionados. Para ello, nos centraremos en la segunda aproximación considerada anteriormente, puesto que formará parte fundamental en la metodología de trabajo de esta tesis, e introducimos en esta sección los aspectos de notación necesarios para su formulación.

Esta aproximación ya ha sido usada en trabajos desarrollados sobre modelado de algoritmos de álgebra lineal [34], donde formalmente el tiempo de ejecución de un programa puede ser modelado en la forma: $t(s, AP, SP)$, donde s es el tamaño del problema, AP son los parámetros algorítmicos y SP son los parámetros del sistema. Los parámetros algorítmicos son aquellos cuya modificación proporciona diferentes versiones del algoritmo, y su variación produce diferencias en los tiempos de ejecución pero no en la solución que proporcionan al problema. Estos parámetros pueden ser modificados con el fin de reducir los tiempos de ejecución. Algunos parámetros algorítmicos típicos son: el número de procesadores a utilizar de los disponibles en un sistema homogéneo, el número de filas y columnas de procesadores en algoritmos sobre mallas, el tamaño de bloque en la distribución cuando se consideran distribuciones cíclicas, el tamaño de bloque de cómputo en algoritmos maestro/esclavo, etc. En el caso de un sistema heterogéneo, el número de parámetros se incrementa de forma considerable debido a que cada procesador puede requerir un tamaño de bloque diferente, o a que el número de procesos a asignar en cada procesador puede variar u otras razones. El número de parámetros algorítmicos puede ser ciertamente elevado y contribuir de manera notable a hacer que el modelo de ejecución de tiempo sea más complejo. Los parámetros del sistema SP son aquellos que dependen directamente de la arquitectura a considerar, algunos de estos parámetros que se suelen considerar son: el coste de una operación aritmética, o el coste de una operación aritmética particular (por ejemplo, podría tener interés incluir un parámetro para multiplicaciones, otro para comparaciones, etc.), o simplemente el coste de una operación básica. El coste

de las operaciones aritméticas suele ser denotado por t_c. En un sistema heterogéneo el coste de este tipo de operaciones varía en función del procesador considerado. Los costes de las operaciones de comunicación deben ser también incluidos en el modelo como parámetros del sistema. La forma habitual de modelar las comunicaciones es a través de los parámetros tiempo de arranque de la comunicación, *start-up time* (t_s), y del tiempo de envío por palabra, *word-sending time* (t_w), de modo que el tiempo de transferencia de n datos lleva asociado un coste de transferencia de $t_s + n \cdot t_w$. Claramente, el coste de los parámetros de comunicación puede variar en función de la rutina de comunicación utilizada. Generalmente, suele ser más sencillo representar en el modelo los parámetros AP que los SP. Los parámetros SP dependen de la arquitectura destino y sólo es necesario calcularlos una vez en cada nuevo sistema.

En sistemas homogéneos los componentes computacionales tienen todos idéntica capacidad computacional, y la red de comunicaciones tiene la misma capacidad entre cada dos elementos del sistema. Así, los parámetros que representan estas capacidades (típicamente t_c, t_s y t_w) tendrán el mismo valor en cada procesador y entre cada dos procesadores. En el caso de sistemas heterogéneos esas capacidades varían de un procesador a otro y entre cada dos procesadores, con lo que no tendremos parámetros individuales, sino que si suponemos el sistema compuesto por P procesadores, nombrados de 0 a $P-1$, a cada parámetro aritmético le corresponderá un vector de P componentes ($t_c = \left(t_{c_0}, t_{c_1}, \ldots, t_{c_{P-1}}\right)$), donde cada componente representa el coste de una operación básica en un procesador; y a cada parámetro de comunicaciones le corresponde un array de dimensión $P \times P$: $t_s = \left(t_{s_{0,0}}, \ldots, t_{s_{0,P-1}}, \ldots, t_{s_{P-1,P-1}}\right)$ y $t_w = \left(t_{w_{0,0}}, \ldots, t_{w_{0,P-1}}, \ldots, t_{w_{P-1,P-1}}\right)$, donde $t_{s_{ij}}$ y $t_{w_{ij}}$ representan el *start-up* y el *world-sending time* de comunicaciones entre un proceso en el procesador i y otro en el j, y cuando $i = j$ entre dos procesos en el mismo procesador.

En el caso de adaptación de software homogéneo a un entorno heterogéneo, que como veremos es una parte importante de esta tesis, los parámetros algorítmicos recogerán también el número de procesos a asignar a cada procesador, de modo que el modelo se usará para reasignar las tareas o procesos, que originariamente se repartían de forma equitativa entre los elementos del entorno homogéneo, sobre el entorno heterogéneo.

Una vez un programa ha sido modelado mediante la función $t(s, AP, SP)$ la evaluación de este modelo sobre unos parámetros algorítmicos específicos AP_0 y sobre un sistema concreto SP_0 nos proporcionaría el tiempo de ejecución del programa sin necesidad de realizar su ejecución, que puede ser muy costosa en tiempo. Dada una arquitectura paralela, caracterizada por SP_0, la optimización de un programa sobre ella consiste en buscar los parámetros AP_{min} que hacen que la función *t()* alcance

su valor mínimo. Dependiendo del sistema y el programa a optimizar, y del nivel de precisión que se requiera en la optimización, el número de parámetros a considerar puede ser muy elevado y el modelo matemático ser bastante complejo.

Este enfoque, que considera la construcción de un modelo matemático, tiene ventajas e inconvenientes. Por un lado, se trata de una forma sencilla de abstraer la autooptimización como un proceso de búsqueda de parámetros que optimizan la función tiempo de ejecución. Además, la aproximación es genérica pues basta con instanciar el modelo con los parámetros adecuados para poder aplicarla. Sin embargo, implica la construcción de un modelo matemático que refleje con exactitud el tiempo de ejecución de la rutina a optimizar. Dado que el modelo depende fuertemente de las características físicas de los sistemas a utilizar, a veces es complicado poder recoger en un modelo matemático la enorme heterogeneidad de los elementos de que puede constar un sistema informático. Otro aspecto a considerar es la dificultad de la operación de minimización del modelo, que también crece con el número de parámetros dado que se trata de minimizar funciones vectoriales de varias variables.

En cuanto a los parámetros del sistema hay que hacer algunas consideraciones, al ser complicada su estimación:

- t_c. Si no se conoce su valor a través de la información proporcionada por el fabricante del hardware, se puede calcular dividiendo el tiempo empleado para ejecutar iterativamente una serie de instrucciones aritméticas básicas por el número de operaciones que se han realizado. En otras ocasiones este valor se estima a partir de la ejecución de versiones reducidas del programa a modelar.

- t_s y t_w. Se pueden calcular haciendo uso de la técnica de *ping-pong* que consiste en realizar un envío de datos de un procesador a otro de tal forma que éste devuelva esos datos al procesador que los envió, controlando el tiempo total del proceso. Se realizan un conjunto de experimentos con configuraciones diferentes en cuanto a tamaños de datos de envío y se calculan los parámetros del sistema a través de alguna de estas opciones:

  - Tomar la media de los parámetros de los valores obtenidos para las distintas ejecuciones.
  - Ajuste por mínimos cuadrados: a veces las diferentes ejecuciones del *ping-pong* proporcionan resultados dispares para los valores de t_s y t_w, y se puede realizar un ajuste por mínimos cuadrados.
  - La información para distintos tamaños se puede almacenar en forma de tabla, y realizar interpolación lineal para obtener los valores para nuevos

tamaños de problema a partir de los valores almacenados.

Uno de los problemas más importantes para lograr reducir el tiempo de ejecución de los problemas a resolver es el adecuado reparto de los recursos hardware, lo cual se denomina mapeo. Se pueden desarrollar técnicas automáticas, de manera que se consiga este objetivo sin intervención humana, incluyéndose así el mapeo en el proceso de autooptimización. A continuación procedemos a estudiar sus características y las diferentes técnicas que existen para abordarlo.

1.3. El problema de mapeo

El concepto de mapeo se asocia a un importante problema en computación en paralelo que ha sido profusamente estudiado prácticamente desde el comienzo de la disciplina. Se trata de un problema de gran interés y de difícil solución. A lo largo de la literatura encontramos ligeras variantes en la formulación del concepto de acuerdo con la aproximación seguida por cada autor. Por ejemplo en [104] se plantea que el problema del mapeo puede establecerse como el de ajustar un algoritmo o un problema a los recursos hardware con el fin de optimizar su resolución. El problema suele complicarse por el hecho de que normalmente el tamaño de un problema es mayor que el número de elementos de procesamiento disponibles en la máquina. En esos casos hay que contar con estrategias que permitan particionar la computación y asignarla a los recursos disponibles. Como resultado de la gestión del problema del mapeo podrían resolverse cuestiones como: ¿Qué plataforma es más adecuada para una aplicación dada?, ¿Con qué eficiencia puede una aplicación ser ejecutada sobre una plataforma determinada? Algunos de los subproblemas que aparecen asociados al problema del mapeo son los siguientes:

- El análisis de dependencias y la transformación de algoritmos
- El particionado, asignación y planificación
- El equilibrado de carga estático y dinámico
- El estudio de las variaciones en el algoritmo

A continuación describiremos algunos aspectos relacionados con estos subproblemas.

1.3.1. Análisis de dependencias y transformación de algoritmos

Asociado al problema del mapeo se encuentra la transformación de algoritmos de una forma a otra más adecuada a la plataforma destino. Los algoritmos pueden transformarse desde su forma secuencial hasta la versión paralela o de una versión paralela a otra.

La forma de detectar el paralelismo y la transformación de algoritmos es básicamente un problema de estudiar dependencias de datos en procesos y entre procesos. Obviamente, una transformación de algoritmos es válida únicamente cuando mantiene la equivalencia de los algoritmos. Se trata de un análisis orientado a identificar el paralelismo existente, y este paralelismo depende de la naturaleza del problema y del algoritmo utilizado por el programador.

Cuando la arquitectura destino es de naturaleza heterogénea es frecuente que la transformación del algoritmo se realice desde la versión paralela del algoritmo desarrollado para un sistema homogéneo hacia una versión que trabaje en el sistema heterogéneo.

1.3.2. Particionado, asignación y planificación

Particionado, Asignación

El problema del particionado puede formularse como el de dividir un problema en subproblemas más pequeños para su resolución. El particionado es esencial cuando el tamaño de un problema de cómputo es superior al número de procesadores físicos disponibles. El tamaño de un problema puede venir expresado de muchas formas diferentes, por ejemplo, como el número de puntos en el conjunto de índices, el número de datos de entrada, el número de filas de una matriz u otros. En general el problema no es sencillo ya que el particionado podría introducir efectos laterales no deseados que degraden la estabilidad numérica de los algoritmos.

El particionado especifica también las unidades secuenciales de cómputo en el programa y de aquí la granularidad de la ejecución. Además, puede llevar asociado:

- Un particionado de datos manteniendo el algoritmo a ejecutar.

- Un particionado o transformación del código en el que, por ejemplo, se asocian secciones o iteraciones distintas del código con distintos procesos, manteniendo los datos de ejecución. En este caso el particionado puede conseguirse mediante transformaciones en el algoritmo. Una aproximación al problema suele consistir

en dividir el conjunto de índices del algoritmo entre bandas y asignar estas bandas al espacio de procesadores.

- El particionado tanto de datos como del código en función de las características del problema a resolver.

El tipo de particionado viene generalmente impuesto por el tipo de dependencias en el cálculo que produce el algoritmo y se trata de buscar un equilibrio entre el número de procesos que se generan, habitualmente superior al número de procesadores de que se dispone, y su posterior asignación a los procesadores físicos, teniendo en cuenta que las comunicaciones debidas al intercambio y a la transferencia de datos y procesos, y las sincronizaciones entre procesos impuestas por las dependencias de datos juegan un papel importante en el tiempo final de ejecución del algoritmo.

En muchas ocasiones se asume que los procesos simplemente se distribuyen y planifican entre los procesadores sin más discusión acerca de los efectos sobre los tipos de procesadores y sus velocidades. Nótese que la asignación de dos procesos a dos procesadores diferentes permite que su ejecución pueda proceder en paralelo hasta que se hace necesario establecer una comunicación entre ambos procesos. En ese momento, se establece la comunicación y la computación puede proceder. En el otro extremo, en caso de que los dos procesos fueran asignados al mismo procesador, la ejecución sería secuencial, sin embargo no sería necesario incurrir en el incremento de tiempo asociado a la comunicación interprocesador. Claramente el particionado debe ir acompañado de una asignación adecuada de los procesos o tareas generadas a los procesadores y su posterior planificación para ejecución, puesto que una mala asignación puede originar un descenso en el rendimiento.

Para algunos autores, el punto de partida para el particionado, asignación y planificación es la construcción de la representación gráfica de los programas. Se trata de una abstracción que proporciona características del rendimiento e ignora el resto de los aspectos, tales como los semánticos. Herramientas habituales para la representación de programas son los grafos de flujos de datos en el caso secuencial o los grafos dirigidos acíclicos (*dags*). En los *dags*, los nodos internos representan una expresión o una sentencia. Las aristas representan dependencias de datos. Un bloque básico en un *dag* representa a un bloque básico de cómputo. La representación gráfica puede contener información de la estructura del programa, el paralelismo, frecuencias de ejecución, y costes de comunicación y tiempo de ejecución.

Una técnica de particionado propuesta en [104] consiste en comenzar con una partición inicial de grano fino como la proporcionada por el *dag* e ir iterativamente incrementando la granularidad hasta conseguir una partición más gruesa, dando lu-

gar a grafos de dependencias de procesos. Los nodos son los procesos y las aristas las dependencias entre procesos. Para cada iteración es necesario computar la función de coste y seleccionar la partición que minimiza la función de coste. La función de coste del particionado es una combinación de dos términos: el término asociado al camino crítico y el término asociado al *overhead*. El término asociado al *overhead* se decrementa con el número de iteraciones porque un movimiento hacia una granularidad más gruesa no puede incrementar el *overhead* total.

Asignación, Planificación

La planificación puede definirse como una función que asigna procesos a procesadores y el objetivo es el de distribuir la carga sobre todos los procesadores con el fin de obtener su máxima eficiencia, minimizando la comunicación de los datos, lo que llevaría a un tiempo de ejecución total más corto. Las políticas de asignación pueden ser clasificadas como estáticas o dinámicas:

- Asignación estática. Bajo las políticas de asignación estática, los procesos se asignan a los procesadores por el programador o por el compilador antes de su ejecución. No hay sobrecarga en tiempo de ejecución y se incurre en la sobrecarga de asignación una sola vez cuando los programas van a ser ejecutados varias veces sobre diferentes secuencias de datos, aunque en muchas ocasiones, pequeños cambios en las secuencias de entrada obligan a una nueva asignación estática.

- Asignación dinámica. Bajo las políticas de asignación dinámica, las tareas se asignan a los procesadores en tiempo de ejecución. Este esquema ofrece mejor utilización de los procesadores, pero al precio de asignaciones adicionales. La asignación dinámica puede ser distribuida o centralizada. En la asignación distribuida, hay una bolsa de tareas y cualquier procesador libre puede tomar procesos de esa bolsa. Una tarea puede distribuirse sobre varios procesadores. En la asignación centralizada, los procesos se asignan a procesadores mediante mecanismos de asignación bajo un control centralizado. Con la asignación centralizada, pueden aparecer cuellos de botella cuando el número de procesadores crece.

1.3.3. Equilibrado de carga

Durante la fase de ejecución puede ocurrir, además, que algunos procesadores completaran sus tareas antes que otros y permanecieran ociosos porque el trabajo

no fue repartido equitativamente, porque algunos procesadores operaran más rápido que otros, o por ambas situaciones. Idealmente, se desea que todos los procesadores trabajen continuamente sobre las tareas de modo que el tiempo de ejecución sea mínimo. Alcanzar este objetivo de reparto equilibrado de tareas sobre el conjunto total de procesadores se conoce como equilibrado de carga, de modo que, en ocasiones, resolver el problema del mapeo implícitamente implica resolver el problema del equilibrado de carga. Este hecho hace que, algunos autores [132] consideren que resolver el problema del mapeo consiste en resolver el problema de la asignación y planificación de los procesos de forma óptima haciendo referencia únicamente a la resolución del problema del equilibrado de carga. Se asume en este caso que se parte de algoritmos sobre los que, de forma implícita, ya ha sido realizado el análisis de dependencias, el particionado o la transformación adecuada.

A lo largo de esta tesis, se asumirá que el problema del mapeo estará vinculado en términos generales a resolver el problema de la asignación y planificación de los procesos y que cuando sea necesario se realizará también la transformación de los algoritmos y el análisis de dependencias. En general el problema del mapeo es un problema computacionalmente intratable NP [132], de ahí que la mayoría de las propuestas para abordar su resolución sean de tipo heurístico. Esta dificultad inherente a este problema hace que se hayan obtenido soluciones eficientes para algunos problemas particulares [15, 89], para otros problemas sin embargo, sólo se pueden obtener soluciones aproximadas haciendo uso de métodos heurísticos [9, 51, 121, 134].

1.3.4. Estudio de variaciones en los algoritmos

Mapear aplicaciones a computadores paralelos y equilibrar el trabajo de los procesadres paralelos no es una tarea fácil. Pequeños cambios en los tamaños de los problemas al utilizar diferentes algoritmos o diferentes aplicaciones pueden producir efectos no deseados e incurrir en degradación del rendimiento. Por ejemplo, hay que analizar cómo afectan al rendimiento del algoritmo, cambios en la granularidad, en los tamaños de vectores, en el uso y la gestión de la memoria, en el tipo de transformación del espacio de índices realizado, ... Las técnicas de mapeo que llevan asociadas funciones de coste como mecanismo de optimización, presentan en su mayoría la ventaja de proporcionar estrategias para el análisis de las variaciones de los algoritmos.

1.3.5. Técnicas de mapeo

Describimos a continuación algunas de las técnicas que históricamente han venido siendo utilizadas para abordar el problema del mapeo:

- Algoritmos Aleatorios: Se realiza una asignación aleatoria de procesos a procesadores.
- Algoritmos Round Robin: Se establecen estrategias de asignación de procesos a procesadores de manera circular. Cuando el último procesador recibe su asignación, se comienza con el primero. La granularidad de la asignación determina en cada iteración de asignación el tamaño de los bloques de procesos. En determinados problemas la búsqueda de la asignación óptima consiste en buscar el tamaño óptimo del bloque [3, 9, 69].
- Bisección Recursiva: Se realiza una división recursiva de los procesos en subgrupos de igual carga computacional y que minimicen el volumen de comunicaciones. Implica la existencia de algún mecanismo de evaluación de la carga a priori [45].
- Algoritmos Heurísticos: Se utiliza algún algoritmo heurístico que permita optimizar la asignación de procesos a procesadores de acuerdo a algún criterio. Cuando el criterio es el de minimizar el tiempo de ejecución la técnica suele implicar el uso de alguna función del coste temporal de los procesos y de las comunicaciones asociadas [106].
- Grafos de precedencia: Esta aproximación consiste en formar un grafo de dependencias del programa en el que los nodos son procesos y los arcos son sus relaciones de dependencia. Una estimación del tiempo de ejecución puede proporcionarse también. Una vez que el grafo de dependencias ha sido construido, se puede realizar el mapeo sobre la arquitectura destino. Esta aproximación proporciona buenos rendimientos, pero el trabajo necesario para obtener el grafo de dependencias puede ser considerable si no se dispone de herramientas que automaticen su generación. La aproximación implica la obtención del camino crítico en el grafo de dependencias así como los tiempos de ejecución más temprano y mas tardío de cada proceso [115].
- Optimización Analítica: Se dispone de una función de analítica del coste del algoritmo y analíticamente se resuelve el problema de encontrar los parámetros que minimizan la ejecución. Presenta la ventaja de ser independiente de la arquitectura destino, pero el inconveniente de la dificultad asociada al proceso de minimización incluso para funciones relativamente simples [3, 12].
- Recorridos en árbol: Se formula el problema de asignación como un problema de tipo enumerativo en el que se consideran todas las posibles soluciones expre-

sadas en forma de árbol de soluciones. Cada rama del árbol representa a una posible asignación. El proceso enumerativo suele ir acompañado de funciones de coste que permiten evaluar la optimalidad en cada nodo del árbol. A lo largo de la tesis se han utilizado este tipo de árboles como herramientas de representación y evaluación de la asignación de procesos a procesadores. Es por esto que posteriormente se mostrará en más detalle este tipo de estrategia.

Un factor importante que afecta a la asignación es el carácter (homogéneo/ heterogéneo) de los procesadores que componen el sistema. Cuando el sistema es heterogéneo algunas de las estrategias de asignación previamente presentadas, deben ser revisadas para poder ser aplicadas con efectividad.

1.3.6. Técnicas de mapeo aplicadas a sistemas heterogéneos

Cuando el sistema consta de elementos con diferentes características computacionales es posible emplear diferentes enfoques para realizar el mapeo. En particular todas las técnicas descritas en la sección anterior pueden ser aplicadas teniendo en cuenta el carácter heterogéneo del sistema donde sea necesario. Haremos énfasis en esta sección, en dos estrategias habitualmente aplicadas al caso heterogéneo:

- La estrategia HoHe [83], consiste en diseñar procesos que trabajan sobre volúmenes de datos distintos y asignar un proceso a cada procesador que se utilice. El volumen de trabajo dependería de las características del procesador sobre el que va a ejecutarse el proceso. Se tiene una distribución homogénea de procesos y una distribución heterogénea de datos.

- La estrategia HeHo [83], en la que cada proceso trabaja con un volumen idéntico de datos. Se crea un número de procesos independiente y habitualmente superior al número de procesadores, y se asigna un número variable de procesos a cada procesador en función de las capacidades de computación y comunicación de cada procesador. Se tiene, en este caso, una distribución heterogénea de los procesos en los procesadores, y una distribución homogénea de los datos en los procesos.

La estrategia HoHe conlleva la reprogramación del código, lo que supone un importante trabajo de rediseño de los algoritmos que han sido desarrollados para los sistemas homogéneos. La estrategia HeHo, sin embargo, permite que se pueda utilizar el mismo código que en sistemas homogéneos, y sólo será necesario disponer de un método eficiente de asignación de procesos a procesadores. En este caso, se plantea

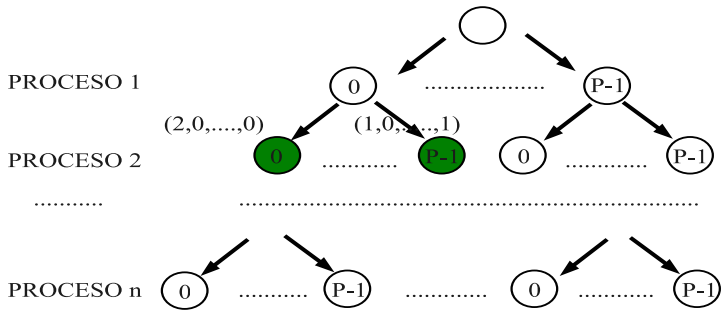

Figura 1.1: Árbol de asignación de procesos a procesadores.

un problema de asignación de procesos a procesadores con el fin de obtener un tiempo total de ejecución reducido.

1.3.7. El árbol de asignación de procesos

En la figura 1.1 podemos ver un ejemplo de la estructura del árbol de asignación. Recordamos que el árbol de asignación se utiliza para la resolución del problema de asignación de procesos a procesadores, y analizamos su recorrido por medio de metaheurísticas. Llamamos P al número de procesadores de que disponemos en el sistema. Cada nivel i representa la posible asignación del proceso número i a uno de los posibles procesadores del sistema. La altura del árbol es el máximo número de procesos que se mapean, no estando limitada al no estarlo el número de procesos que se pueden emplear, pero se puede establecer un valor máximo para limitar la búsqueda, o se puede utilizar un método con el que no se exploren nodos que correspondan a números muy elevados de procesos. Cada nodo del árbol representa una de las posibles asignaciones y tiene asociado su correspondiente tiempo de ejecución [39]. Por lo tanto, para obtener una asignación satisfactoria debemos disponer de un modelo realista del tiempo de ejecución de la rutina, para lo que puede utilizarse el modelado basado en parámetros.

Se puede ver que existen dos nodos coloreados en el nivel 2. El primero tiene el valor (2,0,0,..,0). Dichos valores representan la asignación realizada en esta parte del árbol: los procesos números 1 y 2 (2 procesos) han sido asignados al procesador número 0 y no se han asignado procesos al resto de procesadores. El segundo nodo tiene el valor (1,0,..,1) porque el proceso número 1 ha sido asignado al procesador número 0 y el proceso número 2 al procesador número $P-1$. Otra posibilidad sería representar para cada proceso el procesador al que se asigna: el nodo (2,0,...,0) se

correspondería en este caso con el (0,0,-1,-1,...) al no estar limitada la altura del árbol (y si estuviera limitada debería estarlo a un valor bastante mayor que P, obteniéndose un vector con muchos componentes), donde los dos primeros valores indican que se asignan los procesos 1 y 2 al procesador número 0 y con el valor -1 se indica que no se asigna el proceso correspondiente a ningún procesador.

1.3.8. Aportación de las metaheurísticas

Tal y como veremos a lo largo de esta tesis, la búsqueda a través del árbol se puede hacer usando técnicas de recorrido exhaustivo (*backtracking*, ramificación y acotación... [23, 30]). Cuando este recorrido se hace sobre sistemas de tamaño reducido, el resultado puede ser satisfactorio al no suponer un tiempo de ejecución muy elevado y obtenerse la asignación óptima; sin embargo sobre sistemas complejos implica un excesivo consumo de tiempo incluso en el caso de que se usen estrategias de poda de nodos.

Por lo tanto, usar estas técnicas puede ser adecuado para pequeños sistemas, pues el tiempo para determinar la asignación de procesos a procesadores no será alto, pero no para grandes sistemas pues implicaría recorrer gran parte de un árbol gigantesco. En este caso es posible utilizar un método de avance rápido que puede proporcionar una asignación alejada de la óptima [39].

Otra posibilidad es la utilización sistemática de heurísticas que se han mostrado efectivas para problemas computacionalmente complejos. Por medio de diversas técnicas metaheurísticas (temple simulado, búsqueda tabú, búsqueda dispersa, GRASP... [47, 117]), se pueden obtener distribuciones cercanas a la óptima con un tiempo de decisión reducido, como se verá en el capítulo 5.

1.4. Hipótesis y objetivos

1.4.1. Hipótesis de investigación

La hipótesis de esta tesis se centra en que es posible hacer uso del modelado del tiempo de ejecución del software con el fin de estimar el tiempo de ejecución de los algoritmos paralelos. Una vez estimado el tiempo de ejecución el propio software podría ser capaz de decidir por sí mismo y sin intervención del usuario cómo ajustar los parámetros adecuados para lograr la reducción de su tiempo de ejecución en cualquier sistema informático. De esta manera se evitaría rediseñar las rutinas para obtener versiones eficientes para los distintos sistemas actuales o los que se pudieran diseñar en el futuro, así como liberar al usuario del mantenimiento y optimización del

software permitiéndole trabajar sólo en su ámbito de investigación.

Con este trabajo se pretenden alcanzar objetivos que supongan una evolución dentro del actual contexto de técnicas de optimización de software.

1.4.2. Objetivos generales

El objetivo fundamental de esta tesis es conseguir la autooptimización de código desarrollado en esquemas algorítmicos paralelos iterativos. Para estos esquemas, se busca que el propio software sea capaz de lograr que sus prestaciones se optimicen (autooptimización) sin la intervención del usuario, con independencia de la plataforma que se esté usando.

Como caso particular se hace necesario el estudio, desarrollo e implementación de técnicas que permitan la adaptabilidad de las librerías que se han desarrollado durante años para sistemas homogéneos hacia entornos heterogéneos. Esta adaptación presenta diversas dificultades técnicas que hacen necesario considerar las características físicas del entorno heterogéneo al tener esto una importancia considerable de cara a obtener buenas prestaciones en el software.

1.4.3. Objetivos parciales

Para alcanzar el objetivo general anteriormente expuesto se plantean los siguientes objetivos parciales:

- Desarrollar metodologías para autooptimizar código homogéneo sobre sistemas homogéneos. Debido a la simplificación que supone trabajar con sistemas homogéneos y la importancia que éstos tienen (supercomputadores, *multicore*...) inicialmente estudiaremos la optimización de código sobre estos sistemas.

- Desarrollar metodologías para autooptimizar código homogéneo en sistemas heterogéneos. Una vez estudiado el punto anterior se pretenden desarrollar técnicas para conseguir la adaptación sobre sistemas heterogéneos, lo que implicará una complejidad mayor al tener que considerar las características físicas de los distintos componenentes, por lo que usaremos varias aproximaciones:

    - Métodos exactos. Se trata de emplear métodos exhaustivos de búsqueda para encontrar soluciones de los modelos matemáticos parametrizados de tiempos de ejecución. Como veremos, los diferentes métodos obtienen soluciones óptimas pero a cambio de generar tiempos excesivamente grandes en encontrar estas soluciones. Estos métodos podrán ser útiles o no dependiendo del tipo de problema a resolver.

  - Métodos aproximados. Se trata de emplear métodos que permiten obtener soluciones no óptimas pero sí bastante buenas a cambio de no producir tiempos inasumibles para encontrar estas soluciones. Aquí ocurrirá también que dependiendo del tipo de problema y de la naturaleza de los sistemas su uso será o no aceptable.
  - Técnicas metaheurísticas. Con esta última aproximación, estudiaremos técnicas heurísticas que nos permitan obtener buenas soluciones en problemas para cuya resolución no son válidos los enfoques anteriores.

- Aplicaciones en esquemas paralelos iterativos. Nos vamos a centrar en los esquemas paralelos iterativos como código a autooptimizar, pero la metodología de trabajo deberá ser válida para otro tipo de esquemas. En particular, las técnicas desarrolladas se podrían aplicar a la resolución paralela eficiente de problemas de alto coste computacional tanto en el ámbito académico como de distintas áreas científicas (genética, química, ingeniería ...)

1.5. Metodología

En esta sección planteamos la metodología general de trabajo que se propone en esta tesis con el fin de alcanzar los objetivos fijados. La metodología planteada es, en sí misma, una propuesta de aproximación genérica a la resolución del problema de la autooptimización del software. El esquema general para esta metodología puede verse en la figura 1.2 organizado por fases:

- Fase de **Diseño**: se procede al diseño de la rutina a emplear. Para esta fase se puede hacer uso de alguna metodología de ingeniería del software que permita construir el código buscando la corrección y la exactitud. Una vez hecho esto, se debe construir una función o modelo analítico que represente el coste temporal de ejecución y que considerará los parámetros algorítmicos y los del sistema.

- Fase de **Instalación**: se obtienen los parámetros SP sobre el sistema actual. SP representa los parámetros del sistema, sobre los cuales no podemos ejercer ningún tipo de influencia, sino que vienen determinados por las características tecnológicas del sistema que estamos utilizando para ejecutar nuestros algoritmos, y que pueden ser calculados con el fin de determinar el grado de incidencia que van a tener sobre el tiempo de ejecución final. Algunos parámetros del sistemas son, como ya se ha visto, el tiempo de realizar una operación aritmética básica t_c, el tiempo de inicio de una comunicación entre procesos t_s (*start-up*

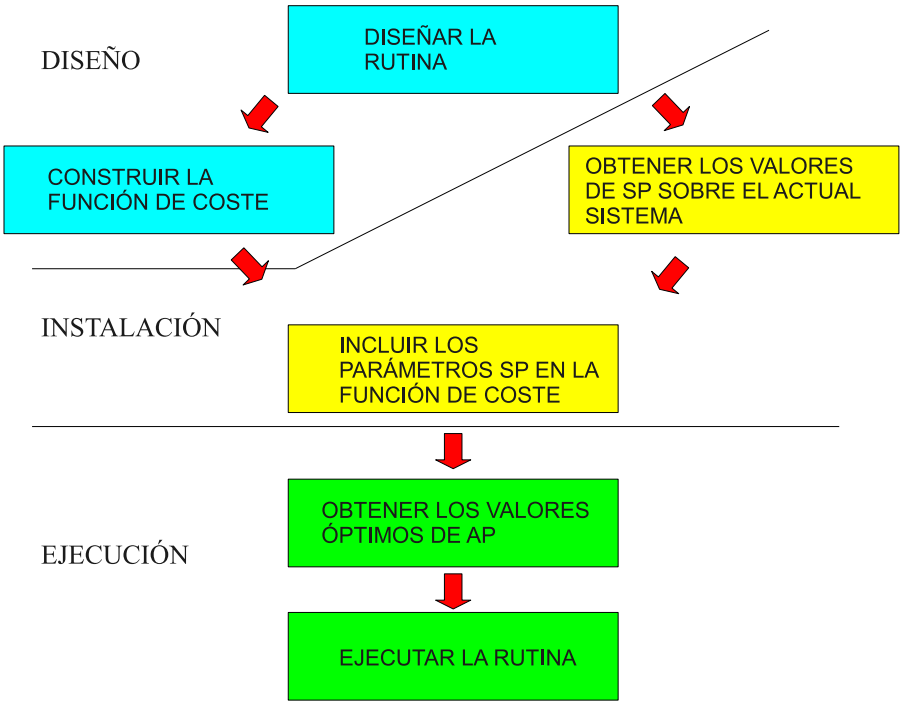

Figura 1.2: Arquitectura software de un sistema de autooptimización de software.

time), el tiempo de envío de un dato básico entre dos procesos t_w (*word-sending time*), el tiempo de realizar una comparación entre datos... Una vez estimados los parámetros SP se incluyen en la función de coste para que ésta refleje fielmente el comportamiento real de la rutina.

- Fase de **Ejecución**: se calculan los valores AP con los que se obtiene el menor tiempo teórico conforme al modelo del problema. A diferencia de los anteriores, estos parámetros vienen determinados por el algoritmo que se emplea, y pueden ser seleccionados de cara a optimizar los tiempos de ejecución. Algunos de estos parámetros pueden ser: el tamaño del bloque en algoritmos de álgebra lineal, el número de procesos a considerar, el número de procesadores a emplear, el esquema de comunicaciones entre procesos... En algunos casos se pueden aplicar métodos exactos para obtener estos valores, pero en la mayoría de los casos suele ser complicado y por ello se hace necesario usar diferentes técnicas heurísticas que permitan lograr una aproximación buena. Una vez calculados, se procedería a la ejecución de la correspondiente rutina.

Tal y como se ha comentado, una de las dificultades que surge en la aproximación es la propia construcción del modelo analítico durante la fase de diseño.

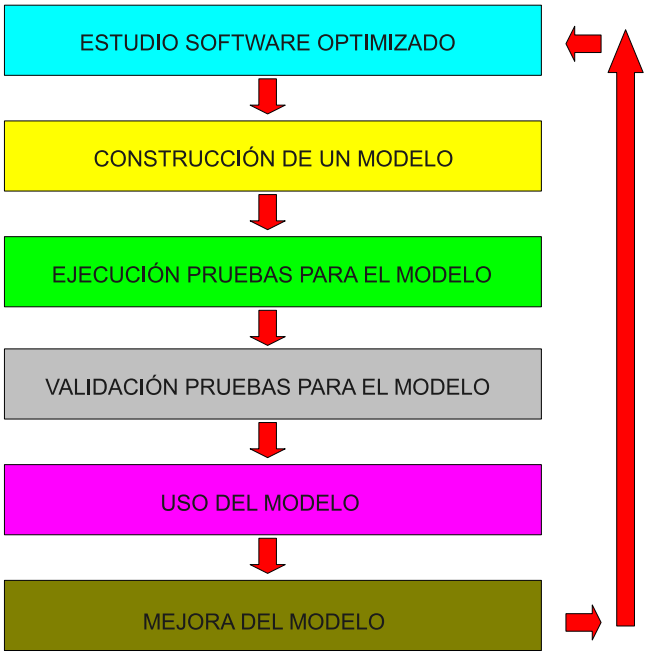

Figura 1.3: Proceso de modelado.

Surge, por tanto, la necesidad de abordar esta fase también de forma metódica. La figura 1.3 muestra el conjunto de pasos que consideramos que deben seguirse durante el proceso de modelado y que a continuación pasamos a detallar:

- **Estudio del software optimizado**: En primer lugar, se debe hacer un estudio del código que se pretende auotooptimizar. En nuestro caso, se trata inicialmente de diferentes algoritmos iterativos, aunque como hemos dicho anteriormente pretendemos la extensión de la metodología a otros tipos de esquemas algorítmicos. Para ello, hay que identificar las características computacionales del código: determinar su complejidad, tamaño del problema...

- **Construcción de un modelo**: Una vez identificados los elementos más relevantes hay que construir un modelo matemático que refleje el tiempo de ejecución del esquema en que se va a introducir autooptimización. Una forma de estimar el tiempo de ejecución es ponerlo en función de dos tipos de parámetros: aquellos que dependen de las características del sistema (parámetros del sistema) y aquellos que dependen de la naturaleza del problema a resolver (parámetros algorítmicos), tal y como se ha visto en la sección de optimización y autooptimización del código.

- **Diseño de pruebas**: Una vez que se ha construido el modelo, se deben crear una serie de pruebas que nos permitirán evaluar si dicho modelo se asemeja a la realidad.

- **Pruebas del modelo**: Se harán simulaciones de las pruebas diseñadas y además se llevarán a cabo ejecuciones en diversos sistemas reales tanto homogéneos como heterogéneos con el fin de probar la bondad del modelo desarrollado.

- **Validación del modelo**: Se deben comparar los resultados obtenidos con simulaciones del modelo desarrollado con respecto a los que se obtienen en sistemas reales. Cuando los resultados sean similares se puede concluir que el modelo es fiable, sin embargo cuando las diferencias sean significativas el modelo no habría sido bien diseñado y sería necesario hacer ajustes y volver a validarlo.

- **Uso del modelo**: Si se consigue construir un modelo fiable, se dispondrá de una herramienta adecuada que permitirá que los algoritmos iterativos puedan ser optimizados de manera automática. A través de una serie de tests iniciales donde se consideran las características físicas del entorno y las algorítmicas del software a emplear, el software debe ser capaz de elegir una configuración de parámetros adecuada para obtener buenas prestaciones en la ejecución.

- **Mejora del modelo**: Uso de diferentes técnicas, tanto de búsqueda exhaustiva como métodos heurísticos con el fin de reducir el tiempo empleado en la obtención de parámetros que van a permitir obtener una buena solución en el modelo del problema a resolver.

1.6. Consideraciones computacionales

En esta sección mostramos las características del contexto computacional (hardware y software) sobre el que se desarrolla la experiencia computacional de esta tesis.

1.6.1. El hardware

Las sistemas utilizados y sus características son:

- Red del laboratorio de computación paralela (**SUNEt**) de la Facultad de Informática de la Universidad de Murcia. Se trata de un sistema formado por estaciones del mismo tipo (5 estaciones SUN Ultra 1) junto con una estación SUN Ultra 5, que es aproximadamente 2.5 veces más rápida que el resto, unidas

en una red Ethernet 10. El sistema operativo es SOLARIS, el lenguaje utilizado es C propietario de SUN, y se usa mpich.

En los experimentos, el primer proceso se ejecuta en la estación más rápida (SUN Ultra 5), por lo que se puede apreciar en los tiempos de ejecución que existe un salto considerable entre la ejecución en un único procesador a hacerlo en más. Se han calculado los tiempos de ejecución hasta para 7 procesos (6 máquinas). En este caso se ejecutan dos procesos sobre el primer procesador (que es el más rápido), por lo que tal y como se verá, en algunos casos se produce una cierta mejora en los tiempos de ejecución usando más procesos que procesadores.

- Red del laboratorio de arquitectura (**PenFE**) de la Facultad de Informática de la Universidad de Murcia. Se trata de una red con 17 ordenadores Pentium III a 850 Mhz, 256Mb RAM, interconexión de red FastEthernet 100, lo que hace más fácil que en el sistema anterior obtener mejores resultados con la versión paralela que con la secuencial. El sistema es linux RedHat 7.3 y se ha utilizado gcc y mpich. En los experimentos se han utilizado sólo 7 procesadores para contrastar los resultados con los obtenidos en el resto de sistemas.

- Sistema **ORIGIN 2000** del Centro Europeo de Paralelismo de Barcelona (CEPBA) de la Universidad Politécnica de Cataluña (UPC). Se trata de un sistema Silicon Graphics con 64 procesadores MIPS R10000, cada uno con 4 Mb de Caché, 12 Gb de RAM y 360 Gb de almacenamiento. Con una velocidad teórica de 32 Gflops. El sistema es IRIX6.4. Se ha utilizado como compilador C y como herramienta mpi los propietarios del sistema. Se han empleado hasta 8 procesadores.

- Sistema **HPC160** del Servicio de Apoyo a la Investigación Tecnológica (SAIT) de la Universidad Politécnica de Cartagena (UPCT). Sistema paralelo de memoria distribuida HPC160 compuesto por 4 nodos. Cada nodo contiene 4 procesadores que comparten memoria. Los procesadores son Alpha EV68CB a 1 GHz con 8 Mb de caché de nivel 2. La capacidad pico teórica del sistema es de 32 Gflops, con 16 Gbytes de memoria y unos 300 Gbytes de almacenamiento en disco. El sistema operativo es Tru64 Unix. Se ha utilizado como compilador C y como herramienta mpi los propietarios del sistema. Se han utilizado 2 nodos (hasta 8 procesadores).

- Sistema **KIPLING** del Departamento de Sistemas Informáticos y Computación (DSIC) de la Universidad Politécnica de Valencia (UPV). Actualmente se trata

de un cluster de alto rendimiento (high performance), tipo CoPS (Conjunto de PC's) compuesto por 2 máquinas homogéneas de arquitectura similar, a nivel de la tecnología de sus procesadores. Cada una de estas máquinas, dispone de 500Mb de RAM y está compuesta por un arreglo de dos procesadores de tipo Intel Xeon 2.20 Ghz, lo que supone un conjunto de 4 nodos. El entorno paralelo a nivel de cluster se completa con una red de alta velocidad asistida por medio de un switch con tecnología Fast Ethernet a 100/1000 Mbps como ancho de banda, con una latencia teórica de 80ms. Cuando se llevaron a cabo los experimentos que aparecen descritos en el capítulo 5, el sistema se componía de 4 nodos de los cuales 2 eran biprocesadores y 2 monoprocesadores. El sistema operativo es linux y se utiliza gcc y mpich.

- Sistema **TORC** del Innovative Computing Laboratory de la Universidad de Tennessee. Se trata de una red de 21 nodos de diferentes tipos: 10 Dual 550 Mhz Pentium III, tres Pentium III 600 Mhz, dos Pentium II 450 Mhz, tres AMD Athlon 1.2 Ghz, un Pentium 4 1.7 Ghz, un Pentium 4 1.5 Ghz, y un DEC Alpha Clone EV5. Los procesadores están conectados a través de tres redes: 100Mbit, Myrinet y Giganet. Sólo se ha empleado la red 100Mbits en los experimentos y sólo alguno de los procesadores dual (DPIII), un Pentium III (SPIII) a 600 Mhz, uno de los AMD Athlon (Ath) y el Pentium 4 a 1.7 Ghz (17P4). Los equipos funcionan con distintas versiones de linux, y se ha utilizado gcc y mpich.

En la tabla 1.1 se resumen las características de los sistemas hardware y del software utilizado.

Tabla 1.1: Características de los sistemas utilizados en los experimentos.

Sistema	localización	proc/núcleos	S.O.	C	MPI
SUNEt	Murcia	6	Solaris	propi.	mpich
PenEt	Murcia	17	RedHat 7.3	gcc	mpich
ORIGIN 2000	Barcelona	64	IRIX6.4	propi.	propi.
HPC160	Cartagena	16	Unix Tru64	propi.	propi.
KIPLING	Valencia	6	linux	gcc	mpich
TORC	Tennessee	31	linux	gcc	mpich

1.6.2. El software

Como se ha indicado, el lenguaje utilizado ha sido C, en unos casos se ha utilizado el propio del sistema y en otros gcc. En la compilación se ha utilizado siempre la opción de optimización máxima (-O3 o -fast), dependiendo del sistema.

Los equipos informáticos que se han empleado son sistemas paralelos constituidos por diferentes configuraciones y cantidades de nodos de procesadores donde se hace necesario el uso de software específico adaptado al paralelismo. Se han utilizado esquemas de paso de mensajes, y para las comunicaciones se ha empleado la librería MPI [123], en unos casos la propia del sistema y en otros la mpich, que es de libre distribución.

La principal ventaja al establecer un estándar para el paso de mensajes, como MPI, es la portabilidad y su facilidad de uso.

Existen distintas posibilidades de reparto de los datos del problema entre los procesos y procesadores que se van a emplear en la resolución del mismo:

- Se puede asignar un proceso a cada procesador y con un volumen de datos idéntico para cada proceso. Esto corresponde a un modelo de programa homogéneo en un sistema homogéneo.

- Se puede asignar un proceso a cada procesador, pero que el volumen de datos en cada proceso sea distinto. Normalmente ese volumen será proporcional a la velocidad relativa del procesador. Corresponde a programas heterogéneos, que son apropiados para sistemas heterogéneos. Como se ha visto en la sección de técnicas de mapeo, se utiliza la denominación HoHe (distribución de procesos en el sistema Homogénea, y distribución de datos en los procesos Heterogénea) [83].

- Se puede asignar más de un proceso a cada procesador siendo igual el volumen de datos de cada proceso pero con un número de procesos por procesador proporcional a la velocidad del procesador. Corresponde a programas homogéneos que se ejecutan en sistemas heterogéneos sin modificarse, sino resolviendo algún problema de asignación. Como también se ha visto en la sección de técnicas de mapeo, se utiliza la denominación HeHo (distribución de procesos en el sistema Heterogénea, y distribución de datos en los procesos Homogénea) [83].

Se ha preferido repartir el trabajo de tal forma que cada proceso que vaya a intervenir procese una cantidad de datos que sean resueltos como si de un conjunto de pequeños problemas "secuenciales" se tratara. MPI considera que cada proceso

trabajará con un código común pero gestionando datos diferentes, haciendo posible la comunicación de datos entre procesos puesto que los procesos necesitarán datos que serán calculados por otros procesos.

Además se han desarrollado algoritmos que implementan técnicas de búsqueda aproximada y de metaheurísticas que se verán en los capítulos de autooptimización en sistemas heterogéneos y en el de uso de metaheurísticas en el mapeo de procesos a procesadores respectivamente.

1.7. Descripción por capítulos

Esta memoria se organiza en capítulos con el siguiente contenido:

- Capítulo 1: Introducción, Objetivos y Metodología. Se presenta una introducción general donde se hace un estudio del estado del arte y se establece el objetivo a conseguir, que no es otro que la posibilidad de construir software autooptimizable para emplearse tanto en sistemas homogéneos como heterogéneos. En esta primera parte se hace un recorrido inicial sobre el problema que se pretende abordar, clarificando los conceptos que se van a usar durante la tesis. Además se analizan en profundidad sus características, estudiando diferentes posibles soluciones, y se explican detalladamente las herramientas hardware y software desarrolladas y empleadas para atacar el problema.

- Capítulo 2: Esquemas iterativos y Programación dinámica. Se analizan diferentes variantes de esquemas iterativos, a los que se podrán aplicar las técnicas que se proponen en la tesis, y se estudian versiones iterativas para algoritmos de programación dinámica, que se utilizan para validar las propuestas realizadas.

- Capítulo 3: Autooptimización en sistemas homogéneos. Se estudia la posibilidad de usar técnicas de optimización automática en el diseño de algoritmos iterativos paralelos que van a ser usados en sistemas homogéneos, es decir, en sistemas con características computacionales y de comunicaciones muy similares.

- Capítulo 4: Autooptimización en sistemas heterogéneos. A diferencia del capítulo anterior, aquí se van a estudiar las posibilidades de usar técnicas de optimización automática en el diseño de algoritmos iterativos paralelos que van a ser usados en sistemas heterogéneos, es decir, en sistemas con características computacionales y de comunicaciones diferentes.

- Capítulo 5: Metaheurísticas en el proceso de autooptimización. En este capítulo se estudia el uso de técnicas metaheurísticas en el mapeo óptimo de procesos a

procesadores, si es posible sin intevención humnana, y se analiza su uso en la fase de decisión dentro de la metodología de autooptimización introducida en el capítulo previo, comparando los resultados obtenidos tanto en simulaciones como en ejecuciones en sistemas reales.

- Capítulo 6: Conclusiones y trabajos futuros. Por último se muestran las conclusiones obtenidas como resultado del estudio del problema planteado, y se proponen nuevas vías de investigación para proseguir con el trabajo realizado.

1.8. Conclusiones

En este capítulo se ha hecho una introducción al problema que se trata en esta tesis. Se ha situado el problema explicando la importancia que tiene la optimización en el campo de la computación científica y cómo esta optimización ha supuesto un esfuerzo económico y temporal durante años para construir librerías cuyo código ha sido probado exhaustivamente. También hemos visto que dicho código ha sido diseñado considerando las características físicas de los sistemas en los que se va a emplear, sistemas que es de prever sean diferentes en el futuro. Por ello, en lugar de rehacer de nuevo todo el código desarrollado, en esta tesis se propone el desarrollo de técnicas de diversa naturaleza que permitan que, sin intervención de los usuarios finales, los sistemas informáticos sean capaces de adaptar el código optimizado. Ello tiene como principal ventaja que el usuario final no va a emplear tiempo en ajustar el software al sistema informático de que disponga en cada momento, con lo cual podrá dedicarse en exclusividad a su labor de investigación. Nos centramos en un tipo específico de algoritmos como son los iterativos y en concreto en aquellos que emplean la técnica de programación dinámica, que es muy empleada para resolver problemas científicos comunes, pero nuestra intención es poder generalizar las técnicas a desarrollar y poderlas aplicar a otros esquemas de programación. Se trata de un objetivo ambicioso pues conlleva el diseño y la construcción de software adaptativo que sea capaz de hacer reutilizable el software optimizado previamente sin mermar sus prestaciones iniciales.

Capítulo 2

Esquemas iterativos y Programación dinámica

En el capítulo anterior se ha planteado el interés de las técnicas de optimización y autooptimización para lograr reducir los tiempos de ejecución de códigos de elevado coste computacional sobre sistemas informáticos tanto homogéneos como heterogéneos. En este capítulo centramos el análisis en un tipo particular de algoritmos como son aquellos que siguen el esquema iterativo, habitualmente usados para resolver problemas científicos y académicos, que consisten en la ejecución repetitiva de un conjunto de instrucciones. Estudiaremos estos esquemas no sólo desde un punto de vista de ejecución secuencial sino también en su paralelización, tanto en sistemas de naturaleza homogénea como heterogénea. Profundizaremos en el desarrollo de metodologías que abarquen la heterogeneidad en la computación y en la comunicación de procesos. Además se pretende su extensión a otros esquemas diferentes.

Más aún, con el fin de estudiar estos esquemas y desarrollar las técnicas, usaremos como prueba de concepto los algoritmos de programación dinámica. La programación dinámica es una técnica algorítmica que ha sido ampliamente estudiada y aplicada a diferentes campos científicos como la investigación operativa, las ciencias de la computación, la biología, y generalmente, sus algoritmos se apoyan en esquemas iterativos.

2.1. Esquemas algorítmicos iterativos

Los esquemas algorítmicos son patrones básicos de resolución de problemas, y se estudian tanto en libros de programación secuencial [24, 30, 58] como paralela [7, 74, 133]. Constituyen un conjunto de esquemas de referencia para la resolución de una gama amplia de problemas reales, aunque en muchos casos la solución de un

problema no se consigue con la aplicación de un único esquema, sino que hay que realizar modificaciones de algún esquema básico o combinación de varios.

En el campo secuencial hay una serie de esquemas algorítmicos ampliamente difundidos y estudiados (divide y vencerás, avance rápido, programación dinámica, *backtracking*, ramificación y poda...) [24, 30], y lo mismo ocurre en computación paralela, donde se pueden estudiar versiones paralelas de los esquemas secuenciales, pero además hay también esquemas propiamente paralelos, como son *pipeline*, maestro-esclavo, bolsa de tareas, paralelismo/particionado de datos...

El estudio de esquemas algorítmicos permite realizar un análisis de mayor nivel que el desarrollado con algoritmos específicos. Se pueden estudiar características comunes a una gama de algoritmos que comparten el mismo esquema. Por ejemplo, se puede realizar un análisis general del tiempo de ejecución, obtener esquemas paralelos comunes a algoritmos que comparten el mismo esquema secuencial, analizar técnicas de optimización o de implementación comunes...

Además, se pueden desarrollar esqueletos que implementen algún esquema [71, 116], y el usuario del esqueleto debe completar únicamente algunas funciones de él para obtener el algoritmo que desea implementar siguiendo el esquema al que corresponde el esqueleto. Esto permite tener esqueletos donde el usuario sólo programa funciones secuenciales, obteniéndose así una paralelización transparente al usuario. También es posible incorporar en los esqueletos técnicas de optimización automática, con lo que se descarga al usuario de la toma de decisiones para obtener ejecuciones con tiempo de ejecución reducido.

En esta tesis nos centramos en un tipo particular de algoritmos o esquemas algorítmicos, los que siguen un esquema iterativo.

Entendemos como **esquema iterativo** aquel en que la ejecución se divide en un número de pasos que se repiten, y donde cada paso empieza tras haber acabado el anterior. Hay algunas variantes de este esquema básico. Por ejemplo, el número de iteraciones puede ser fijo o depender de un cierto criterio de parada, en cuyo caso tras cada iteración (o tras algunas iteraciones concretas) se comprueba el criterio de parada, y acaba la computación cuando éste se cumple. También puede haber variación en la computación en cada iteración, que puede ser constante o depender de la iteración en que se encuentre... Así, podemos tener varios esquemas algorítmicos iterativos y, aunque nos centremos en algunos esquemas y ejemplos concretos, la metodología de trabajo debe ser general a las variaciones del esquema básico.

Algunos de los algoritmos clásicos y aplicaciones científicas que se resuelven con esquemas iterativos son [24, 30, 58]:

- Los algoritmos de programación dinámica suelen seguir este esquema. Se resuelve un problema de un cierto tamaño basándose en soluciones de tamaños menores, que se han resuelto previamente. Así, cada decisión o grupo de decisiones a tomar se puede ver como un paso del proceso iterativo. Ejemplos clásicos de problemas que se resuelven por programación dinámica son el problema de la mochila, el de las monedas, el de la parentización óptima de la multiplicación de matrices, el análisis de secuencias de ADN [3, 92]...

- El algoritmo de Dijkstra de cálculo de caminos mínimos tiene un esquema iterativo: se usa para determinar el camino más corto desde un vértice origen al resto de vértices en un grafo dirigido y con pesos en cada arista. La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices. Cuando se obtiene el camino más corto desde el vértice origen al resto de vértices que componen el grafo, el algoritmo se detiene.

- Los algoritmos genéticos [47, 60, 77], así como la mayoría de metaheurísticas, trabajan en sucesivas iteraciones (generaciones), en las que se realiza siempre la misma computación: selección, cruce y mutación; y la computación acaba cuando se cumple algún criterio de convergencia. En general, los métodos metaheurísticos siguen el mismo esquema iterativo básico [117, 127].

- Muchos algoritmos numéricos siguen un esquema iterativo: multiplicación de matrices por el método de Cannon, resolución de sistemas de ecuaciones [13]... Este tipo de esquemas tiene gran importancia en computación científica, donde se resuelven en muchos casos problemas numéricos, por ejemplo, la iteración de Jacobi es un algoritmo de álgebra lineal usado para resolver sistemas lineales del tipo $Ax = b$. Esto se logra creando una iteración que converge finalmente a una aproximación al valor de x. La sucesión se crea desde un valor inicial $x(0)$, que generará el valor $x(n)$ (la aproximación a x). Matemáticamente vemos que A es una matriz que se descompone de la forma $A = D + L + U$, donde U es una matriz triangular superior, D una matriz diagonal y L la matriz triangular inferior.

- Muchas simulaciones se discretizan en el tiempo y en el espacio de manera que se obtienen los valores en los distintos puntos del espacio en un instante dado en función de los valores en puntos vecinos en el instante anterior, y utilizando las ecuaciones que representan la evolución temporal (ecuaciones diferenciales) para obtener los valores en cada punto. La relajación de Jacobi y el juego de la vida

son ejemplos básicos [7], pero hay multitud de aplicaciones de distintos campos que siguen este esquema: simulaciones climáticas [82], de contaminación [93], de cálculo de órbitas [67], simulación de la ecuación del calor [133], simulación de los *n-cuerpos* [133]...

En muchos de estos ejemplos los problemas a resolver son de un coste muy alto, lo que hace que sea interesante el desarrollo de algoritmos paralelos eficientes en sistemas de cómputo de altas prestaciones. Además, los usuarios que resuelven estos problemas, habitualmente, no son expertos en paralelismo, con lo que es conveniente desarrollar técnicas de autooptimización que les permitan un uso eficiente de los recursos de manera transparente. Así, el objetivo de este trabajo será desarrollar estas técnicas para esquemas iterativos y sobre sistemas paralelos. En las secciones siguientes analizaremos las ideas generales de los esquemas con los que se va a trabajar en el resto de capítulos de la tesis.

2.1.1. Esquemas secuenciales

Un esquema iterativo básico se muestra en el algoritmo 2.1. Como hemos indicado, se ejecutan sucesivas iteraciones hasta que se cumple un criterio de convergencia.

Algoritmo 2.1: Esquema iterativo básico.

```
while no se cumple condición de fin do
    computación
    actualizar condición de fin
end
```

Si suponemos un cierto número de iteraciones ($numiter$) y llamamos $t_{comp}(iter)$ al tiempo de computación y $t_{cond}(iter)$ al tiempo de actualización y comprobación de la *condición de fin* en cada iteración $iter$, el coste del tiempo de ejecución se puede representar por:

$$\sum_{iter=1}^{numiter} (t_{comp}(iter) + t_{cond}(iter)) \tag{2.1}$$

Que queda en la forma $numiter \cdot (t_{comp} + t_{cond})$ si los costes en las distintas iteraciones coinciden.

En algunos casos el coste de la actualización y comprobación de la condición es despreciable respecto al coste de la computación. En ocasiones no se realiza la comprobación porque hay un número fijo de iteraciones o puede que el número de iteraciones

en que se realiza la comprobación sea pequeño. En estos casos no es necesario incluir el término t_{cond}. Además, si la computación tiene el mismo coste en todas las iteraciones, el coste total es $numiter \cdot t_{comp}$.

Este esquema básico lo siguen algunos problemas típicos. Por ejemplo:

- En algoritmos genéticos [47, 60, 77] el coste de la evaluación suele ser muy pequeño (comparar dos valores) en relación con la computación, que tiene un coste alto de cruce, mutación y evaluación de las funciones de *fitness*; y el coste en las distintas iteraciones coincide.

- En los métodos iterativos de resolución de sistemas de ecuaciones [13] el coste de la computación es de orden $O(n^2)$ por iteración (multiplicación matriz-vector), mientras que la condición de fin se evalúa calculando la norma de un vector de diferencias, con orden $O(n)$.

La situación no siempre es tan simple. Por ejemplo, en algunos algoritmos de programación dinámica se suele rellenar una tabla por filas, y el número de iteraciones coincide con el número de filas (decisiones a tomar), pero el coste de cálculo de cada fila de la tabla puede variar en función del dato sobre el que se está operando, con lo que en la ecuación 2.1 los valores $t_{comp}(iter)$ son distintos para los distintos valores de $iter$. Esto ocurre por ejemplo en el problema de las monedas [58], que se utilizará en los capítulos posteriores, pero no ocurre en el de la mochila 0/1, pues en ese caso se toma el mínimo siempre de dos valores (incluir o no el objeto en la mochila).

En muchos algoritmos no se conoce a priori el número de iteraciones, por ejemplo, en la relajación de Jacobi, en los métodos iterativos de resolución de sistemas de ecuaciones, en muchas metaheurísticas... También es posible que la comprobación de la convergencia no sea de coste reducido. En la relajación de Jacobi [58] el coste de cada iteración es de orden $O(n^2)$, pues se evalúa el valor en cada punto de una malla $n \times n$ en función de sus vecinos; y la evaluación de la convergencia tiene el mismo coste, pues hay que comparar los valores en toda la malla con los de la iteración anterior. Así, puede ser preferible no evaluar la convergencia en las primeras iteraciones.

En ocasiones el coste de cada iteración no depende del número de iteración con un valor conocido, sino que depende de la forma de la entrada. Esto ocurre en el problema de las monedas, donde se calcula el mínimo de una serie de valores, que van desde no dar esa moneda hasta el número máximo de veces que se puede dar. Como este máximo depende de la cantidad que se esté devolviendo, N, de la cantidad de monedas del tipo i que estamos considerando, q_i, y del valor de la moneda, v_i, se tiene que el tiempo en la iteración i depende de la entrada, $\sigma = (N, q_i, v_i)$, y no del tamaño

de la entrada. En estos casos el coste de la computación en cada iteración es de la forma $t_{comp}(iter, \sigma)$.

En algunos problemas que se resuelven de manera iterativa es posible considerar el esquema básico de varias formas. Por ejemplo, en programación dinámica donde se rellena la tabla de subsoluciones hasta llegar a la solución del problema original, el bucle en el algoritmo 2.1 se puede considerar que recorre todos los subproblemas, con lo que la computación consiste en obtener el óptimo para un subproblema, o que recorre cada una de las decisiones que se toman (las filas de la tabla), y a cada decisión están asociados varios subproblemas, con lo que la computación corresponde al cálculo de los óptimos de todos los subproblemas asociados a esa decisión (una fila de la tabla).

A la hora de obtener esquemas paralelos suele interesar considerar esquemas con computación de grano grueso, pues en caso contrario las comunicaciones o sincronizaciones tendrían mucho peso en el tiempo total de ejecución. Por esta razón introducimos la variable *granularidad* que se empleará para dar mayor o menor peso a la computación en relación a las comunicaciones dentro de la ejecución del algoritmo.

Un ejemplo de esto lo encontramos en los algoritmos genéticos paralelos con esquema de islas [95], donde cada proceso itera localmente sobre la subpoblación que contiene, y la migración entre procesos se realiza cada cierto número de generaciones. Al aumentar el número de generaciones entre migraciones se aumenta la granularidad de la computación, aunque se puede ralentizar la convergencia (aumentar el número de iteraciones) al trabajarse durante más tiempo con poblaciones locales.

En esta tesis nos centramos en el comportamiento de esquemas paralelos iterativos. Dado que hay distintas posibilidades para estos esquemas, su comportamiento y la metodología de optimización pueden variar dependiendo del sistema de cómputo al que están destinados. En este trabajo consideramos dos tipos de modelos de cómputo paralelos, los sistemas homogéneos y los heterogéneos (cuyas características han sido explicadas en el capítulo anterior). En la siguiente sección comentaremos las ideas generales de los esquemas iterativos paralelos homogéneos, y en las sucesivas los esquemas heterogéneos y las nociones básicas de ejecución de esquemas homogéneos en sistemas heterogéneos. En los restantes capítulos de esta tesis se estudian con algunos ejemplos teórica y experimentalmente las técnicas de optimización propuestas para estos esquemas en los distintos tipos de sistemas.

2.1.2. Esquemas paralelos homogéneos

Un esquema iterativo paralelo básico se obtiene añadiendo al esquema secuencial (algoritmo 2.1) una parte de comunicación o sincronización tras cada iteración, con lo que se obtiene el esquema del algoritmo 2.2, donde p es el número de procesos que se van a emplear en su resolución. En programación con espacio de direcciones compartido habrá sincronización, y mediante paso de mensajes lo que tendremos serán comunicaciones, que aseguran además la sincronización.

Algoritmo 2.2: Esquema iterativo paralelo básico.

```
forall P_i, i = 0, 1, ..., p − 1 do
    while no se cumple condición de fin do
        computación
        comunicación o sincronización
        actualizar condición de fin
    end
end
```

Así, la ecuación del tiempo de ejecución 2.1 se transforma en:

$$\sum_{iter=1}^{numiter} (t_{comp}(iter) + t_{comm}(iter) + t_{cond}(iter)) \tag{2.2}$$

donde $t_{comm}(iter)$ corresponde al tiempo de comunicación o sincronización tras cada paso de computación. Si los costes de las distintas iteraciones coinciden queda $numiter \cdot (t_{comp} + t_{comm} + t_{cond})$.

La comprobación de la condición de fin incluye normalmente una parte de comunicación además de una de computación, ya que el criterio de convergencia será global a todos los procesos. En este caso el coste por iteración del algoritmo se descompone en dos partes de computación con dos partes intercaladas de comunicación.

Igual que en los esquemas secuenciales, en algunos casos el coste de la actualización y comprobación de la condición es despreciable respecto al coste de computación y comunicación. En ocasiones, no se realiza la comprobación porque hay un número fijo de iteraciones, o puede que el número de iteraciones en que se realiza la comprobación sea pequeño. En todos estos casos puede no incluirse el término t_{cond}. Además, si las partes de computación y las de comunicación tienen el mismo coste en todas las iteraciones, el coste total es $numiter \cdot (t_{comp} + t_{comm})$.

Algunos de los ejemplos comentados en el caso secuencial siguen este esquema básico:

- En el algoritmo de Cannon para la multiplicación de matrices [7, 74] las matrices de dimensión $n \times n$ se distribuyen en una malla de $r \times r$ procesos ($p = r^2$), con un bloque de tamaño $\frac{n}{r} \times \frac{n}{r}$ de cada matriz en cada proceso. Suponiendo que las matrices están distribuidas inicialmente de la manera apropiada (si no lo están se tiene un coste inicial de distribución de los datos), el algoritmo trabaja en r pasos. En cada paso cada proceso multiplica los dos bloques de las matrices que tiene, con coste $2\left(\frac{n}{r}\right)^3$, y a continuación envía un bloque a la fila de procesos anterior y otro a la columna de procesos anterior (las filas y columnas de procesos forman una topología lógica de toro), con lo que el coste de las comunicaciones es $4\left(t_s + \left(\frac{n}{r}\right)^2 t_w\right)$ (fase de diseño de la figura 1.2) si suponemos que las comunicaciones se realizan en paralelo y que cada proceso interviene en cuatro comunicaciones, dos envíos y dos recepciones. Tras el último paso de computación no hay comunicación. Así, el coste es:

$$2\frac{n^3}{r^2} + 4(r-1)t_s + 4n^2\frac{r-1}{r^2}t_w \tag{2.3}$$

- En los algoritmos genéticos paralelos [95] no conocemos el número de iteraciones (*numiter*) a priori, ya que se acabará cuando se cumpla una condición de convergencia, que normalmente consiste en que no se haya mejorado la mejor solución en un determinado número de iteraciones. Consideramos el esquema de islas y que se realizan migraciones cada determinado número de generaciones (g). Con estas suposiciones el coste se puede modelar como:

$$\frac{numiter}{g} \cdot \left(g\left(t_{cruc} + t_{muta}\right) + t_{migr} + t_{cond}\right) \tag{2.4}$$

  donde suponemos que el coste computacional en cada iteración viene dado por el coste del cruce de individuos (t_{cruc}) y el de mutación (t_{muta}), y llamamos t_{migr} al coste de migración, que será un coste de comunicación y cuyo valor depende de la forma en que ésta se haga: si se migra un gran número de individuos y se transfieren a muchas islas el coste de comunicación será grande, pero puede que la convergencia sea rápida al compartir más información las poblaciones; y si se migran pocos individuos o entre grupos localizados de procesos la comunicación será menos costosa pero la convergencia puede ser más lenta. El término t_{cond} corresponde a la comprobación de la condición de fin, que se puede realizar tras cada iteración o aprovechar las migraciones para llevar a cabo las comunicaciones necesarias.

En el modelo aparecen parámetros del sistema que representan el coste de operaciones aritméticas básicas y de las comunicaciones. En las ecuaciones 2.3 y 2.4 no hemos incluido el coste de las operaciones aritméticas básicas, pero aparecería multiplicando al número de operaciones. Por ejemplo, en los algoritmos géneticos: $t_{cruc} = t_c \cdot n_{cruc}$, con t_c el coste de una operación básica y n_{cruc}, el número de operaciones básicas en el cruce.

Comentamos brevemente cuáles serían los parámetros de computación y cómo se estimarían en los dos ejemplos anteriores:

- En la ecuación 2.3 tendríamos $2\frac{n^3}{r^2}t_c$, donde t_c es el parámetro que representa el coste de una operación aritmética básica, que en este caso es el de una suma o multiplicación de números reales (flop). Este valor puede estimarse en la instalación de la rutina simplemente ejecutando secuencialmente una versión reducida de la operación que se está aplicando, que es la multiplicación de matrices.

- En el algoritmo genético hay varios tipos de operaciones aritméticas básicas, pues se seleccionan individuos aleatoriamente, se calcula la función de *fitness*, se cruzan los individuos..., y las operaciones aritméticas en la comprobación de la condición de fin también son diferentes. Así, aparecerían parámetros aritméticos distintos para distintas operaciones, y estos dependen también del algoritmo genético concreto y del problema al que se está aplicando. En vez de identificar todos los parámetros y estimarlos por separado puede ser preferible ejecutar secuencialmente una iteración del algoritmo e identificar el parámetro que afecta a la computación: $(n_{cruc} + n_{muta}) \cdot t_{c_{iter}}$, y hacer lo mismo con la condición de la comprobación de la convergencia: $n_{cond} \cdot t_{c_{cond}}$.

En cuanto a la estimación de los parámetros de comunicación, si se modelan mediante el modelo lineal clásico $t_s + m \cdot t_w$, la estimación de los valores de t_s y t_w mediante un *ping-pong* podría no ser adecuada al no reflejar el tipo de comunicaciones que se realizan en el algoritmo. En los dos ejemplos que estamos considerando se podría proceder de la siguiente forma:

- En el algoritmo de Cannon se puede realizar en la instalación una ejecución de la parte de comunicación del algoritmo, para varios tamaños de matrices, y hacer un ajuste para obtener los valores de t_s y t_w en la ecuación 2.3.

- En el algoritmo genético también se pueden ejecutar en la instalación las partes de comunicación en la migración y en la comprobación de la condición de fin, y estimar así los valores de t_s y t_w que aparecen en los términos t_{migr} y t_{cond}.

En los capítulos posteriores se analizará la estimación de los parámetros más detalladamente, estudiando el modelo teórico y describiendo la fase de instalación para varios ejemplos, y analizando teórica y experimentalmente la utilización de distintas técnicas de estimación.

El parámetro algorítmico a decidir (figura 1.2) será el número de procesadores a usar ($P = r^2$ en la ecuación 2.3). Este parámetro se decide en tiempo de ejecución: a partir del modelo y los parámetros del sistema obtenidos en la instalación, y del tamaño del problema a resolver, se obtiene el número de procesadores con que se obtiene el menor valor del tiempo de ejecución teórico, y se ejecuta con ese número de procesadores (estamos considerando un proceso por procesador).

Como hemos indicado en el caso secuencial, el modelo de tiempo de ejecución puede ser más complicado:

- Si iteraciones distintas tienen costes diferentes no es posible estimar los parámetros con la ejecución de una iteración o de una única comunicación y será necesario identificar las operaciones que se ejecutan en cada iteración, para ejecutar en la instalación las funciones correspondientes a cada una de las funciones identificadas. Si el tipo de operaciones se repite, aunque los costes sean distintos, bastará con ejecutar una iteración para identificar los parámetros, aunque en la fórmula los costes dependan del número de iteración ($t_{comp}(iter)$, $t_{comm}(iter)$ y $t_{cond}(iter)$).

- Puede que el coste de la computación o de las comunicaciones sea distinto en procesadores distintos, y además que varíe con el número de iteración. En este caso, los valores de cada operación en la ecuación 2.2 serán aquellos que se obtienen en el procesador que más tarda en ella, quedando la ecuación en la forma:

$$\sum_{iter=1}^{numiter} \left(\max_{i=0,\ldots,P-1} \{t_{comp}(iter,i)\} + \max_{i=0,\ldots,P-1} \{t_{comm}(iter,i)\} + \max_{i=0,\ldots,P-1} \{t_{cond}(iter,i)\} \right) \tag{2.5}$$

  donde por cada operación e iteración se contabiliza el tiempo máximo de entre todos los procesadores. Es posible que haya solapamiento entre computación y comunicación, lo que no se contempla en la fórmula.

  El problema de optimización a resolver será obtener el número de procesadores (determinar P) de todos los disponibles en el sistema con el que la fórmula anterior tenga valor mínimo.

2.1.3. Esquemas paralelos heterogéneos

Consideramos en esta sección el modelo de cómputo heterogéneo y los esquemas iterativos sobre este modelo.

El esquema básico de un algoritmo iterativo para un sistema heterogéneo coincide con el homogéneo (algoritmo 2.2): se llevan a cabo sucesivas iteraciones hasta que se cumple un criterio de convergencia, y en cada iteración hay una parte de computación seguida de otra de comunicación. La diferencia en el caso heterogéneo es que la velocidad de computación y de comunicación en los distintos procesadores varía, lo que debe tenerse en cuenta para optimizar el tiempo de ejecución, y conlleva el diseño de algoritmos heterogéneos, en los que se asigna un volumen de datos distinto a cada procesador, a diferencia del caso homogéneo donde todos los procesos trabajan con el mismo volumen.

El modelo de tiempo será similar al homogéneo. Si en la ecuación 2.2 suponemos que el coste en cada iteración es el mismo queda: $numiter \cdot (t_{comp} + t_{comm} + t_{cond})$. Nos centramos en el coste de una única iteración, e incluimos en la computación y la comunicación las partes de computación y de comunicación que se realizan en la comprobación de la condición de fin, con lo que hay que estimar sólo dos términos. En el caso homogéneo puede que el coste de la computación sea el mismo en cada proceso, aunque también puede variar, en cuyo caso se modela el tiempo con el máximo de todos los procesadores, como se hace en la ecuación 2.5. En el caso heterogéneo el coste en cada procesador varía incluso aunque se asigne el mismo volumen de datos a cada procesador. Como se asigna distinto volumen de datos a cada procesador, contabilizamos en cada uno el número de operaciones aritméticas básicas (llamamos $n_{comp}(i)$ al número de operaciones en el procesador i) y multiplicamos ese número por el coste de una operación aritmética básica en cada procesador (t_{c_i}), con lo que modelamos el coste aritmético en el procesador i por $n_{comp}(i) \cdot t_{c_i}$, y el coste de computación en una iteración será el máximo de los costes en todos los procesadores:

$$\max_{i=0,\ldots,P-1} \{n_{comp}(i) t_{c_i}\} \tag{2.6}$$

Para modelar las comunicaciones se puede proceder de forma similar. Una posibilidad es considerar el número de operaciones de inicio de comunicaciones ($n_{inicom}(i,j)$) y el número de datos transferidos entre dos procesos ($n_{dattra}(i,j)$), y que el coste de las comunicaciones viene dado por el máximo de los inicios de comunicaciones y de datos transferidos:

$$\max_{i=0,\ldots,P-1;\ j=0,\ldots,P-1} \left\{ n_{inicom}(i,j) t_{s_{ij}} \right\} \tag{2.7}$$

y

$$\max_{i=0,\ldots,P-1;\ j=0,\ldots,P-1} \left\{ n_{dattra}(i,j) t_{w_{ij}} \right\} \tag{2.8}$$

donde para cada par de procesos se considera el número de inicio de comunicaciones entre esos dos procesos multiplicado por el coste de inicio de una comunicación entre dichos procesos, y se toma como coste total de inicio de comunicación en el sistema el máximo entre pares de procesos. De la misma forma se hace para la transferencia de datos.

De este modo el modelo de tiempo queda:

$$\sum_{iter=1}^{numiter} \left(\max_{i=0,\ldots,P-1} \{ n_{comp}(i) t_{c_i} \} + \max_{i=0,\ldots,P-1;\ j=0,\ldots,P-1} \left\{ n_{inicom}(i,j) t_{s_{ij}} \right\} + \max_{i=0,\ldots,P-1;\ j=0,\ldots,P-1} \left\{ n_{dattra}(i,j) t_{w_{ij}} \right\} \right) \tag{2.9}$$

También es posible usar otros modelos, pero la metodología de trabajo sería la misma. Por ejemplo, en la comunicación se puede considerar que un procesador realiza las comunicaciones una tras otra, con lo que los tiempos de inicio de comunicaciones y de envío de datos desde él no se solapan, y la fórmula quedaría:

$$\sum_{iter=1}^{numiter} \left(\max_{i=0,\ldots,P-1} \{ n_{comp}(i) t_{c_i} \} + \max_{i=0,\ldots,P-1} \left\{ \sum_{j=0}^{P-1} \left(n_{inicom}(i,j) t_{s_{ij}} + n_{dattra}(i,j) t_{w_{ij}} \right) \right\} \right) \tag{2.10}$$

Con la distribución de los datos en el sistema se intentará que el trabajo esté equilibrado, con lo que los valores de los que se tomen esos máximos deberían coincidir o estar muy cerca entre los distintos procesadores. Además, puede que el equilibrio de la computación afecte al de las comunicaciones, con lo que no es suficiente realizar una distribución del trabajo de manera que $n_{comp}(i) t_{c_i} = n_{comp}(j) t_{c_j},\ \forall\ i, j$. En algunos casos, cuando el coste de las comunicaciones es mucho menor que el de la computación se considera sólo el coste de computación y se decide una distribución que proporcione la igualdad anterior.

En general se puede plantear un problema de optimización que considera todas las posibles asignaciones de datos a los procesadores y de todas ellas tomar aquella que proporciona el menor valor según el modelo que se esté utilizando. Por ejemplo, con la ecuación 2.10 el problema de optimización sería:

$$\min_{\sigma \in Asignaciones} \left\{ \sum_{iter=1}^{numiter} \left(\max_{i=0,\ldots,P-1} \{ n_{comp}(i,\sigma) t_{c_i} \} + \right. \right.$$
$$\left. \left. \max_{i=0,\ldots,P-1} \left\{ \sum_{j=0}^{P-1} \left(n_{inicom}(i,j,\sigma) t_{s_{ij}} + n_{dattra}(i,j,\sigma) t_{w_{ij}} \right) \right\} \right) \right\} \tag{2.11}$$

Usamos como ejemplo el esquema de islas de los algoritmos genéticos. El tamaño de la subpoblación que se asigna a cada procesador puede ser proporcional a la velocidad relativa del procesador: si un procesador tiene velocidad 1 y otro velocidad 2, el segundo trabajará con una población con el doble de individuos que el primero. Como el coste es proporcional al tamaño de la población, esta asignación puede asegurar equilibrio en la computación dependiendo de cómo se realicen las mutaciones. Si el número de generaciones entre migraciones es grande no compensa preocuparse por el coste de las comunicaciones, pero si hay un número reducido de generaciones entre migraciones, las comunicaciones tienen importancia en el tiempo de ejecución, y se puede decidir que desde procesadores con menor capacidad de comunicación migren menos elementos que de otros con mayor velocidad de comunicación. Los tamaños, y por tanto los costes, de la computación y de la comunicación en cada procesador son independientes y pueden decidirse estos tamaños por separado, simplificándose la obtención de distribuciones adecuadas.

Otros algoritmos son de más difícil adaptación a sistemas heterogéneos. Esto ocurre con el algoritmo de Cannon y con otros algoritmos de álgebra lineal, donde hay un esquema de comunicaciones en malla de procesos, y el tamaño de las comunicaciones viene dado por el volumen de datos asignado a cada proceso. El problema de obtener asignaciones óptimas no es una tarea fácil, y hay muchos trabajos de sintonización de rutinas básicas de álgebra lineal a sistemas heterogéneos [17, 19, 32, 44, 88].

Discutimos brevemente cómo se realizaría la estimación de los parámetros del sistema, SP, en un entorno heterogéneo (fase de instalación de la figura 1.2). Consideramos sistemas heterogéneos de dos tipos:

- **Estáticos**. Las capacidades de computación y comunicación de los distintos componentes son distintas pero no varían con el tiempo. En este caso los parámetros del sistema se pueden estimar en la instalación de la rutina. Los parámetros de computación (el vector t_c) se pueden estimar a partir de las velocidades relativas de los procesadores, o se puede resolver en secuencial en cada uno de los procesadores una versión reducida del problema.

Es probable que para realizar la estimación de la parte de comunicaciones y debido a las dificultades que esto entraña se deban utilizar diferentes posibilidades. Se puede utilizar los valores de t_s y t_w entre cada dos procesadores obtenidos por medio de un *ping-pong*. Como veremos en los capítulos siguientes, esta opción puede no dar resultados satisfactorios, y puede ser preferible ejecutar una versión reducida de la rutina de comunicación que se ejecuta en el algoritmo. El principal problema es que el coste de esta rutina puede venir determinado por la asignación de datos que se haya hecho a los procesadores, y por lo tanto la rutina puede ejecutarse de muchas formas distintas pudiéndose obtener prestaciones distintas. Una posibilidad consiste en realizar experimentos con varias configuraciones y tomar la media de los parámetros o almacenar los parámetros de comunicación como una tabla en función de algunas configuraciones básicas. Esta dificultad en la estimación de los parámetros de comunicaciones hace que en una gran parte de los trabajos de optimización en entornos heterogéneos se estudie el caso simplificado que no incluye comunicaciones, considerándose éstas de mucho menor coste que las computaciones, cosa que ocurre en muchas de las aplicaciones que es interesante resolver en paralelo.

- **Dinámicos**. Son sistemas de carga variable en que la capacidad de los componentes varía con el tiempo. Esto puede ocurrir tanto si el sistema es homogéneo como si es heterogéneo, por ejemplo si es compartido por varios usuarios que envían sus trabajos de forma concurrente. En este caso los valores de los parámetros obtenidos en la instalación de la rutina pueden no representar las condiciones del sistema en el momento de la ejecución. Habría que estimar los parámetros en la ejecución, pero el tiempo de estimación debe ser reducido pues supone una sobrecarga al tiempo de resolución del problema. Otra posibilidad es realizar experimentos exhaustivos en la instalación para obtener unos valores de los parámetros que se modificarán en el momento de la ejecución con unas ejecuciones reducidas [36, 114] o con valores del comportamiento actual del sistema obtenidos con alguna herramienta de monitorización tipo NWS [108].

Estamos considerando algoritmos heterogéneos en los que se asigna un proceso por procesador pero donde cada proceso trabaja con un volumen de datos distinto. En este caso el parámetro algorítmico a determinar será el tamaño del bloque de datos que se asigna a cada procesador: $b = (b_0, \ldots, b_{P-1})$, con lo que tenemos tantos parámetros como número de procesadores. En el momento de la ejecución (figura 1.2) se utiliza el modelo de tiempo, con los parámetros del sistema sustituidos en él y con los valores que representan el tamaño del problema que se está resolviendo, para

obtener los tamaños de los bloques de datos con que se obtiene el menor valor del tiempo teórico, y se resuelve el problema con esa distribución.

El número de parámetros a decidir es grande si el sistema lo es, y el número de posibles valores de los parámetros también es amplio. Por tanto, el problema que se plantea no es simple, y puede que no sea posible resolverlo de manera óptima en tiempo de ejecución debido a la sobrecarga que supondría. Una alternativa que se usa normalmente y que se estudiará en este trabajo consiste en usar aproximaciones heurísticas o metaheurísticas (fase de ejecución de la figura 1.2) de manera que aunque no se obtenga la solución óptima, sí una cercana a ella, y sea posible realizar una ejecución con tiempo reducido y con sobrecarga mínima de toma de decisión. Otra alternativa es iniciar la ejecución con alguna distribución por defecto (por ejemplo, asignando a todos los procesadores el mismo volumen de datos) y utilizar alguna técnica de sintonización para modificar la distribución en iteraciones sucesivas usando la información sobre tiempos de computación y de comunicación obtenidos en las iteraciones iniciales [53].

Del mismo modo que en el caso secuencial y homogéneo, podemos tener un esquema en el que los costes de las distintas iteraciones sean diferentes y por tanto sería conveniente obtener tamaños de bloques de datos a distribuir en función del número de iteración. En este caso se complica el problema de optimización, ya que el número de parámetros es ahora $numiter \cdot P$.

2.1.4. Ejecución de esquemas homogéneos en sistemas heterogéneos

Una posibilidad para resolver problemas eficientemente en un sistema heterogéneo es el desarrollo de algoritmos heterogéneos tal como se ha comentado en la sección anterior. El desarrollo de estos algoritmos es más complejo que el de los algoritmos homogéneos correspondientes, y hay una gran cantidad de trabajos en este campo [17, 18, 44, 83, 87].

Otra posibilidad consiste en utilizar algoritmos homogéneos (se consideran todos los procesos con el mismo volumen de computación) y utilizar alguna técnica para asignar los procesos lógicos del algoritmo homogéneo a los procesadores físicos del sistema heterogéneo [83]. Se pueden utilizar estrategias generales [28] o estrategias específicas para distintos esquemas algorítmicos. En esta tesis analizamos estrategias para esquemas iterativos.

Con la primera aproximación se pueden obtener algoritmos eficientes, pero con un coste alto de reprogramación de algoritmos homogéneos clásicos. Para que la segunda

aproximación sea una buena alternativa es necesario obtener una buena estrategia de distribución (una estrategia con un coste de ejecución reducido y con la que se obtenga un buen equilibrio de la carga). El problema de asignación es de gran dificultad, por lo que no es posible obtener la solución óptima en un tiempo reducido. Normalmente es preferible utilizar alguna heurística (fase de ejecución de la figura 1.2) que se adapte al problema o alguna técnica metaheurística. En este trabajo se analiza la aproximación de utilizar técnicas metaheurísticas para abordar el problema de encontrar la solución óptima.

Una posible aproximación heurística inicial podría consistir en considerar un esquema de algoritmo homogéneo asignando sólo un proceso por procesador y decidir el número de procesadores a usar. Se ordenan los procesadores de más rápidos a más lentos y se obtiene el tiempo teórico con un proceso (en el procesador más rápido), con dos procesos (en los dos más rápidos)..., hasta que el valor del modelo teórico del tiempo aumenta. Se resuelve el problema con el número de procesadores con el que se obtiene el menor tiempo teórico. En algunos casos se obtienen resultados satisfactorios si se han estimado bien los valores de los parámetros del sistema, y en sistemas con carga variable si se estiman bien los valores en el momento de la ejecución con una sobrecarga mínima [36].

Para un sistema heterogéneo puede ser preferible asignar un número de procesos distinto a cada procesador dependiendo de su capacidad de cómputo y de la de comunicación de la red. Así, el tamaño del bloque de computación sería fijo para todos los procesos, y el nuevo parámetro algorítmico que aparece es un vector donde se almacena el número de procesos que se asigna a cada procesador ($d = (d_0, \ldots, d_{P-1})$). El número total de procesos viene dado por $p = \sum_{i=0}^{P-1} d_i$, y en principio puede ser ilimitado aunque para cada problema habrá un límite que puede venir establecido por el tamaño de problema. Los parámetros algorítmicos a determinar serían los propios del problema junto con p y d. En este caso también el problema general de optimización del modelo de tiempo variando los parámetros algorítmicos tiene alta complejidad computacional y se deben usar aproximaciones heurísticas o metaheurísticas, tal como analizaremos en los capítulos siguientes.

2.2. Esquemas iterativos de programación dinámica

En esta tesis se pretende analizar y diseñar diversos métodos de optimización que se apliquen sobre los diferentes esquemas iterativos con el fin de reducir su tiempo de ejecución. De entre los problemas iterativos existe una variante de aquellos que

emplean la denominada técnica de programación dinámica. La programación dinámica es una importante técnica de resolución de problemas que ha sido ampliamente usada en campos tan diversos como la teoría de control, la investigación operativa, la biología o las ciencias de la computación. Se trata de un método que habitualmente se enmarca entre las técnicas enumerativas de resolución de problemas. La técnica de programación dinámica se utiliza con frecuencia cuando la solución a un problema puede ser vista como el resultado de una secuencia de decisiones. Muchos de estos problemas necesitan una enumeración completa de las secuencias de decisiones y se elige la mejor de ellas. En esta técnica se obtiene la secuencia óptima haciendo uso explícito del principio de optimalidad establecido por Bellman, según el cual cualquier subsecuencia de una secuencia óptima debe ser también óptima. En ocasiones, la programación dinámica reduce drásticamente el espacio de búsqueda descartando para su consideración algunas secuencias de decisiones que no pueden ser óptimas. La resolución de problemas mediante esta técnica se caracteriza porque los datos en cada momento se van calculando a partir de datos calculados previamente, que deben se guardados, conforme a un proceso iterativo. Gráficamente, se puede representar la resolución de este tipo de algoritmos con una tabla (o con una serie de variables) en la cual los valores de las filas, columnas o diagonales se calculan a partir de las filas, columnas o diagonales calculadas anteriormente, tal y como se ve en las figuras 2.2, 2.3 y 2.4 respectivamente, que serán explicadas con detalle en la siguiente sección. Una cuestión importante es definir la/s tabla/s que son utilizadas por el algoritmo, y sobre todo la forma en que son rellenada/s para recomponer la solución final a partir de los datos que se han ido almacenado en ella/s durante el proceso computacional. El dato que permite solucionar el problema suele hallarse en la última celda de la última columna de la tabla, por lo que, generalmente, para obtener la solución habrá que calcular previamente toda la tabla.

Desde que el método fuera originalmente presentado por Bellman en la década de los 50, han sido presentadas diversas formalizaciones, por ejemplo las presentadas en [76, 84] en las que se establece el principio de optimalidad como una condición de monotonicidad. Aproximaciones más genéricas las encontramos en [40, 79], mientras en el primer caso se aborda el problema desde la perspectiva de la teoría de autómatas en el segundo caso se hace uso del paradigma de la programación funcional.

En términos generales podemos afirmar que para poder aplicar la técnica, hay que definir un conjunto de ecuaciones recurrentes, las ecuaciones funcionales de la programación dinámica, que permiten expresar la solución a problemas grandes en función de los pequeños. Como en todo tratamiento recurrente, aunque no se hace de forma recursiva, hay que definir los casos base que van a permitir la ejecución de las

instrucciones recurrentes. Es precisamente esta ecuación funcional la que determina la dimensión de la tabla de programación dinámica e impone ciertas restricciones al recorrido de la tabla en función de las dependencias de datos. En función del tipo de recurrencia, podemos encontrar los problemas con recurrencias simples de tipo monádico multietapa como los recogidos en [79] o problemas con recurrencias no monádicas y no multietapa como los que se muestran en [54, 91, 130]. Las recurrencias multietapa suelen permitir recorridos por filas o columnas de la tabla de programación dinámica, mientras los problemas no multietapa suelen admitir recorridos por diagonales.

Mostramos a continuación algoritmos clásicos y aplicaciones que se resuelven mediante programación dinámica. Alguno se utilizará como prueba de concepto de la metodología propuesta en esta tesis:

- **El problema de la mochila 0/1:** Se trata de un problema clásico en programación dinámica [24] y que se usa frecuentemente como ejemplo con el que ilustrar la técnica. Consiste en introducir un cierto número de objetos, cada uno de ellos lleva asociado un peso y un beneficio, en una mochila de tal forma que se maximice el beneficio de los objetos transportados.

  La aproximación de programación dinámica puede formularse haciendo uso de las siguientes variables que describen el problema:

  - M = Capacidad de la mochila
  - n = Número de objetos disponibles
  - w_i = Peso del objeto i, $1 \leq i \leq n$, $w_i > 0$
  - v_i = Beneficio obtenido al emplear el objeto i, $1 \leq i \leq n$, $v_i > 0$
  - $Mochila[i, m]$ = Beneficio obtenido considerando sólo los i primeros objetos (de los n originales) con una capacidad de la mochila m
  - $Mochila$ = Tabla de tamaño $n \times M$ donde se irán guardando los cálculos
  - $x_i = 0/1$. Selección ($x_i = 1$) o no ($x_i = 0$) del objeto para introducirlo en la mochila
  - $\sum_{i=1}^{n} x_i \, v_i$ devuelve el valor de beneficio total

  Se pueden usar dos tablas. La primera contiene el valor del cálculo realizado, tabla $Mochila$. La segunda permite reconstruir la solución final a partir del último valor, es decir, qué objetos se van a seleccionar para introducirlos en la mochila.

Los casos base del problema son:

- Si $i < 0$ o $M < 0$ entonces no hay solución: $Mochila[i, M] = -\infty$, pues se aplica un máximo
- Si $i = 0$ o $M = 0$, la solución es no incluir ningún objeto: $Mochila[i, M] = 0$

Es decir, cuando la capacidad de la mochila sea negativa el beneficio obtenido será un mínimo con el fin de mantener la solución parcial actual al aplicar el máximo, como vemos a continuación.

Podemos definir el problema de forma recurrente, en función de que en cada paso se use o no el objeto i:

- Si no se usa el objeto i: $Mochila[i, M] = Mochila[i - 1, M]$
- Si se usa: $Mochila[i, M] = v_i + Mochila[i - 1, M - w_i]$

donde $Mochila[i, j]$ contiene el beneficio máximo usando los i primeros objetos y peso j.

Se trata de calcular $Mochila[n, M]$ y para ello la ecuación recurrente sería:

$$Mochila[i, j] = \text{máx}\{Mochila[i - 1, j], v_i + Mochila[i - 1, j - w_i]\}, \\ 1 \leq i \leq n, 1 \leq j \leq M \quad (2.12)$$

de modo que $Mochila[n, M]$ contiene la solución al problema que considera los n objetos y la capacidad total de la mochila M.

Para rellenar la tabla habría que inicializar los casos base y para todo i, desde 1 hasta n, y j desde 1 hasta M aplicar la ecuación de recurrencia 2.12.

Se puede disponer de una tabla auxiliar cuyos valores 0/1 (0 para indiciar que no se elige el objeto correspondiente y 1 que sí) sirvan para almacenar las decisiones que se han ido tomando en cada iteración, y a partir de la cual se obtiene la solución final. Otra opción, es calcular las decisiones tomadas a partir del valor óptimo $Mochila[n, M]$ y analizando las decisiones que se tomaron para cada objeto i hasta obtener la solución, obteniéndose así el vector de decisiones (x_1, x_2, ..., x_n) de la siguiente forma:

- Si $Mochila[i, j]$ = Mochila[i-1,j] entonces la solución no usa el objeto i, $x_i = 0$

- Si $Mochila[i,j]$ = Mochila[i-1,j-w_i] + v_i entonces sí se usa el objeto i, $x_i = 1$
- Si $Mochila[i,j]$ = Mochila[i-1, j-w_i] + v_i y $Mochila[i,j] = Mochila[i-1,j]$ entonces se puede usar el objeto i o no (existe más de una solución óptima)

Se acaba el proceso computacional cuando se llega a $i = 0$ o a $j = 0$. Tal y como se ve, la forma de resolver el problema es a través de un cálculo recurrente de datos que se van guardando en una tabla y cuyo último valor permitirá calcular la solución final.

Obsérvese que se trata de un problema multietapa puesto que en la ecuación funcional la dependencia de datos se produce entre filas consecutivas de la tabla. En este caso el recorrido natural en la tabla sería por filas, aunque las dependencias de datos permitirían también realizar recorridos por columnas o diagonales.

- **El problema del cambio de monedas:** Se trata también de un problema clásico de la programación dinámica [24] que puede ser considerado como una variante del problema anterior. Consiste en devolver una cantidad de dinero empleando el menor número posible de monedas de entre un surtido de monedas existentes de distintos valores.

  El problema puede ser abordado desde diversas perspectivas, por ejemplo, considerar una moneda de cada tipo, un número determinado o un número ilimitado de monedas de cada tipo, etc. En primer lugar se mostrará la nomenclatura utilizada para resolver el problema que nos ocupa haciendo uso de la técnica de programación dinámica. El problema puede modelarse haciendo uso de las siguientes variables:

  - N = Cantidad total a devolver
  - n = Número de tipos de monedas
  - v_i = Valor de la moneda de tipo i, $1 \leq i \leq n$, $v_i > 0$
  - q_i = Cantidad de monedas de tipo i disponibles, $1 \leq i \leq n$, $q_i > 0$
  - c = Tabla de tamaño $n \times N$ donde se irán guardando los cálculos
  - $c[i,j]$ = Mínimo número de monedas para devolver la cantidad j utilizando monedas de los tipos 1 hasta i
  - $c[n,N]$ = Solución del problema completo a partir de la cual se reconstruye la solución final

Se pueden usar dos tablas. La primera contiene el valor del cálculo realizado, tabla c. La segunda permite reconstruir la solución final a partir del último valor, es decir cuántas monedas de cada tipo se han de utilizar para devolver la cantidad exacta.

Los casos base del problema son:

- $c[i, j] = 0$, si $i > 0$ y $j = 0$,
- $c[i, j] = \infty$, si $i \leq 0$ o $j < 0$,

Es decir cuando la cantidad a devolver (j) es 0 el número de monedas a devolver es 0. Cuando no se utilice ningún tipo de moneda (i), es decir, cuando nos posicionamos en una columna fuera de la tabla, el mínimo número de monedas a utilizar será infinito con el fin de mantener la solución parcial actual al aplicar el mínimo, como vemos a continuación.

Se trata de calcular $c[n, N]$, y para ello la ecuación recurrente, considerando una cantidad ilimitada de monedas de cada tipo, sería:

$$c[i, j] = \min_{k=0,1,\ldots,\ \left\lfloor \frac{j}{v_i} \right\rfloor} \{c\left[i-1, j-k \cdot v_i\right] + k\}, 1 \leq i \leq n, 1 \leq j \leq N \qquad (2.13)$$

Básicamente lo que hay que decidir es la cantidad de monedas de tipo i que se van a emplear para devolver la cantidad j, siempre y cuando haya tal cantidad de las monedas usadas disponibles.

En función del número de monedas de cada tipo y de su valor existirá o no solución.

Tal y como se ve en la figura 2.1, cuando estemos haciendo el cálculo de la celda (i, j), necesitaremos los valores de las celdas (i, j') donde $j' = j - k \cdot v_i$, con $k = 0, \ldots, \left\lfloor \frac{j}{v_i} \right\rfloor$.

Otro ejemplo que suele ser utilizado para ilustrar la técnica de la programación dinámica es la multiplicación encadaneda de matrices. Se trata de un problema de interés porque la recurrencia que se obtiene al resolverlo coincide exactamente con la recurrencia que surge en otros muchos problemas (triangulación de polígonos, plegado de secuencias de ARN, etc.). Introducimos a continuación el problema:

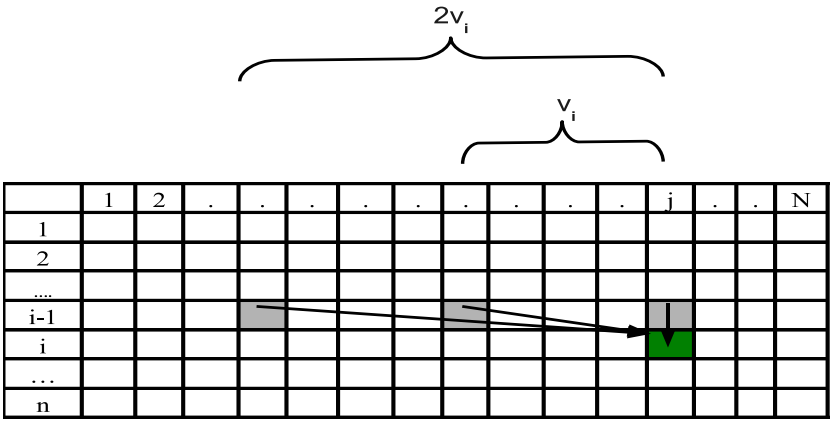

Figura 2.1: Esquema de la ejecución secuencial de la resolución del problema de las monedas, donde se ve que para calcular el valor de una celda es necesario contar con los datos de celdas calculados previamente.

- **Multiplicación encadenada de matrices:** Se trata de un problema clásico de programación dinámica [24] en el que hay que decidir cómo multiplicar una serie de matrices de tal forma que se minimice el número de operaciones aritméticas a realizar. Puesto que el producto es asociativo, habrá muchas formas de realizarlo. Cada colocación de los paréntesis indica un orden en el que se realizan las operaciones de dos matrices. Según este orden, el número total de multiplicaciones escalares necesarias puede variar considerablemente. Usando programación dinámica se puede obtener la posición de los paréntesis con la que se realiza un menor número de multiplicaciones elementales.

  Las variables a utilizar para describir el problema son:

  - n = Número de matrices disponibles
  - M_i = Matriz número i que se va a multiplicar, $1 \leq i \leq n$
  - M = Matriz resultado de multiplicar todas las matrices M_i, $1 \leq i \leq n$
  - $NMulti$ = Tabla de tamaño $n \times n$ donde se guarda el número mínimo de multiplicaciones para los distintos subproblemas. De la tabla sólo se utiliza la parte triangular superior
  - $NMulti[i, j]$ el número mínimo de productos escalares necesarios para realizar la multiplicación entre la matriz i y la j (con $i \leq j$), es decir:

$M_i \cdot M_{i+1} \cdot \ldots \cdot M_j$

- $d[0..n]$ contiene las dimensiones de las matrices a multiplicar, donde la matriz M_i será de dimensión $d[i-1] \times d[i]$

Los casos base del problema son:

- Si $i = j$ entonces $NMulti[i, j] = 0$. No es necesario realizar ninguna operación
- Si $i = j - 1$, entonces $NMulti[i, j] = d[i-1] \cdot d[i] \cdot d[i+1]$. Sólo existe una forma de hacer el producto de matrices

La ecuación de recurrencia sería:

- Si no se da ninguno de los casos anteriores, entonces podemos hacer la primera multiplicación por una posición k, con $i \leq k < j$: $(M_i \ldots M_k) \cdot (M_{k+1} \ldots M_j)$
- El resultado será el valor mínimo:

$$NMulti[i, j] = \min_{i \leq k < j} \{NMulti[i, k] + NMulti[k+1, j] + d[i-1] \cdot d[k] \cdot d[j]\} \tag{2.14}$$

La solución se obtiene con el valor $NMulti[1, n]$.

En este caso el problema es no multietapa puesto que en cualquiera de los recorridos (filas, columnas o diagonales), se producen dependencias entre etapas no consecutivas. Aunque los tres recorridos son posibles, en particular el recorrido por diagonales facilita la aproximación a su resolución paralela.

La forma de rellenar la tabla en los esquemas de programación dinámica puede ser: de arriba hacia abajo por filas (figura 2.2), por columnas de izquierda a derecha (figura 2.3) o por diagonales (figura 2.4). En los problemas de la mochila y de las monedas el recorrido natural es por filas, y en el de la multiplicación de matrices por diagonales. En estas figuras aparece en color verde la celda que se desea calcular y en gris las celdas, filas, columnas o diagonales que intervendrán para ello.

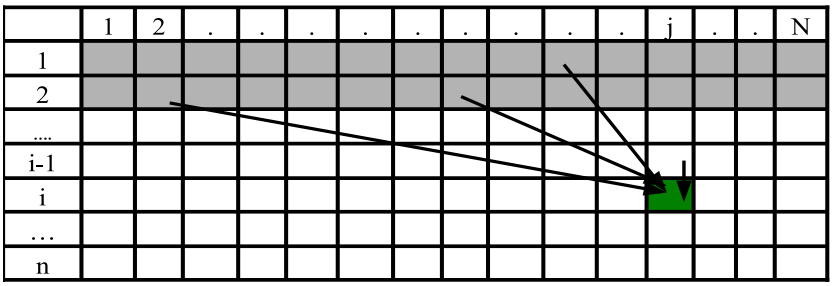

Figura 2.2: Cálculo de datos por filas. Para calcular el valor de cada celda hace falta haber calculado previamente los de filas anteriores.

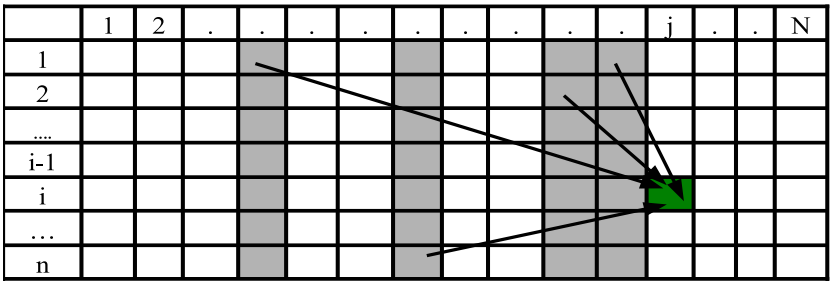

Figura 2.3: Cálculo de datos por columnas. Para calcular el valor de cada celda hace falta haber calculado previamente los de columnas anteriores.

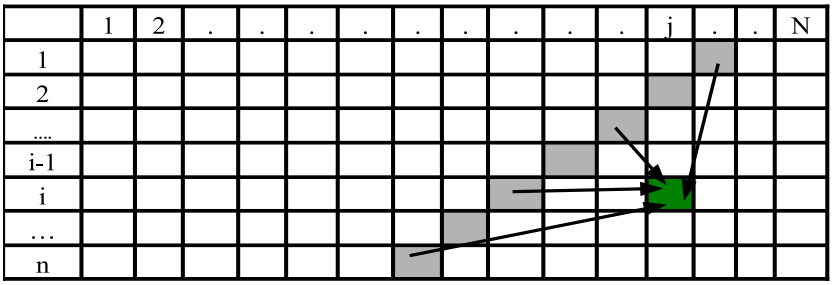

Figura 2.4: Cálculo de datos por diagonales. Para calcular el valor de cada celda hace falta haber calculado previamente los de diagonales anteriores.

2.2.1. Esquemas secuenciales

El algoritmo 2.3 muestra un esquema secuencial básico para un algoritmo de programación dinámica. El algoritmo se corresponde con una instanciación particular del esquema general presentado en la sección previa para algoritmos iterativos. En esta instanciación, la variable *etapas* indica al número de iteraciones del algoritmo mientras que la variable *estados* representa al número de elementos que hay que computar en cada iteración. Obsérvese que el número de estados en cada *etapa* (iteración) puede depender de la iteración en cuestión tal y como ocurre en el problema de la multiplicación encadenada de matrices.

Algoritmo 2.3: Esquema secuencial de programación dinámica. $f[i][j]$ representa al valor óptimo de acuerdo con la ecuación de recurrencia.

```
for i = 0; i ≤ etapas; i++ do
    for j = 0; j ≤ estados(i); j++ do
        calcular la solución óptima f[i][j] para el estado j de la etapa i
    end
end
```

En el algoritmo 2.3 los valores $f[i][j]$ vienen determinados por la ecuación de recurrencia del problema en cuestión y se almacenan en la tabla de programación dinámica. Las etapas del algoritmo podrían indicar filas, columnas o diagonales en la tabla. Cuando se recorren iterativamente las filas de la tabla, se evalúan estados o celdas de la tabla que dependen de valores que están en filas previamente calculadas. El mismo razonamiento es válido cuando la etapa representa a columnas o a diagonales. Las figuras representan distintos tipos de recorrido.

- Los datos se pueden calcular fila a fila de tal forma que para calcular una fila sea necesario usar datos calculados en filas anteriores (figura 2.2).
- Los datos se pueden calcular columna a columna de tal forma que para calcular una columna sea necesario usar datos calculados en columnas anteriores (figura 2.3).
- Los datos se pueden calcular por diagonales o antidiagonales usando datos de filas y/o columnas anteriores (figura 2.4).

2.2.2. Esquemas paralelos homogéneos

La paralelización de los algoritmos de programación dinámica ha sido objeto de estudio durante las últimas décadas. Estas paralelizaciones han estado orientadas fun-

damentalmente hacia arquitecturas homogéneas. Así, podemos encontrar que diversos autores han presentado soluciones paralelas para ecuaciones de recurrencia específicas sobre arquitecturas homogéneas específicas [11, 12, 20, 52, 94, 101, 119, 120]. Es cierto también que aunque las propuestas de procedimientos de paralelización generales están limitados a clases de recurrencia, sugiriéndose en la mayoría de los casos, versiones paralelas para algoritmos de la misma clase, en [105] ya han sido presentadas propuestas de generalización.

La revisión sobre los algoritmos de programación dinámica paralelos muestra dos grandes líneas de actuación en cuanto a la estrategia de paralelización. La primera de ellas asigna las celdas de la tabla de programación dinámica a los procesadores para que estas sean computadas en paralelo. La tabla se descompone en diferentes bloques que pueden estar compuestos por una o varias celdas, y la evaluación de estos bloques es asignada a diferentes procesadores. Se realizarían sincronizaciones bajo demanda a través de comunicaciones punto-a-punto entre los procesadores implicados. Esta es la situación que aparece cuando, por ejemplo, en el problema de la mochila se asigna la computación de una fila de la tabla a cada procesador y las columnas se computan en paralelo siguiendo un esquema *pipeline*. Las sincronizaciones en este caso se realizarían únicamente entre procesadores consecutivos. El segundo método generalmente aplicado hace uso de un esquema iterativo paralelo como el que aparece en el algoritmo 2.4. Los procesadores evalúan iterativamente y en parelelo cada una de las etapas del algoritmo de programación dinámica siguiendo el modelo de programación SPMD. Cada paso de la computación viene seguido de una sincronización para intercambiar datos entre los procesadores. Este paso de sicronización en muchos casos implica a una operación colectiva. Las etapas deben contener conjuntos de celdas que dependan sólo de celdas que hayan sido computadas en iteraciones previas. En el caso de problemas como el de la mochila o el de las monedas, las etapas estarían conformadas por las filas de la tabla, mientras que en el caso de la multiplicación encadenada de matrices serían las diagonales las que constituirían las etapas. En [74] pueden verse diversos casos y estrategias de recorrido.

Tal y como ya ha sido comentado, el rendimiento de cualquier algoritmo paralelo está fuertemente condicionado por la ratio computación/comunicación y por la adaptación del algoritmo a la arquitectura. En el caso particular de los algoritmos de programación dinámica, normalmente suelen mostrar comportamientos óptimos cuando se analizan únicamente desde el punto de vista teórico en el que se dispone de tantos procesadores como el tamaño de la entrada y las comunicaciones presentan comportamientos ideales. Sin embargo, es cierto que muchos de estos algoritmos presentan un rendimiento bastante pobre cuando se ejecutan en las arquitecturas ac-

Algoritmo 2.4: Esquema paralelo SPMD de programación dinámica. $f[i][j]$ representa al valor óptimo de acuerdo con la ecuación de recurrencia. Se asigna un proceso a cada columna de la tabla.

```
for i = 0; i ≤ etapas; i++ do
    forall proceso j; 0 ≤ j ≤ estados(i) do
        calcular la solución óptima f[i][j] para el estado j de la etapa i
        comunicación de resultados
    end
end
```

tuales. La implementación de estos algoritmos está fuertemente condicionada por la asignación real de los procesos a los procesadores físicos, la granularidad de la arquitectura y el caso particular de problema a ejecutar. Nuevamente, para preservar la optimalidad de estos algoritmos es necesario considerar la combinación adecuada de estos factores durante el proceso de optimización.

Podemos encontrar varias aproximaciones orientadas a la búsqueda de estos factores para la programación dinámica. La estrategia más extendida suele estar orientada a la descomposición del espacio de iteraciones en bloques de acuerdo a la arquitectura destino, utilizando una técnica que frecuentemente es conocida como *tiling*. El proceso de *tiling* puede consistir en buscar la geometría adecuada para los bloques, buscar las dimensiones adecuadas para los bloques o realizar ambas búsquedas. Los bloques serán computados independientemente por distintos procesadores estableciendo comunicaciones punto-a-punto bajo demanda. Esta es la aproximación propuesta en [2, 11, 12, 101, 102], y generalmente se aplica a problemas específicos (o a familias de problemas). En la mayoría de los casos se plantea la función analítica de coste y se obtiene de forma analítica también una expresión para los parámetros algorítmicos en función de los parámetros de la arquitectura. La aproximación es interesante puesto que, conocida la arquitectura y la instancia particular, basta con sustituir en la fórmula analítica para conseguir los parámetros algorítmicos óptimos en tiempo de ejecución. Esta aproximación plantea el inconveniente de que el cálculo analítico de los parámetros algorítmicos presenta una enorme dificultad y es sólo de validez práctica para problemas simples y sobre arquitecturas simples. En [68] también se presenta una estrategia con la que optimizar la ejecución de una clase de algoritmos de programación dinámica *pipeline* mediante la minimización heurística de la función analítica de coste. Aunque la metodología general es similar a la aplicada en esta tesis, la estrategia se centra en resolver el problema de la asignación de los procesadores virtuales de un *pipeline* sobre un anillo físico de procesadores.

Cuando en el esquema algorítmico iterativo 2.4 el número de procesadores físicos

disponible coincide con el número de estados de la etapa, se asigna directamente un proceso a cada procesador. En la práctica esto no suele ser posible puesto que el número de estados por etapa suele ser muy elevado, y hay que realizar la asignación de procesos virtuales a procesos físicos. Esta asignación puede realizarse mediante estrategias de asignación cíclicas, por bloques o combinaciones de ambas. Otra opción es considerar el esquema que se muestra en el algoritmo 2.5, en el que el algoritmo paralelo hace directamente la asignación de valores a cada procesador físico. En el algoritmo 2.5 la asignación se realiza por bloques, pero también podría ser cíclica o cíclica por bloques. Se asume que hay un proceso asignado a cada procesador y que cada proceso dispone de bloques de igual tamaño para realizar la computación.

Algoritmo 2.5: Esquema paralelo SPMD de programación dinámica. $f[i][j]$ representa al valor óptimo de acuerdo con la ecuación de recurrencia. El procesador con identificador $P_k, k = 0, \cdots, P-1$ calcula un bloque de celdas en cada fila.

```
for i = 0; i ≤ etapas; i++ do
    for j = k * estados(i)/P + 1; j < (k + 1) * estados(i)/P + 1; j++ do
        calcular la solución óptima f[i][j] para el estado j de la etapa i
        comunicación de resultados
    end
end
```

En esta tesis abordaremos el proceso de autooptimización de algoritmos iterativos de programación dinámica sobre sistemas homogéneos y heterogéneos considerando como parámetros algorítmicos AP en el modelo analítico aquellos que respondan a las siguientes cuestiones:

- ¿Cuántos procesos van a intervenir en la resolución del problema?
- ¿Cuántos procesadores se van a utilizar?
- ¿Cuántos procesos se van a asignar a cada procesador?
- ¿Qué cantidad de datos va a calcular cada proceso?

Estas cuestiones no son fáciles de considerar de forma general. En la figura 2.5 se muestra la forma en que se obtendrían los valores de una tabla cuyas columnas son calculadas por diferentes procesos. Se puede ver que los procesos necesitarán intercambiar datos entre sí para poder hacer sus propios cálculos. En los próximos capítulos se abordarán estas cuestiones aplicadas al ámbito de los esquema iterativos.

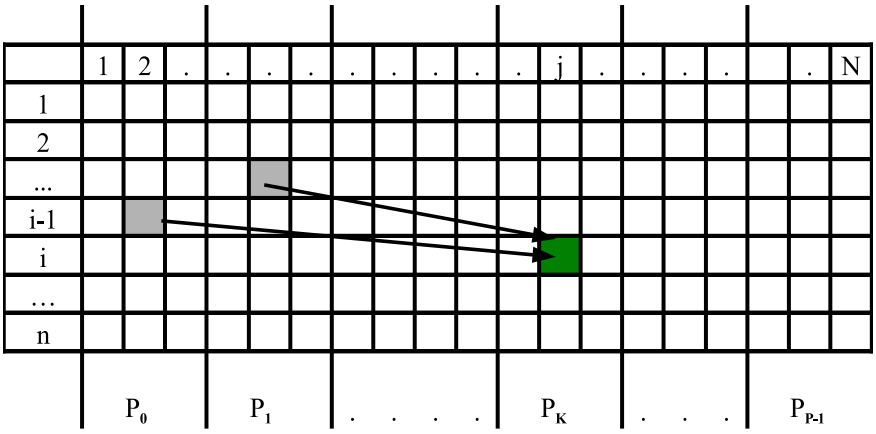

Figura 2.5: Ejecución en paralelo de un problema de programación dinámica donde los datos se calculan por filas. Tal y como se ve es posible que para que el proceso P_k calcule el dato (i, j), necesite datos que son calculados por otros procesos.

2.2.3. Esquemas paralelos heterogéneos

El esquema de programación dinámica paralela para sistemas heterogéneos coincide con el presentado en el algoritmo 2.5 pero en el que el tamaño del bloque debe ser ajustado a la capacidad del procesador (algoritmo 2.6). Obsérvese que esta estrategia impone la modificación del algoritmo desarrollado para la arquitectura homogénea. Se asume que hay un proceso asignado a cada procesador y que cada proceso dispone del bloque de cómputo adecuado a su potencia computacional. Si se considera que la heterogeneidad en el sistema es estática y que el número de procesos involucrados es igual al número de procesadores físicos, el problema de optimización consistirá en calcular el tamaño de bloque óptimo para cada proceso.

En [53] ha sido presentado un mecanismo alternativo que permite adaptar dinámicamente el tamaño del bloque para códigos iterativos homogéneos. Se trata de una aproximación que, en tiempo de ejecución de la rutina, obtiene la carga computacional de cada procesador en función de la instancia que está siendo calculada. A medida que avanza la ejecución del algoritmo, el tamaño del bloque se ajusta en función de esta carga computacional. Esta aproximación requiere la modificación de la rutina homogénea y además la instrumentación del código para poder establecer el mecanismo de ajuste dinámico.

Algoritmo 2.6: Esquema paralelo SPMD de programación dinámica. $f[i][j]$ representa al valor óptimo de acuerdo con la ecuación de recurrencia. El procesador $P_k, k = 0, \cdots, P-1$ calcula un bloque de celdas en cada fila.

```
for i = 0; i ≤ etapas; i++ do
    for j = primero(P_k); j < primero(P_k) + tam(P_k); j++ do
        calcular la solución óptima f[i][j] para el estado j de la etapa i
        comunicación de resultados
    end
end
```

2.2.4. Esquemas paralelos homogéneos en sistemas heterogéneos

Para adaptar esquemas paralelos homogéneos a sistemas heterogéneos, y asumiendo nuevamente un sistema heterogéneo estático, una opción consiste en partir del código del algoritmo 2.4 en el que cada proceso puede tener asignado un conjunto de valores a computar. Se trataría de reasignar los procesos a los procesadores de acuerdo a la capacidad de cómputo de estos. Esta aproximación ha sido considerada en [106] para algoritmos de programación dinámica basados en estrategias *pipeline*. Se utiliza la función analítica de coste y se aplica una metaheurística durante el proceso de optimización de la misma. Claramente la estrategia de optimización utilizada para la reasignación de los procesos es crítica y condiciona el éxito de la propuesta. A lo largo de los siguientes capítulos, proponemos aplicar estrategias exactas, heurísticas y metaheurísticas al problema de optimización del algoritmo 2.4 en sistemas heterogéneos.

2.3. Conclusiones

En este capítulo se han analizado las características generales de los esquemas iterativos, tanto secuenciales como paralelos, y algunos aspectos particulares de los esquemas según el sistema sea homogéneo u heterogéneo. También se han presentado esquemas iterativos para problemas de programación dinámica, y algunos problemas típicos que se resuelven por programación dinámica.

La finalidad del capítulo no ha sido hacer un estudio detallado ni de los esquemas iterativos ni de la programación dinámica, sino establecer la notación que se utilizará en capítulos posteriores, mostrar distintas variantes a las que se podrían aplicar los métodos desarrollados en los siguientes capítulos, y explicar los problemas que se utilizarán como caso de prueba en el resto del trabajo.

Capítulo 3

Autooptimización en sistemas homogéneos

Como se ha visto en capítulos previos, la autooptimización de algoritmos puede suponer una reducción del trabajo de mantenimiento y adaptación de software. Las técnicas de ajuste automático han sido usadas en el diseño de rutinas paralelas en los últimos años. Estas técnicas han sido desarrolladas en diferentes campos [25, 49], y especialmente sobre rutinas de álgebra lineal [27, 42, 85]. La comunidad científica está dedicando un importante esfuerzo al estudio y análisis de la autooptimización de esquemas algorítmicos paralelos. La autooptimización en estos sistemas permitiría el diseño de esqueletos para ser incluidos en lenguajes paralelos de alto nivel [109], y el desarrollo de esquemas para ser usados en sistemas paralelos [72]. Pero la extensión a esquemas algorítmicos paralelos de las metodologías usadas para obtener rutinas optimizadas automáticamente [34] o una jerarquía de librerías optimizadas [35] todavía no ha sido analizada.

Por ello, en este capítulo se muestran las posibilidades de la autooptimización en el uso de sistemas homogéneos para los esquemas paralelos iterativos de programación dinámica. Consideramos como homogéneos no sólo a aquellos sistemas cuyos elementos tienen las mismas características físicas sino también a aquellos formados por elementos muy similares. Se estudia la posibilidad de incluir técnicas de optimización automática en el diseño de algoritmos paralelos de programación dinámica, como ejemplo de esquema iterativo. La idea principal es lograr una aproximación automática al número óptimo de procesadores a usar y establecer un esquema adecuado de reparto de la carga de trabajo. Se persigue la aplicación de la técnica no sólo para un esquema sencillo sino que además pueda usarse en problemas con diferentes características y así obtener conclusiones sobre su aplicación a otros esquemas iterativos.

3.1. Autooptimización de esquemas de programación dinámica

La solución a los problemas de programación dinámica se puede obtener con un planteamiento básico que también se puede aplicar a otros esquemas algorítmicos, como por ejemplo el método de avance rápido en el algoritmo de Dijkstra [24, 30]. Un posible esquema consiste en obtener la solución completando una tabla bidimensional donde las columnas representan el tamaño del problema a resolver y las filas el tamaño de las subsoluciones. Cada entrada en la tabla contiene la solución óptima para un problema del tamaño representado por la columna y con el número de decisiones representado por la fila. La solución final es aquella que se obtiene a partir de la celda de la última fila y columna. Se puede usar otra tabla para almacenar las decisiones que se han tomado en cada paso del algoritmo para obtener la solución óptima.

Para poder desarrollar un modelo parametrizado del tiempo de ejecución es necesario empezar estableciendo los parámetros que intervienen en él. Estos parámetros son parámetros algorítmicos y del sistema, pero también los que determinan el tamaño del problema a resolver. Obviamente el tamaño depende del problema concreto, pero hay algunas características comunes a problemas que siguen el mismo esquema. En el caso de esquemas de programación dinámica donde se utiliza una tabla bidimensional para almacenar las soluciones óptimas de subproblemas, se tiene que el valor de *etapas* en el esquema básico de programación dinámica (algoritmo 2.3) coincide con el número de filas de la tabla, y en algunos problemas el número de posibles estados para cada etapa ($estados(i)$ en el algoritmo) coincide con el número de columnas, que llamaremos N.

En una versión paralela, tal como se vio en el algoritmo 2.5, el trabajo a realizar en cada iteración (cada *etapa*) consiste en una computación seguida de un intercambio de información, que es necesario para que los procesadores implicados en la computación tengan disponibles los datos generados por otros procesadores y que necesitan para los *estados* que les corresponde calcular. El número de procesadores en el sistema lo identificamos con P y al número de procesos usados en la resolución del problema le llamamos p, con $1 \leq p \leq P$. Como consideramos un programa homogéneo con idéntico volumen de computación en cada proceso, y un sistema homogéneo donde los P procesadores tienen idéntica capacidad computacional y la red de interconexión es simétrica entre cada par de procesadores, se asigna un proceso a cada procesador seleccionado para participar en la computación, por lo que p representa tanto el número de procesos como de procesadores seleccionados. Así, el parámetro algorítmico a determinar es el número de procesos (o procesadores) usado en la resolución del

problema, p.

En este capítulo se analiza este esquema básico de paso de mensajes. Hay algunas variaciones de estos esquemas secuencial y paralelo [70], pero consideramos que el análisis que aquí se hace será válido para otros esquemas similares.

El objetivo final de la autooptimización es el de obtener rutinas con las que se seleccionen valores de determinados parámetros que nos permitan obtener tiempos de ejecución reducidos sin intervención del usuario. En este caso el valor que hay que obtener es el número de procesadores a usar en la resolución del problema. Como mencionamos en el capítulo anterior, el tiempo de ejecución de la rutina se debe modelar basándose en algunos parámetros que representan las características del sistema donde el problema se va a resolver. Por otro lado, el tiempo de ejecución depende de los parámetros algorítmicos (en este caso el número de procesadores) que se seleccionan para reducir el tiempo de ejecución.

En el primer capítulo se analizaron las ideas generales de la optimización basada en modelos parametrizados del tiempo de ejecución. Ahora analizaremos la aplicación de esta metodología con el esquema del problema de las monedas con programación dinámica, para el que veremos un esquema algorítmico, analizaremos los parámetros del sistema y algorítmicos que influyen en el tiempo de ejecución, obtendremos un modelo parametrizado de este tiempo, y describiremos el proceso de instalación de la rutina y de ejecución para una entrada concreta. Estos son los diferentes pasos que se ven en la figura 1.2.

Dentro de la fase de **diseño**, y una vez se ha diseñado la rutina, que en nuestro caso resolverá el problema de las monedas, se debe construir su función de coste, a la que habrá que incorporar los parámetros algorítmicos y del sistema. Además, en algunos casos el valor de los parámetros del sistema depende del tamaño del problema y de los valores de los parámetros algorítmicos: *SP(s,AP)*. En diferentes sistemas los valores óptimos de los parámetros algorítmicos pueden ser diferentes y consecuentemente estos valores dependen de los parámetros del sistema. Por lo tanto el tiempo de ejecución se modela en la forma: $t(s, AP, SP(s, AP))$ [34], donde s representa el tamaño del problema, AP los parámetros algorítmicos, y SP los parámetros del sistema.

En los esquemas de programación dinámica, consideramos que el tamaño del problema inicialmente se corresponde con el tamaño de la tabla. En el caso particular del problema de las monedas se determina por la cantidad de dinero a devolver (N) y el número (n) de diferentes valores de monedas de que se dispone, que son los parámetros que determinan el tamaño de la tabla, pero los valores de las monedas, v_i, y las cantidades usadas de cada tipo, q_i, también influyen en el tiempo de ejecución, de ahí que sean parte del tamaño del problema: $s = (n, N, v, q)$.

Tal y como ha sido comentado, una forma típica de representar el tiempo de un programa con paso de mensajes es $t(n,p) = t_{comp}(n,p) + t_{comm}(n,p)$, donde n es el tamaño del problema, p el número de procesadores que se utilizan, numerados desde 0 hasta $p-1$, t_{comp} el tiempo de ejecución de la parte computacional y t_{comm} el coste de las comunicaciones. Además, el coste de comunicaciones se puede representar como una función de t_s y t_w [74]. Por otro lado, $t_{comp}(n,p)$ depende del número de operaciones computacionales realizadas y del coste de una operación computacional en el algoritmo, t_c. Por lo tanto, los parámetros del sistema son $SP = (t_c, t_s, t_w)$.

Para obtener una versión de autoajuste de la rutina hay que obtener los parámetros del sistema sobre el que se ejecutará la misma (fase de **instalación** en la figura 1.2). Una vez obtenidos se deben incorporar a la función de coste temporal para obtener el tiempo de ejecución teórico para el sistema particular en el que estamos trabajando.

Dentro de la fase de **ejecución** hay que determinar los parámetros algorítmicos. En nuestro caso el número de procesadores a usar (p) es uno de ellos y otro parámetro sería el tamaño de bloque (b) de la distribución cíclica de datos. En los experimentos realizados sólo se ha considerado el parámetro p.

De esta forma, el tiempo de ejecución para el problema de las monedas puede ser modelado como:

$$t\left(n, N, v, q, t_c(s,p,b), t_s(s,p,b), t_w(s,p,b), p, b\right) \tag{3.1}$$

donde comparando con la ecuación $t(n, AP, SP)$, tendríamos $s = (n, N, v, q)$, $SP = (t_c(s,p,b), t_s(s,p,b), t_w(s,p,b))$, $AP = (p,b)$, y los valores de p y b deben ser seleccionados para obtener el mínimo valor de t para un valor particular de t_c, t_s y t_w en el sistema en cuestión para el problema que se pretende resolver. Una vez que el trabajo se ha dividido en n pasos con computación y comunicaciones, el tiempo paralelo tiene la forma $t = n \cdot (t_{comp} + t_{comm})$, y sólo es necesario obtener los valores de t_{comp} y t_{comm} en cada paso. En la figura 2.5 puede verse que las N computaciones en cada paso son asignadas a p procesadores de manera equitativa pues todos ellos tienen iguales prestaciones.

En el problema de devolución de monedas básicamente lo que hay que decidir para monedas de tipo i es la cantidad de monedas que se utilizan, que puede ser desde cero hasta un máximo posible determinado por la cantidad de monedas de ese tipo y el número de monedas que se pueden dar sin exceder la cantidad a devolver. Para obtener la solución se aplica la ecuación 2.13.

Si las columnas de la tabla se asignan usando una distribución por bloques contiguos, a cada procesador se le asignan $\frac{N}{p}$ columnas consecutivas y el procesador con

mayor carga de trabajo es el $p-1$, el cual tiene las columnas $N-\frac{N}{p}+1,\ldots,N$, y el trabajo asignado a este procesador tiene coste:

$$\sum_{j=N-\frac{N}{p}+1}^{N}\left(1+min\left\{\left\lfloor\frac{j}{v_i}\right\rfloor,q_i\right\}\right) \tag{3.2}$$

Haciendo un estudio sobre el coste computacional de la anterior ecuación y considerando el número de procesadores a emplear se puede deducir que:

- Cuando la cantidad j es grande y los valores de las monedas v_i no son demasiados grandes la ecuación 3.2 se puede aproximar por:

$$\sum_{j=N-\frac{N}{p}+1}^{N}(1+q_i)=(1+q_i)\frac{N}{p} \tag{3.3}$$

  y debido a que el coste secuencial es aproximadamente $(1+q_i)N$, la eficiencia alcanzable es del 100 %.

- Cuando q_i es grande, o incluso cuando está disponible un número ilimitado de monedas de valores v_i, la ecuación 3.2 se aproxima a:

$$\sum_{j=N-\frac{N}{p}+1}^{N}\left(1+\left\lfloor\frac{j}{v_i}\right\rfloor\right)\approx\frac{N}{p}+\left(\frac{N^2}{p}-\frac{N^2}{2p^2}+\frac{N}{2p}\right)\frac{1}{v_i}\in o\left(\frac{N^2}{v_ip}\right) \tag{3.4}$$

La máxima eficiencia alcanzable, en este caso, con esta distribución de datos es del 50 %, debido a que el coste secuencial de un paso es $o\left(\frac{N^2}{2v_i}\right)$.

Una distribución cíclica de las columnas a los procesadores daría una eficiencia teórica del 100 %, pero requeriría mayor coste de comunicaciones.

En el paso de comunicaciones, cada procesador P_i, con $i=0,1,\ldots,p-2$, envía los valores que ha calculado a los procesadores $P_{i+1},P_{i+2},\ldots,P_{p-1}$, tal como se ve en la figura 3.1. Así el procesador P_0 haría $p-1$ envíos de datos a los otros procesadores, el procesador P_1 haría $p-2$ envíos de datos, y por último el procesador P_{p-2} haría un único envío de datos. Por lo tanto el número de comunicaciones sería:

$$\sum_{i=1}^{p-1}(p-i)=\frac{p(p-1)}{2} \tag{3.5}$$

Puesto que en cada envío se transfieren $\frac{N}{p}$ datos, la cantidad total de datos transferida sería $\frac{N(p-1)}{2}$.

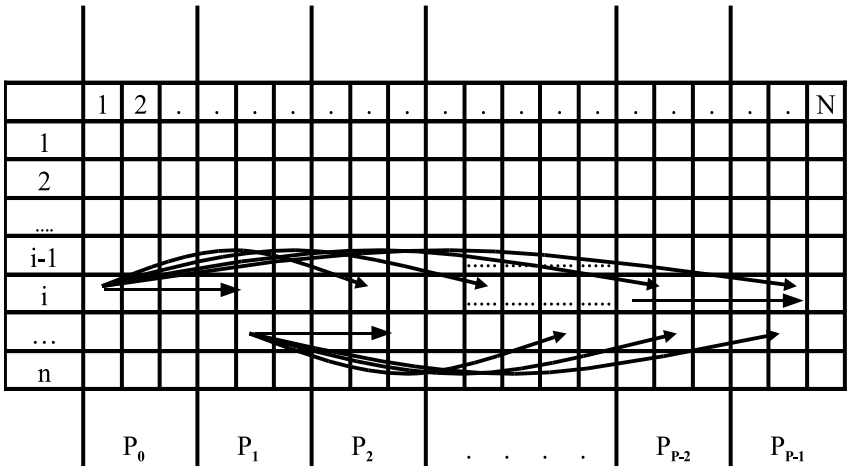

Figura 3.1: Envío de datos entre procesadores.

En una red tipo Ethernet, si consideramos que solo se puede enviar cada vez un mensaje por el bus, el coste de las comunicaciones se puede representar en la forma:

$$t_{comm} = \frac{p(p-1)}{2} t_s + \frac{N(p-1)}{2} t_w \tag{3.6}$$

Si la red de comunicación permite que las comunicaciones se lleven a cabo en paralelo, el coste de las comunicaciones es el correspondiente al proceso que más comunicaciones realiza, que es P_0, y el coste queda:

$$t_{comm} = (p-1)t_s + \frac{N(p-1)}{p} t_w \tag{3.7}$$

Una vez que se ha obtenido el tiempo teórico durante la fase de **diseño** (figura 1.2) el siguiente paso es decidir cómo obtener los valores de los parámetros del sistema, dentro de la fase de **instalación**. Estos valores se pueden obtener cuando la rutina (o la librería que contiene a la rutinas) se instala. Los parámetros del sistema (t_c, t_s y t_w en el problema de las monedas) se pueden obtener ejecutando en el momento de la instalación una versión reducida de la rutina o de parte de la rutina donde aparecen los parámetros.

Usando la técnica del *ping-pong*, explicada en el primer capítulo, se pueden calcular los valores de t_s y t_w, pero éste no es el tipo de rutina que aparece en la fase de comunicaciones en el esquema que estamos considerando, y su uso para estimar t_s y t_w puede producir una diferencia entre el comportamiento de la rutina y el compor-

tamiento del modelo, y por lo tanto se obtendría una mala predicción.

En la tabla 3.1 se muestran los valores de t_s y t_w (expresados en segundos) obtenidos con esa técnica para diferentes plataformas usadas en esta tesis, cuyas características han sido explicadas en la subsección 1.6.1 de herramientas hardware del primer capítulo, y sobre las que se muestran resultados experimentales en la próxima sección. Los valores obtenidos en los diferentes sistemas o entre distintos nodos en un mismo sistema, y la variación en el coste relativo entre el tiempo de inicio de la comunicación y de envío de un dato, deja patente que es necesario un proceso de autooptimización como el que proponemos, que ajuste la rutina al sistema sobre el que se instala. En el caso del laboratorio de computación paralela, en el sistema SUNEt, se han calculado los valores t_s y t_w teniendo en cuenta las comunicaciones entre procesadores iguales (SUN1 - SUN1) y entre procesadores diferentes (SUN1 - SUN5). Además en el sistema HPC160 se han calculado dichos valores considerando las comunicaciones entre procesos que se encuentran en un mismo nodo (HPC160 intranodo) o en nodos diferentes (HPC160 internodo).

Tabla 3.1: t_s y t_w, en segundos, para diferentes sistemas.

	SUN1 SUN1	SUN1 SUN5	PenFE	Origin 2000	HPC160 intranodo	HPC160 internodo
t_s	0.00170	0.00179	0.000808	0.000279	0.000221	0.000163
t_w	3.53E-06	3.49E-06	3.90E-06	1.71E-07	2.23E-08	7.81E-08

Hay grandes diferencias de tiempos en los diferentes sistemas empleados, tal como se ve en la figura 3.2, donde se muestra una comparativa de los tiempos de comunicaciones obtenidos en ellos. Se puede observar cómo las diferencias en el coste de las comunicaciones entre SUNEt y PenFE con respecto a Origin y HPC160 aumentan notablemente conforme crece el tamaño del mensaje. Ello se debe a que las comunicaciones se realizan en estos dos últimos sistemas a través de la memoria compartida (en el caso de HPC160 cuando la comunicación es intranodo). En la gráfica 3.3 se comparan los tiempos de las comunicaciones en SUNEt (izquierda) y HPC160 (derecha) variando el tamaño de los mensajes y mostrando los tiempos en escala logarítmica. Se observa que en el caso de SUNEt los valores de las comunicaciones entre procesadores iguales (SUN1-SUN1) son prácticamente iguales (se solapan en la figura) que entre procesadores diferentes (SUN1-SUN5). Para un sistema con comunicaciones diferentes, como HPC160, sí que se aprecia en la gráfica que cuando el tamaño del mensaje crece las comunicaciones internodo se hacen más costosas que las intranodos.

Para aclarar más lo anterior, en la gráfica 3.4 se muestra una comparativa entre los tiempos de comunicaciones. Para SUNEt (izquierda) se expresa el cociente de los tiempos de SUN1-SUN1 entre los de SUN1-SUN5. Se puede observar que, con independencia del tamaño del mensaje, el coste de comunicación es prácticamente igual (el cociente es aproximadamente 1). En la figura a la derecha se muestra el cociente entre los tiempos de comunicaciones internodo e intranodo en HPC160. Aquí sí que se aprecian diferencias notables, pues cuando el tamaño del mensaje aumenta, el coste de las comunicaciones internodo triplica al de las intranodo.

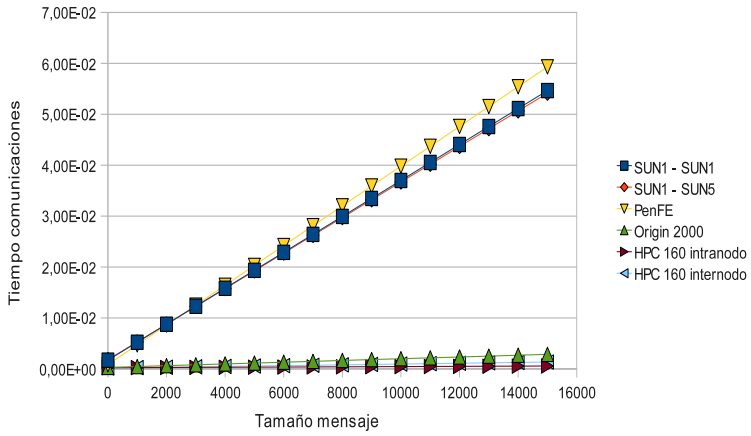

Figura 3.2: Tiempos de comunicaciones en diferentes sistemas.

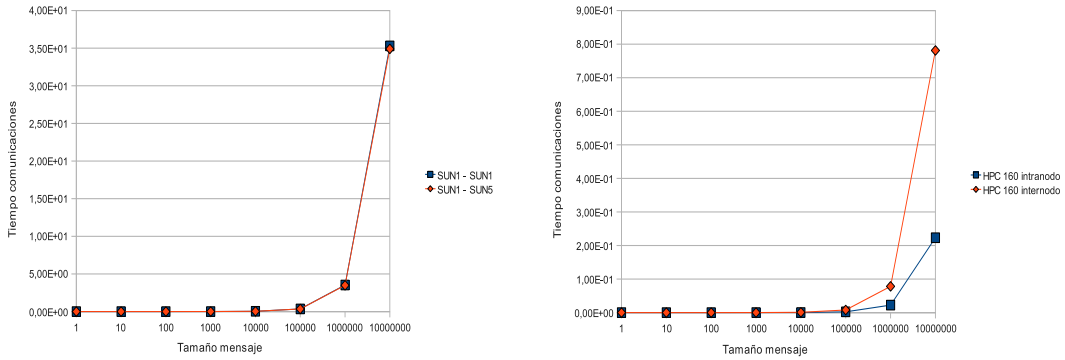

Figura 3.3: Tiempos de comunicaciones en escala logarítmica en SUNEt (izquierda) y HPC160 (derecha) al variar el tamaño del mensaje multiplicando por 10.

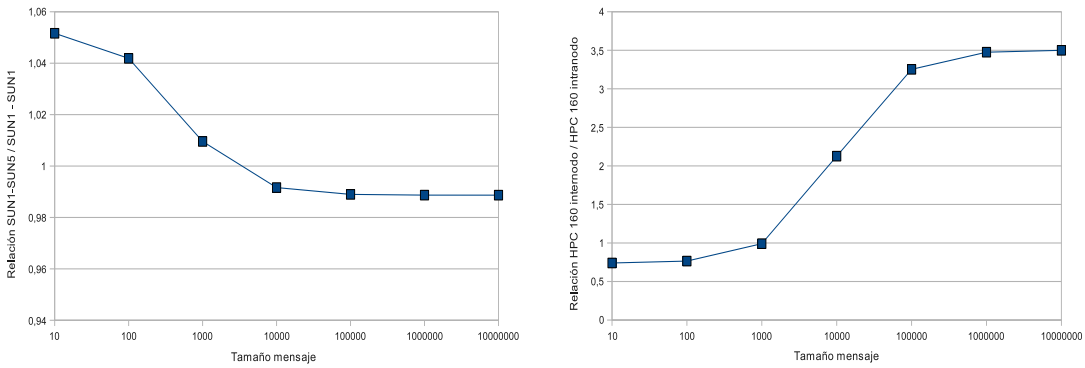

Figura 3.4: Ratio de tiempos de comunicaciones en SUNEt (izquierda) y HPC160 (derecha) al variar el tamaño del mensaje.

Otra posibilidad es estimar los valores de t_s y t_w ejecutando una versión reducida de la rutina que contenga las comunicaciones que en ella se realizan, variando los valores de los parámetros algorítmicos (en nuestro caso p), y el coste de las comunicaciones se modelaría como una función de estos parámetros.

El valor de t_c se puede estimar ejecutando una pequeña versión del problema en secuencial en un procesador en el sistema. Si el sistema fuera heterogéneo, como sería el caso del laboratorio de computación paralela con las máquinas SUN, la prueba se realizaría para cada tipo de procesador y así se obtendrían diferentes valores de t_c. Este caso se estudiará en el siguiente capítulo.

Dentro de la fase de **ejecución**, la rutina decidirá los valores de los parámetros algorítmicos que producen un menor tiempo teórico de ejecución usando la ecuación 3.1, y se ejecutará con estos valores de los parámetros. Los mejores valores se obtienen sustituyendo en el modelo parametrizado los posibles valores de los parámetros algorítmicos, y considerando para cada parámetro del sistema los valores correspondientes a los valores de los parámetros algorítmicos que se están utilizando. Esto se debe a que como hemos mencionado los parámetros del sistema podemos considerarlos como una función de los parámetros algorítmicos y del tamaño del problema ($t_c(s,p)$, $t_s(s,p)$, $t_w(s,p)$).

3.2. Resultados experimentales

Con el fin de estudiar la autooptimización en sistemas homogéneos se ha programado el problema de las monedas descrito anteriormente y se han realizado pruebas de su versión paralela sobre diferentes sistemas homogéneos: dos redes de ordenadores

distintas de la Facultad de Informática de la Universidad de Murcia (SUNEt, PenFE), el sistema del Centro Europeo de Paralelismo de Barcelona (ORIGIN 2000), y un sistema de la Universidad Politécnica de Cartagena, HPC160, todos ellos descritos en la subsección de hardware 1.6.1, y los resultados se muestran a continuación. Aunque SUNEt y HPC160 cuando se usan varios nodos no son sistemas homogéneos, en este capítulo los experimentos se harán con rutinas homogéneas y no consideraremos las características de heterogeneidad de los sistemas.

3.2.1. Análisis del paralelismo en los distintos sistemas

Los experimentos se han realizado de la siguiente forma. Como entrada se proporciona el número de procesadores en el sistema sobre el que se van a ejecutar los programas, la cantidad de dinero a devolver, el número de monedas que se van a generar aleatoriamente y la granularidad, que es un valor que se utiliza para aumentar el coste computacional de los algoritmos. Variando la granularidad se pretende estudiar el paralelismo en el esquema que sigue el problema de las monedas, pues estamos interesados en el esquema algorítmico más que en el problema concreto. El valor y el número de monedas de cada tipo son generados al azar entre unos valores establecidos. Todas las pruebas se han realizado utilizando 10 tipos distintos de monedas sin limitación del número de monedas disponibles y cuyos valores se calculan aleatoriamente.

Se han hecho 2 versiones del software, denominadas Versión A y Versión B, con el fin de probar distintas posibilidades que reduzcan los tiempos de ejecución, como a continuación se explica. En las dos versiones la tabla de subsoluciones a rellenar se distribuye con bloques adyacentes de columnas entre los procesos: si la tabla es una matriz $n \times N$ y se utilizan p procesos (consideramos que los procesos son numerados de 0 hasta $p-1$), al proceso i le corresponden las columnas de la $\frac{N}{p}i$ a la $\frac{N}{p}(i+1)-1$ (consideramos las columnas numeradas de 0 hasta $N-1$). Cada paso corresponde al cálculo de los subproblemas de una fila, que se hace en función de valores de la fila anterior. Así, en la parte de comunicación es necesario que unos procesos envíen a otros algunos de los valores que han calculado en la parte de computación. Las dos versiones se diferencian en la forma en que se hacen las comunicaciones:

- Versión A: Cada proceso envía a todos los demás la totalidad de sus datos calculados en el paso de computación anterior.
- Versión B: Cada proceso no envía la totalidad de sus datos a todos los procesos que vienen a continuación, sino que calcula qué datos van a necesitar estos procesos y se los envía.

En la Versión A hay comunicaciones innecesarias de datos, pero se facilita la programación y se evita la sobrecarga de calcular los datos a enviar y los procesos destino y origen de los datos que se comunican.

En la Versión B se emplea un tiempo en cálculos adicionales, pero se asegura que los datos que se envían son exactamente los que hacen falta, reduciéndose el tiempo de comunicaciones.

En las tablas 3.2, 3.3, 3.4, 3.5 y 3.6 aparecen los resultados correspondientes a las ejecuciones sobre los distintos sistemas. La primera columna representa la cantidad de dinero que se pretende devolver, la segunda es el número de procesos que se van a emplear en la resolución del problema y a continuación se muestran los tiempos obtenidos con cada versión del algoritmo para granularidad 10, 50 y 100. Tal y como se ha indicado, la granularidad es un índice que sirve para proporcionar mayor peso a la sección de computación en el tiempo total de ejecución. El mejor tiempo para cada ejecución (granularidad, tamaño y versión) aparece en negrita.

Comentamos los resultados obtenidos en los distintos sistemas intentando extraer algunas conclusiones y comparando los resultados según las características del sistema.

- En SUNEt (tabla 3.2):

  - Se observa a grandes rasgos que cuando el tamaño del problema es grande no se reduce el tiempo de ejecución usando paralelismo, pues el menor tiempo se alcanza siempre en ejecución monoprocesador. Esta disparidad se debe al tipo de red (Ethernet) y a que el primer proceso se asigna a un SUN5, que es aproximadamente 2.5 veces más rápido que el resto de procesadores, con lo que hacen falta al menos tres procesos para obtener un tiempo de ejecución menor que el secuencial.
  - También hay que tener en cuenta que debido a que este sistema consta de 6 procesadores de los cuales 5 son iguales en cuanto a velocidad, en las ejecuciones con 7 procesos al procesador más rápido se le asignan 2 procesos. En la tabla se muestran en itálica los mejores tiempos para ejecuciones entre 2 y 6 procesos, en las que todos los procesadores que se utilizan son SUN1.
  - En los casos en que el tamaño es pequeño se observa que cuando la granularidad es grande sí que interesa utilizar paralelismo, pues empleando varios procesos se reduce el tiempo de ejecución, pero cuando el tamaño del problema es grande siempre se obtiene el mejor tiempo de ejecución con la ejecución secuencial. Esto se debe a que el procesador donde se realiza

Tabla 3.2: Tiempos de ejecución, en segundos, en el sistema SOLARIS/SUN (SUNEt) de la red del laboratorio de computación paralela de la Universidad de Murcia, con distinto número de procesos, cantidades a devolver y granularidad simulada.

Tamaño	procs.	Versión A Granularidad			Versión B Granularidad		
		10	50	100	10	50	100
10000	1	**0.91**	4.45	8.91	**0.91**	4.54	9.18
10000	2	1.27	6.21	12.35	1.27	6.19	12.32
10000	3	*1.05*	4.72	9.37	1.03	4.71	9.34
10000	4	1.31	5.10	10.03	*1.01*	3.98	7.96
10000	5	1.23	4.45	8.71	1.31	3.56	7.11
10000	6	1.42	*4.05*	*7.86*	1.45	*3.39*	*6.70*
10000	7	1.55	**3.83**	**7.51**	1.58	**3.10**	**6.13**
50000	1	**4.54**	22.41	44.91	**4.58**	22.82	45.91
50000	2	6.83	39.61	63.93	7.29	32.47	63.32
50000	3	6.58	27.05	53.35	6.41	33.55	54.89
50000	4	*5.85*	26.90	51.61	*5.84*	21.21	41.88
50000	5	11.47	21.51	42.62	12.10	*18.41*	34.75
50000	6	6.67	*19.93*	*37.66*	6.01	25.04	*31.09*
50000	7	7.08	**17.25**	**33.34**	6.44	**16.89**	**28.28**
100000	1	**8.96**	**44.37**	**89.25**	**9.15**	**45.40**	**90.56**
100000	2	14.01	62.51	123.12	14.04	62.83	123.70
100000	3	*12.38*	*54.15*	*106.44*	*12.42*	*54.46*	*106.96*
100000	4	14.85	63.77	125.38	13.50	57.25	112.47
100000	5	15.78	64.29	125.98	13.90	58.46	115.04
100000	6	15.39	64.80	126.60	15.68	60.34	118.03
100000	7	15.79	59.58	112.56	16.55	55.49	109.42
500000	1	**46.01**	**227.31**	**452.71**	**46.85**	**232.59**	**464.57**
500000	2	70.38	313.68	617.75	70.20	313.71	617.57
500000	3	*62.38*	*272.65*	*535.23*	*62.92*	*273.01*	*536.05*
500000	4	72.74	317.98	625.56	66.03	284.86	559.45
500000	5	72.79	318.91	627.75	68.27	292.62	574.43
500000	6	75.21	320.18	628.47	70.22	300.05	585.95
500000	7	73.41	298.82	579.62	71.83	280.40	543.61

Tabla 3.3: Tiempos de ejecución, en segundos, en el sistema PenFE con red ETHERNET 10/100 del laboratorio de arquitectura de la Universidad de Murcia, con distinto número de procesos, cantidades a devolver y granularidad simulada.

Tamaño	procs.	Versión A Granularidad			Versión B Granularidad		
		10	50	100	10	50	100
10000	1	**0.56**	2.76	5.52	**0.55**	2.75	5.55
10000	2	0.74	2.04	3.92	0.76	2.03	3.66
10000	3	0.93	1.92	3.19	0.96	1.91	3.16
10000	4	0.91	1.75	2.93	0.89	1.77	2.93
10000	5	0.91	**1.72**	2.91	0.93	**1.73**	2.92
10000	6	0.92	1.76	**2.84**	0.92	1.76	**2.83**
10000	7	0.97	1.76	2.91	0.97	1.75	3.02
50000	1	**2.99**	14.63	29.12	**2.91**	14.48	29.15
50000	2	3.53	10.13	18.31	3.55	10.17	18.27
50000	3	4.70	9.77	17.35	4.38	9.72	17.33
50000	4	5.29	10.20	17.76	5.29	10.07	17.13
50000	5	5.06	9.53	16.34	5.15	9.82	16.35
50000	6	4.65	9.48	16.62	4.74	9.44	16.77
50000	7	4.58	**9.09**	**15.83**	4.98	**8.85**	**15.38**
100000	1	**6.38**	31.13	62.11	**6.17**	30.84	61.66
100000	2	7.29	21.05	38.25	7.24	21.08	38.21
100000	3	8.03	19.77	35.48	8.13	19.66	35.16
100000	4	11.38	18.87	30.97	11.67	**18.56**	30.59
100000	5	13.37	20.02	30.08	12.67	19.44	29.99
100000	6	12.98	**19.28**	**28.96**	13.33	19.78	30.52
100000	7	12.92	20.13	31.75	13.13	20.27	**30.21**
500000	1	**31.04**	152.63	303.93	**30.37**	151.29	302.83
500000	2	36.71	107.83	196.13	36.29	107.05	195.86
500000	3	39.48	**101.17**	186.58	39.16	102.88	186.22
500000	4	40.12	103.47	186.33	40.03	103.15	185.61
500000	5	40.78	103.15	184.58	40.44	102.74	184.27
500000	6	41.02	103.22	**158.66**	40.89	102.84	184.72
500000	7	41.77	101.89	182.51	41.08	**101.88**	**181.83**

Tabla 3.4: Tiempos de ejecución, en segundos, en el sistema ORIGIN 2000 del CEP-BA, con distinto número de procesos, cantidades a devolver y granularidad simulada.

Tamaño	procs.	Versión A Granularidad			Versión B Granularidad		
		10	50	100	10	50	100
10000	1	0.78	3.72	7.30	0.76	3.36	6.94
10000	2	0.45	1.98	4.23	0.49	2.00	4.11
10000	3	0.35	1.63	3.16	0.34	1.60	3.17
10000	4	0.30	1.36	2.81	0.32	1.39	2.73
10000	5	0.30	1.24	2.47	0.29	1.26	2.54
10000	6	0.31	1.31	2.49	**0.28**	**1.19**	**2.39**
10000	7	**0.26**	**1.22**	**2.42**	0.30	1.21	2.45
50000	1	4.27	20.15	37.76	3.45	16.75	33.96
50000	2	2.22	10.89	22.05	2.08	10.19	21.27
50000	3	1.67	7.89	15.70	1.69	7.70	15.04
50000	4	1.45	6.68	13.10	1.37	6.02	12.80
50000	5	1.35	6.30	12.88	1.36	6.28	12.41
50000	6	**1.24**	**5.80**	11.51	**1.21**	5.70	11.41
50000	7	1.25	5.92	**11.48**	1.28	**5.59**	**10.89**
100000	1	8.54	42.39	75.87	9.00	41.99	85.30
100000	2	5.00	22.66	42.01	4.76	20.98	42.32
100000	3	3.53	15.08	32.64	3.57	15.71	32.80
100000	4	3.42	15.38	31.19	3.06	14.31	27.59
100000	5	**2.99**	15.20	30.43	3.12	15.30	28.74
100000	6	3.08	14.61	28.27	3.01	13.93	27.13
100000	7	3.00	**13.90**	**27.87**	**2.80**	**13.61**	**26.48**
500000	1	45.16	216.89	451.27	45.33	202.01	426.38
500000	2	25.25	126.72	245.27	26.92	126.74	253.95
500000	3	24.13	123.40	225.82	21.79	108.37	242.84
500000	4	24.40	114.56	222.85	25.75	**100.13**	221.48
500000	5	22.54	**99.56**	221.69	22.52	111.20	200.34
500000	6	23.10	116.16	212.56	**21.77**	101.76	207.14
500000	7	**20.71**	102.24	**197.81**	21.84	101.07	**197.81**

Tabla 3.5: Tiempos de ejecución, en segundos, en el sistema HPC160 de la Universidad Politécnica de Cartagena, usando procesadores que se encuentran en el mismo nodo, con distinto número de procesos, cantidades a devolver y granularidad simulada.

Tamaño	procs.	Versión A Granularidad		
		10	50	100
10000	1	0.16	0.77	1.54
10000	2	0.09	0.44	0.87
10000	3	0.08	0.33	0.66
10000	4	**0.06**	**0.28**	**0.55**
50000	1	0.79	3.87	7.73
50000	2	0.47	2.28	4.53
50000	3	0.49	2.21	4.38
50000	4	**0.36**	**1.51**	**3.00**
100000	1	1.58	7.77	15.50
100000	2	0.94	4.57	9.13
100000	3	0.91	4.52	8.79
100000	4	**0.90**	**4.44**	**8.69**
500000	1	8.42	42.36	95.38
500000	2	4.90	24.47	47.13
500000	3	4.66	22.31	44.55
500000	4	**4.51**	**21.88**	**43.53**

Tabla 3.6: Tiempos de ejecución, en segundos, en el sistema HPC160 de la Universidad Politécnica de Cartagena, usando procesadores que se encuentran en diferentes nodos, con distinto número de procesos, cantidades a devolver y granularidad simulada.

Tamaño	procs.	Versión A Granularidad		
		10	50	100
10000	2	**0.09**	0.44	0.94
10000	3	0.08	0.33	0.69
10000	4	0.06	0.28	0.55
10000	5	0.16	0.25	0.50
10000	6	0.25	0.24	0.49
10000	7	0.28	**0.23**	**0.47**
10000	8	0.26	0.25	0.50
50000	2	0.48	2.29	4.60
50000	3	0.46	2.20	4.37
50000	4	0.30	1.42	2.81
50000	5	0.29	1.31	2.60
50000	6	0.34	1.26	2.52
50000	7	**0.27**	1.03	**2.05**
50000	8	0.28	**1.01**	2.33
100000	2	0.98	4.62	9.11
100000	3	0.93	4.41	8.80
100000	4	0.94	4.38	8.68
100000	5	0.91	4.34	8.63
100000	6	0.93	4.37	8.64
100000	7	**0.47**	2.43	**4.43**
100000	8	0.61	**2.25**	4.46
500000	2	5.00	23.31	46.56
500000	3	4.74	22.50	44.69
500000	4	4.74	22.13	43.79
500000	5	4.65	21.85	43.39
500000	6	4.69	21.80	43.28
500000	7	**4.59**	**21.51**	**42.77**
500000	8	4.70	21.61	42.77

la ejecución secuencial (SUN5) es más rápido que el resto de procesadores. Si consideramos sólo los SUN1 (mejores tiempos en itálica) vemos que con una configuración homogénea sí se consigue una reducción en el tiempo de ejecución con la versión paralela.

- En cualquier caso, la reducción de tiempos entre el mejor y el peor caso es un porcentaje bastante pequeño debido a que las características de la red propicia que se produzcan colisiones y reenvíos constantes. Además, con mensajes grandes (mayor tamaño de problema) se producen más colisiones y la sobrecarga por las comunicaciones es mayor, lo que hace que con los tamaños menores el mejor tiempo se obtenga con un número mayor de procesadores que con los tamaños mayores, donde el óptimo se sitúa en 3 procesadores.
- En cuanto a la bondad de una u otra versión, se ve en la figura 3.5 que los resultados con la versión B son ligeramente mejores que con la versión A, principalmente al aumentar el número de procesadores, lo que es lógico ya que en la versión B se optimizan las comunicaciones.

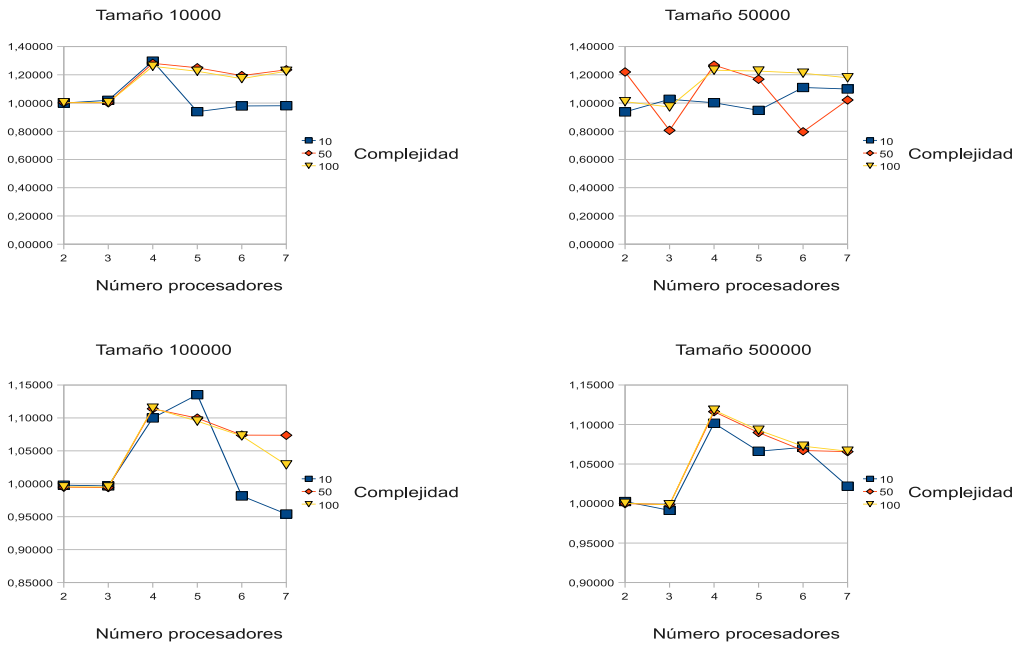

Figura 3.5: Cociente de los tiempos de ejecución entre las versiones A y B en el sistema SUNEt, para distintos tamaños y granularidades variando el número de procesadores.

- En PenFE (tabla 3.3):

  - Se observa que con independencia del tamaño del problema y de la granularidad, prácticamente no hay diferencias significativas entre los tiempos de ejecución obtenidos con las 2 versiones del algoritmo (lo que ocurre en todos los sistemas salvo en SUNEt), aunque cuando el tamaño del problema es grande sí existen algunas diferencias en cuanto al número de procesos que proporcionan esos valores mínimos. Esto ofrece la posibilidad de emplear polialgoritmos, es decir, usar algoritmos que, en nuestro caso, y en función del número de procesos y de la granularidad a emplear, ejecuten una u otra versión del algoritmo con el fin de reducir los tiempos de ejecución.
  - También hay que decir que siempre compensa utilizar más de un proceso cuando la granularidad es grande. En este caso el uso del paralelismo es más favorable que en el sistema anterior debido a las características de la red (mayor velocidad) y de los parámetros del sistema. La reducción de tiempos entre el mejor y el peor caso es un porcentaje cercano al 50 %. Aquí también influye el hecho de que todos los procesadores sean iguales y además sólo se asigna un proceso por procesador.

- En ORIGIN 2000 (tabla 3.4):

  - Ocurre igual que en el caso anterior: con independencia del tamaño del problema y de la granularidad, prácticamente no hay diferencias significativas entre los tiempos de ejecución obtenidos con las 2 versiones del algoritmo.
  - Se ve a grandes rasgos que siempre es conveniente paralelizar con independencia del tamaño del problema y de la granularidad utilizada. Esta situación es comprensible por las características del sistema ORIGIN 2000, que es de memoria virtual compartida y las comunicaciones son mucho más rápidas que en los dos sistemas anteriores (tabla 3.1).
  - El tiempo de ejecución paralelo se reduce unas 3 veces respecto al secuencial. En este caso también influye el hecho de que todos los procesadores sean iguales, y que sólo se asigna un proceso por procesador.

- En HPC160 (tablas 3.5 y 3.6 donde se muestran los resultados obtenidos ejecutando los procesos sobre procesadores que se encuentran en un mismo nodo o en nodos diferentes respectivamente):

- En la tabla 3.5 sólo aparecen resultados con hasta 4 procesos (asignando como máximo uno a cada procesador, pues cada nodo tiene 4 procesadores). En la tabla 3.6 aparecen resultados con hasta 8 procesos (los 4 primeros procesos se ejecutan en cada uno de los 4 procesadores de un nodo y los restantes en procesadores de otro nodo). Los primeros 3 datos de la columna de granularidad 10 y tamaño 10000 son iguales a los de la tabla anterior, sin embargo a partir de 5 procesadores se aprecia un aumento considerable en el tiempo de ejecución al utilizarse dos nodos. Este salto en el tiempo no parece coherente con los resultados para tamaños y granularidad mayores, y puede deberse al mayor coste proporcional de las comunicaciones para tiempos tan reducidos.
- Se observa que la paralelización también es deseable en HPC160, con independencia de que los procesos se ejecuten en un mismo nodo o en nodos diferentes. Cuando el tamaño del problema es pequeño, la reducción de tiempo es considerable mientras que cuando es grande, la reducción es un porcentaje bastante pequeño. Este comportamiento es extraño y puede deberse a que el coste de las operaciones de comunicación no varía linealmente con el tamaño de los mensajes, lo que no se tiene en cuenta si se usa el modelo $t_s + n \cdot t_w$ para el envío de n datos.
- En ambos casos sólo se han hecho pruebas con la versión A del algoritmo, ya que se ha visto que en las ejecuciones en sistemas diferentes no hay diferencias significativas en cuanto a tiempos de ejecución obtenidos ni en cuanto al número de procesos a emplear que permiten reducir estos tiempos.

En la tabla 3.7 se muestra una comparativa que refleja el número de procesos que se deben emplear en los diferentes sistemas para obtener los mínimos tiempos de ejecución en función del tamaño del problema y de la granularidad correspondiente. Estos resultados se han obtenido sólo para la versión A puesto que, como ha quedado comprobado en las tablas anteriores, no hay diferencias significativas en los resultados entre las versiones A y B sobre los diferentes sistemas. En el caso del sistema SUNEt los resultados no tienen en consideración el primer procesador ya que éste es más rápido que los otros 5. Para el sistema HPC160 sólo se han considerado los resultados de la tabla 3.6 y no los de la tabla 3.5 puesto que en ella sólo aparecen la ejecuciones con hasta 4 procesos. Además, los resultados son los obtenidos pudiendo emplear hasta 6 procesos debido a que los diferentes sistemas tienen diferentes números de procesadores, con el fin de que los resultados puedan ser comparados.

Tabla 3.7: Comparativa del número de procesos con el que se obtiene el menor tiempo de ejecución en los diferentes sistemas, usando hasta 6 procesos.

Cantidad	Granularidad	Sistema			
		SUNEt	PenFE	ORIGIN 2000	HPC160
10000	10	3	1	4	5
10000	50	6	5	5	6
10000	100	6	6	5	6
50000	10	4	1	6	5
50000	50	6	6	6	6
50000	100	6	5	6	6
100000	10	3	1	5	5
100000	50	3	4	6	5
100000	100	3	6	6	5
500000	10	3	1	5	5
500000	50	3	3	5	6
500000	100	3	6	6	6

Se observa que en diferentes sistemas el número óptimo de procesos a utilizar varía y que no siempre es lo mejor utilizar el mayor número posible de procesadores. Así, se muestra la conveniencia de utilizar una técnica de autooptimización que decida el número de procesos en función del tamaño de problema y de parámetros que reflejan el comportamiento del sistema.

En la tabla 3.8 se muestra el *speedup* de los diferentes sistemas considerando también sólo la versión A. Para cada sistema se han obtenido para cada tamaño y granularidad el máximo *speedup* alcanzado al variar el número de procesadores utilizados, y se ha calculado la media de todos estos valores. Como se puede observar, en los sistemas SUNEt y PenFE la ganancia con la paralelización no es importante, debido a las continuas colisiones que se producen durante las comunicaciones debido a las características de la red, mientras que en los otros sistemas al disponer de redes diferentes la ganancia es evidente. En el caso del sistema HPC160, se obtienen mejores resultados al considerar las ejecuciones internodo con más de 4 procesos en relación a las ejecuciones intranodo que sólo emplean hasta 4 procesos (uno por procesador). A pesar de que las comunicaciones entre nodos son más costosas, el hecho de emplear más cantidad de procesos hace que se obtengan mejores tiempos de ejecución, sobre todo cuando el tamaño del problema no es demasiado grande (con independencia de la granularidad).

Tabla 3.8: Media de los *speedup* máximos de los diferentes sistemas variando el tamaño del problema y la granularidad.

Sistema	SUNEt	PenFE	Origin 2000	HPC160 intranodo	HPC160 internodo
speedup máximo	1.08287	1.50725	2.39445	2.23616	2.83555

Para comparar con más detalle el comportamiento en los distintos sistemas, mostramos en la figura 3.6 el *speedup* para la versión A con diferentes configuraciones de ejecución. La gráfica 3.6, a) muestra el *speedup* al variar el número de procesadores, para los valores máximos de granularidad y tamaño. La b) lo muestra en el caso en que varía la granularidad, para el tamaño máximo y 6 procesadores, y la c) cuando varía el tamaño, para la máxima granularidad y 6 procesadores. En HPC160 se han considerado los resultados con dos nodos (tabla 3.6), y en SUNEt se ha tomado como tiempo secuencial el obtenido en SUN5 multiplicado por 2.5, que sería aproximadamente el tiempo secuencial en SUN1.

De los resultados mostrados en las gráficas se puede deducir que:

- Al aumentar el número de procesadores (figura 3.6 a)) se observa en general un pequeño aumento en el *speedup*, salvo en SUNEt, donde el *speedup* disminuye a partir de 3 procesadores. Así, a partir de dos procesadores la mejora que se obtiene con la paralelización es mínima. Esto es normal si tenemos en cuenta la poca granularidad de la computación en relación con las comunicaciones, cuyo coste aumenta considerablemente (de forma cuadrática) al aumentar el número de procesadores. Además, en SUNEt y PenFE la reducción en las prestaciones puede ser debida al incremento de colisiones en las comunicaciones debido al tipo de redes. Las colisiones no se contemplan en el modelo teórico de las comunicaciones, pero podrán tenerse en cuenta en el análisis empírico de las rutinas.

  La poca ganancia conseguida con el paralelismo no es un problema para el objetivo que perseguimos, pues no estamos tratando de optimizar los algoritmos, sino de modelarlos de manera que se tomen decisiones acertadas a la hora de ejecutarlos. De esta manera, un esquema con granularidad fina, en sistemas de distintas características en los que es difícil modelar de forma precisa las comunicaciones, es un buen banco de pruebas para la metodología de optimización que se propone.

- Cuando se aumenta la granularidad (figura 3.6 b)), que es un parámetro que sirve para incrementar el peso de la parte de computación en el tiempo de

ejecución total, el comportamiento es el esperable: se produce una pequeña mejora en el rendimiento de la paralelización, y la mejora es más apreciable en SUNEt y PenFE, donde las comunicaciones son más costosas.

- Cuando aumenta el tamaño del problema (figura 3.6 c)), se observa que se produce un empeoramiento de los resultados en todos los sistemas salvo en HPC160. Este comportamiento no es el esperable ya que se aumentan linealmente tanto el coste de la computación como el de las comunicaciones. Esta anomalía nos confirma en la idea de incluir en la metodología análisis empíricos con los que corregir los modelos teóricos.

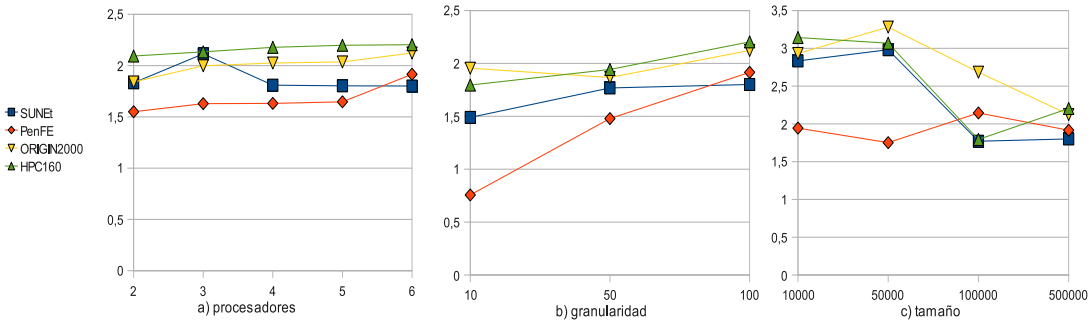

Figura 3.6: *Speedup* en los distintos sistemas: a) variando el número de procesos, con granularidad=100 y tamaño=500000; b) variando la granularidad, con tamaño=500000 y procesadores=6; c) variando el tamaño, con granularidad=100 y procesadores=6.

Las pruebas anteriores se han hecho para comprobar la utilidad de la paralelización para este tipo de esquemas algorítmicos y la disparidad de comportamiento en los distintos sistemas. Ha quedado comprobado que la paralelización es una buena opción para reducir los tiempos de ejecución en diferentes sistemas homogéneos, pero los resultados difieren en los distintos sistemas, por lo que es necesaria una metodología para la autooptimización del software para el sistema donde se instale. La ganancia que se obtiene con el uso del paralelismo también está condicionada por el tamaño del problema (figura 3.6, c)), lo que no es intuitivo para un usuario no experto, y ese fenómeno no se incluye en el modelo con una estimación de los parámetros de comunicaciones hecha con *ping-pong*. Así, es necesario un modelo con el que tomar las decisiones en función de la granularidad, el tamaño y el número de procesadores de que se dispone, y que incluya parámetros del sistema que correspondan al tipo de operaciones que se realizan en la rutina.

3.2.2. Evaluación de la autooptimización

En este apartado analizamos la metodología de autooptimización propuesta para sistemas homogéneos. Se utiliza el esquema del problema de las monedas sobre dos de las redes de procesadores anteriormente descritas: la red del laboratorio de computación paralela (SUNEt) y la red del laboratorio de arquitectura (PenFE). Se utilizan estas redes porque, como se ha comprobado en la subsección anterior, son en las que más dificultad hay para prever el comportamiento de la rutina, pero la metodología es válida para otros sistemas.

Tal y como se ha comentado, el esquema del problema de las monedas ha sido usado para mostrar experimentalmente cómo es la metodología de trabajo. El problema se resuelve con un algoritmo que tiene un coste secuencial de $O\left(\frac{N^2}{2v_i}\right)$ en el paso i. Los resultados experimentales se han obtenido con una versión paralela que hace uso de una distribución por bloques de las columnas de las tablas. Los experimentos se han hecho con valores grandes de q_i, el coste aritmético puede ser aproximado con la ecuación 3.4 y el de comunicaciones con la ecuación 3.6 pero, tal y como se ha mencionado, este coste puede estar lejos del real en una red Ethernet. Queremos estudiar cómo trabaja el método para el esquema general del que el problema de las monedas es un caso particular. Para esto los experimentos se han realizado con diferentes tamaños del problema, lo que significa variar los valores de N, v_i y q_i (el valor de n no influye en el estudio porque la computación se descompone en n pasos) y también se ha variado la granularidad de la computación.

Para obtener el valor de t_c es suficiente resolver un pequeño problema secuencial. Esto se puede hacer durante la instalación de la librería que contiene el esquema de programación dinámica y el valor se almacena para ser usado por las rutinas en tiempo de ejecución. Este valor puede ser almacenado en algún fichero o incluido en el código del esquema anterior al instalar la librería.

El coste de una ejecución particular puede ser modelado en tiempo de ejecución multiplicando el número de operaciones por el valor de t_c. Si el valor de t_c depende de algunos parámetros algorítmicos puede ser almacenado como una tabla $t_c(AP)$. Este no es el caso aquí, y t_c tiene un único valor.

En SUNEt, se obtienen valores diferentes de t_c, uno para SUN Ultra 5, y otro para SUN Ultra 1. Debido a la diferencia de velocidades es suficiente con estimar en tiempo de instalación el valor del parámetro en Ultra 5 (t_{c_5}). El valor para SUN Ultra 1 es $t_{c_1} = 2.5t_{c_5}$.

En PenFE todos los procesadores tienen igual velocidad y el valor de t_c se obtiene en tiempo de instalación en sólo uno de ellos.

Los valores de t_s y t_w se pueden obtener con un clásico *ping-pong*. Los valores estimados para SUNEt y PenFE se pueden ver en la tabla 3.1.

Aplicando estos valores al modelo dado por las ecuaciones 3.4 y 3.6 se obtiene el número óptimo teórico de procesadores a usar. Llamamos cp1 al método de estimación de los parámetros de comunicaciones por *ping-pong*.

En cp1 los valores de t_s y t_w son constantes y en algunos sistemas pueden no reflejar bien el comportamiento real de la rutina. Esto sucede en las redes que estamos considerando, donde las colisiones cuando el número de procesadores incrementa pueden hacer que sea mejor representar los valores de t_s y t_w como una función del número de procesadores. Tendríamos así $t_s(p)$ y $t_w(p)$, y estos valores pueden ser calculados durante el tiempo de instalación como una tabla en función de p. Para obtener esta tabla, se realizan diferentes ejecuciones con distintos tamaños del sistema (número de procesadores). Llamamos a este método de estimación de los parámetros de comunicación cp2. El tiempo de instalación necesario para la estimación de los parámetros en los experimentos fue de 6 segundos en PenFE, y de 9 segundos en SUNEt. Estos valores son bastante reducidos si los comparamos con los tiempos de ejecución que se van a producir con la ejecución de la rutina no optimizada. En los experimentos, se han resuelto problemas con $N = 10000$, y estos valores se han usado para estimar el tiempo de ejecución.

Cuando el valor de t_w depende también de N y no sólo de p, este valor se puede estimar mejor como una tabla bidimensional ($t_w(N, p)$) resolviendo en tiempo de ejecución algunos problemas seleccionados pero variando los valores de p y N. Llamamos a este método de estimación cp3. Hemos experimentado con valores $p = 2, 4, 6$ y $N = 10000, 100000$. En la instalación se han empleado 76 segundos en SUNEt, y 35 segundos en PenFE. Este incremento puede producir una mejor estimación de los parámetros del sistema y consecuentemente una mejor selección del número óptimo de procesadores que se deben emplear para resolver el problema, y además es todavía un tiempo de instalación reducido.

Cuando se resuelve un problema para diferente número de procesadores o de tamaño del problema, los valores de los parámetros se obtienen a través de la interpolación de los valores almacenados. En los experimentos se ha usado interpolación lineal para estimar t_w debido a que la variación teórica respecto a p y N es lineal.

En la tabla 3.9 se compara el número de procesadores seleccionados para cada uno de los 3 métodos anteriores con aquel con el que se obtiene el tiempo de ejecución más bajo. Los valores se han obtenido resolviendo problemas de diferentes tamaños ($N = 10000, 50000, 100000, 500000$) y con diferente granularidad en cada paso ($granularidad = 10, 50, 100$). Se muestra el número de procesadores estima-

dos para cada uno de los métodos de estimación considerados (cp1, cp2, cp3), y se han resuelto los problemas para todos los posibles procesadores, mostrándose en las columnas etiquetadas con tmb el número de procesadores con los que se obtiene el tiempo más bajo.

Tabla 3.9: Número de procesadores seleccionados por los diferentes métodos de estimación para la solución del problema y número de procesadores que proporciona un tiempo de ejecución más bajo. El tamaño del problema y el coste computacional en cada paso varían.

Granularidad												
	10				50				100			
SUNEt												
Tamaño	tmb	cp1	cp2	cp3	tmb	cp1	cp2	cp3	tmb	cp1	cp2	cp3
10000	1	1	1	1	6	6	1	6	6	6	1	6
50000	1	1	1	1	1	6	1	4	6	6	1	5
100000	1	1	1	1	1	6	1	4	5	6	1	5
500000	1	1	1	1	1	6	1	4	1	6	1	5
PenFE												
Tamaño	tmb	cp1	cp2	cp3	tmb	cp1	cp2	cp3	tmb	cp1	cp2	cp3
10000	1	6	1	1	5	7	1	7	6	7	5	7
50000	1	6	1	1	7	7	1	6	7	7	7	7
100000	1	6	1	1	4	7	1	6	6	7	7	7
500000	1	6	1	1	7	7	1	6	6	7	7	7

Se comprueba que el comportamiento de los tres métodos de estimación es diferente:

- cp1 usa una estimación de t_s y t_w donde no se reflejan las características de la red, y por lo tanto la estimación del tiempo de ejecución es optimista y se considera normalmente como óptimos más procesadores de los necesarios para obtener los tiempos de ejecución más bajos. Este hecho es más notable en PenFE por el tipo de red. En 10 de los casos se selecciona el número óptimo de procesadores, y en los 14 casos en que se falla el número seleccionado es mayor que el óptimo.

- cp2 estima los valores de los parámetros resolviendo un problema de tamaño y grano de computación reducido (para reducir el tiempo que se emplea durante la instalación). La variación de los parámetros del sistema con respecto a los parámetros algorítmicos no se refleja bien en el modelo y el número de procesadores estimados en este caso se queda lejos del óptimo. En 9 de los casos el

número de procesadores seleccionados es menor que el óptimo, y sólo en 2 es mayor.

- cp3 usa diferentes tamaños de problema y número de procesadores para estimar los valores de los parámetros del sistema, y la variación de los parámetros con respecto a N y p es mejor. La tendencia se invierte respecto a cp2: en 3 casos el número de procesadores seleccionados es menor que el óptimo y en 9 es mayor.

Si para cada método de estimación sumamos las diferencias entre el número de procesadores que seleccionan y el número de procesadores con un tiempo de ejecución más bajo, los valores que se obtienen son 49 para cp1, 41 para cp2, y 23 para cp3. Esto significa que la diferencia entre el tiempo de ejecución obtenido más bajo y el obtenido con los diferentes métodos, considerando un valor medio, debe ser mejor con la estimación dada por cp3.

Se comparan los tiempos de ejecución en la tabla 3.10. Para cada tamaño de problema, grano de computación y método de estimación, se muestra el cociente entre el tiempo obtenido con el número de procesadores estimados por el método y el tiempo de ejecución más bajo. Un valor de 1 significa que la estimación del número óptimo de procesadores se ha hecho correctamente. Se muestran los valores medios de los cocientes para cada método de estimación. El mejor método es cp3, con el que se obtiene una desviación del 5 % con respecto al tiempo de ejecución más bajo. Los otros métodos dan una desviación de 17 % (cp1) y 15 % (cp2). Por lo tanto, se ve con claridad que cp3 es el mejor método de estimación, ya que la variación de t_w con N y p se refleja mejor.

¿Es preferible proporcionar al usuario una rutina con capacidad de autooptimización tal y como acabamos de ver o es mejor proporcionársela con indicaciones sobre cómo usarla? En la tabla 3.11 se comparan los tiempos para diferentes tipos de usuarios modelados con aquellos que se obtendrían usando el anterior método cp3. Se muestran los cocientes con respecto a los tiempos de ejecución más bajos. Hemos considerado tres tipos de usuarios:

- Un usuario voraz (uv) sería el que usase el mayor número de procesadores disponibles para resolver el problema, con lo que obtendría en muchos casos un tiempo de ejecución lejos del óptimo.

- Un usuario más conservador (uc) decidiría reducir el número de procesadores a emplear (puede ser $\frac{p}{2}$) porque conoce el beneficio que se puede obtener usando paralelismo sin utilizar demasiados procesadores, pues sabe que esto podría incrementar el coste de las comunicaciones.

Tabla 3.10: Cociente entre el tiempo de ejecución obtenido con el número de procesadores estimado con los diferentes métodos de estimación y el tiempo de ejecución más bajo. Se varía el tamaño del problema y el coste computacional.

Granularidad									
	10			50			100		
SUNEt									
Tamaño	cp1	cp2	cp3	cp1	cp2	cp3	cp1	cp2	cp3
10000	1	1	1	1	1.33	1	1	1.44	1
50000	1	1	1	1.52	1	1.13	1	1.13	1.07
100000	1	1	1	1.04	1	1.13	1.23	1.6	1
500000	1	1	1	1.25	1	1.19	1.20	1	1.18
PenFE									
Tamaño	cp1	cp2	cp3	cp1	cp2	cp3	cp1	cp2	cp3
10000	1.64	1	1	1.02	1.60	1.02	1.02	1.02	1.02
50000	1.56	1	1	1	1.61	1.04	1	1	1
100000	2.03	1	1	1.07	1.65	1.10	1.10	1.10	1.15
500000	1.32	1	1	1	1.50	1.01	1.15	1.15	1.15
ratios									
cp1/tmb				cp2/tmb			cp3/tmb		
1.173				1.149			1.044		

- Un usuario experto en paralelismo (ue) que decidiría usar diferente número de procesadores (1, $\frac{p}{2}$ o p) dependiendo del grano de computación.

Tabla 3.11: Cociente entre el tiempo de ejecución obtenido con el número de procesadores usado por diferentes tipos de usuarios y el tiempo de ejecución más bajo. Se varía el tamaño del problema y el coste computacional.

Granularidad												
	10				50				100			
SUNEt												
Tamaño	uv	uc	ue	cp3	uv	uc	ue	cp3	uv	uc	ue	cp3
10000	2.31	2.23	1	1	1	1.33	1.33	1	1	1.41	1	1
50000	2.19	1.51	1	1	1.52	1.08	1.08	1.13	1	1.20	1	1.07
100000	3.78	1.45	1	1	1.04	1.15	1.15	1.13	1.23	1.19	1.23	1
500000	1.87	1.44	1	1	1.25	1.14	1.14	1.19	1.20	1.11	1.20	1.18
PenFE												
Tamaño	uv	uc	ue	cp3	uv	uc	ue	cp3	uv	uc	ue	cp3
10000	1.73	1.66	1	1	1.02	1.12	1.12	1.02	1.02	1.12	1.02	1.02
50000	1.53	1.57	1	1	1	1.08	1.08	1.04	1	1.10	1	1
100000	2.03	1.26	1	1	1.07	1.05	1.05	1.10	1.10	1.23	1.10	1.15
500000	1.35	1.27	1	1	1	1.01	1.01	1.01	1.15	1.18	1.15	1.15
ratios												
	uv/tmb				uc/tmb				ue/tmb			cp3/tmb
	1.433				1.245				1.070			1.044

En la figura 3.7 se resumen los resultados obtenidos por los distintos usuarios modelados y con los tres métodos de estimación utilizados. Se muestran los cocientes de los tiempos de ejecución respecto al menor tiempo (los ratios que aparecen en las tablas 3.10 y 3.11). En cada caso los cocientes se han obtenido como la media de los cocientes con los diferentes tamaños y granularidad que aparecen en las tablas. Se observa que el tiempo de ejecución obtenido para cada uno de los usuarios modelados es peor que el obtenido con cp3, e incluso en algunos casos hay una diferencia sustancial. El problema es que en las comparativas anteriores hemos considerado que se puede disponer de un usuario experto (ue) que sea capaz de determinar por su experiencia previa el número de procesadores a emplear. Se trata de usuarios teóricos donde el usuario experto además también sería experto en el problema y en el sistema en cuestión. De todas formas, podemos decir que el uso de técnicas de autooptimización es muy útil (incluso para usuarios expertos en paralelismo) con el fin

de lograr una adaptación al sistema que reduzca el tiempo de ejecución de cara a resolver el problema de manera óptima usando paralelismo.

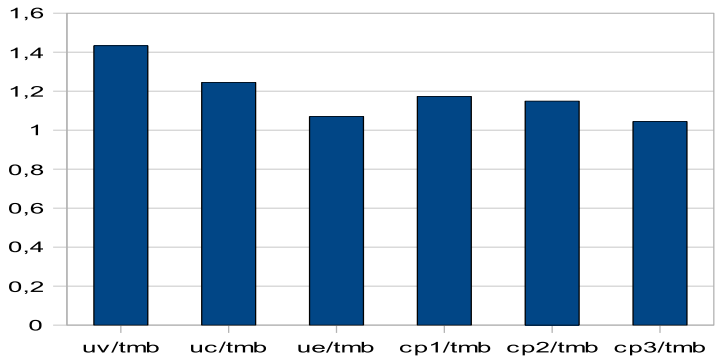

Figura 3.7: Cocientes de los tiempos de ejecución obtenidos por los distintos usuarios modelados y con los diferentes métodos de estimación de los parámetros con respecto al menor tiempo de ejecución.

3.3. Conclusiones

En este capítulo se ha hecho un estudio sobre la aplicación de técnicas de autooptimización a esquemas paralelos iterativos en un entorno homogéneo. Para ello se ha usado como ejemplo el esquema de programación dinámica del problema de las monedas, y se ha variado la granularidad pues estamos interesados en el esquema y no en el problema concreto. El estudio se ha realizado sobre sistemas de diferentes características pero considerándolos en todos los casos como homogéneos. En primer lugar se han obtenido resultados en 4 sistemas diferentes, con lo que se ha mostrado la ventaja que tiene la paralelización de los esquemas algorítmicos que hemos elegido para esta tesis. Además, por los resultados obtenidos al hacer pruebas sobre 2 de las anteriores redes, se puede decir que con la metodología propuesta es posible la no intervención del usuario para conseguir la autooptimización en el caso de los anteriores esquemas paralelos. El hecho de usar redes de ordenadores con prestaciones físicas similares supone una ventaja para el desarrollo de técnicas de autooptimización pues simplifica muchísimo el estudio teórico y la construcción de modelos. A pesar de esto, es muy usual disponer de sistemas formados por procesadores con características y rendimientos dispares, y por ello en el capítulo siguiente se hará un estudio exhaustivo sobre el uso de técnicas de autooptimización sobre sistemas heterogéneos y se comprobará la viabilidad de éstas.

Capítulo 4

Autooptimización en sistemas heterogéneos

Tras haber estudiado en el capítulo previo la autooptimización sobre sistemas homogéneos y conociendo que los sistemas informáticos suelen estar compuestos por elementos con características computacionales diferentes, en este capítulo se analiza la viabilidad técnica de la optimización automática en el diseño de algoritmos paralelos iterativos sobre sistemas heterogéneos. La idea principal consiste en aproximar automáticamente los valores óptimos de un número de parámetros algorítmicos de manera que se obtengan tiempos reducidos de ejecución. En este nuevo escenario, las decisiones que se deben tomar para reducir los tiempos de ejecución serán el número y tipos de procesadores a usar, el número de procesos a emplear y su distribución a los diferentes procesadores. De esta forma, las ideas vistas en el capítulo anterior para sistemas homogéneos se pueden usar en combinación con una solución al problema de asignación de tareas a procesadores.

Con ello, los usuarios tendrían a su disposición rutinas que se ejecutarían eficiente e independientemente de la experiencia del usuario en computación heterogénea y esquemas algorítmicos iterativos y, además, las rutinas se adaptarían automáticamente a una nueva red de procesadores o a un nueva configuración de la red.

4.1. Modelado para optimización automática sobre sistemas heterogéneos

Existen diferentes maneras de resolver problemas sobre sistemas heterogéneos de forma eficiente:

- Una forma consiste en diseñar algoritmos específicos para este tipo de sistemas con una distribución adecuada de datos entre procesos, tal y como ha sido

desarrollado en otros trabajos [17, 83].

- Otra aproximación puede ser desarrollar métodos (exactos o aproximados) con el fin de asignar procesos a procesadores a la vez que se hace uso de algoritmos homogéneos ya estudiados (algoritmos paralelos donde se consideran como idénticos los procesos de una topología lógica) para obtener la solución de los problemas.

Lo ideal sería hacer uso de la primera aproximación pues permitiría algoritmos muy eficientes pero ello conllevaría un alto coste por la reprogramación de los algoritmos previamente realizados.

Para que la segunda aproximación sea válida es necesario usar alguna técnica que permitiera obtener, en un tiempo de ejecución asumible, un reparto eficiente de la carga de trabajo a realizar. Una posible estrategia sería usar las rutinas existentes eficientemente sobre sistemas heterogéneos con un coste de programación adicional bajo y asumible [83]. Se genera un número de procesos iguales que se asignarán a determinados procesadores del sistema con el fin de reducir el tiempo de ejecución. Para lograr este objetivo se debe usar algún modelo teórico del tiempo de ejecución formado por parámetros que representan las características del sistema, tal y como se ha visto en el capítulo 1. Además, el coste del tiempo de decisión debe ser reducido, pues supone un coste adicional que añadir al de resolver el problema. Cuando el sistema es heterogéneo, distribuido o de carga variable, los valores de estos parámetros varían de un procesador a otro, estática y dinámicamente, y también con la asignación de procesos a procesadores. El problema de asignación general es NP-completo [90, 134].

Es evidente que con una situación tan compleja, no es posible en general alcanzar una selección óptima de procesadores, procesos y mapeo de estos sobre aquellos. Sin embargo, se están dedicando importantes esfuerzos para resolver estos tipos de problemas, tanto en sistemas con máquinas idénticas [48, 51, 80] como en sistemas heterogéneos y distribuidos [14, 121, 128]. En algunos casos, se puede aproximar a un tipo específico de problema (p.e. el esquema Maestro-Esclavo [15] o la computación iterativa [89]). Normalmente, se obtiene la solución aproximada del problema de asignación con algún método heurístico [51, 121, 128, 134].

En definitiva, tal y como ocurre en el caso homogéneo, es deseable que la técnica para decidir los parámetros a usar en la solución del problema sea desarrollada de forma que pueda ser automatizada para ser incluida en la librería junto con el algoritmo, de modo que sea posible obtener algoritmos o librerías con capacidad de autooptimización. Hecho esto, el sistema podría trabajar eficientemente con las ruti-

nas disponibles sin intervención del usuario, independientemente del sistema donde se están ejecutando las rutinas o incluso con independencia de las condiciones del sistema en un momento determinado.

Tradicionalmente la asignación de procesos a procesadores se ha tratado como un problema de asignación de un grafo de tareas [122]. En este capítulo se propone una aproximación diferente. A lo largo de los años, se han desarrollado rutinas y librerías para sistemas homogéneos cuyos tiempos de ejecución teóricos responden a fórmulas muy conocidas. Cuando estas fórmulas son ajustadas con precisión, pueden ser usadas para decidir sobre el número de procesadores, la topología de la red, la distribución de datos y otros parámetros a utilizar en la solución del problema. Si se considera un sistema heterogéneo con una rutina homogénea (una rutina paralela donde se consideran todos los procesos con las mismas capacidades computacionales y de comunicaciones), se puede emplear la misma fórmula modificada para decidir los parámetros óptimos con el fin de reducir los tiempos de ejecución, y ahora se hace necesario decidir el número de procesos y el mapeo de procesos a procesadores. De esta forma, la recodificación del código existente ya programado y probado no sería necesaria, y la ejecución satisfactoria sobre entornos heterogéneos se confía a la selección de parámetros y la asignación de procesos. La técnica es suficientemente genérica como para ser aplicada a algoritmos de diferente tipo.

Con el fin de conseguir este objetivo, seguimos la metodología propuesta en capítulos anteriores, para lo que se expresa el tiempo de ejecución de un algoritmo paralelo como una función de algunos parámetros algorítmicos y del sistema.

Tal y como se ha visto en el capítulo 1, los parámetros del sistema representan las características del sistema y pueden ser el coste de una operación aritmética o el coste de una operación aritmética particular (puede ser incluido un parámetro para la multiplicación y otro para la comparación), el tiempo de inicio de una comunicación (t_s) y el tiempo de envío de un dato (t_w). En sistemas heterogéneos habrá un valor distinto en cada procesador para cada operación aritmética, y valores distintos de parámetros de comunicación entre cada dos procesadores, con lo que se tienen vectores para los parámetros aritméticos y arrays bidimensionales para los de comunicaciones.

Los parámetros algorítmicos (AP) son aquellos que pueden ser modificados con el fin de obtener tiempos de ejecución reducidos, a diferencia de los parámetros del sistema que son fijos. Igual que en sistemas homogéneos, el objetivo es obtener parámetros algorítmicos que proporcionen tiempos de ejecución reducidos. Para ello habrá que adaptar a algoritmos homogéneos para sistemas heterogéneos la metodología de diseño de rutinas autooptimizables estudiada en el capítulo anterior.

Una aproximación simplificada sería considerar el valor de cada parámetro del sistema como el máximo de los valores en todos los procesadores (o entre cada par de procesadores para las comunicaciones), para después cambiar los valores estimados de un parámetro del sistema por el del procesador o red disponible en tiempo de ejecución (usando interpolación lineal) [36]. La rutina sería ejecutada con los parámetros algorítmicos seleccionados y con un proceso en cada uno de los procesadores seleccionados. De esta manera no se incluiría en la ejecución la heterogeneidad y sólo se tendrían en cuenta los procesadores usados en la solución final.

En un sistema heterogéneo podría ser preferible usar un número diferente de procesos y procesadores y asignar un número diferente de procesos a cada procesador usado, dependiendo de la capacidad computacional de cada procesador y de la capacidad de comunicaciones de la red empleada. De esta forma, el número de parámetros algorítmicos se incrementa respecto al caso homogéneo con un nuevo vector de parámetros, $d = (d_0, \ldots, d_{P-1})$ (con P el número de procesadores en el sistema, numerados de 0 a $P-1$), que representa el número de procesos asignados a cada procesador, y siendo $\sum_{i=0}^{P-1} d_i$ el número total de procesos. Ahora la fórmula teórica del tiempo de ejecución será más compleja debido a que los valores de los diferentes costes aritméticos varían de un procesador a otro y también con el número de procesos asignados a cada procesador, y los valores de los parámetros de comunicaciones también son diferentes entre cada par de procesos.

Con el fin de clarificar la notación usada, se explican a continuación las variables empleadas:

- Como se ha dicho, P representa el número de procesadores del sistema.
- T representa el número de tipos diferentes de procesadores. Una vez que se considera un sistema heterogéneo, los P procesadores del sistema se pueden agrupar en T tipos de procesadores (procesadores de exactamente el mismo tipo o similar pero agrupados con el fin de simplificar el problema de asignación). El número de procesadores de cada tipo se representa por un array $(p_0, p_2, \ldots, p_{T-1})$, donde p_i es el número de procesadores del i-ésimo tipo.
- Como se ha mencionado, el array $(d_0, d_1, \ldots, d_{P-1})$ representa el número de procesos asignados a cada procesador. Cuando $d_i = 0$ el procesador i no participa en la solución del problema.
- El número de procesos generados se representa por D, donde $D = \sum_{i=0}^{P-1} d_i$.
- Como se vio en el capítulo 1, el coste de una operación computacional básica

en cada procesador i es t_{c_i}, y los tiempos de inicio de comunicación y envío de un dato del procesador i al procesador j son $t_{s_{i,j}}$ y $t_{w_{i,j}}$, respectivamente.

Un algoritmo paralelo tiene un tiempo teórico de ejecución según la fórmula:

$$t(s, D) = t_{comp}(s, D) + t_{comm}(s, D) \tag{4.1}$$

donde s representa el tamaño del problema, D el número de procesos usados en la solución del problema, t_{comm} el coste de comunicaciones (incluye un término con t_s y otro con t_w) y t_{comp} es el coste de computación según la fórmula:

$$t_{comp}(s, D) = t_c \cdot n_{comp}(s, D) \tag{4.2}$$

donde t_c es el coste de una operación aritmética básica y n_{comp} el número de operaciones aritméticas básicas. n_{comp} y t_{comm} son divididos en diferentes partes, y en cada parte se consideran los valores de los procesadores con mayor carga computacional y mayor coste de comunicaciones. En entornos homogéneos los valores de t_c, t_s y t_w son iguales en diferentes procesadores, y la selección del número de procesadores y la topología lógica a usar (los parámetros que representan la topología aparecerían en n_{comp} y t_{comm}) son seleccionados para minimizar el valor de la fórmula 4.1. En un sistema heterogéneo, también se hace necesario seleccionar el número de procesos a usar (D) y el número de procesos asignados a cada procesador (d). Los valores de t_c, t_s y t_w están afectados por el número de procesos asignados a cada procesador y son una función de d ($t_c\,(d_0, d_1, \ldots, d_{P-1})$, $t_s\,(d_0, d_1, \ldots, d_{P-1})$ y $t_w\,(d_0, d_1, \ldots, d_{P-1})$). La forma en la que estos valores varían con la asignación de procesos también depende de las características computacionales del sistema y de comunicaciones de la red. Se podría considerar el coste de una operación aritmética básica en el procesador i como $d_i \cdot t_{c_i}$, y que el coste de comunicación no varía con el número de procesos que se asignan a un procesador.

Debido a que el problema de asignación es demasiado complejo para ser resuelto eficientemente, en la siguiente sección se considera de forma general una aproximación heurística para un esquema de programación dinámica.

En este capítulo estudiamos la distribución de procesos a procesadores para reducir el tiempo de ejecución cuando se utiliza un algoritmo homogéneo en un sistema heterogéneo. Esta aproximación permite abordar la paralelización eficiente en sistemas heterogéneos de los mismos algoritmos homogéneos estudiados en el capítulo anterior, por lo que se analizan las modificaciones en la metodología de autooptimización para adaptarla al nuevo tipo de sistema. Se utilizarán diversas técnica heurísticas para abordar este problema, y en el siguiente capítulo se abordará la aplicación sistemática

de metaheurísticas. El proceso de selección de los parámetros adecuados para lograr una ejecución eficiente se explica a continuación.

4.2. Un esquema de programación dinámica en sistemas heterogéneos

De nuevo haremos uso de un problema de programación dinámica, en este caso para ilustrar la técnica de autooptimización de esquemas paralelos iterativos en sistemas heterogéneos. Se vuelve a utilizar el "problema de las monedas", ya explicado en los anteriores capítulos. La solución se obtiene completando una tabla bidimensional donde las columnas representan el tamaño del subproblema correspondiente (la cantidad a devolver) y cada fila representa la decisión sobre un tipo de monedas (el número de monedas del tipo correspondiente que se dan). La versión del "problema de las monedas" utilizada es aquella en la que hay que devolver una cantidad C con un número mínimo de monedas, de entre las que se dispone de n tipos, con valores $v_1, v_2, \ldots, v_n$, y con una cierta cantidad de monedas de cada tipo, $q_1, q_2, \ldots, q_n$.

En una versión de paso de mensajes en un sistema heterogéneo la tabla (o las tablas) se puede asignar a los procesadores de diferentes formas:

- Se puede asignar un número diferente de columnas a cada procesador, dependiendo de la velocidad relativa de los procesadores (figura 4.1.a). En la figura a) se puede ver que el procesador P_1 recibe más carga de trabajo que P_0 al tener mayor velocidad.

- O se puede utilizar un algoritmo homogéneo, con un número de procesos a los que se asignan la misma cantidad de columnas. El número de procesos es distinto del de procesadores en este caso, y se asignará un número distinto de procesos a cada procesador dependiendo de la velocidad relativa (figura 4.1.b).

El proceso de selección de los parámetros se divide en cuatro pasos:

Paso 1: Identificación de parámetros

El primer paso consiste en la identificación de los parámetros del sistema y algorítmicos que aparecen en el tiempo de ejecución teórico.

Como en el caso homogéneo, el tamaño del problema viene dado por la cantidad a devolver (C), el número de tipos de monedas (n), y los valores y cantidades de monedas de cada tipo, que vienen dados por los arrays v y q.

En el caso homogéneo teníamos un único parámetro algorítmico, el número de procesadores, pero en el caso heterogéneo hay que decidir el número de procesos que

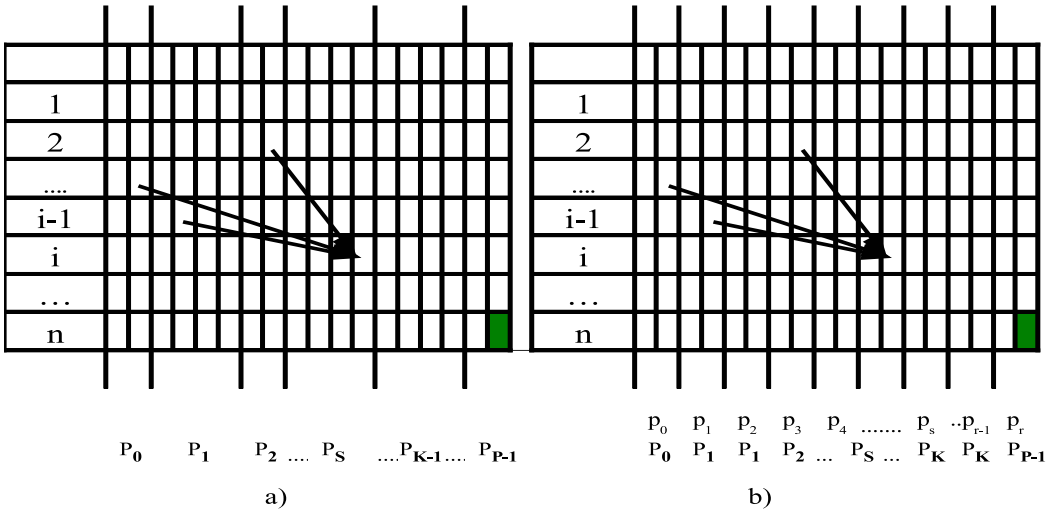

Figura 4.1: Distribución del trabajo y asignación de procesos a procesadores en un esquema de programación dinámica, en algoritmos heterogéneos (a) y homogéneos (b).

se asigna a cada procesador en el sistema. Si disponemos de P procesadores, hay que determinar el array de asignaciones, $d = (d_0, d_1, \ldots, d_{P-1})$, donde d_i indica el número de procesos que se asignan al procesador i, y $D = \sum_{i=0}^{P-1} d_i$ es el número total de procesos. La asignación se puede representar de otras maneras, pero lo más importante es que el número de parámetros a determinar aumenta respecto al caso homogéneo, y puede hacerse muy grande en un sistema de gran dimensión.

Paso 2: Modelado de la rutina

Una vez que se han determinado los parámetros se modela el tiempo de ejecución en función de ellos. En el problema de las monedas tenemos:

$$t\,(n, C, v, q, t_c(n, C, v, q, D, d), t_s(n, C, v, q, D, d), \\ t_w(n, C, v, q, D, d), D, d) \tag{4.3}$$

donde el tamaño del problema se representa por $s = (n, C, v, q)$, y hay que seleccionar los valores de los parámetros algorítmicos (D y d) con los que se obtiene el menor valor de t para los valores particulares de los parámetros del sistema en que trabajemos (t_c, t_s y t_w). Ahora tenemos valores diferentes de los parámetros del sistema en cada procesador (t_c se representa con un vector $t_c = \left(t_{c0}, t_{c1}, \ldots, t_{cP-1}\right)$), y t_s y t_w por matrices de tamaño $P \times P$, donde $t_{s_{i,j}}$ y $t_{w_{i,j}}$ representan los tiempos de inicio de una

comunicación y de envío de un dato entre procesos en los procesadores i y j. Es usual restringir estos parámetros del sistema a los procesadores de los que disponemos, pues el coste de una operación básica de computación o comunicación dependerá del procesador al que se asignen los procesos, y por eso los arrays t_c, t_s y t_w tienen dimensiones P y $P \times P$.

Como el trabajo se divide en n iteraciones, con computación seguida de comunicación, sólo es necesario obtener los costes de computación (t_{comp}) y comunicación (t_{comm}) en una iteración.

En cada iteración se reparte el trabajo entre los procesos de la siguiente forma: las C columnas se dividen equitativamente en bloques de columnas contiguas y se asignan de forma consecutiva entre los D procesos, con $\frac{C}{D}$ columnas para cada uno de ellos. El proceso con más trabajo es el número $D-1$, cuyo coste en la iteración i es:

$$\sum_{j=C-\frac{C}{D}+1}^{C} \left(1 + min\left\{\left\lfloor \frac{j}{v_i} \right\rfloor, q_i\right\}\right) \tag{4.4}$$

Esta fórmula se utiliza como coste de un proceso, y el modelo en un sistema heterogéneo se puede obtener de distintas maneras. Cuanto más se aproxime el modelo a la realidad mejor serán las decisiones que se tomen, pero la metodología de trabajo es independiente de la precisión del modelo. Como primera aproximación consideraremos como coste aritmético en un procesador el que se obtiene multiplicando el número de operaciones básicas de un proceso por el número de procesos asignados al procesador y por el coste de una operación básica en ese procesador, y como coste de las comunicaciones entre dos procesos el coste de las comunicaciones entre los procesadores donde se encuentran. De esta manera consideramos que la capacidad computacional de un procesador se divide linealmente entre los procesos que contiene, y que la capacidad de comunicación de la red no se ve afectada por el número de procesos que se asigna a un procesador. El tiempo de ejecución del algoritmo se obtiene como suma del máximo coste aritmético en todos los procesadores y el máximo coste de comunicaciones entre cada dos procesadores, tal y como se ve en la siguiente fórmula:

$$\max_{i=0,\ldots,P-1} \{t_{comp}(i)\} + \max_{i=0,\ldots,P-1;\ j=0,\ldots,P-1} \{t_{comm}(i,j)\} \tag{4.5}$$

Paso 3: Obtención de los parámetros del sistema

Una vez obtenido el modelo teórico del tiempo de ejecución, hay que decidir cómo obtener los valores de los parámetros del sistema. Estos valores se pueden obtener cuando se instala la rutina o la librería que la contiene, y se estiman normalmente

ejecutando una versión reducida de la rutina o de la parte de la rutina donde aparecen los parámetros.

Los valores de t_s y t_w se pueden obtener haciendo un *ping-pong* entre cada dos procesadores en el sistema, pero ya vimos en el capítulo anterior que una estimación de este tipo puede llevar a estimaciones no satisfactorias pues un *ping-pong* no es el tipo de comunicación que se utiliza en la rutina. En el caso homogéneo el coste de las comunicaciones se podía estimar mejor ejecutando comunicaciones del tipo del que se usan en la rutina y considerando su coste como función del tamaño del problema y del número de procesadores que se utilizan, con lo que se tiene un array unidimensional o bidimensional de parámetros. En el caso heterogéneo una solución de este tipo es mucho más compleja, pues los valores pueden depender del tamaño del problema y del array de asignaciones (d), que es de dimensión P, con lo que si se obtuviera cada valor de t_s y t_w en función de todas las posibles combinaciones su dimensión sería $S \times p \times \ldots^{P} \ldots \times p$ (con S un valor máximo del tamaño y p el número máximo de procesos por procesador), lo que es impracticable incluso aunque no consideremos todos los posibles valores del tamaño y de las asignaciones, sino sólo unos valores representativos. En los experimentos veremos algunas aproximaciones que simplifican la forma de estimar los parámetros y con las que se obtienen resultados satisfactorios.

El caso de los parámetros aritméticos es más sencillo, y los valores de t_c se pueden obtener resolviendo en secuencial en cada procesador en el sistema un problema de tamaño reducido.

Paso 4: Selección de los valores de los parámetros algorítmicos

En el momento de la ejecución, la rutina decidirá los valores de los parámetros algorítmicos con los que se tiene menor tiempo de ejecución teórico, y se resuelve con esos valores. Se pueden obtener comparando los tiempos de ejecución dados por el modelo para todas las posibles combinaciones de los parámetros algorítmicos y sustituyendo en el modelo los valores de los parámetros del sistema en función de los algorítmicos. El problema en el caso heterogéneo es que hay muchas posibles combinaciones de los parámetros algorítmicos, y no es posible evaluar en un tiempo razonable los tiempos para cada una de ellas. Así, se hace necesaria una aproximación que no evalúe todas las posibilidades sino algunas de las más prometedoras. En este capítulo, en la sección de experimentos, se mostrará cómo utilizar heurísticas con métodos aproximados, y en el siguiente capítulo se analizará el uso de metaheurísticas.

4.3. El problema de asignación

El problema de obtener los valores de los parámetros algorítmicos con los que se obtiene el menor tiempo de ejecución teórico puede representarse mediante un árbol de asignaciones. Cada nodo del árbol representará una posible combinación de los parámetros, y tendrá asociado el tiempo teórico correspondiente a esa combinación. De esta manera, el problema de optimización consiste en recorrer todo el árbol (o partes del árbol donde se sabe que se encontrará la solución óptima) para obtener el nodo con menor coste asociado. Un posible ejemplo de árbol de asignaciones se analizó en el capítulo 1, y aquí analizamos el tipo de árboles que utilizaremos en sistemas heterogéneos en los que consideramos los procesadores agrupados por tipos.

Como hemos dicho, si consideramos algoritmos homogéneos en los que el orden en que se asignen los procesos a los procesadores no importa al ser todos los procesos idénticos, se puede representar la asignación con un array $d = (d_0, d_1, \ldots, d_{P-1})$, donde d_i almacena el número de procesos que se asigna al procesador i. Otra posibilidad consiste en almacenar para cada proceso el procesador al que se asigna. Así, una vez que se ha decidido cuántos procesos emplear, la solución se representa por un vector, a, cuyo número de componentes será teóricamente ilimitado (es posible tener un número ilimitado de procesos). El número de componentes estará limitado por el número máximo de niveles alcanzables en el árbol. Cada valor a_i representa el procesador donde se asigna el proceso i. En este caso el espacio de soluciones se representa por un árbol en el que cada nivel representa un proceso, y cada nodo hijo de un nodo dado representa uno de los P procesadores a los que el proceso puede ser asignado. Si agrupamos los P procesadores en T tipos, se pueden considerar para cada nodo T descendientes, reduciéndose el número de nodos en el árbol y consecuentemente el tiempo de optimización. Es conveniente agrupar los procesadores en tipos conforme a sus afinidades de computación y comunicación (aun no siendo todos los procesadores que se agrupan en un tipo exactamente iguales) pues si no el árbol resultante sería grande y por lo tanto su recorrido costoso. Cada nodo del árbol representa una posible solución al problema de asignación, donde ésta puede estar formada por cualquier número de procesos. En el nivel l del árbol se consideran las asignaciones de l procesos. No es importante el orden en el que los procesos son asignados a los procesadores, y además más de un proceso puede ser asignado a un procesador. El árbol tiene la forma de la figura 4.2 para un sistema con 2 tipos de procesadores (el árbol es representado sólo hasta el nivel 4), y la de la figura 4.3 para un sistema con 4 tipos de procesadores (el árbol es representado hasta el nivel 2).

Considerar un número reducido de procesadores no es muy realista pero, como

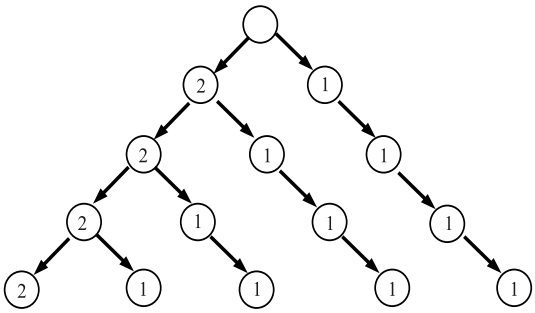

Figura 4.2: Árbol de asignaciones de nivel 4 para 2 tipos de procesadores.

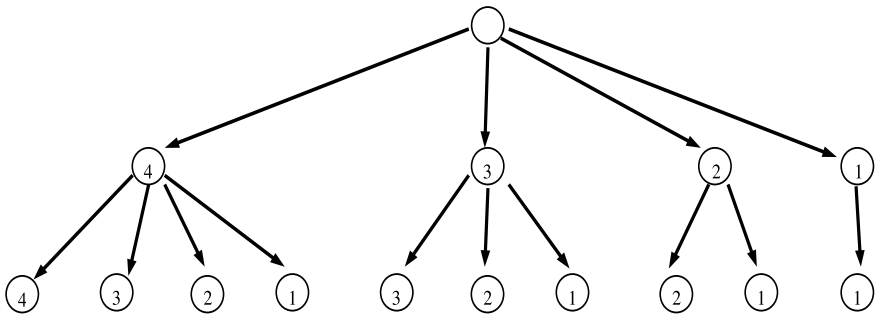

Figura 4.3: Árbol de asignaciones de nivel 2 para 4 tipos de procesadores.

se verá más adelante, la situación no está lejos de la realidad si consideramos una red donde puede haber una gran cantidad de procesadores pero normalmente de un número reducido de tipos diferentes (esto ocurre por ejemplo en las 2 redes consideradas en las pruebas realizadas, una con 6 procesadores de 2 tipos diferentes (SUNEt) y otra con 21 procesadores de 4 tipos distintos (TORC)). Cuando haya una gran cantidad de tipos diferentes de procesadores, con el fin de reducir el tamaño del árbol de asignación se pueden considerar como del mismo tipo procesadores con características parecidas.

Se trata de resolver un problema de optimización que puede ser abordado usando técnicas de recorrido y poda, como *backtracking* o *branch and bound*, en el árbol de soluciones. La función que hay que optimizar es el tiempo de ejecución teórico que se estima en cada nodo con los valores de los parámetros del sistema correspondientes a los parámetros algorítmicos (D y d) representados por el nodo.

Este problema de asignación es bien conocido que es NP [90], y sería muy costoso resolver el problema general de optimización, pero en la práctica se pueden usar estrategias para eliminar nodos del árbol. Analizaremos en las secciones siguientes el uso de técnicas heurísticas en combinación con métodos exhaustivos de recorrido del árbol de asignaciones. Haremos el estudio en dos sistemas reales y estudiaremos el comportamiento de los métodos propuestos en simulaciones de sistemas de grandes dimensiones.

Tal como se hace con métodos exhaustivos de recorrido del árbol, consideraremos que cada nodo tiene asociada una serie de valores:

- Tiempo de Ejecución Estimado ($EET(nodo)$).
- Tiempo de Ejecución más Bajo ($LET(nodo)$).
- Tiempo de Ejecución más Alto ($GET(nodo)$).

En cada nodo, $LET(nodo)$ y $GET(nodo)$ son los límites inferior y superior para la solución óptima que se obtendría con los nodos descendientes de *nodo*. Hay varias posibilidades para obtener estas cotas [26], y algunas de las más comunes y que usaremos en este trabajo son:

- $LET(nodo)$ se obtiene considerando que todos los procesadores no ocupados participan en la computación, con lo que se usa toda la potencia computacional disponible y se reduce al máximo el tiempo de ejecución. En la sección de resultados experimentales se mostrará con unos ejemplos cómo se calcula.

- $GET(nodo)$ se obtiene, por ejemplo, a través de una técnica de avance rápido, seleccionando en cada paso la asignación al procesador de los disponibles que proporciona el menor tiempo de ejecución teórico.

El problema de optimización consiste en encontrar el nodo con $EET(nodo)$ más bajo. $LET(nodo)$ y $GET(nodo)$ se usarán para limitar el número de nodos a evaluar y la altura del árbol. Tanto en el esquema *backtracking* como en el *branch and bound*, se comparan los valores $LET(nodo)$ y $MEET = \text{mín}_{nodo \in nodos\ evaluados}\{GET(nodo)\}$ en cada nodo con el fin de continuar por la rama de ese nodo cuando $LET(nodo) \leq MEET$. En la siguientes secciones se muestran los resultados obtenidos en los dos sistemas previamente referenciados y se explica cómo obtener la solución óptima a través de la búsqueda en el árbol.

4.4. Resultados experimentales

Los experimentos se han realizado en los sistemas SUNEt y TORC, cuyas características han sido explicadas en la sección de hardware del capítulo 1.

Se ha usado el esquema del problema de las monedas, explicado en el capítulo 2 y en secciones previas de este capítulo, para mostrar experimentalmente cómo funciona la metodología, pero lo que queremos es estudiar cómo trabajaría el método para un esquema general. Otros esquemas algorítmicos siguen esquemas paralelos similares al del problema de las monedas, por lo que se emplearía la misma metodología.

El problema se resuelve con un algoritmo de coste secuencial de aproximadamente $\theta\left(\left(1 + q_i\right) C\right)$ en cada paso i. Se han obtenido los resultados experimentales para una versión paralela con una distribución por bloques de las columnas de las tablas. Tal y como ocurre en el caso de sistemas homogéneos, el coste aritmético es el de la fórmula 3.2, y el coste de comunicaciones sería modelado por la fórmula 3.6, pero este coste puede estar lejos del real en una red Ethernet, debido a que las colisiones dificultan modelar teóricamente las comunicaciones, y posiblemente los valores de t_s y t_w se obtendrían mejor a partir de la ejecución de partes seleccionadas del algoritmo, tal como comprobamos en el capítulo anterior que era preferible en un sistema homogéneo.

Se han realizado los experimentos con diferentes tamaños del problema, lo que significa variar el valor de C, v_i y q_i (el valor de n no influye en el estudio debido a que la computación se descompone en n pasos). También se ha variado el grano de la computación en cada paso.

Como en el caso de los sistemas homogéneos, para obtener el valor de t_c es suficiente resolver un pequeño problema en modo secuencial en cada procesador del

sistema. Esto se puede hacer durante la instalación de la librería que contiene el Esquema de Programación Dinámica, y el valor puede almacenarse para ser usado por las rutinas en tiempo de ejecución. Este valor se almacenaría en algún fichero o se incluiría en el código del esquema antes de instalar la librería. De esta forma tenemos almacenado el coste de las computaciones elementales, y el coste estimado de una ejecución particular puede ser modelado en tiempo de ejecución multiplicando el número de operaciones computacionales básicas por el valor de t_c. Cuando el valor de t_c depende de algún parámetro algorítmico, puede ser almacenado como una tabla $t_c(AP)$. Este no es el caso en el problema con el que estamos trabajando, t_c tiene un único valor en cada procesador, y en tiempo de ejecución para cada procesador el tiempo teórico asociado a un proceso se multiplica por el número de procesos asignados al procesador. Para los procesadores duales en TORC, debemos considerar un proceso por procesador cuando se asignan dos procesos a un nodo, y el valor de t_c no cambia. En SUNEt, se han obtenido dos valores diferentes de t_c, uno para SUN Ultra 5, y otro para SUN Ultra 1. Debido a que SUN Ultra 1 es aproximadamente 2.5 veces más lento que SUN Ultra 5, es suficiente estimar el valor del parámetro en Ultra 5 (t_{c_5}) en tiempo de instalación. El valor para SUN Ultra 1 es $t_{c_1} = 2.5t_{c_5}$. En TORC se obtienen valores en cada tipo de procesador.

Los valores de t_s y t_w han sido obtenidos usando el clásico *ping-pong* explicado en el capítulo anterior. De esta forma, los valores estimados en SUNEt son $t_s = 904\mu\ sec$ y $t_w = 3.56\mu\ sec$. En TORC, se obtiene una tabla de valores para t_s (tabla 4.1) y otra para t_w (tabla 4.2). Recordamos que de la red TORC [125] detallada en la sección de hardware del capítulo 1, de 21 nodos de diferentes tipos, sólo se ha empleado alguno de los procesadores dual (DPIII), un Pentium III (SPIII) 600 Mhz, uno de los AMD Athlon (Ath) y el Pentium 4 1.7 Ghz (17P4). Los valores de la diagonal corresponden a comunicaciones entre procesos en el mismo procesador y para DPIII entre dos procesos en el mismo nodo. DPIII aparece dos veces debido a que las comunicaciones entre procesos en dos nodos son diferentes a aquellas entre procesos en diferentes procesadores del mismo nodo. No aparecen valores en la diagonal para 17P4, debido a que no es posible comunicar procesos asignados a este procesador. Como es lógico, los valores de t_s y t_w son menores en comunicaciones dentro de un nodo que intranodo. Se observa que el coste de inicio de comunicación (tabla 4.1) varía entre procesadores de distintos tipos, pues incluye operaciones internas, pero la diferencia en el coste de envío de un dato (tabla 4.2) entre procesadores distintos es mínima y se debe sólo a la toma de datos.

Las grandes diferencias de los valores de los parámetros en los diferentes sistemas producirán predicciones y selecciones de parámetros algorítmicos totalmente diferen-

Tabla 4.1: Valor de t_s (en μ *sec*) entre cada par de procesadores en TORC.

	DPIII (nodo 1)	DPIII (nodo 2)	SPIII	Ath	17P4
DPIII (nodo 1)	21	68	88	76	84
DPIII (nodo 2)	68	21	88	76	84
SPIII	88	88	44	141	150
Ath	76	76	141	16	69
17P4	84	84	150	69	

Tabla 4.2: Valor de t_w (en μ *sec*) entre cada par de procesadores en TORC.

	DPIII (nodo 1)	DPIII (nodo 2)	SPIII	Ath	17P4
DPIII (nodo 1)	0.10	0.38	0.38	0.38	0.37
DPIII (nodo 2)	0.38	0.10	0.38	0.38	0.37
SPIII	0.38	0.38	0.09	0.37	0.37
Ath	0.38	0.38	0.37	0.05	0.37
17P4	0.37	0.37	0.37	0.37	

tes. En TORC la velocidad de comunicación es prácticamente la misma entre los diferentes procesadores (especialmente para t_w, donde las diferencias son debidas a los experimentos) debido a que la red usada es la misma, pero entre dos procesos en un procesador dual las comunicaciones son más rápidas.

Con estos valores (no dependientes del tamaño del problema y los parámetros algorítmicos) y usando el modelo dado por las fórmulas 3.2 y 3.6, el número óptimo teórico de procesadores y el número de procesos a emplear en cada procesador puede ser estimado y el problema se resuelve con este número de procesadores y la asignación de procesos. Llamamos a esta estimación de los parámetros de comunicación cp1.

En cp1 los valores de t_s y t_w son constantes para cada par de procesadores, y en algunos sistemas pueden no reflejar de manera fidedigna el comportamiento del sistema, pues los costes de inicio de comunicación y de envío de un dato pueden depender de parámetros como el tamaño de la comunicación y el número de procesos que intervienen en ella. Además, en redes tipo Ethernet el número de colisiones aumenta cuando el número de procesos crece. Por estos motivos, consideramos otro método de estimación de los parámetros al que llamamos cp2, en el que los valores de t_s y t_w se representan como una función del número de procesos (no se ha variado el tamaño de los mensajes en la estimación de t_s y t_w). Los valores pueden ser calculados durante la

instalación como una tabla que depende de D. Para obtener la tabla, se deben realizar ejecuciones para un tamaño de problema seleccionado. Se han hecho las estimaciones con los valores más pequeños del tamaño del problema y granularidad de entre los que se han hecho experimentos ($C = 10000$ y $granularidad = 10$), para tener un tiempo reducido de estimación de los parámetros. Para cada número de procesos el valor de los parámetros de comunicaciones es el de la tabla calculada con esos experimentos. Estos valores han sido usados para estimar el tiempo de ejecución para los otros casos considerados. Los experimentos han sido realizados resolviendo problemas de diferentes tamaños ($C = 10000, 50000, 100000, 500000$) y diferente granularidad en cada paso ($granularidad = 10, 50, 100$).

A continuación se muestra cómo el problema de optimización puede ser usado en combinación con los dos métodos de predicción en cada uno de los dos sistemas considerados. Los resultados de algunas simulaciones muestran la aplicación práctica del método a sistemas de mayor dimensión que SUNEt y TORC.

4.4.1. Predicciones en SUNEt

Se puede suponer que la solución del problema de optimización es una tarea muy costosa pero, como veremos, no siempre es así, y los métodos podrían ser incluidos en la librería para decidir cuántos procesos y qué procesadores usar en la solución del problema durante el tiempo de instalación o incluso en tiempo de ejecución.

En SUNEt hay dos tipos de procesadores, y el árbol de solución es binario (figura 4.4). Cada nodo se etiqueta como 5 o 1, indicando que el correspondiente proceso se asigna a una SUN Ultra 5 o a una SUN Ultra 1. Los procesos asignados a SUN Ultra 1 se reparten equitativamente entre todos los procesadores. Cuando un tipo de procesadores tiene asignado un número de procesos más grande que el número de procesadores del mismo tipo, entonces más de un proceso se asigna a algún procesador y se incrementa el coste aritmético en dicho procesador.

Inicialmente se asigna cada proceso al procesador con mayor capacidad computacional (SUN Ultra 5), y un nodo que representa una asignación a SUN Ultra 1 no tiene descendientes etiquetados como 5. De cualquier forma, hay que limitar la altura del árbol, pues representa el número de procesos a emplear. Una posibilidad es obtener un tiempo computacional máximo alcanzable. Esto se puede estimar resolviendo un pequeño problema (por ejemplo, $C = 10000$ y $granularidad = 10$) en SUN Ultra 5, y estimando el mínimo tiempo computacional para un cierto C y *granularidad*. Se estima el coste de comunicaciones para cada número de procesos usando los valores más bajos de t_s y t_w y la fórmula 3.6. Por lo tanto, para cada nodo del árbol, el tiempo

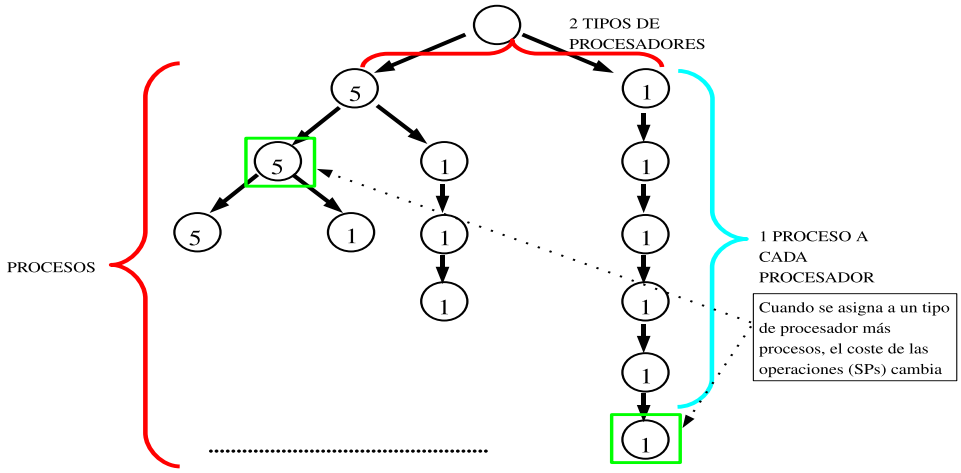

Figura 4.4: Árbol de soluciones para SUNEt.

de ejecución más bajo alcanzable es la suma del coste mínimo de computación y del coste de comunicación de ese nodo. Cuando este coste estimado más bajo es mayor que el valor mínimo de $GET(nodo)$ en los nodos explorados, no hay necesidad de explorar los sucesivos niveles.

El árbol con l niveles tiene $\frac{(l+2)(l+1)}{2}$ nodos. No es una cantidad especialmente grande si consideramos que algunos nodos pueden ser eliminados según el criterio anterior. En general, con l niveles y T tipos de procesadores, el número de nodos en el árbol de asignación es:

$$\left(\begin{array}{c} T-1 \\ 1 \end{array} \right) + \left(\begin{array}{c} T \\ 2 \end{array} \right) + \ldots + \left(\begin{array}{c} T+l-1 \\ l \end{array} \right) = \left(\begin{array}{c} T+l \\ l \end{array} \right) \tag{4.6}$$

y se hace evidente la necesidad de eliminar nodos.

La única diferencia entre cp1 y cp2 es la forma en la que se estima el tiempo de ejecución en cada nodo. La tabla 4.3 compara los tiempos de ejecución obtenidos con el número de procesos seleccionados por cada método (columnas tres y cuatro), para los diferentes valores considerados de C y *granularidad*, con los tiempos de ejecución experimentales más bajos. Se ha usado el error relativo respecto de *masbajo* ($\frac{|metodo-masbajo|}{masbajo}$) para representar la desviación del tiempo de ejecución con respecto al más bajo obtenido. La media de las desviaciones es 0.67 para cp1 y 0.17 para cp2, lo que representa desviaciones del 67 y 17 por ciento respecto al tiempo más bajo obtenido en los experimentos. Si consideramos sistemas con SUN Ultra 5 y un número de SUN Ultra 1 variando de 2 a 5, la media de las desviaciones para cp1 es

0.42, y para cp2 es 0.12 (tabla 4.4). Las predicciones de cp2 son mejores debido a que en cp1 las características de la red no se consideran de cara a las predicciones. Si se compara el número de procesos seleccionados por cada uno de los métodos y aquéllos que proporcionan un tiempo de ejecución más bajo, la diferencia media es 4.6 con cp1 y 2.2 con cp2. Se muestra la media de las desviaciones con respecto al tiempo de ejecución más bajo (PROMEDIO), y la desviación de la suma de los tiempos totales de ejecución con respecto a la suma de los tiempos más bajos (TOTAL). En las tablas 4.3 y 4.4 también se comparan los tiempos que diferentes tipos de usuario pueden obtener (columnas cinco, seis y siete) con aquellos obtenidos con cp1 y cp2 (columnas tres y cuatro).

Tabla 4.3: Desviación del tiempo de ejecución con el número de procesos seleccionados con los métodos cp1 y cp2 y diferentes tipos de usuarios, con respecto a los tiempos de ejecución obtenidos más bajos. Para diferentes valores de C y *granularidad*. En SUNEt con los seis procesadores.

C	*granularidad*	cp1	cp2	ue	uc	uv
10000	10	0.90	**0.00**	**0.00**	0.56	0.53
10000	50	0.30	**0.25**	0.31	0.82	0.34
10000	100	**0.18**	0.22	**0.18**	0.72	0.34
50000	10	2.06	**0.00**	**0.00**	0.92	2.22
50000	50	**0.22**	0.23	0.42	1.16	0.47
50000	100	**1.07**	1.48	1.24	3.24	1.92
100000	10	0.94	**0.00**	**0.00**	0.90	1.89
100000	50	0.62	**0.00**	0.07	0.74	0.67
100000	100	**0.03**	0.15	0.72	0.97	0.87
500000	10	0.86	**0.00**	**0.00**	0.88	1.02
500000	50	0.49	**0.00**	0.03	0.72	0.62
500000	100	0.37	**0.00**	0.44	0.42	0.41
PROMEDIO		0.67	**0.17**	0.28	1.04	0.94
TOTAL		0.43	**0.05**	0.32	0.67	0.60

Si se proporciona a un usuario una rutina con capacidad de autooptimización, la desviación del tiempo de ejecución más bajo podría ser mejor con cp2, pero ¿sería una buena alternativa o sería preferible proporcionar la rutina al usuario junto con algunas indicaciones sobre cómo usarla?

Al igual que se ha hecho en el capítulo de autooptimización en sistemas homogéneos, consideraremos las decisiones tomadas por tres tipos de usuarios para poderlas comparar con los resultados obtenidos:

Tabla 4.4: Desviación media del tiempo de ejecución con el número de procesos seleccionados con los métodos de predicción cp1 y cp2 y diferentes tipos de usuarios con respecto a los tiempos de ejecución más bajos obtenidos. En SUNEt.

sistema	cp1	cp2	ue	uc	uv
completo	0.67	**0.17**	0.28	1.04	0.94
número variable de SUN Ultra 1	0.42	**0.12**	0.15	0.60	0.81

- Un usuario voraz (uv) usaría el máximo número de procesadores disponibles para resolver el problema (un proceso por procesador), y por lo tanto obtendría en la mayoría de los casos un tiempo de ejecución lejano del óptimo.

- Un usuario más conservador (uc) decidiría usar un número reducido de procesadores, ya que conoce los beneficios que se pueden obtener usando paralelismo y también que si se usan demasiados procesadores se incrementa el coste de comunicaciones, con lo cual disminuye este beneficio.

- Un usuario experto (ue) decidiría usar un número diferente de procesos (1, 3 y P) dependiendo de la granularidad de la computación. Este usuario experto estaría especializado en computación heterogénea, esquemas de programación dinámica y el sistema en el que se está resolviendo el problema. Es obvio que sería complicado disponer de un usuario con estas características debido a lo subjetivo de las decisiones que debería tomar.

Podemos ver que los tiempos de ejecución obtenidos por cada uno de estos usuarios son peores que los obtenidos con cp2, y en algunos casos la diferencia es sustancial. En la tabla 4.4 se muestra la desviación de cp1 y cp2 y de cada uno de los anteriores usuarios para un sistema completo y cuando se varía el número de SUN Ultra 1 disponibles de 2 a 5. En la tercera fila se muestra la media de las desviaciones de los cuatro sistemas. cp2 también es el método preferido si se consideran las diferentes configuraciones del sistema, aunque los resultados obtenidos por el hipotético "usuario experto" se acercan a los obtenidos con cp2. Se puede concluir que el uso de técnicas de autooptimización en sistemas heterogéneos es útil para proporcionar a los usuarios rutinas que se adapten automáticamente al sistema donde se ejecuten y que obtengan tiempos de ejecución reducidos de una manera transparente al usuario, lo que es especialmente interesante en entornos heterogéneos, debido a su mayor complejidad.

4.4.2. Predicciones en TORC

En este sistema se consideran procesadores de cuatro tipos diferentes: 17P4, Ath, SPIII y DPIII. t_c se obtiene resolviendo un problema pequeño en cada uno de los procesadores. Los tiempos de ejecución, en segundos, son 0.16 para 17P4, 0.20 para Ath, 0.45 para SPIII y 0.49 para DPIII. Se usan estos tiempos para estimar los tiempos de ejecución para problemas con mayor tamaño o granularidad.

El árbol de soluciones es parecido al de SUNEt, pero en este caso cada nodo tiene un máximo de cuatro hijos (figura 4.5). En la figura, los números 0, 1, 2, y 3 representan los tipos de procesador 17P4, Ath, SPIII y DPIII, en este orden. Los procesadores de los tipos 17P4, Ath y SPIII son monoprocesadores, y por lo tanto, cuando más de un proceso se asigna a uno de estos procesadores, éstos son asignados al mismo procesador. Tal como estaba configurado el sistema, en la realización de los experimentos al procesador 17P4 sólo se podía asignar un proceso, lo que produce una reducción en el número de nodos del árbol. Para los procesadores del tipo DPIII, hay ocho nodos disponibles, con dos procesadores por nodo. Los procesos asignados a DPIII son dos por cada nodo. En el caso de que más de 16 procesos se asignen a DPIII, se asignarían de forma cíclica procesos adicionales a los procesadores.

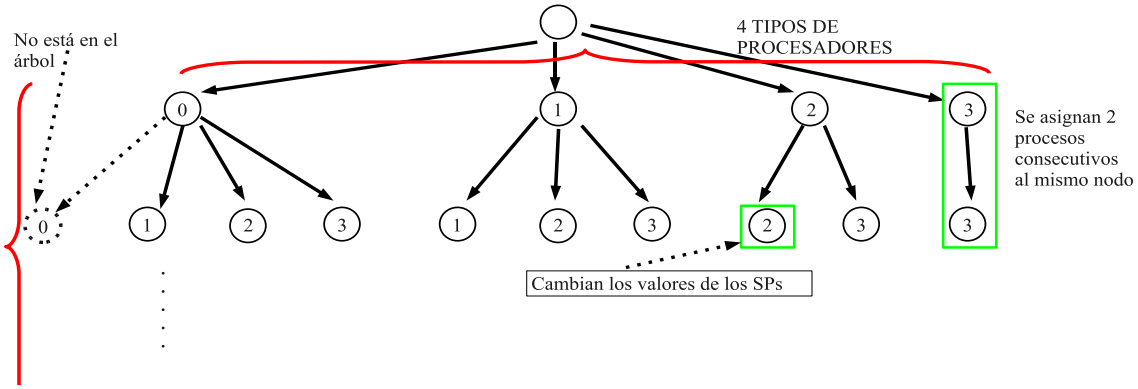

Figura 4.5: Árbol de soluciones para TORC.

Se puede resolver el problema de optimización usando un algoritmo de *backtracking* o *branch and bound*. Dada una posible configuración, por ejemplo 1 Ath y 2 DPIII, los procesadores 17P4 y SPIII no participarían en la solución del problema en los nodos descendientes, y por lo tanto se calcularía el tiempo de computación teórico más bajo con una capacidad de computación total sin considerar la capacidad de estos procesadores. Para obtener el coste de comunicación, se usan los valores más altos de t_s y t_w entre procesadores en la solución parcial. De esta forma, el tiempo

de ejecución estimado en un nodo ($EET(nodo)$) se obtiene de las fórmulas 3.2 y 3.6, con los costes computacionales y de comunicaciones obtenidos según las fórmulas:

$$t_c = max_{i=0,1,...,P-1}\{d_i t_{c_i}\} \tag{4.7}$$

$$t_s = max_{i,j=0,1,...,P-1/d_i\neq 0,d_j\neq 0}\{t_{s_{i,j}}\} \quad t_w = max_{i,j=0,1,...,P-1/d_i\neq 0,d_j\neq 0}\{t_{w_{i,j}}\} \tag{4.8}$$

Para obtener el tiempo de ejecución más bajo en un nodo ($LET(nodo)$) el valor de t_c se obtiene usando el máximo alcanzable cuando se usan todos aquellos procesadores que no están descartados. Por ejemplo, si consideramos el array de tipos y procesadores (0,0,0,1,1,1,2,2,2,3,3,3) (lo que significa cuatro tipos de procesadores y tres procesadores de cada tipo) y una asignación parcial (0,2,1,0), en la que se ha descartado a todos los procesadores de tipo 0 y un procesador de tipo 1 pero aún no se ha explorado todo el árbol para los procesadores de tipo 2 y 3, con lo que los procesadores de estos tipos podrían intervenir en la computación, siendo así el máximo speedup que se podría obtener el de la asignación que no desecha ninguno de los procesadores no descartados: $r = (0, 2, 3, 3)$ (dos procesadores de tipo 1, y tres procesadores de cada uno de los tipos 2 y 3), y el máximo speedup tendría la forma:

$$s_{total} = \sum_{i=0}^{T-1} r_i s_i \tag{4.9}$$

donde s_i es el speedup de los procesadores del tipo i. A partir de este valor del máximo speedup alcanzable, se obtiene $LET(nodo)$, con el tiempo computacional calculado como el tiempo secuencial dividido por s_{total}. El valor de los parámetros de comunicaciones es el mismo que el usado por EET (fórmula 4.8).

$GET(nodo)$ se obtiene con una aproximación *greedy*. Para cada uno de los hijos del nodo *nodo* se calculan los EET, y el nodo con el EET más bajo se incluye en la solución. El proceso acaba cuando el EET más bajo del hijo es mayor o igual que el de su padre. Así, la computación en cada nodo no es constante, y tiene un orden $O(T \cdot l)$ en el peor caso.

Cuando el coste estimado más bajo en un nodo es más grande que el de la solución parcial óptima, no se explora el subárbol que cuelga de él. En la tabla 4.5, se muestra la configuración teórica óptima para diferentes valores de C y *granularidad*. Esta configuración es la que se obtiene realizando la selección en el árbol de asignaciones tal como se ha explicado. En los resultados experimentales, la configuración óptima tiene un 17P4 y un Ath en todos los casos. También se han realizado experimentos en una

configuración con Ath, SPIII y DPIII. En este caso, se muestran las configuraciones seleccionadas en la tabla 4.6.

Tabla 4.5: Configuración teórica óptima de procesos, variando C y *granularidad*. En TORC, con 1 17P4 + 1 Ath + 1 SPIII + 8 DPIII.

C	*granularidad*	configuración
50000	10	1 17P4 + 1 Ath
50000	50	1 17P4 + 1 Ath + 2 DPIII
50000	100	1 17P4 + 1 Ath + 2 DPIII
100000	10	1 17P4 + 1 Ath
100000	50	1 17P4 + 1 Ath + 2 DPIII
100000	100	1 17P4 + 1 Ath + 2 DPIII
500000	10	1 17P4 + 1 Ath
500000	50	1 17P4 + 1 Ath + 1 SPIII + 1 DPIII
500000	100	1 17P4 + 1 Ath + 1 SPIII +1 DPIII

Tabla 4.6: Configuraciones teórica y experimental óptimas de procesos, variando C y *granularidad*. En TORC, con 1 Ath + 1 SPIII + 8 DPIII.

C	*granularidad*	conf. teórica	conf. óptima
50000	10	2 Ath + 1 SPIII	2 Ath + 1 SPIII + 6 DPIII
50000	50	2 Ath + 1 SDIII	2 Ath + 1 SPIII + 8 DPIII
50000	100	2 Ath + 2 DPIII	2 Ath + 1 SPIII + 8 DPIII
100000	10	2 Ath + 1 SPIII	2 Ath + 1 SPIII
100000	50	2 Ath + 1 DPIII	2 Ath + 1 SPIII + 8 DPIII
100000	100	2 Ath + 1 DPIII	2 Ath + 1 SPIII + 8 DPIII
500000	10	2 Ath + 1 SPIII	2 Ath + 1 SPIII
500000	50	2 Ath + 1 SPIII	2 Ath + 1 SPIII + 1 DPIII
500000	100	2 Ath + 1 SPIII	2 Ath + 1 SPIII

El número de procesos seleccionados es pequeño y los procesadores a los que se asignan son los más rápidos. Esto junto con el bajo coste computacional en cada paso implica que la reducción en el tiempo de ejecución no sea significativa. Nuestra meta no es obtener una gran reducción en los tiempos de ejecución, sino obtener sin intervención del usuario un tiempo de ejecución cercano al tiempo de ejecución más bajo alcanzable.

En las tablas 4.7 (la primera configuración de sistema considerada) y 4.8 (sistema sin el 17P4) se muestra la desviación con respecto al tiempo experimental más

bajo del tiempo de ejecución obtenido con la selección de los métodos cp1 y cp2 y con selecciones que corresponderían a usuarios expertos (ue), conservadores (uc) y voraces (uv). También se muestran en ellas el promedio de las desviaciones y las desviaciones de los tiempos totales de ejecución. Podemos ver que el promedio con cp2 es claramente el mejor en la primera configuración (tabla 4.7), pero no ocurre lo mismo en la segunda configuración (tabla 4.8), donde los usuarios experto y voraz obtienen desviaciones medias más bajas. En la última fila de las tablas se muestra la desviación del tiempo total de las diferentes ejecuciones, ya que es más importante que el promedio de la desviación. En este caso cp2 obtiene el valor más bajo en ambas configuraciones aunque en la segunda configuración el valor que el usuario experto modelado obtendría está cercano al obtenido con cp2.

Tabla 4.7: Desviación con respecto al tiempo de ejecución más bajo obtenido experimentalmente, del tiempo de ejecución con los parámetros seleccionados con cp1 y cp2 y con usuarios ue, uv y uc. En TORC, con 1 17P4 + 1 Ath + 1 SPIII + 8 DPIII.

C	*granularidad*	cp1	cp2	ue	uc	uv
50000	10	**0.00**	**0.00**	0.18	0.43	0.73
50000	50	0.40	**0.00**	0.24	0.27	0.21
50000	100	0.39	**0.00**	0.05	0.25	0.08
100000	10	**0.00**	**0.00**	0.14	0.67	0.43
100000	50	0.61	0.01	0.43	0.90	**0.00**
100000	100	0.72	0.07	**0.00**	0.82	0.04
500000	10	**0.00**	**0.00**	0.14	0.66	0.95
500000	50	0.61	**0.00**	0.48	0.76	1.07
500000	100	0.61	**0.00**	0.92	0.76	0.84
PROMEDIO		0.37	**0.01**	0.29	0.61	0.48
TOTAL		0.56	**0.01**	0.59	0.73	0.84

En TORC el número de configuraciones diferentes es demasiado grande para experimentar con todas ellas, y el tiempo experimental mínimo considerado es aquel de un total de 11 ejecuciones con configuraciones que producirían tiempos de ejecución bajos. También es más difícil en este sistema hacer una selección de la distribución de procesos y consecuentemente sería más útil tener un mecanismo para la selección automática de parámetros. Los usuarios modelados harían la siguiente selección:

- El usuario voraz (uv) no es experto en paralelismo, y usa un proceso en cada uno de los nodos disponibles.
- El usuario conservador (uc) utiliza la mitad del número usado por uv, con un

Tabla 4.8: Desviación con respecto al tiempo de ejecución más bajo obtenido experimentalmente, del tiempo de ejecución con los parámetros seleccionados con cp1 y cp2 y con usuarios ue, uv y uc. En TORC, con 1 Ath + 1 SPIII + 8 DPIII.

C	*granularidad*	cp1	cp2	ue	uc	uv
50000	10	0.35	0.38	0.40	0.51	**0.00**
50000	50	0.88	0.48	0.48	0.37	**0.07**
50000	100	0.88	0.49	**0.02**	0.33	0.10
100000	10	**0.07**	**0.07**	0.09	0.34	0.26
100000	50	0.83	0.53	0.53	0.93	**0.04**
100000	100	1.02	0.68	**0.07**	1.13	0.09
500000	10	**0.00**	0.09	0.06	0.32	0.41
500000	50	0.01	**0.00**	**0.00**	0.24	0.32
500000	100	**0.00**	0.12	0.18	0.36	0.32
PROMEDIO		0.45	0.32	0.20	0.50	**0.18**
TOTAL		0.14	**0.12**	0.18	0.36	0.32

proceso en cada uno de los nodos más rápidos.

- El usuario experto (ue) conoce la computación heterogénea y el sistema que se utiliza, así como el problema que se está resolviendo, y usa un proceso para problemas con coste computacional bajo, el mismo número de procesos que uv para problemas con coste computacional alto y la mitad de los procesos para problemas con coste computacional medio pero con dos procesos en cada nodo de DPIII y Ath.

Hay una clara diferencia entre los resultados con las dos configuraciones de los sistemas. Podemos observar la dificultad de modelar un "usuario experto" para un sistema heterogéneo, pues no siempre se obtienen los mejores resultados con ue, sino que en la tabla 4.8 el "usuario voraz" tiene mejor promedio, e incluso mejor que con los dos métodos de estimación de los parámetros. Pero hay que tener en cuenta que la reducción del tiempo de ejecución se compara en la última fila en las tablas 4.7 y 4.8. La figura 4.6 muestra el cociente de los tiempos totales de ejecución obtenidos por los tres usuarios modelados y con los dos métodos de estimación de los parámetros con respecto al menor tiempo de ejecución experimental. Se observa que en términos de tiempos absolutos sí hay una reducción considerable en el tiempo de ejecución usando los dos métodos de estimación, y especialmente cp2, que usa valores no constantes de los parámetros de comunicación.

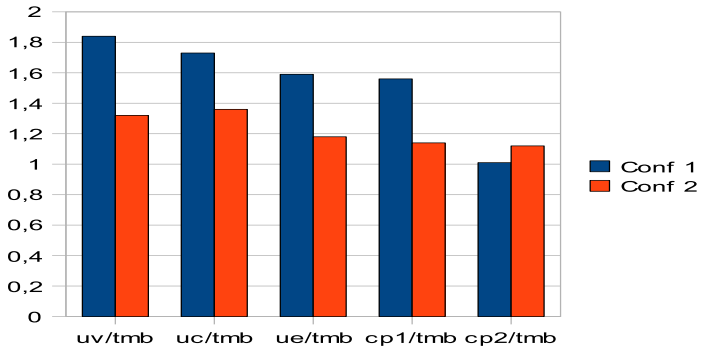

Figura 4.6: Cocientes de los tiempos de ejecución obtenidos por los distintos usuarios modelados y con los diferentes métodos de estimación de los parámetros con respecto al menor tiempo de ejecución, para las dos configuraciones de TORC consideradas.

4.5. Resultados de algunas simulaciones

En las secciones previas, se ha mostrado con dos sistemas de dimensiones reducidas la viabilidad del método propuesto. En esta sección se realiza un estudio similar con simulaciones de sistemas de mayor dimensión, utilizando la técnica de *backtracking* y las estimaciones de EET, LET y GET consideradas en las secciones previas. Adicionalmente, se han hecho algunos experimentos utilizando estimaciones *greedy* utilizando árboles de asignaciones combinatorios y permutacionales, y con LET obtenido con una técnica *greedy*. La técnica *greedy* utilizada en cada caso es la explicada en la sección anterior, en la que se van añadiendo procesos mientras se reduce el tiempo teórico. Con estas simulaciones se pretende justificar la conveniencia y la posibilidad de utilizar la técnica de autooptimización propuesta en sistemas de grandes dimensiones, como pueden ser entornos distribuidos, en los que el número de parámetros aumentaría considerable y es inviable utilizar un método exhaustivo para resolver el problema de optimización.

El comportamiento del método propuesto se analiza en los dos sistemas considerados y en seis sistemas simulados. Se simulan sistemas con 5, 10, 20, 30, 50 y 100 tipos de procesadores (SIMUL5, SIMUL10, SIMUL20, SIMUL30, SIMUL50 y SIMUL100). En la tabla 4.9 se muestran el número de tipos de procesadores, nodos y procesadores para cada sistema simulado.

En la tabla 4.10 y la figura 4.7 se muestran los resultados de las simulaciones. Los resultados han sido obtenidos para $granularidad = 10$ y $C = 500000$, y usando la estimación cp1. Los métodos analizados son:

- $L = Ba$, $G = EE$ es el método *backtracking* cuando se estima LET como se ha

Tabla 4.9: Número de tipos de procesadores, nodos y procesadores en cada sistema simulado.

sistema	tipos de procesadores	número de nodos	número de procesadores
SUNEt	2	6	6
TORC	4	11	19
TORC1	3	10	18
SIMUL5	5	15	21
SIMUL10	10	20	40
SIMUL20	20	70	134
SIMUL30	30	60	132
SIMUL50	50	250	1250
SIMUL100	100	291	613

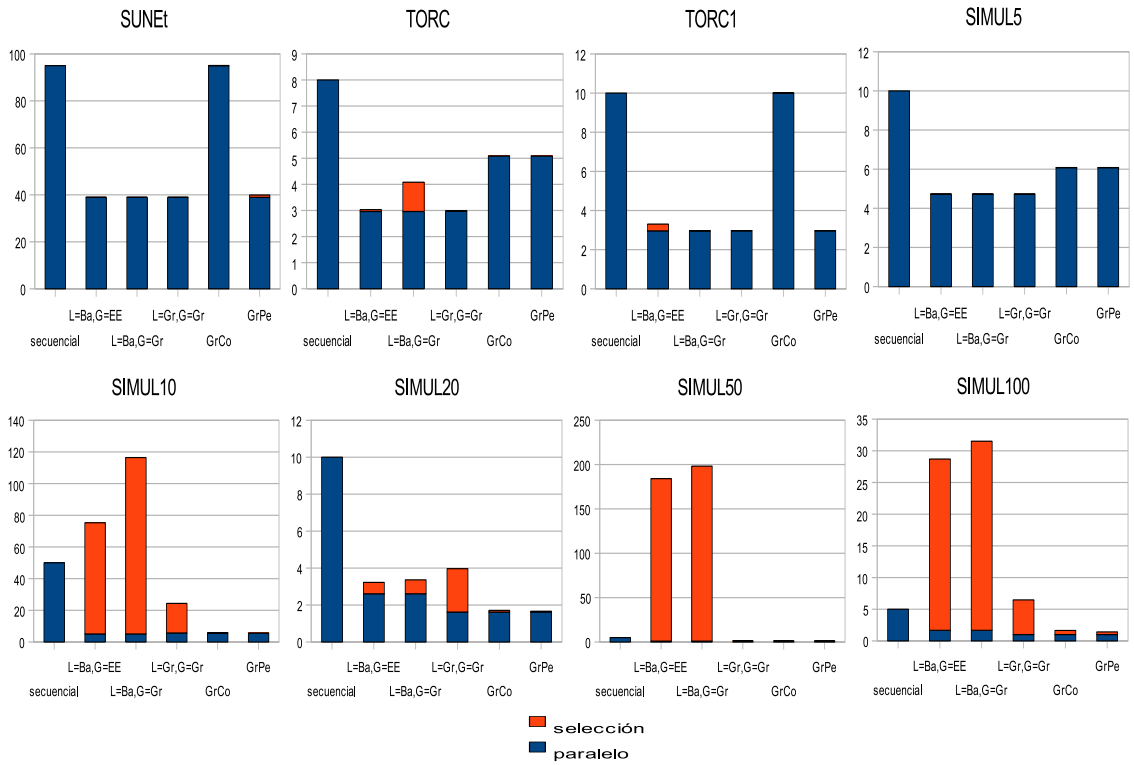

Figura 4.7: Tiempos de selección y paralelos (en segundos) de los resultados de simulaciones en distintos sistemas.

Tabla 4.10: Resultados de la simulación en diferentes sistemas. Tiempos de ejecución en segundos.

sist/secuencial		L=Ba,G=EE	L=Ba,G=Gr	L=Gr,G=Gr	GrCo	GrPe
SUNEt	ti. sel.	0.006	0.001	0.001	0.072	0.001
95.00	nivel	69	9	6	2	6
	ti. par.	38.93	38.93	38.93	95.00	38.93
TORC	ti. sel.	0.071	0.122	0.017	0.001	0.001
8.00	nivel	100	100	27	2	2
	ti. par.	2.96	2.96	2.97	5.09	5.09
TORC1	ti. sel.	0.347	0.013	0.003	0.001	0.001
10.00	nivel	100	100	17	2	16
	ti. par.	2.96	2.96	2.97	10.00	2.97
SIMUL5	ti. sel.	0.006	0.005	0.003	0.001	0.001
10.00	nivel	11	8	6	3	3
	ti. par.	4.72	4.72	4.72	6.08	6.08
SIMUL10	ti. sel.	70.206	111.369	18.654	0.009	0.004
50.00	nivel	16	16	11	11	11
	ti. par.	5.07	5.07	5.72	5.72	5.72
SIMUL20	ti. sel.	0.616	0.754	2.346	0.092	0.046
10.00	nivel	5	5	12	12	12
	ti. par.	2.61	2.61	1.62	1.62	1.62
SIMUL30	ti. sel.			1339.434	0.377	0.378
50.00	nivel			8	13	13
	ti. par.			10.14	6.28	6.28
SIMUL50	ti. sel.	183.105	197.215	5.471	0.649	0.417
5.00	nivel	5	5	5	5	5
	ti. par.	1.00	1.00	1.00	1.00	1.00
SIMUL100	ti. sel.	27.029	29.849	5.471	0.649	0.417
5.00	nivel	3	3	5	5	5
	ti. par.	1.67	1.67	1.00	1.00	1.00

explicado previamente y los valores de GET y EET coinciden.

- En $L = Ba$, $G = Gr$ el valor de GET se obtiene con el método *greedy* explicado antes. De esta forma se analizan menos nodos pero en ellos el coste es muy grande, lo que hace que en algunos casos el coste de resolver el problema de optimización con este método sea mayor que con el anterior (TORC, SIMUL10, SIMUL50 y SIMUL100 en la figura 4.7).

- $L = Gr$, $G = Gr$ usa una aproximación *greedy* para calcular LET. Para cada nodo, se incluye aquel hijo que aumenta menos el coste de operaciones aritméticas en la solución, y el coste de las operaciones de comunicaciones se mantiene constante. El proceso continúa hasta que el tiempo de ejecución en un nodo es más grande que el de su padre. Los valores de LET y GET son más cercanos ahora que en los casos previos y esto produce una reducción del número de nodos examinados.

- $GrCo$ es un método *greedy* en el árbol considerado, que es un tipo de árbol combinatorio con repeticiones (figura 4.8.b). Esta aproximación obtiene rápidamente una asignación, pero la selección puede estar lejana de la óptima si en algún paso del método *greedy* se toma una decisión no adecuada. Esto ocurre por ejemplo en SUNEt y TORC1, donde el tiempo obtenido con $GrCo$ se dispara. Considerar un árbol combinatorio implica que si se descarta un tipo de procesadores no se usarán estos procesadores en sucesivos pasos.

- En $GrPr$ se evita el inconveniente del método anterior aplicando la técnica *greedy* en un árbol permutacional con repeticiones (figura 4.9.b). De esta forma, cuando se ha descartado un tipo de procesador en un paso, este tipo vuelve a ser considerado en pasos sucesivos.

Recordamos que un árbol combinatorio es aquel en el que, de cara a la solución final, no importa el orden en el que son elegidos sus nodos al recorrerlo, mientras que en un árbol permutacional sí que importa. En nuestro contexto los niveles de un árbol de asignación representan los diferentes procesos que van a intervenir en la computación mientras que los nodos del árbol representan el procesador al cual se va a adjudicar el correspondiente proceso. Para entender esto, usamos como ejemplo la asignación a tres procesadores de hasta 3 procesos. En las figuras 4.8 (árboles combinatorios sin a) y con repetición b)) y 4.9 (árboles permutacionales sin a) y con repetición b)) se muestran cómo serían los correspondientes árboles teniendo en cuenta lo anteriormente explicado.

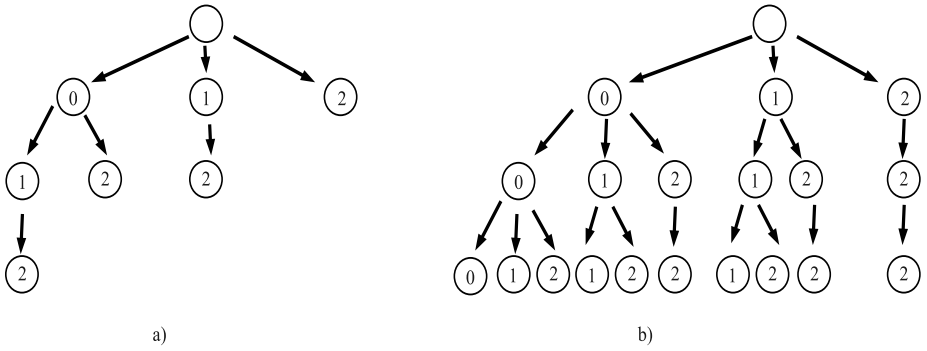

Figura 4.8: Árboles combinatorio sin (a) y con repetición (b).

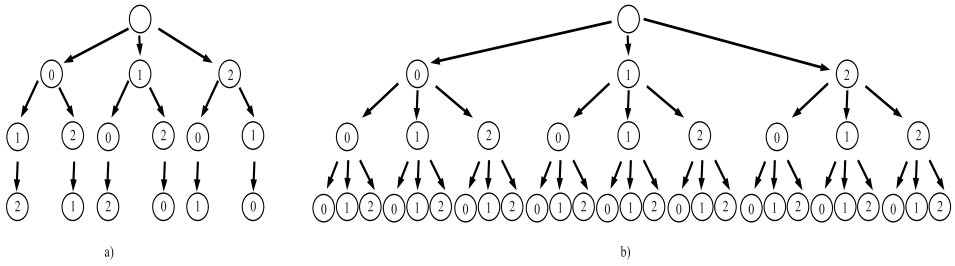

Figura 4.9: Árboles permutacional sin (a) y con repetición (b).

En la tabla 4.10, en la columna uno se muestra el tiempo de ejecución secuencial simulado junto con el nombre del sistema, y para cada sistema se muestran en tres filas los tiempos de selección, el nivel hasta el que se ha llegado en el árbol de asignaciones y el tiempo paralelo asociado al nodo donde se ha encontrado la solución del problema de optimización. Las columnas de la tres a la siete muestran esos valores para los cinco métodos utilizados en la aproximación del problema de optimización.

En la figura 4.7 se muestran los tiempos paralelos simulados (en azul) y los tiempos de selección de los parámetros (en naranja) para distintos sistemas reales y simulados (no se incluye SIMUL30 porque no se han realizado experimentos con todos los métodos y porque el tiempo de selección es excesivamente alto con L=Gr,G=Gr). Se incluye también el tiempo secuencial como primera columna en los gráficos. El tiempo total de ejecución en paralelo utilizando el correspondiente método de selección será la acumulación del tiempo de selección y el simulado.

Comentamos algunas conclusiones que se pueden obtener de los resultados que se muestran:

- El tiempo de selección es reducido en todos los casos para los sistemas de menor tamaño (SUNEt, las dos configuraciones de TORC y SIMUL5) pero no para el resto de sistemas. Esto se refleja claramente en la primera fila de gráficas de la figura 4.7, donde en los gráficos de la primera fila se incluyen los tiempos para los sistemas de menor dimensión.

  En la mayoría de los sistemas, el mapeo óptimo se obtiene con un nivel de árbol bajo (fila 2 para cada sistema en la tabla 4.10), lo que lleva a unos tiempos de ejecución bajos en el proceso de selección cuando el número de tipos de procesadores no es muy grande, pero para sistemas con una mayor cantidad de tipos de procesadores (simulaciones de SIMUL10 hasta SIMUL100) es necesario un proceso de selección más eficiente.

- Así, cuando el tamaño del sistema crece, una posibilidad es incluir alguna heurística para la selección de los parámetros. Esto se hace al aplicar técnicas *greedy*, pero en la mayoría de los sistemas esto lleva a que el tiempo de ejecución paralelo estimado quede lejos del óptimo, ya que se restringen las zonas que se exploran dentro del árbol de decisiones. Esto no se aprecia claramente en los sistemas de mayor dimensión, pero en algunos de ellos no es posible la aplicación de una técnica exacta en la mayoría de los casos (esto depende de valores particulares de los parámetros que afectan al tiempo de ejecución), por lo que en algunas de las simulaciones (SIMUL20, SIMUL30 y SIMUL100) los métodos de *backtracking* se han aplicado sólo hasta un cierto nivel, y es por

eso por lo que las aproximaciones *greedy* proporcionan los tiempos de ejecución paralelos más bajos.

- La utilización de un método *greedy* para intentar reducir el número de nodos que se exploran no proporciona mejoras en los experimentos realizados. Si comparamos los tiempos de selección con $L = Ba$, $G = EE$ y $L = Ba$, $G = Gr$ vemos que son muy parecidos, siendo normalmente mayores los de $L = Ba$, $G = Gr$ cuando se explora hasta el mismo nivel, ya que con la técnica *greedy* se realiza más trabajo en cada nodo. Algunas veces (SUNEt y SIMUL5) se logra reducir el nivel hasta el que se explora el árbol. En cualquier caso, el tiempo paralelo modelado que se obtiene es siempre el mismo.

- Mejores resultados se obtienen cuando se aplica el método *greedy* para acotar inferiormente el tiempo de ejecución a partir de un nodo ($L = Gr$, $G = Gr$). Se obtiene una reducción en el número de niveles que se exploran y consecuentemente también en el tiempo de selección. Aun así, para sistemas grandes el tiempo de selección sigue siendo excesivo, por ejemplo en SIMUL30 más de 20 minutos, para un tiempo secuencial de 50 segundos.

- Cuando se utiliza un árbol permutacional el número de nodos es mayor que en uno combinatorio, pero esto puede no afectar al tiempo de selección si se utiliza una técnica con la que decidir recorrer sólo algunas partes del árbol. Además, como comentamos anteriormente, el método *greedy* usado, en un árbol combinatorio descartaría posibles combinaciones que no se desechan con la representación permutacional. Esto hace que el método $GrPe$ en algunos casos explore más niveles y obtenga mejores tiempos que el $GrCo$.

En algunos casos el tiempo de ejecución de métodos exhaustivos es demasiado grande para ser considerado para la autooptimización, incluso aunque se utilicen técnicas heurísticas. A pesar de esto, la selección automática de procesadores (por un método exacto cuando es posible o por un método de aproximación en otros casos) es preferible a la selección por parte de los usuarios. En la tabla 4.11 se compara el tiempo de ejecución pronosticado con $L = Gr$, $G = Gr$ y $GrPe$ con el esperado por una selección hecha por los tres usuarios voraz, conservador y experto (modelados tal y como se ha explicado en las secciones previas). De hecho, el usuario experto selecciona los procesadores con una aproximación *greedy* pero con un número fijo de decisiones y sin considerar el coste de comunicación, y debido a la complejidad de los sistemas es difícil tener un usuario realmente experto que pueda tomar decisiones satisfactorias, lo que hace que muchas veces las decisiones que toma sean peores que

las tomadas por otros tipos de usuarios. Si comparamos los resultados obtenidos por los usuarios modelados, observamos que, aunque de los tres usuarios el experto y el conservador toman cada uno la mejor decisión en tres de las simulaciones y el usuario voraz toma en cinco de las simulaciones la peor decisión de entre los tres usuarios, la diferencia en el tiempo total entre ue y uv es mínima, y el tiempo total de uc es menor porque ue y uv toman en SIMUL5 y SIMUL50 decisiones muy alejadas de la óptima. En la figura 4.10 se muestran los cocientes de los tiempos de ejecución de los distintos métodos y usuarios con respecto a los obtenidos con $GrPe$. Un valor menor que uno indica que el método o usuario correspondiente toma una mejor decisión que $GrPe$. Vemos que en la mayoría de los casos las decisiones tomadas con $GrPe$ (y también con $L = Gr$, $G = Gr$) son mejores que las tomadas por los distintos usuarios, y algunas veces la diferencia es bastante grande a favor de los métodos de selección automática.

Tabla 4.11: Comparación de los tiempos paralelos estimados con métodos de selección automática usando algoritmos *backtracking* y *greedy* y con las decisiones tomadas por usuarios voraz, conservador y experto.

sistema	L=Gr,G=Gr	GrPe	ue	uc	uv
SIMUL5	**4.72**	6.08	18.26	10.44	18.26
SIMUL10	**5.72**	**5.72**	6.16	9.03	6.29
SIMUL20	**1.62**	**1.62**	3.97	2.51	4.00
SIMUL30	10.14	6.28	**5.44**	8.86	7.30
SIMUL50	**1**	**1**	13.14	6.50	13.14
SIMUL100	**1**	**1**	8.51	3.91	9.06
TOTAL	24.2	**21.7**	55.48	41.25	58.05

Se han realizado experimentos con otras posibles combinaciones de métodos exhaustivos y heurísticas, pero los resultados no difieren de los que aquí se muestran. Mejores estimaciones de LET y GET en cada nodo, utilización de heurísticas más adaptadas al problema con que se trabaja y el uso de otras técnicas exhaustivas, podrían dar lugar a tiempos de ejecución más bajos, y harían el método aplicable a un rango más amplio de sistemas. También es posible incluir en la metodología de selección automática la predicción obtenida por estos usuarios modelados (u otros que se podrían modelar), y combinar distintos tipos de selección imponiendo un límite en el tiempo dedicado a la selección por cada uno se ellos.

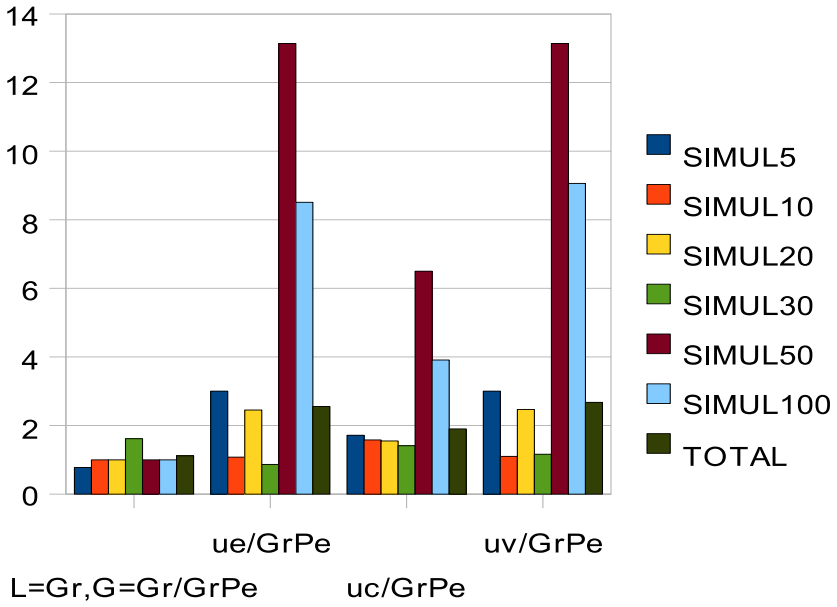

Figura 4.10: Cocientes de los tiempos obtenidos con distintos métodos y usuarios con respecto al obtenido con el método $GrPe$.

4.6. Conclusiones

En este capítulo se ha extendido a sistemas heterogéneos la metodología de autooptimización de esquemas paralelos iterativos. Se ha utilizado el esquema del "problema de las monedas" para estudiar esta metodología propuesta y se ha probado el método en dos redes de procesadores y con varias simulaciones. La metodología propuesta se puede aplicar a otros esquemas de programación dinámica, iterativos e incluso a otros campos (en [32] se aplica esta técnica con algunas modificaciones a factorizaciones matriciales). Los experimentos realizados confirman que se pueden obtener tiempos de ejecución reducidos sin intervención del usuario haciendo uso de técnicas de autooptimización, pero hace falta desarrollar métodos de selección más sofisticados para mejorar la selección en el caso de sistemas heterogéneos muy complejos. Se podría mejorar el problema de optimización desarrollando mejores modelos teóricos y haciendo más experimentos, pero se considera que una mejor alternativa puede ser la utilización sistemática de heurísticas por medio de métodos metaheurísticos, lo que se estudiará en el siguiente capítulo.

Capítulo 5

Metaheurísticas en el proceso de autooptimización

En capítulos previos ha quedado constatado que la optimización del tiempo de ejecución de un algoritmo paralelo puede obtenerse mediante el uso de funciones que modelan analíticamente el coste de dicha ejecución. Generalmente la función de coste incluye un conjunto de parámetros que modelan el comportamiento del sistema y del algoritmo. Con el fin de obtener el tiempo de ejecución óptimo, algunos de estos parámetros deben ser ajustados de acuerdo al problema de entrada y a la arquitectura destino. Si este ajuste se realiza de forma automática se obtienen tiempos de ejecución reducidos sin intervención del usuario e independientemente del sistema computacional. Se formula un problema de optimización en el que el tiempo de ejecución modelado se utiliza para estimar tales parámetros. Debido al número de parámetros en el modelo, la minimización analítica suele ser descartada y se opta por la minimización numérica. Pueden utilizarse técnicas algorítmicas de búsqueda exhaustiva para resolver el problema de optimización pero cuando el número de parámetros o el tamaño del sistema de cómputo crecen, el método es impracticable debido al elevado tiempo de cómputo empleado en su aplicación, que supondría una sobrecarga adicional al tiempo que se pretende minimizar. El uso de métodos aproximados para guiar la búsqueda y reducir el tiempo de decicisión es una alternativa. Sin embargo, la dependencia sobre el algoritmo modelado (pérdida de generalidad) y la mala calidad de las soluciones debido a la presencia de muchos valores de óptimos locales en las funciones objetivo, son también inconvenientes para el uso de estas técnicas. El problema se convierte en particularmente difícil en sistemas complejos compuestos de un gran número de procesadores heterogéneos que resuelven aplicaciones científicas no triviales, tal y como se ha visto en el capítulo previo. El uso de metaheurísticas permite el desarrollo de aproximaciones válidas para resolver problemas generales con

un gran número de parámetros. Una ventaja conocida de los métodos metaheurísticos es la capacidad para proporcionar soluciones de alta calidad en tiempos de ejecución razonables, manteniendo al mismo tiempo su generalidad. Por ello, el uso de metaheurísticas es una alternativa apropiada para resolver el problema de minimización en un tiempo reducido (obtención de los valores en la fase de **Ejecución** de la figura 1.2), posibilitando así la autooptimización en sistemas complejos. Proponemos en este capítulo combinar el modelado analítico de la función de coste con el uso de métodos metaheurísticos en la minimización, lo que contribuye al desarrollo de una alternativa real para minimizar el tiempo de ejecución paralela en sistemas complejos.

5.1. Metaheurísticas y problemas de optimización

Un problema de optimización combinatoria puede representarse como:

$$\begin{gathered} \text{mín}\, f(x) \text{ o } \text{máx}\, f(x) \\ x \in X \\ f : X \rightarrow Y \end{gathered} \tag{5.1}$$

en el que se trata de optimizar (minimizar o maximizar en función del contexto) el valor de la función $f()$, teniendo en cuenta que x puede tener una estructura vectorial o matricial y toma valores discretos, enteros o incluso binarios en el conjunto X (habitualmente muy numeroso) de soluciones alternativas. $f()$ representa a la función objetivo a optimizar, x a las variables de decisión, X al espacio de soluciones que contiene al conjunto de soluciones factibles e Y es el conjunto donde toma valores la función a maximizar (o minimizar), y suele ser el conjunto de los naturales o los reales. X, denotado también como *región factible* o *espacio de búsqueda*, puede venir representado en la forma de restricciones que deben cumplir las variables de decisión, y su naturaleza dependerá de cada problema. Se trata de encontrar la mejor solución (solución óptima), x^*, entre un conjunto de soluciones alternativas factibles. Este esquema de optimización aparece en diversos problemas del ámbito científico y de la ingeniería, y suele ser habitual que el problema se plantee de una forma relativamente sencilla pero que su resolución sea ciertamente complicada. A lo largo de los años se ha demostrado que muchos problemas de optimización combinatoria pertenecen a la clase de Problemas NP-completos, es decir, la solución óptima se obtiene utilizando algoritmos que emplean un tiempo de computación que crece de forma superpolinomial (normalmente exponencial) en el tamaño del problema y, por consiguiente, en muchos casos la solución óptima no se puede obtener en un tiempo razonable y ha de

optarse por métodos aproximados. Las metaheurísticas, como métodos aproximados, han venido desarrollándose desde los años 80 y han dado muestras de su eficacia en una amplia variedad de problemas complejos [47, 60].

El término *metaheurística* fue originalmente introducido por Fred Glover [63], y hace referencia a una amplia clase de conceptos algorítmicos orientados a la optimización y a la resolución de problemas. Como definición, consideramos la presentada en [117] como la unión de los conceptos formulados en [63] y en [129]:

"*Una metaheurística es un proceso iterativo maestro que guía y modifica las operaciones de heurísticas subordinadas para producir de manera eficiente soluciones de alta calidad. Puede manipular una única solución completa o incompleta, o una colección de soluciones en cada iteración. La heurística subordinada podría consistir en procedimientos de alto (o bajo) nivel, en búsquedas locales simples, o simplemente métodos constructivos. Estos métodos con el tiempo han llegado a incluir cualquier procedimiento para resolver problemas que empleen estrategias para superar las trampas de optimalidad local en espacios de solución compleja, especialmente aquellos procedimientos que utilizan una o más estructuras de vecindad como medios para definir movimientos admisibles para transitar de una solución a otra, o para construir o destruir soluciones en procesos constructivos y destructivos.* "

El propio Glover junto a Laguna [61] plantea una definición en los siguientes términos:

"*Metaheurística se refiere a una estrategia maestra que guía y modifica a otras heurísticas para producir soluciones más allá de aquellas que normalmente se generan en una búsqueda de óptimos locales.*"

Ambas definiciones recogen las siguientes propiedades para estos métodos:

- Guían el procedimiento de búsqueda.
- Poseen mecanismos para evitar el estancamiento en óptimos locales.
- Son procedimientos generales, no específicos.
- Pueden incluir uno o más procedimientos heurísticos.
- Suelen hacer uso de la experiencia mediante retroalimentación del procedimiento.
- Normalmente poseen una componente aleatoria.
- Su objetivo final es encontrar soluciones óptimas o cuasióptimas en tiempos computacionales razonables.

Como se sabe, las metaheurísticas al igual que los algoritmos genéricos aproximados permiten manejar grandes sistemas en tiempos de ejecución reducidos [60]. Históricamente han venido siendo aplicadas en diferentes campos de la computación para solucionar problemas complejos que requerían elevados tiempos de decisión, como por ejemplo el problema del ancho de banda, rutas de vehículos bajo demanda, problemas de flujo en redes, búsqueda de cadenas de ADN, gestión de tráfico aéreo, biología molecular, geofísica, robótica, reconocimiento de patrones, exploración de datos [22, 47, 73, 77, 112]... Todos ellos son problemas de optimización combinatoria en los que se trata de optimizar una función objetivo sobre un espacio de búsqueda. Con frecuencia se aplican a problemas que no tienen algoritmos (exactos o aproximados) que proporcionen soluciones satisfactorias en tiempo de ejecución y en calidad de la solución obtenida.

Situándonos en el ámbito de esta tesis, hacemos notar que, también se han aplicado con éxito métodos metaheurísticos en la resolución de determinados problemas de mapeo sobre sistemas específicos [47, 60, 77], lo que abre el camino al proceso de generalización que aquí proponemos. En nuestro caso se trata de estudiar su aplicación en combinación con la técnica de optimización de esquemas paralelos iterativos basada en el modelado parametrizado del tiempo de ejecución para lograr que la asignación de procesos a procesadores se haga, en la medida de lo posible, sin interveción humana. Por ello, en la ecuación 5.1 la función a optimizar f sería la ecuación del tiempo de ejecución según el modelo paramétrico, siendo X el producto cartesiano de los conjuntos en los que pueden tomar valores cada uno de los parámetros del algoritmo e Y es el conjunto de los números reales, pues es donde el tiempo de ejecución toma valores. Por ejemplo, en un esquema paralelo iterativo en un sistema heterogéneo la función f tomará valores en todas las posibles asignaciones de datos a los procesadores (ecuación 2.10), y se trata de obtener el menor tiempo de entre los de todas las posibles asignaciones (ecuación 2.11).

5.2. Clasificación de Metaheurísticas

Las metaheurísticas se pueden clasificar atendiendo a diferentes criterios. Cada clasificación proporciona un punto de vista diferente del conjunto de metaheurísticas propuestas en la literatura. Algunas metaheurísticas podrían encajar en más de una de las clases obtenidas y posiblemente otros criterios proporcionarían agrupaciones distintas. Atendiendo al tipo de procedimiento utilizado por la metaheurística nos limitaremos a mencionar algunas de las técnicas más conocidas y utilizadas en los últimos años [1, 61, 75]:

- Metaheurísticas basadas en trayectorias: hacen referencia a un conjunto de métodos que se caracterizan por un sistema de búsqueda de soluciones que "dibuja" una trayectoria en la región factible. La información proporcionada por la trayectoria puede utilizarse para decidir, a priori, sobre el comportamiento del algoritmo y su efectividad final. Dicha efectividad se fundamenta en una buena combinación del algoritmo elegido, la codificación del problema y el caso particular sobre el que estemos trabajando. Como ejemplos tenemos: búsqueda local, arranque múltiple [98], temple simulado [46, 60], búsqueda tabú [60, 64, 65], búsqueda por entornos variables (VNS) [103].

- Metaheurísticas constructivas: construyen una solución añadiendo, iterativamente, elementos a una estructura inicialmente vacía. Estos métodos están formados por un conjunto de estrategias o técnicas utilizadas en la elección de los elementos que conformarán la solución. Dentro de ellas destaca, por su aplicación e influencia en otras técnicas, la técnica GRASP [118]. Otros ejemplos son: las técnicas *greedy* y los sistemas de hormigas [29].

- Metaheurísticas basadas en poblaciones: procesan en cada iteración un conjunto de soluciones normalmente denominado *población*, de forma que se desarrolla un procedimiento inspirado en la naturaleza. El éxito final dependerá del método utilizado para manipular las soluciones de la población. Si la manipulación de esta población sigue principios o analogías con la naturaleza, se dice que el procedimiento es Bioinspirado. Entre los métodos de poblaciones más utilizados, los más estudiados en la literatura y a los que se les ha encontrado un mayor campo de aplicación están los sistemas de hormigas y los algoritmos evolutivos (EA), entre los que podemos destacar los algoritmos genéticos [47, 60, 77], los meméticos [107] y la búsqueda dispersa (*scatter search*) [47, 78, 96, 97].

- Metaheurísticas híbridas: es un novedoso tipo de procedimiento metaheurístico, surgido en los últimos tiempos, que se caracteriza por la combinación de varias metaheurísticas que, trabajando en conjunto, realizan la búsqueda de soluciones más eficientemente que trabajando por separado. Pueden encontrarse tres formas originales de hibridación:

    - Inclusión de componentes de una metaheurística como componentes subordinados de otra metaheurística. En este tipo de hibridación podemos encontrar la aplicación de una búsqueda local multiarranque como componente integrado de una metaheurística más avanzada, como los algoritmos genéticos [113].

- Concatenación de varios algoritmos trabajando simultáneamente que intercambian información sobre la región de búsqueda. Este tipo de metaheurísticas híbridas se suele denominar búsqueda cooperativa [126].
- Integración de métodos exactos y aproximados [75], que se ejecutan simultáneamente o de forma secuencial y que combinan la exactitud de los primeros en la intensificación de la búsqueda con la aleatorización de los segundos en la diversificación.

De entre todas estas técnicas hay que destacar por su reiterado uso las técnicas de algoritmos genéticos, búsqueda dispersa, búsqueda tabú, temple simulado, GRASP y su hibridación. Cada una de ellas tiene un recorrido histórico individual y siguen diferentes paradigmas y filosofías. El éxito de un método particular depende del método en sí mismo, de su adecuación al problema (*tuning*) y de la combinación de la heurística y la aleatoriedad.

5.3. El esquema general

Todas las metaheurísticas existentes comparten ideas y difieren en algunos componentes. Las visiones unificadas, presentadas por ejemplo en [117, 127], proporcionan una interesante perspectiva del campo, con un algoritmo general para las distintas metaheurísticas. Un algoritmo general como el que se presenta en el código 5.1, en el que se muestra un procedimiento iterativo de manipulación de soluciones, puede instanciarse para obtener diferentes metaheurísticas simplemente cambiando algunos elementos clave (las funciones Inicializar, CondiciondeFin, ObtenerSubconjunto, Combinar, Mejorar e IncluirSoluciones) y los tamaños de los conjuntos de soluciones involucrados (S, SS, $SS1$ y $SS2$).

A continuación se describen los elementos y operadores del algoritmo anterior:

- S es el conjunto inicial formado por las soluciones factibles de partida, que son proporcionadas por el procedimiento Inicializar. Estos elementos representan soluciones temporales al problema a resolver, y a partir de ellos se van generando en sucesivas iteraciones mejores aproximaciones a la solución del problema. Pasada la etapa de generación de soluciones iniciales, S pasa a contener el conjunto actual de soluciones, que es sometido a diversas transformaciones. El conjunto S se conoce con distintos nombres en función de la metaheurística en cuestión, así, por ejemplo, el conjunto S en la búsqueda dispersa se conoce como conjunto de referencia y en los algoritmos genéticos como población. En

Algoritmo 5.1: Esquema general de un método metaheurístico.

```
Inicializar(S);
while no se cumple CondiciondeFin(S) do
    SS = ObtenerSubconjunto(S);
    if |SS| > 1 then
        SS1 = Combinar(SS);
    else
        SS1 = SS;
    end
    SS2 = Mejorar(SS1);
    S = IncluirSoluciones(SS2);
end
```

los métodos basados en poblaciones, como los genéticos, a las soluciones del conjunto S se les llama individuos.

- El operador Inicializar se utiliza para crear los elementos del conjunto inicial S, y devuelve un conjunto de soluciones de partida. Estas soluciones se pueden obtener aleatoriamente o bien a través de algún proceso que utiliza criterios relacionados con la naturaleza del problema a resolver. Algunas particularidades de este operador en las distintas metaheurísticas son:

  - En los algoritmos genéticos se suele utilizar una población con muchos individuos, lo que hace que en ocasiones el método se ralentice para problemas con instancias de tamaño grande.
  - La búsqueda dispersa trabaja con un número reducido de individuos en S (típicamente alrededor de 20). Esto puede producir un menor tiempo de decisión que en los algoritmos genéticos, lo que la hace atractiva como técnica en aquellos casos en los que es deseable obtener la solución en tiempos de respuesta rápido.
  - En métodos constructivos como la búsqueda tabú y los métodos GRASP se trabaja con un conjunto S con un único elemento.

- SS es un conjunto de elementos extraídos de S con el fin de combinarlos para generar nuevos elementos que potencialmente mejorarán la solución del problema. El conjunto toma distintas formas en función del método:

  - En los algoritmos genéticos SS está formado por los elementos que serán seleccionados para ser cruzados por pares.

  - En la búsqueda dispersa, tal y como se verá más adelante, estará formado por varios elementos que podrán ser seleccionados conforme a diferentes criterios para ser combinados (dos a dos o por grupos mayores de elementos) con el fin de obtener buenas soluciones del problema a resolver.
  - En los métodos constructivos (búsqueda tabú, GRASP...) $SS = S$ puesto que $|S| = 1$.

- El operador ObtenerSubconjunto selecciona soluciones a partir del conjunto de soluciones actual para ser combinadas o mejoradas. Es habitual seleccionar elementos de entre los mejores del conjunto introduciendo algún criterio de aleatoriedad. Se pueden usar diferentes criterios dependiendo de la técnica metaheurística empleada:

  - En los algoritmos genéticos los individuos con mejor función objetivo tienen más probabilidad de ser seleccionados.
  - En la búsqueda dispersa es posible seleccionar todos los individuos para combinarlos entre ellos, o seleccionar los mejores según la función objetivo para ser combinados con los "más dispersos" respecto a los mejores.
  - En la búsqueda tabú y en GRASP esta función no es necesaria pues $|S| = 1$, y el único elemento que se está utilizando es el que se usará para mejorarlo.

- $SS1$ es el conjunto de elementos resultantes de aplicar el operador Combinar conforme a algún criterio a los elementos de SS. Se busca que a partir de $SS1$ (mediante la combinación) se pueda aportar "variedad" o dispersión en las soluciones temporales que se van obteniendo. De este modo las soluciones de una iteración pueden mejorar las de iteraciones anteriores, o bien alejarse de las soluciones de partida en el espacio de búsqueda para poder progresar hacia otras zonas no exploradas.

- En la función Combinar, se combinan los elementos seleccionados para generar $SS1$. Nuevamente, la operación de combinación se conoce con distintos nombres de acuerdo a los métodos. Con esta combinación se busca que los diferentes elementos se puedan "mezclar" con el fin de obtener soluciones diferentes. En la operación de combinación se introducen elementos que proporcionan aleatoriedad sobre los resultados obtenidos, y admite distintas instanciaciones:

  - En los algoritmos genéticos el operador se conoce como operador de cruce, y en él se cruzan pares de individuos obteniendo nuevos descendientes (generalmente dos nuevos individuos) que heredan parte de las características

de cada antecesor. Algunos de los descendientes generados se añadirán a la población en IncluirSoluciones.

- En la búsqueda dispersa se puede proceder como en los algoritmos genéticos pero hay otras posibilidades combinándose los elementos de diferentes maneras generando uno o más elementos que se añaden al conjunto. Como se verá al final del capítulo, en el apartado de resultados experimentales, se pueden dividir los elementos en dos grupos: un grupo que contenga aquellos elementos con mejores valores de la función objetivo y otro con los más dispersos respecto a los mejores. Se pueden combinar todos los elementos por pares (o combinar más de dos elementos cada vez), o se pueden combinar entre sí elementos en los dos grupos.
- En la búsqueda tabú y los métodos GRASP esta función no es necesaria, ya que sólo hay un elemento con el que trabajar, por lo que los conjuntos $SS1$ y SS coinciden.

- $SS2$ es el conjunto de elementos resultantes de aplicar el operador Mejorar a los elementos de $SS1$. Contendrá elementos obtenidos a partir de los que se acaban de generar en Combinar, pudiéndose modificar todos los elementos generados, sólo unos pocos según una cierta probabilidad, o en algunos casos no mejorar ningún elemento, con lo que sería $SS2 = \emptyset$. En los métodos que trabajan con un único elemento (por ejemplo, búsqueda tabú y GRASP) coincide con $SS1$.

- En general, en la función Mejorar se trata de orientar el recorrido hacia buenas soluciones intensificando la búsqueda en una región del espacio que puede estar próxima o no a las soluciones encontradas hasta el momento. En las operaciones de mejora, tal y como ocurre con las operaciones de combinación, se integran elementos que proporcionan cierta aleatoriedad sobre los resultados obtenidos. Se puede intentar mejorar los elementos generados por combinación, o mejorar el elemento actual, o también aplicar cambios a algunos de los elementos para intentar diversificar la búsqueda. Esta operación será diferente dependiendo de la técnica empleada:

  - En los algoritmos genéticos son operaciones simples conocidas como mutaciones en las que se introduce un fuerte componente de aleatoriedad para generar nuevas soluciones. Se seleccionan unos pocos individuos a los que se aplica el operador de mutación, en este caso con el fin de diversificar la población para evitar caer en óptimos locales.

- En la búsqueda dispersa se realizan búsquedas sistemáticas en la vecindad de los elementos del conjunto $SS1$ con el fin de realizar mejoras.
- En la búsqueda tabú algunas soluciones de la vecindad de la solución actual se analizan, excluyendo las que forman parte de una lista tabú.
- En GRASP esta función suele consistir en una búsqueda local que mejora el individuo seleccionado. Se suelen usar métodos *greedy* o analizar todos los individuos de la vecindad.

Como se observa, es frecuente que en esta operación los algoritmos apliquen búsquedas locales o incluso que hibridicen con otros métodos.

- Con el operador IncluirSoluciones se crea un nuevo conjunto de soluciones haciendo uso de alguna técnica de selección. Selecciona algunos elementos de $SS2$ para ser incluidos en S para la próxima iteración. Se pretende mejorar el conjunto de soluciones seleccionados o la solución con la que se trabaja. Por ejemplo, se pueden seleccionar los mejores elementos generados para incluirlos en el conjunto S, eliminando de éste las peores soluciones. Cada método aplica el operador de diferente forma:

  - En los algoritmos genéticos forman la nueva población S los mejores elementos de entre todos aquellos disponibles (los del conjunto original, los descendientes y los generados mediante mutación).
  - En la búsqueda dispersa se seleccionan los mejores elementos junto con aquellos que estén alejados de ellos. De este modo se intenta conseguir que la búsqueda no se realice únicamente en una área del espacio de búsqueda, sino que se disperse para evitar caer en óptimos locales.
  - En la búsqueda tabú y en GRASP se toma como la siguiente solución el mejor elemento de aquellos analizados.

- En la CondiciondeFin no suele haber diferencias significativas entre los distintos métodos. Hay varias posibilidades:

  - Pueden realizarse un número fijo de iteraciones, pero se suele utilizar criterios que tengan en cuenta la mejora de las soluciones.
  - Se puede continuar la ejecución mientras se obtengan mejoras a las soluciones obtenidas, ya sea porque mejora el mejor elemento, la media de los valores, u otro criterio.

- Se puede acabar si en una iteración no hay cambios en la población.
- Se pueden combinar varios de los criterios anteriores. Por ejemplo, el criterio de parada puede ser que se alcance un número máximo de iteraciones o un número máximo sin cambios en el mejor elemento, o en la media de los valores o en el conjunto de referencia.
- Cuando se intenta establecer criterios comparativos entre varios métodos, es habitual también fijar un tiempo durante el que los métodos están en ejecución y a su finalización se comprueba cuál de los métodos ha proporcionado la mejor solución.
- En la búsqueda tabú un criterio de parada podría ser la imposibilidad de usar elementos que no formen parte de la lista tabú (soluciones que fueron visitadas en el pasado reciente).

Además de los elementos que aparecen explícitamente en el algoritmo general de una metaheurística, hay dos aspectos internos que deben ser comentados adicionalmente:

- En primer lugar, obsérvese que en el proceso iterativo que representa el algoritmo 5.1 puede verse implicado un elevado número de soluciones que deben ser evaluadas. Por tanto, el proceso de evaluación de soluciones se convierte en un elemento crítico que condiciona el rendimiento final del método. La evaluación de las soluciones dependen fuertemente de cómo están representadas para un problema dado, y es por esto que las soluciones a un problema de optimización deben ser codificadas de manera adecuada para conseguir un tratamiento eficiente en el método heurístico. Por ejemplo, en los problemas de asignación o en problemas del viajante del comercio, es frecuente representar las soluciones mediante permutaciones, esto es, para un problema de tamaño n una solución puede venir dada por la permutación $\{\pi_1, ..., \pi_n\}$ del conjunto $\{1, ..., n\}$, en problemas de la mochila nos podríamos encontrar con soluciones del tipo $I \subset \{1, ..., n\}$, también para problemas de tamaño n. Generalmente el tiempo de evaluación de una solución de estas características es $O(n)$. En este tipo de problemas ésta es la representación natural de una solución que se obtiene al formular el problema de optimización combinatoria. Esta misma representación natural puede ser utilizada como estructura de datos en el código que implementa al método. Sin embargo, en muchas ocasiones la representación natural ofrece soluciones en la forma de estructuras matriciales bidimensionales, como ocurre en problemas de flujo en redes o en problemas de rutas. La evaluación de una

solución representada de este modo implica un coste de evaluación por solución de $O(n^2)$ y el impacto de esta evaluación podría resultar excesivo, produciendo un método altamente ineficiente como consecuencia del elevado número de evaluaciones que se producen. Es por esto que en estas situaciones es deseable analizar posibles mecanismos de representación alternativos a los que surgen de manera natural, con el fin de mejorar la eficiencia de la metaheurística. Ésta es la estrategia seguida en [112] para problemas de flujo en redes en la que se transforma la estructura matricial en una estructura lineal de $O(n)$. En ella la semántica de los índices ha sido variada con el fin de que la solución pueda contener toda la información asociada al problema.

- Un segundo aspecto a tener en cuenta es que las operaciones de mejora o de generación de nuevas soluciones, como ocurre en las operaciones Combinar, Mejorar y algunas otras, se nutren de las soluciones cercanas a soluciones que previamente han sido generadas. Este hecho hace que sea necesario disponer de procedimientos que permitan realizar movimientos de una solución a otra en entornos reducidos del espacio de búsqueda. Es necesario definir una estructura de vecindad $N : X \rightarrow P(X)$, donde $P(X)$ es el conjunto potencia de X, aunque se suelen usar vecindades con un número preestablecido n de elementos. Así, para cada $x \in X$, el conjunto $N(x) \in P(X)$ proporciona el conjunto de soluciones vecinas a x. $N(x)$ es la vecindad de x, y $x' \in N(x)$ es una solución vecina de x. Asociado a la estructura de vecindad se encuentra un mecanismo de generación de movimientos que proporciona el conjunto de vecinos de una solución dada mediante la modificación de algún atributo o a través de la combinación de varios de ellos. La generación de un movimiento en un paso o iteración del algoritmo consiste en la transición de una solución x hacia otra solución $x' \in N(x)$. Tal y como ocurre con la representación de las soluciones, la eficiencia de las metaheurísticas está fuertemente condicionada por la estructura de entorno seleccionada así como por las estrategias de generación de movimientos asociadas a la misma.

En esta sección hemos presentado las ideas básicas de las metaheurísticas y un esquema general para ellas, y en las siguientes secciones nos vamos a centrar en intentar resolver el problema de la asignación de procesos a procesadores mediante metaheurísticas para esquemas iterativos paralelos centrándonos, tal y como se ha hecho en capítulos previos, en instancias particulares de problemas de programación dinámica. En el proceso de decidir qué método metaheurístico seleccionar se ha pretendido encontrar un cierto equilibrio entre generalidad, eficiencia y facilidad de im-

plementación. Es un hecho establecido que los algoritmos genéticos suelen ser bastante genéricos y además relativamente sencillos de implementar. Sin embargo, para problemas complejos y de gran tamaño, su eficiencia tiende a disminuir. Los algoritmos GRASP generalmente son sencillos de implementar y muy rápidos en la ejecución, pero pueden quedar atrapados en óptimos locales, por lo que es frecuente encontrar el método implementado en combinación con otros más sofisticados. La búsqueda tabú incorpora una estructura, la lista tabú, que en función de las características del problema puede ser más o menos sofisticada en su implementación. Su eficacia, además dependerá de las estrategias seguidas para incluir o excluir elementos de la lista. La revisión de trabajos que hemos realizado en esta tesis, nos hace pensar que la búsqueda dispersa cubre bien los objetivos de generalidad, eficiencia y facilidad de implementación que buscamos. Las siguientes secciones estarán orientadas a describir la técnica general de la búsqueda dispersa y su aplicación al problema que nos ocupa. De manera similar se pueden aplicar otras metaheurísticas a este problema [26], y a otros problemas similares de asignación de tareas a procesadores [5, 33, 37, 56, 57].

5.4. La búsqueda dispersa (*Scatter search*)

La búsqueda dispersa, conocida en inglés como *scatter search*, es un método evolutivo propuesto en los años setenta por Fred Glover [62] que se diferencia de otras metaheurísticas en los principios de unificación y recombinación de soluciones que utiliza. Aunque el método fue propuesto hace más de dos décadas, no alcanzó la importancia que tiene hoy en día hasta 1998, con una nueva publicación del anterior autor [66] en la que se recopilan diferentes aplicaciones del método realizadas hasta el momento. A partir de esa aportación, la búsqueda dispersa se ha convertido en una de las metaheurísticas evolutivas de mayor aplicación.

Trabajos más recientes [96, 97] muestran que la búsqueda dispersa es una metaheurística de búsqueda agresiva que consigue encontrar buenas soluciones con un tiempo de búsqueda aceptable en un amplio rango de problemas. Opera sobre un conjunto de soluciones, el conjunto de referencia, que son combinadas entre sí con el fin de obtener otras soluciones mejores. Se parte del conjunto de referencia inicial y se obtienen nuevos elementos como combinación de elementos que se encontraban en el conjunto en la iteración anterior. Al contrario de lo que ocurre en otros algoritmos evolutivos como los algoritmos genéticos, el conjunto de referencia es pequeño, lo que contribuye a que el tiempo de decisión también lo sea.

La idea inicial propuesta para justificar la búsqueda dispersa se fundamenta en los siguientes aspectos:

- En cada paso del método se combinan dos o más soluciones creando regiones de atracción.
- Una versión reducida del método se desarrollaría al seleccionar soluciones en la línea que une dos soluciones dadas.
- Se deben seleccionar previamente los pesos a utilizar en el momento de la selección, en vez de realizar la selección de forma aleatoria.
- Es conveniente realizar combinaciones convexas (donde sólo se consideran los elementos que quedan dentro del espacio formado por los elementos actuales) y no convexas (donde se consideran elementos que quedan fuera del espacio formado por los elementos actuales).

Se introduce la idea de *conjunto de referencia* junto con los procedimientos que actualizan dicho conjunto con "buenas" soluciones. Estos procedimientos serían, por un lado, un método de *búsqueda inteligente* que combina *diversificación* e *intensificación*, consistente en un método de *mejora* con el que optimizar las soluciones del conjunto de referencia, y, por otro, un método para *actualizar* el conjunto de referencia después de realizar las *combinaciones* entre las soluciones incluidas en él. A continuación se describe el significado de cada uno de estos términos:

- **Conjunto de Referencia (RS)**: conjunto formado por las mejores soluciones obtenidas durante el proceso de búsqueda. En esta metaheurística, el concepto de mejor solución se refiere a las soluciones con mejor calidad y a las más distantes del propio conjunto de referencia, asegurando así la diversidad de la búsqueda. Al introducir el concepto de solución más distante, el método considera algún tipo de métrica con el que cuantificar esa distancia. Generalmente se utiliza la distancia euclídea. El Conjunto de Referencia es un conjunto de tamaño $|RS|$, con las soluciones más "representativas"del problema, seleccionadas de un conjunto o población de soluciones iniciales de prueba, P. Normalmente $|P| = 100$ y $|RS| \leq 20$. Estas soluciones evolucionan para proporcionar una exploración inteligente de la región factible, a través de esquemas de intensificación y diversificación que, respectivamente, enfocan la búsqueda hacia regiones prometedoras, deducidas de las soluciones previas, y dirigen la búsqueda hacia nuevas regiones de exploración no visitadas anteriormente.

  En el esquema general dado en el algoritmo 5.1, la función Inicializar generaría el conjunto P, del que selecciona los elementos del conjunto RS, que coincide con el conjunto S en el esquema general.

- **Método de Diversificación**: con este método se consigue un conjunto de soluciones diversas, formado normalmente por las 100 soluciones que constituyen la población inicial P, del que se tomarán aproximadamente 20 que formarán el conjunto de referencia. Estos mecanismos de diversificación u otros, se utilizan también en la actualización del conjunto de referencia. Por ejemplo, para generar RS en cada iteración se puede generar un conjunto con más elementos (con la función Combinar en el esquema 5.1), mejorarlos (con la función Mejorar) y seleccionar (función IncluirSoluciones en el esquema) los que van a constituir el conjunto RS en la siguiente iteración.

- **Método de Mejora**: habitualmente se utiliza una búsqueda local para mejorar las soluciones (función Mejorar en el esquema general), tanto del conjunto de referencia como las resultantes de las combinaciones, antes de confirmar su inclusión en el conjunto de referencia. Este procedimiento intensifica la búsqueda en áreas próximas a las soluciones encontradas.

- **Método de Actualización**: se aplican los mecanismos de diversificación e intensificación para actualizar el conjunto de referencia y guiar la heurística hacia una solución óptima del problema. La diversificación se aplica habitualmente para actualizar un subconjunto del conjunto de referencia con soluciones que no se encuentran en dicho conjunto y que maximizan alguna distancia a él. Por ejemplo, se puede maximizar la distancia al elemento más cercano del conjunto de referencia o la suma de las distancias a los elementos del conjunto. Las soluciones restantes se actualizan mediante un esquema de intensificación. Normalmente, la intensificación se ejecuta mediante la combinación (función Combinar en el esquema) de diferentes soluciones para crear nuevas soluciones iniciales en los procedimientos de búsqueda local, que aparecen como función Mejorar en el esquema general. El conjunto de referencia, formado por $|RS|$ soluciones, se actualiza en cada iteración con el subconjunto en el que se incluyen las $|RS_1|$ soluciones de mejor calidad hasta ese momento y con otro subconjunto que contiene las $|RS_2|$ soluciones más distantes del conjunto de referencia. De esta forma $RS = RS_1 \cup RS_2$.

- **Método de Combinación de Soluciones**: la búsqueda dispersa basa su estrategia en la combinación de elementos del conjunto de referencia. De esta forma, se seleccionan subconjuntos de dicho conjunto y se les aplica un método de combinación que estará adaptado a las características del problema y de la codificación de soluciones. El mecanismo de combinación de soluciones puede

crear uno o más entornos en los que se realizan movimientos de una solución a otra, con métodos que normalmente realizan búsquedas en entornos. Entonces se ejecuta una secuencia de movimientos en el entorno seleccionado para generar soluciones intermedias que constituyen las combinaciones producidas (en la interpretación del significado de *combinación* que se emplea en la búsqueda dispersa). En este sentido, la estructura de entorno produce nuevas soluciones que contienen algunas mezclas de los atributos de los elementos iniciales, mientras que mantiene determinadas propiedades deseables como la factibilidad, mediante las características del entorno. Posteriormente, las soluciones obtenidas con el método de combinación se mejoran con una búsqueda local simple (función Mejorar).

Un esquema de la técnica de búsqueda dispersa se muestra en el algoritmo 5.2. Si se compara con el esquema general de las metaheurísticas (algoritmo 5.1) se encuentran similitudes, pues ambos parten de un conjunto inicial de elementos sobre los que se itera hasta alcanzar una cierta condición de fin, realizándose sobre el conjunto de elementos diferentes operaciones de selección y combinación que tienen como objetivo generar nuevos elementos que mejoren la mejor solución obtenida hasta el momento. Cada una de las acciones que aparecen en el esquema puede ser implementada de diversas formas y de ellas depende la eficiencia y la eficacia del algoritmo obtenido. Así, este esquema representa un modelo básico de la búsqueda dispersa, pero pueden encontrarse una gran cantidad de variantes en función de cómo se instancie cada una de las acciones señaladas. Se describen a continuación algunas posibilidades para las funciones en el algoritmo:

Algoritmo 5.2: Esquema de búsqueda dispersa.

```
CrearPoblación P;
GenerarConjuntoReferencia RS;
while Convergencia no alcanzada do
    Seleccionar elementos a combinar;
    Combinar elementos seleccionados;
    Mejorar elementos combinados;
    ActualizarConjuntoReferencia RS con los elementos más prometedores y
    los más dispersos;
end
```

- **CrearPoblación**:

  En la búsqueda dispersa normalmente se utiliza como población P un conjunto de alrededor de 100 soluciones del que se extraerá el conjunto de referencia.

Para la generación de los elementos del conjunto inicial hay varias posibilidades. Es habitual realizar una generación aleatoria de soluciones factibles y posteriormente aplicar un proceso de mejora sobre ellas utilizando, por ejemplo, una búsqueda local o un método de avance rápido. En la generación aleatoria se puede decidir asignar más probabilidad a elementos o decisiones que proporcionen un mejor valor de la función a optimizar. Otra opción es realizar la generación de cada solución a partir de las proporcionadas por métodos constructivos como GRASP.

- **GenerarConjuntoReferencia**:

  El conjunto de referencia, de un tamaño de alrededor de 20 soluciones, se extrae de la población inicial P seleccionando aproximadamente la mitad de los elementos por calidad y la otra mitad por diversidad. El método debe considerar una función de distancia con la que caracterizar la dispersión. La función de distancia usada generalmente es la distancia euclídea aunque en función del problema se pueden considerar distintas alternativas, como por ejemplo el número de componentes distintos. Además, como hemos mencionado, la distancia a maximizar puede ser al total de elementos del conjunto de referencia o al elemento más cercano de ese conjunto, en cuyo caso se dice que la solución x es la más distante con respecto al conjunto de referencia si maximiza la función de distancia $d(x, RS) = min\{d(x, y)/y \in RS\}$.

- **Convergencia**:

  La regla de parada del algoritmo admite las distintas opciones que se mencionaron para la función CondiciondeFin del algoritmo 5.1. En la búsqueda dispersa normalmente el criterio utilizado para finalizar el algoritmo es que ya no haya ninguna solución nueva en el conjunto de referencia que podamos combinar para producir mejoras, o establecer un número máximo de iteraciones y un número máximo de iteraciones sin mejorar la mejor solución obtenida.

- **Seleccionar**:

  El algoritmo selecciona una cierta cantidad de elementos del conjunto de referencia para la operación de combinación. Las estrategias de selección pueden ser también diversas. La determinación de las soluciones que se toman del conjunto de referencia para combinar se realiza generando subconjuntos de dicho conjunto. Las soluciones de estos subconjuntos se combinan para obtener nuevas soluciones que incorporan buenas propiedades de las soluciones previas. Los subconjuntos pueden generarse, por ejemplo, como sigue:

  - Todos los subconjuntos de dos soluciones.
  - Todos los subconjuntos de tres soluciones.
  - Se pueden obtener conjuntos de tres soluciones obteniendo todos los pares de elementos, y por cada par formar un conjunto de tres soluciones añadiendo un nuevo elemento, de manera que no se obtienen todos los subconjuntos de tres elementos (que quizás sería un número excesivamente grande) sino tantos como subconjuntos hay de dos elementos.
  - Todos los subconjuntos formados por soluciones que contienen a elementos buenos (con mejor valor de la función a optimizar) con elementos malos.

- **Combinar**:

  Cada conjunto de elementos seleccionados se combina de acuerdo a algún criterio. Algunas posibilidades son:

  - Se puede seguir la estrategia para combinar que se utiliza en los algoritmos genéticos: de dos elementos se generan otros dos nuevos elementos combinando partes de los elementos iniciales.
  - Dados varios elementos la combinación puede consistir en generar un nuevo elemento en el espacio delimitado por estos elementos. Con dos elementos se seleccionaría uno en la recta que los une, con tres elementos en el triángulo que delimitan... La combinación puede hacerse tomando la media de los componentes de los elementos a combinar, o con una media ponderada que asigne más peso a elementos con mejor valor de la función a optimizar.
  - También se pueden combinar los elementos componente a componente, seleccionando cada componente aleatoriamente, con mayor probabilidad para los componentes de elementos mejores.

- **ActualizarConjuntoReferencia**:

  En cada iteración se genera un nuevo conjunto de referencia donde se incluyen los elementos de mayor calidad y aquellos más diversos con respecto a estos más prometedores. De esta manera se diversifica la búsqueda intentando evitar caer en óptimos locales. Como hemos mencionado, se pueden considerar como más diversos aquellos con una distancia (euclídea) mayor con respecto a los de más calidad o a aquellos más distintos coordenada a coordenada.

El éxito de la búsqueda dispersa se basa en una adecuada integración de los métodos de selección, combinación y mejora de soluciones para actualizar el conjunto de referencia en cada paso del procedimiento.

5.5. Autooptimización a través de la búsqueda dispersa en el problema de mapeo

Tras haber estudiado en los capítulos previos la autooptimización de algoritmos iterativos paralelos sobre sistemas tanto homogéneos como heterogéneos, se ha comprobado que, en ocasiones, no es posible lograr en un tiempo razonable la asignación de procesos a procesadores con la que se obtiene el menor tiempo de ejecución. En esta sección se analiza la adecuación a este problema de la técnica de búsqueda dispersa para lograr en un tiempo de decisión reducido un mapeo con el que se obtenga de forma automática un tiempo de ejecución cercano al óptimo (autooptimización).

5.5.1. El mapeo como un problema de optimización

El problema de la asignación de procesos a procesadores se puede resolver, como se ha visto en el capítulo de introducción, buscando a través de un árbol de asignaciones, figura 1.1, que incluye todas las posibilidades de asignación de procesos a procesadores, donde en cada nivel se decide sobre a qué procesador asignar el proceso correspondiente y donde cada nodo tiene asociado un tiempo teórico de ejecución que se pretende optimizar, ecuación 5.1.

En el anterior capítulo se ha analizado el recorrido del árbol con métodos de avance rápido y de búsqueda exhaustiva. En la tabla 5.1 se resumen algunos de los experimentos realizados en el sistema heterogéneo **TORC**, y se comparan los tiempos para decidir la asignación usando métodos exactos (*backtracking*) y aproximados (*greedy*) para el esquema de programación dinámica del problema de la devolución de monedas ya explicado previamente. Se muestra el tiempo de asignación (tiempo de obtención del tiempo de ejecución mínimo según el modelo utilizado) y el tiempo de ejecución utilizando los parámetros seleccionados, en TORC con 20 procesadores y en un sistema simulado con 40 procesadores. El uso de técnicas exhaustivas (*backtracking*, *branch and bound*...), implica un excesivo consumo de tiempo incluso en el caso de que se usen estrategias de poda de nodos, tal y como se puede ver en la tabla. Por lo tanto, usar estas técnicas puede ser adecuado para sistemas pequeños, donde el tiempo para determinar la asignación de procesos a procesadores no será alto, pero no para grandes sistemas, pues implicaría recorrer gran parte de un árbol de gran dimensión. Para estos sistemas se pueden usar métodos de aproximación *greedy*, que reducirán el tiempo de búsqueda en el árbol, pero a cambio la asignación obtenida podrá quedar lejos del óptimo en muchos casos, tal como se observa en la columna del tiempo de ejecución.

Tabla 5.1: Comparación de los tiempos de ejecución y asignación (en segundos) entre los métodos de *greedy* y *backtracking* en dos sistemas heterogéneos (real y simulado).

método	TORC, 20 proc.	
	Tiempo asig.	Tiempo ejec.
backtracking	0.122	2.96
greedy	0.001	5.09
método	Simulación, 40 proc.	
	Tiempo asig.	Tiempo ejec.
backtracking	111.369	5.07
greedy	0.009	5.72

Establecer el modelo teórico del tiempo de ejecución requiere un preciso conocimiento de los parámetros del sistema, como son el tiempo de una operación aritmética básica, o el tiempo de inicio de una comunicación y de envío de un dato básico, pero estamos interesados no en la obtención de modelos precisos, sino en la aplicación de técnicas metaheurísticas utilizando estos modelos.

Con la aplicación de la búsqueda dispersa se tiene la intención de obtener asignaciones cercanas a la óptima con un tiempo de decisión reducido, al contrario de lo que ocurre con el recorrido exhaustivo del árbol de asignaciones. Por lo tanto, se plantea el problema de asignación de procesos a procesadores como un problema de optimización cuyo objetivo es obtener la asignación con el menor tiempo de ejecución teórico asociado, ecuación 5.1. Se utiliza la técnica de búsqueda dispersa, que ha probado ser de utilidad en otros problemas complejos de optimización.

5.5.2. Codificación de las soluciones y estructuras de entorno

En secciones previas se indicó que el tipo de codificación utilizado para representar una solución al problema es un factor que puede condicionar el rendimiento final en una metaheurística. Para aplicar la búsqueda dispersa a nuestro problema consideramos que cada solución representa una posible asignación de procesos a procesadores en el sistema (un nodo en el árbol de asignación tal y como aparece explicado en la figura 1.1).

La estructura utilizada para representar una solución es un array $d = (d_0, d_1, \ldots, d_{P-1})$ donde P es el número de procesadores del sistema y d_i el número de procesos asignados al procesador i (figura 1.1). En el caso de recorrido de árboles, si se fija el número de procesos a utilizar se obtiene una solución con rapidez pero se limita el número de posibilidades en las asignaciones, por lo que se consideran

elementos con diferentes números de procesos (nodos en niveles diferentes del árbol de asignaciones). Sin embargo, en la búsqueda dispersa no hay necesidad de establecer un número máximo de procesos con el fin de evitar el excesivo tiempo de decisión.

Como ejemplo ilustrativo para entender la forma de representación de la información, se puede ver que, si por ejemplo se dispone de un sistema heterogéneo con cinco procesadores, la solución $d_1 = (0, 2, 3, 0, 0)$ significa que no se asigna ningún proceso al procesador P_0, 2 procesos se asignan al procesador P_1, 3 a P_2 y ningún proceso a P_3 y P_4. Otra solución posible podría ser, por ejemplo, $d_2 = (0, 2, 2, 1, 0)$. Las soluciones d_1 y d_2 son muy parecidas, pues sólo se diferencian en que la primera no asigna ningún proceso a P_3 y la segunda sí, además de que emplean un número cercano de procesos en el procesador P_2. Esto no tiene por qué implicar que proporcionen tiempos de ejecución similares, pues esto depende de la función de optimización. Lo que sí es cierto es que son soluciones "vecinas", es decir, que se encuentran cercanas debido a su representación espacial y podrían ser generadas a partir de una misma solución conforme a los diferentes pasos de la técnica metaheurística.

Se pueden definir distintos tipos de vecindad o entorno de una asignación dada. Por ejemplo, se puede considerar que las vecinas de una asignación $a = (a_0, a_1, \ldots, a_{P-1})$ son las que se diferencian de ella en un único componente y la diferencia es de una unidad. Así, la asignación $b = (b_0, b_1, \ldots, b_{P-1})$ es vecina de la a si $a_i = b_i \; \forall i \neq k$ para un k entre 0 y $P - 1$, y $|a_k - b_k| = 1$. Con esta definición de vecindad, los vecinos de la asignación d_1 son: $(1, 2, 3, 0, 0)$, $(0, 1, 3, 0, 0)$, $(0, 3, 3, 0, 0)$, $(0, 2, 2, 0, 0)$, $(0, 2, 4, 0, 0)$, $(0, 2, 3, 1, 0)$ y $(0, 2, 3, 0, 1)$.

Otra posibilidad es definir la vecindad de manera que a y b son vecinos si tienen los mismos procesos pero sólo uno de ellos está asignado a un procesador distinto: $\sum_{i=0}^{P-1} a_i = \sum_{i=0}^{P-1} b_i$ y $a_i = b_i \; \forall i \neq k_1, \; i \neq k_2$ para k_1 y k_2 entre 0 y $P-1$, y $|a_{k_1} - b_{k_1}| = 1$ y $|a_{k_2} - b_{k_2}| = 1$. En este caso, los vecinos de d_1 son: $(1, 1, 3, 0, 0)$, $(0, 1, 4, 0, 0)$, $(0, 1, 3, 1, 0)$, $(0, 1, 3, 0, 1)$, $(1, 2, 2, 0, 0)$, $(0, 3, 2, 0, 0)$, $(0, 2, 2, 1, 0)$ y $(0, 2, 2, 0, 1)$.

En la búsqueda dispersa se utiliza la vecindad de un nodo para realizar en ella la búsqueda local o el método de avance rápido que se utiliza para mejorar las soluciones que se generan. Con una vecindad amplia se explorarán más elementos, por lo que la mejora que se realiza puede ser sustancial, pero el tiempo empleado en ella puede ser grande. Para adaptar la búsqueda dispersa a un problema concreto será conveniente considerar distintas vecindades y experimentar con ellas para obtener una con la que se tengan resultados satisfactorios en tiempo de decisión y en la función objetivo a optimizar.

5.5.3. Instanciación del método

En esta subsección se va a detallar la forma en que se han adaptado las operaciones de que consta el método de la búsqueda dispersa para resolver el problema de asignación de tareas a procesadores:

CrearPoblación:

Se crea una población inicial de soluciones donde cada elemento representa una asignación de procesos a procesadores. De entre las distintas posibilidades disponibles para generar soluciones, en nuestra implementación hemos considerado las dos siguientes:

- Generar soluciones en las que se asignan procesos a procesadores de forma aleatoria entre dos valores, independientemente del número de procesadores en el sistema.
- Generar soluciones teniendo en cuenta el número de procesadores del sistema y generar un número de procesos proporcional al de procesadores.

Por ejemplo, si consideramos cinco procesadores y con la primera aproximación el número de procesos que se asigna a cada procesador está entre 0 y 10, se puede generar una asignación $d = (6, 4, 2, 9, 6)$. Si en la segunda aproximación se genera un número de procesos doble del de procesadores, una posible asignación es $d = (1, 1, 3, 3, 2)$.

La primera posibilidad introduce aleatoriedad, mientras que la segunda guía un poco más la búsqueda hacia soluciones factibles. Parece más lógica la segunda, pues en un sistema con muchos procesadores con la primera opción se generarían muchos procesos, lo que ralentizaría el tiempo de ejecución al incrementar las comunicaciones en el programa paralelo. En los resultados experimentales se han considerado ambas posibilidades en la realización de las pruebas para adaptar el método al problema de asignación.

GenerarConjuntoReferencia:

En el conjunto de referencia se incluyen no sólo los elementos de mayor calidad, que en este caso son aquellos que tienen asociados los menores tiempos de ejecución de acuerdo con la función objetivo, sino también aquellos más diversos respecto a los elementos del conjunto (para lo que hay que definir una función de distancia). Como ya hemos comentado, si sólo se consideraran los mejores elementos, es posible que se converja rápidamente a un óptimo local del que tal

vez no sea posible salir. La inclusión de elementos diversos contribuye a explorar un espacio de búsqueda completo para converger a un óptimo global.

Para nuestro problema, se ha seleccionado a partir del conjunto inicial un conjunto de referencia de 20 soluciones, que contiene los mejores elementos y otros dispersos con respecto a los mejores.

Existen diferentes posibilidades para medir la distancia de los elementos respecto a los mejores. Hemos considerado dos casos:

- Podemos considerar como más diversos aquellos con una distancia euclídea mayor con respecto a los de más calidad.
- Otra opción puede ser considerar que son más diversos a aquellos más distintos coordenada a coordenada con respecto a los de más calidad.

Si elegimos la primera opción tendríamos que calcular la media de las distancias euclídeas de cada elemento con respecto a los mejores elementos, es decir, con respecto a aquellos que minimizan la función del tiempo de ejecución según el modelo parametrizado. La distancia para cada elemento se calcularía como la raíz cuadrada de la suma de los cuadrados de las diferencias de componentes (coordenadas) entre cada par de elementos. Siguiendo con el ejemplo anterior, la distancia euclídea de la solución $d_x = (4, 1, 1, 0, 2)$ y $d_y = (7, 5, 5, 2, 2)$ con respecto a las mejores soluciones, si suponemos que son $d_1 = (0, 2, 3, 0, 0)$ y $d_2 = (0, 2, 2, 1, 0)$, sería $dist(x, 1) = 5$, $dist(x, 2) = 4.79$, $dist(y, 1) = 8.36$, $dist(y, 2) = 8.48$ y por lo tanto la distancia euclídea media de d_x sería 4.895 y de d_y sería 8.42.

Si se elige la segunda opción habría que contar el número medio de coordenadas diferentes con respecto a los mejores elementos. En el caso de d_x sería 4.5 y en el caso de d_y 5.

A los elementos del conjunto de referencia inicial se aplica el procedimiento de mejora que se describirá más adelante.

Convergencia:

Aunque es habitual considerar que el criterio para finalizar la ejecución del método es que ya no haya ninguna solución nueva en el conjunto de referencia que podamos combinar, dado que en el problema que estamos abordando es fundamental obtener un tiempo de decisión reducido (que al sumarlo al tiempo de

ejecución del programa paralelo no le añada una sobrecarga excesiva), se ha experimentado con otros criterios menos restrictivos. Existen varias posibilidades, y las consideradas en nuestra implementación son:

- Considerar que la convergencia se alcanza cuando el mejor elemento del conjunto de referencia no mejora al mejor de la anterior etapa.
- Continuar las iteraciones mientras la media de las funciones de los elementos en el conjunto de referencia mejore.

En los resultados experimentales se han considerado las dos opciones. Tal y como se verá a continuación, se obtienen mejores resultados haciendo uso de la segunda opción.

Seleccionar:

A la hora de seleccionar los elementos a ser combinados se pueden seguir diferentes criterios. En las pruebas se han seguido dos opciones:

- Seleccionar todos los elementos para ser combinados todos con todos por pares de elementos. De esta forma se generarán muchos elementos, con lo que la exploración será mejor pero requerirá de un tiempo de ejecución grande.
- Dividir los elementos en dos grupos: aquellos que optimizan la función objetivo por un lado y por otro aquellos que son más dispersos con respecto a los mejores del otro grupo. Se combinarán por pares los elementos de un grupo con los del otro.

Combinar:

Conforme con la notación usada para la codificación de las soluciones, se ha optado por combinar los elementos seleccionados por parejas, es decir, cada par de soluciones se combinan entre sí para formar una nueva solución. En esta nueva solución las nuevas coordenadas se pueden calcular de diferentes formas. Se han utilizado las siguientes:

- Se genera al azar un valor que sirve para elegir como nueva coordenada el valor medio de las coordenadas de cada pareja o bien el valor mínimo de las dos coordenadas de la pareja.

- Se genera al azar un valor que sirve para elegir como nueva coordenada el valor medio de las coordenadas de cada pareja o bien el valor máximo de las dos coordenadas de la pareja.
- Puesto que las coordenadas representan los procesos asignados a cada procesador, se elige aquella coordenada cuyo valor mejor se adapta a la capacidad computacional del procesador que representa. Se puede tomar el menor de los valores en procesadores con menor capacidad computacional y el mayor en los procesadores con mayor capacidad.

Con la última opción, si queremos combinar las soluciones seleccionadas en la fase anterior $d_1 = (0, 2, 3, 0, 0)$ y $d_2 = (0, 2, 2, 1, 0)$, el resultado de la combinación podría ser la solución: $d_x = (0, 2, 3, 1, 0)$. Las dos primeras coordenadas son iguales en ambas soluciones por lo que en la solución generada sería igual, la tercera componente sería un 3 si suponemos que el procesador P_2 es rápido (puede admitir más procesos), la cuarta componente sería un 1 si suponemos que el procesador P_3 es rápido (puede admitir más procesos) y la última componente sería un 0 al coincidir en ambos casos.

Mejorar:

El procedimiento de mejora se aplica a las soluciones del conjunto de referencia inicial y también a los elementos del conjunto resultante de la combinación de elementos. Con esta acción se pretende obtener soluciones que mejoren el valor de la función objetivo, que en nuestro caso es el tiempo modelado del algoritmo paralelo. En nuestra implementación se ha utilizado como método de mejora un método de avance rápido.

Siguiendo con el ejemplo usado hasta ahora, si como resultado de los diferentes pasos del algoritmo (selección, combinación...) se ha obtenido una posible solución $d_z = (6, 2, 0, 0, 1)$, de acuerdo con la estructura de entorno fijada, lo que se haría es generar las posibles soluciones consistentes en añadir un proceso a cada uno de los procesadores, lo cual generaría las posibles soluciones: $d_{z1} = (7, 2, 0, 0, 1)$, $d_{z2} = (6, 3, 0, 0, 1)$, $d_{z3} = (6, 2, 1, 0, 1)$, $d_{z4} = (6, 2, 0, 1, 1)$ y $d_{z5} = (6, 2, 0, 0, 2)$. A continuación se comprobarían los tiempos que proporcionarían cada una de ellas y si el menor fuera mejor que el de d_z, ésta se sustituiría por la asignación con el menor valor.

Hemos considerado la posibilidad de añadir un proceso, pero el criterio podría ser cualquier otro, como, por ejemplo, disminuir un proceso a cada procesador, o añadir o disminuir un proceso a un procesador elegido al azar, o cualquier

otro criterio que se considere, pues hay que recordar que la aleatoriedad es un elemento importantísimo de las metaheurísticas.

ActualizarConjuntoReferencia:

En cada iteración se genera un nuevo conjunto de referencia con el mismo número de elementos que en la iteración anterior. De todos los elementos generados con la combinación y mejora se seleccionan los mejores y los más dispersos, la mitad de cada tipo:

- Los elementos de mayor calidad o más prometedores son aquellos que proporcionan mejores tiempos de ejecución conforme al modelo paramétrico de la función temporal a optimizar.
- Los más dispersos son aquellos elementos que se encuentran a mayor distancia euclídea media de los elementos más prometedores, o bien aquellos que son más diferentes en número de componentes con respecto a los más prometedores.

5.6. Experiencia computacional

Como acabamos de ver, para cada una de las acciones del esquema de la búsqueda dispersa existen múltiples posibilidades, aunque sólo se han explicado aquellas que hemos considerado como más relevantes. Se pueden estudiar todas ellas por separado pero estudiar el efecto que tiene su aplicación y sus posibles combinaciones implicarían un enorme esfuerzo temporal, por ello en los experimentos que se muestran a continuación las acciones Combinar y Mejorar van a ser siempre las mismas. Sin embargo, se han considerado dos posibles opciones para cada una de las acciones generar (funciones CrearPoblación y GenerarConjuntoReferencia), Convergencia, Seleccionar y ActualizarConjuntoReferencia.

Con el fin de analizar la influencia de cada una de las anteriores posibilidades y elegir aquellas que hagan más eficiente la técnica de búsqueda dispersa, hemos realizado experimentos tanto en sistemas reales como con simulaciones de sistemas heterogéneos. Seguimos haciendo uso del problema de las monedas, en el que se pretende minimizar el número de monedas a devolver, y la función objetivo a optimizar en el problema de asignación es el tiempo teórico de ejecución.

En el caso de las simulaciones, los parámetros del sistema han sido elegidos de la siguiente forma: el número de procesadores varía entre 20 y 100, el máximo número de procesos se obtiene en el rango $[15, \ldots, 15 + pdr]$, donde pdr es el número de

procesadores del sistema, y los valores de los parámetros computaciones (t_{c_i}) han sido elegidos aleatoriamente en el rango $[10^{-10}, \ldots, 10^{-10} + 10^{-10} randmax]$, donde $randmax$ es un número específico de cada sistema. Para estimar los valores de los parámetros de comunicaciones (t_s y t_w) en los sistemas reales nos hemos decidido por aplicar un ajuste por mínimo cuadrados considerando el sistema como homogéneo. El uso de estos rangos viene determinado por incluirse en ellos los valores de los sistemas en los cuales se han hecho las pruebas reales.

Para comprobar que la búsqueda dispersa es una buena alternativa a los métodos de búsqueda exhaustiva cuando el tamaño del sistema crece, en la tabla 5.2 se comparan los tiempos modelados de ejecución para distintos tamaños de problema y complejidades. Así mismo en la figura 5.1 se muestra una relación entre los cocientes de los tiempos obtenidos con la búsqueda dispersa para estas diferentes configuraciones con respecto a los tiempos óptimos. Como se puede apreciar, conforme el tamaño del problema aumenta y con independencia de la complejidad, la diferencia porcentual se reduce desde un máximo de casi un 20 % hasta casi un valor nulo. Se considera un sistema heterogéneo con 18 procesadores de tres tipos diferentes y con red Fast-Ethernet. Los valores obtenidos con la búsqueda dispersa están cerca de los óptimos, por lo que consideramos la búsqueda dispersa una buena alternativa para sistemas de mayor complejidad, donde las restricciones de tiempo en la toma de decisión hacen impracticables los métodos exhaustivos.

Se ha utilizado una versión básica de la búsqueda dispersa en la que las funciones tienen la forma:

Tabla 5.2: Comparación del tiempo de ejecución óptimo y el modelado utilizando una búsqueda dispersa básica.

Complejidad:	10		50		100	
Tamaño	Óptimo	*Scatter*	Óptimo	*Scatter*	Óptimo	*Scatter*
100000	0.59	0.69	2.04	2.29	3.85	3.87
500000	2.05	2.27	8.68	8.75	16.53	16.60
750000	2.95	3.02	12.69	12.69	24.28	24.75
1000000	3.87	3.87	16.71	16.71	31.88	31.88

- En la inicialización de la población se asigna un número aleatorio de procesos a cada procesador (entre unos valores mínimo y máximo, pero aumentando la probabilidad de asignar más procesos a procesadores con mayor capacidad computacional). Los elementos generados se mejoran con un avance rápido donde

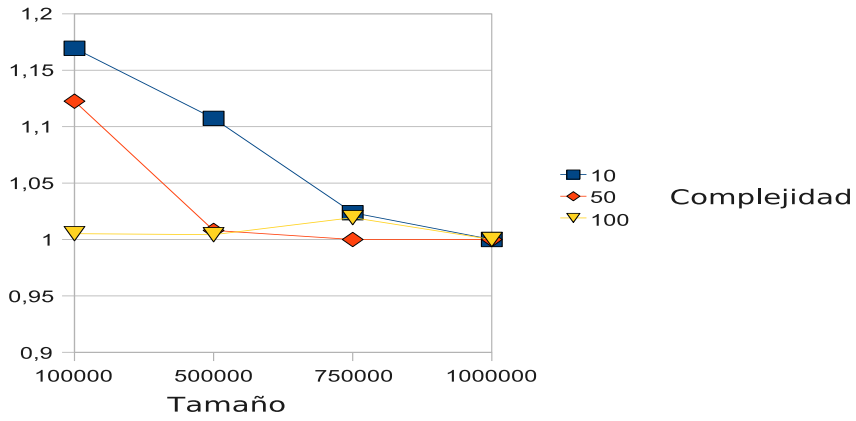

Figura 5.1: Cociente entre los tiempos obtenidos con la técnica de búsqueda dispersa y el valor óptimo.

se consideran vecinos de un elemento los que se diferencian de él en un único componente teniendo un valor más (se asigna un proceso adicional a uno de los procesadores en el sistema).

- El método converge cuando en una iteración la mejor solución no mejora a la mejor solución de la iteración anterior.
- Se seleccionan todos los elementos para combinarlos por pares.
- Los pares de elementos se combinan componente a componente. En cada componente se decide aleatoriamente si se toma el mayor o menor valor que aparece en ese componente en los dos elementos, asignando más probabilidad de aumentar en procesadores con mayor capacidad computacional.
- La misma técnica de avance rápido que se utiliza con la población inicial se usa para mejorar los elementos que se generan por combinación.
- El conjunto de referencia se actualiza incluyendo en él la mitad de los elementos los mejores y la otra mitad los más dispersos. En los experimentos se incluyen los 10 elementos con menor tiempo modelado y los 10 con mayor distancia euclídea a los 10 más prometedores.

Una vez hemos comprobado la utilidad de la búsqueda dispersa en este problema, tenemos que adaptar el método básico a las características del problema considerando distintas implementaciones de las funciones y distintos valores de los parámetros.

Como hay muchas posibles combinaciones, simplificaremos el estudio experimentando por separado con las diferentes acciones.

En la tabla 5.3, se comparan las dos posibilidades consideradas para la generación inicial y el criterio de convergencia. La tabla muestra el porcentaje de experimentos en que gana cada opción (se obtiene menor tiempo total: suma del tiempo de decisión y el del modelo), tanto en la generación del conjunto inicial de referencia como en la convergencia alcanzada. Los resultados confirman lo esperado: que es mejor generar el conjunto de referencia inicial considerando el número de procesadores del sistema y que también es mejor que el algoritmo acabe cuando la media del nuevo conjunto de referencia no mejore la media del conjunto de referencia anterior.

Tabla 5.3: Comparación entre las posibilidades para generar el conjunto inicial de referencia y el criterio de convergencia. Porcentaje de casos en que gana cada una de las opciones.

Generar el conjunto inicial de referencia	Resultados
Considerando número procesos	80 %
Aleatoriamente	20 %
Convergencia alcanzada	**Resultados**
Media	80 %
Mejores elementos	20 %

Una vez que las acciones generar y convergencia han sido fijadas, es necesario elegir las mejores opciones para la selección y la actualización del conjunto de referencia. La tabla 5.4 representa la comparación entre cuatro combinaciones con respecto al método de *backtracking* con poda.

Hemos denominado TO-EU, TO-DI, MD-EU y MD-DI a las simulaciones realizadas. Se pueden seleccionar todos los elementos (TO) para combinarlos, o los mejores y otros dispersos (MD). La distancia que se utiliza para medir la dispersión respecto a los mejores es la distancia euclídea (EU) o la que cuenta el número de componentes distintos (DI). Los números en la tabla representan el porcentaje de simulaciones donde la búsqueda dispersa proporciona un tiempo total de ejecución más bajo (tiempo de decisión junto con el tiempo modelado) con respecto a *backtracking* con poda. El sistema considerado es el mismo que en la tabla anterior.

Los mejores resultados, como se puede comprobar, se han obtenido cuando se seleccionan todos los elementos para ser combinados con todos y cuando el número de diferencias es usado para determinar los elementos "más diferentes" con respecto a los más prometedores.

Tabla 5.4: Comparación entre las posibilidades de las opciones de selección e inclusión. Porcentaje de casos en que mejora la búsqueda dispersa respecto al *backtracking* con poda.

	Seleccionar elementos	
Incluir	Todos (TO)	Mejores/dispersos (MD)
Distancia euclídea (EU)	90 %	85 %
Diferencia de componentes (DI)	91 %	86 %

Para comprobar las bondades de las anteriores 4 configuraciones se han hecho simulaciones para cada una de ellas sobre el problema de las monedas con un tamaño de datos de 500000, el uso de 20 monedas de diferentes valores generadas al azar, 20 procesadores y 67 procesos. La figura 5.2 y la tabla 5.5 muestran los resultados obtenidos para las 4 combinaciones previas en una simulación particular. Diferentes ejecuciones devuelven resultados distintos pero el comportamiento es muy parecido. Los métodos MD-EU y MD-DI tienen tiempos de decisión bajos (0.18 y 0.47 segundos respectivamente) con respecto a los TO-EU y TO-DI (5.89 y 3.33 segundos respectivamente), pero el tiempo total, considerando el tiempo de la solución obtenido, es más bajo en los métodos TO-EU y TO-DI. TO-DI tiene un número menor de iteraciones que TO-EU, pero sus tiempos según el modelo son parecidos, por lo que consideramos TO-EU y TO-DI como las opciones más adecuadas.

Tabla 5.5: Tiempos modelados y de decisión y número de iteraciones para diferentes métodos de selección e inclusión en el conjunto de referencia.

Comcinación	Iteraciones	Tiempo modelado	Tiempo de decisión	Tiempo total
TO-EU	96	1796.04	5.89	1801.93
TO-DI	69	1797.83	5.33	1803.16
MD-EU	7	2237.24	0.18	2237.42
MD-DI	2	2411.21	0.47	2411.68

Una vez seleccionada la mejor combinación, se han realizado simulaciones para comparar el comportamiento de la búsqueda dispersa con el del *backtracking* con poda al variar la granularidad de la computación y el tamaño del problema. En la tabla 5.6 y la figura 5.3 se muestra la comparación entre un *bactracking* con poda y la búsqueda dispersa usando la combinación TO-EU anteriormente seleccionada. La complejidad es un parámetro utilizado para aumentar el peso de la computación

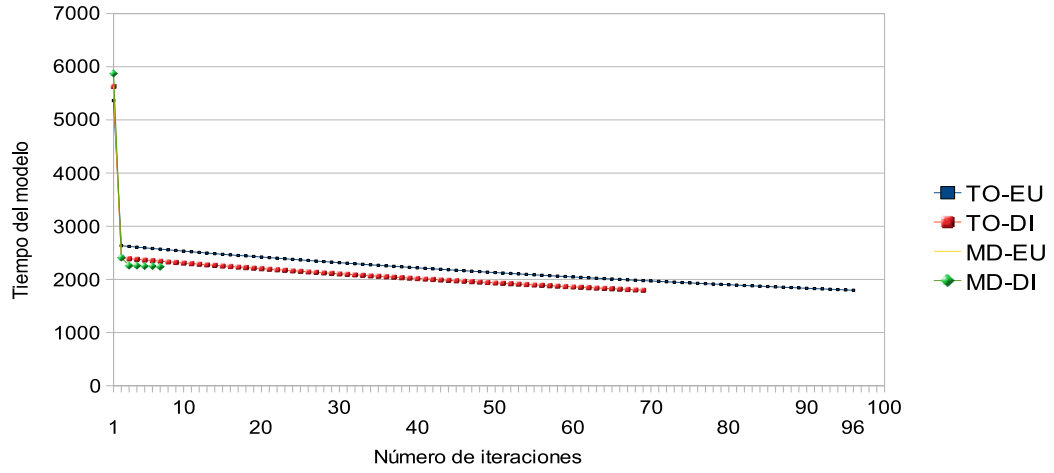

Figura 5.2: Tiempos e iteraciones para diferentes métodos de selección e inclusión en el conjunto de referencia.

en comparación con las comunicaciones. Las pruebas han sido realizadas con valores de la complejidad denominados pequeños ($[1, \ldots, 100]$) y grandes ($[100, \ldots, 400]$). Aquí no influyen las comunicaciones, sólo la computación. El tamaño del problema (en el problema de las monedas sería la cantidad de dinero a devolver) influye tanto en la parte de computación como en la de comunicaciones. Hemos nombrado los distintos valores del tamaño del problema para realizar las pruebas como pequeño ($[1, \ldots, 100000]$), mediano ($[100000, \ldots, 500000]$) y grande ($[500000, \ldots, 1000000]$).

De esta forma se han llevado a cabo pruebas considerando problemas de tamaño pequeño, mediano y grande y también teniendo en cuenta valores grandes y pequeños para el ratio computación/comunicación. Cada porcentaje representa el número de simulaciones donde la correspondiente técnica es mejor con respecto a la otra, es decir, donde el tiempo total de ejecución (tiempo de asignación junto con el tiempo obtenido por el modelo según esta distribución) es más bajo. Para problemas de tamaño pequeños, el *backtracking* con poda es mejor que la búsqueda dispersa con independencia de la complejidad, pero cuando el tamaño del problema o la complejidad crecen, la búsqueda dispersa mejora los resultados. Se puede decir que en general los resultados al aplicar búsqueda dispersa son buenos, en especial cuando los problemas son grandes y complejos.

Tabla 5.6: Comparación entre *backtracking* con poda y búsqueda dispersa.

Tamaño	Complejidad	back.	búsq. disp.
Pequeño $[1, \ldots, 100000]$	Pequeño $[1, \ldots, 100]$	92 %	8 %
Pequeño $[1, \ldots, 100000]$	Grande $[100, \ldots, 400]$	73 %	27 %
Medio $[100000, \ldots, 500000]$	Pequeño $[1, \ldots, 100]$	42 %	58 %
Medio $[100000, \ldots, 500000]$	Grande $[100, \ldots, 400]$	40 %	60 %
Grande $[500000, \ldots, 1000000]$	Pequeño $[1, \ldots, 100]$	22 %	78 %
Grande $[500000, \ldots, 1000000]$	Grande $[100, \ldots, 400]$	5 %	95 %

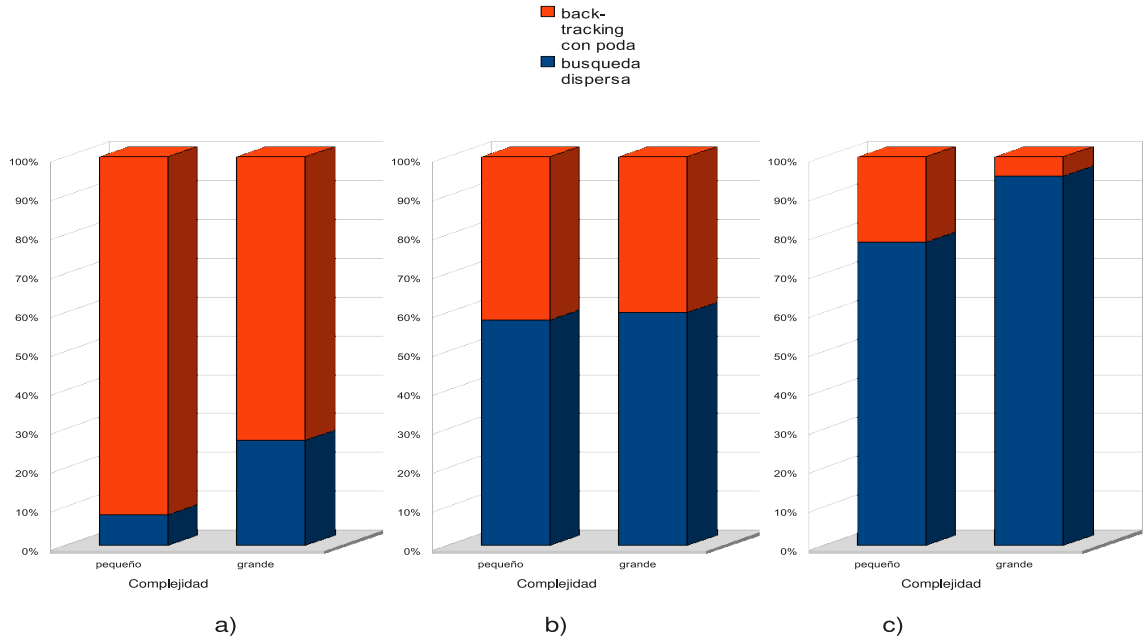

Figura 5.3: Porcentajes de éxito de la búsqueda dispersa con respecto al *backtracking* con poda considerando un tamaño de datos pequeño a), mediano b) y grande c).

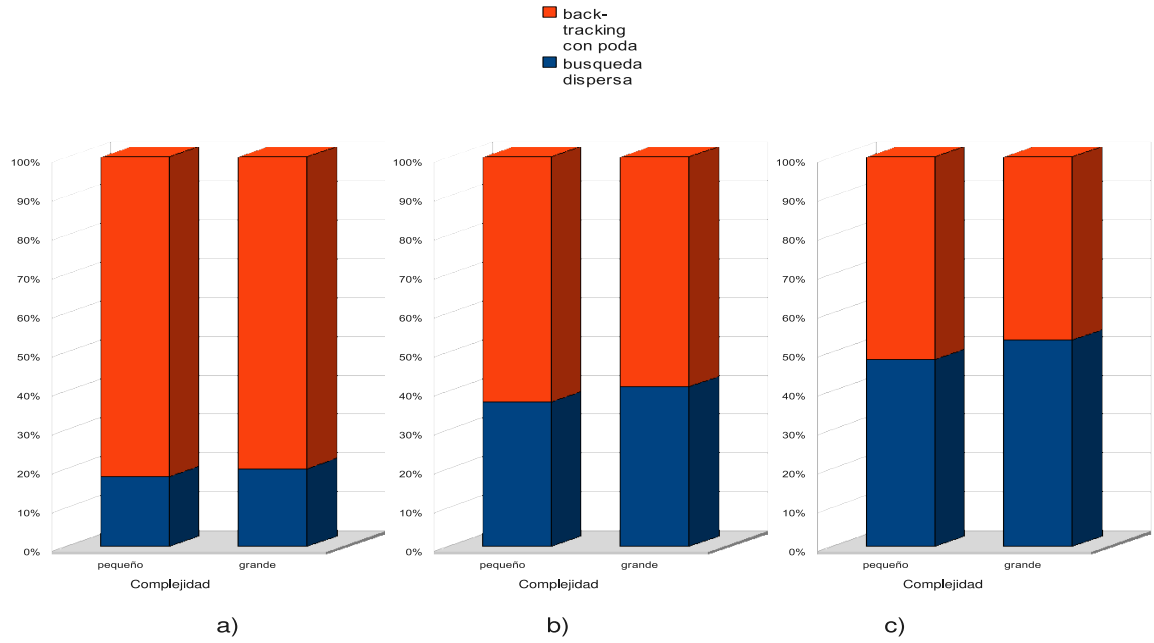

Figura 5.4: Porcentajes de éxito de la búsqueda dispersa con respecto al *backtracking* con poda considerando un tamaño de datos pequeño a), mediano b) y grande c) en KIPLING.

Como se ha dicho anteriormente, también se han hecho ejecuciones de simulaciones sobre diferentes configuraciones, que son mostradas en la tabla 5.7 donde los resultados se agrupan para las simulaciones con tamaño y complejidad crecientes. Para cada combinación de tamaño, complejidad, número de procesadores (*pdrs*) y número de procesos, se han realizado experimentos variando los parámetros tal y como se ha explicado previamente. Cada porcentaje representa el número de simulaciones donde la búsqueda dispersa es mejor que el *backtracking* con poda.

Además, se muestran en la tabla 5.8 y la figura 5.4 los resultados obtenidos en el sistema heterogéneo KIPLING (capítulo 1), compuesto por elementos de diferentes características computacionales y comunicados por una red Fast-Ethernet. Con un tamaño pequeño de problema el *backtracking* con poda es mejor que la búsqueda dispersa con independencia de su complejidad, pero cuando el tamaño crece la búsqueda dispersa resulta igual de buena. Los resultados en este sistema real son peores que los obtenidos en las simulaciones, lo que se debe a que el cluster es muy sencillo (sólo consta de 4 nodos). Así, para sistemas pequeños puede ser preferible utilizar técnicas de búsqueda exhaustiva, pero para sistemas con mayor complejidad la búsqueda dispersa proporciona mejores resultados.

Tabla 5.7: Comparación entre *backtracking* con poda y búsqueda dispersa considerando diferentes tamaños y complejidades.

Tamaño	Complejidad	Sistema1
		20 pdrs
100000	100	100 %
100000	400	100 %
750000	100	100 %
750000	400	100 %
Tamaño	Complejidad	Sistema2
		40 pdrs
100000	100	100 %
100000	400	100 %
750000	100	100 %
750000	400	100 %
Tamaño	Complejidad	Sistema3
		60 pdrs
100000	100	60 %
100000	400	95 %
750000	100	95 %
750000	400	100 %
Tamaño	Complejidad	Sistema4
		80 pdrs
100000	100	40 %
100000	400	95 %
750000	100	95 %
750000	400	100 %
Tamaño	Complejidad	Sistema5
		100 pdrs
100000	100	30 %
100000	400	90 %
750000	100	90 %
750000	400	95 %

Tabla 5.8: Comparación entre *backtracking* con poda y búsqueda dispersa en un sistema real (KIPLING).

Tamaño	Complejidad	back.	búsq. disp.
Pequeño $[1,\ldots,100000]$	Pequeño $[1,\ldots,100]$	82 %	18 %
Pequeño $[1,\ldots,100000]$	Grande $[100,\ldots,400]$	80 %	20 %
Medio $[100000,\ldots,500000]$	Pequeño $[1,\ldots,100]$	63 %	37 %
Medio $[100000,\ldots,500000]$	Grande $[100,\ldots,400]$	59 %	41 %
Grande $[500000,\ldots,1000000]$	Pequeño $[1,\ldots,100]$	52 %	48 %
Grande $[500000,\ldots,1000000]$	Grande $[100,\ldots,400]$	47 %	53 %

En la figura 5.5 se resume gráficamente la comparación de la búsqueda dispersa (versión TO-DI) con el *backtracking*. La figura 5.5.a) muestra el porcentaje de ejecuciones en que el tiempo de decisión más el tiempo modelado es menor con búsqueda dispersa que con *backtracking*. En todos los casos se han realizado 20 ejecuciones. Se muestran resultados para un sistema real con 4 nodos, mono y biprocesadores, con procesadores con distinta velocidad y con red Fast-Ethernet (KIPLING). Este sistema puede considerarse como muy pequeño. Las pruebas se han realizado con problemas pequeños (1 a 100000), medianos (100000 a 500000) y grandes (500000 a 1000000), y con ratios de computación/comunicación pequeños (complejidad entre 1 y 100) y grandes (entre 100 y 400). Para este sistema pequeño el *backtracking* proporciona mejores resultados que la búsqueda dispersa, pero cuando aumenta la complejidad o el tamaño del problema mejora el comportamiento de la búsqueda dispersa. La figura 5.5.b) muestra la comparación para un sistema simulado con 60 procesadores. Se muestran resultados para tamaños medianos y grandes y para complejidades pequeñas y grandes.

Para analizar la variación en el tiempo de decisión de la búsqueda dispersa, la tabla 5.9 y la figura 5.6 muestran el tiempo modelado y el de decisión para la versión TO-DI, para sistemas simulados con 20, 40, 60, 80 y 100 procesadores. El tamaño del problema es 100000 y la complejidad 100 y 400. Los valores corresponden a la media de 30 ejecuciones para cada complejidad y sistema. Se observa que el tiempo de decisión aumenta con el tamaño del sistema simulado. Esto es debido a que los elementos con los que trabaja la búsqueda dispersa son más grandes, y el coste de las funciones básicas en la metaheurística aumenta. Incluso con ese aumento en el tiempo de decisión los resultados de la búsqueda dispersa son satisfactorios, especialmente para problemas con mayor coste computacional.

Como resumen final y de acuerdo con los resultados de los experimentos, se puede

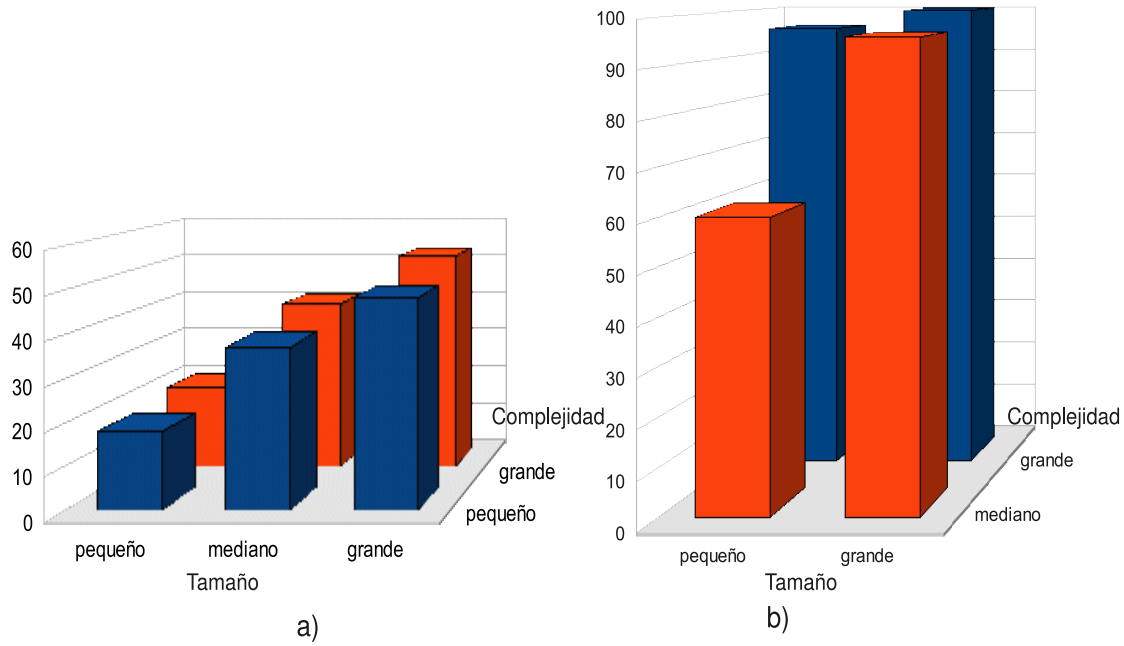

Figura 5.5: Porcentaje de ejecuciones en que la búsqueda dispersa obtiene mejor tiempo total (tiempo modelado más tiempo de decisión) que *backtracking* con eliminación de nodos: a) para un sistema real con 6 procesadores, b) para un sistema simulado con 60 procesadores.

Tabla 5.9: Tiempo modelado y de decisión (en segundos) de la versión TO-DI de la búsqueda dispersa para diferentes complejidades y sistemas simulados.

	Complejidad 100		Complejidad 400	
Procesadores	Tiempo model.	Tiempo decis.	Tiempo model.	Tiempo decis.
20	225.29	0.28	963.10	0.27
40	112.07	1.45	477.12	1.48
60	91.41	4.03	320.39	4.31
80	96.57	8.93	258.45	8.88
100	94.32	15.15	246.54	15.98

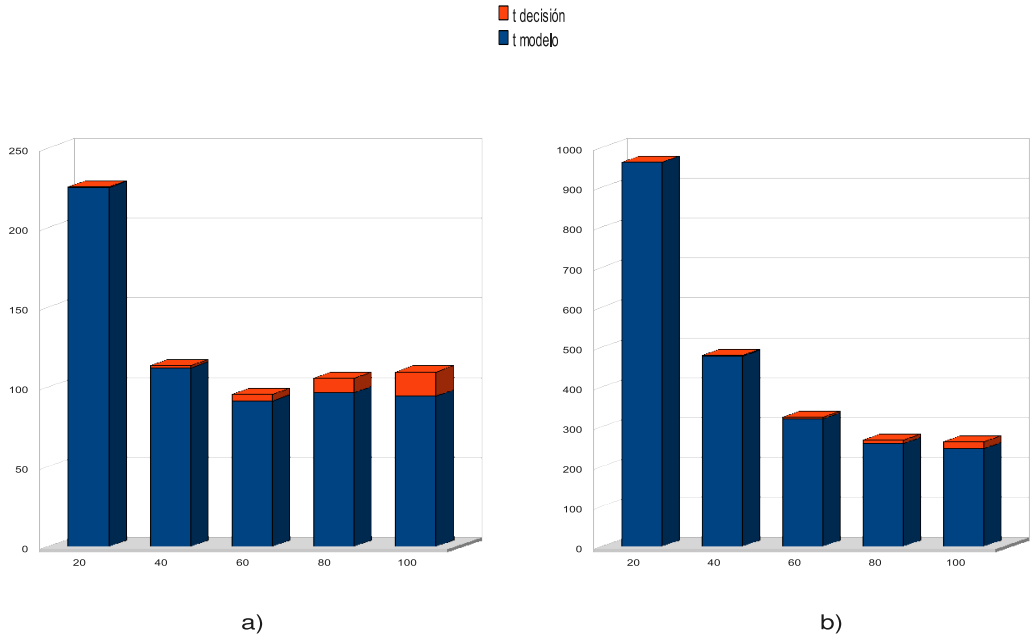

Figura 5.6: Tiempo modelado y de decisión (en segundos) de la versión TO-DI de la búsqueda dispersa para complejidades 100 a) y 400 b).

decir que la búsqueda dispersa es una técnica adecuada para resolver problemas de asignación de tareas a procesadores sobre sistemas heterogéneos, especialmente cuando los sistemas son grandes y se hace impracticable la aplicación de técnicas de búsqueda exhaustiva.

5.7. Conclusiones

En este capítulo hemos estudiado la aplicación de técnicas metaheurísticas al problema de asignación de procesos a procesadores. Las técnicas se basan en la exploración de un árbol de asignación donde cada nodo tiene asociado un tiempo teórico de ejecución. Las técnicas metaheurísticas nos permiten obtener una asignación satisfactoria en un tiempo razonable para este problema, sin la intervención del usuario haciendo posible de este modo la autooptimización. Para ello, es necesario ajustar algunos parámetros con el fin de reducir los tiempos de decisión y por tanto mejorar el tiempo total de ejecución de la solución obtenida.

Se aplica una técnica de búsqueda dispersa para explorar el espacio de búsqueda haciendo uso de dicho árbol y minimizar la función analítica de coste en el problema de asignación de procesos a procesadores en el esquema iterativo de programación

dinámica del problema de las monedas.

El método empleado nos permite obtener una asignación satisfactoria en un tiempo razonable mejorando todos los resultados obtenidos con métodos exactos y heurísticos en los capítulos anteriores. Sin embargo, la dependencia que existe respecto de los parámetros y variantes en el método, hace pensar que es posible reducir aun más los tiempos de decisión y por tanto mejorar el tiempo total de ejecución de la soluciones obtenidas.

Conforme a los resultados obtenidos en las pruebas realizadas, se puede decir que la búsqueda dispersa es una técnica adecuada para la asignación de procesos a procesadores en sistemas heterogéneos de manera automática. En el futuro, tenemos la intención de aplicar otras técnicas heurísticas (búsqueda tabú, temple simulado, algoritmos genéticos, ...) [47, 77] al mismo problema y estudiar la aplicación de los métodos a otros esquemas algorítmicos paralelos [74], tratando de componer un entorno de desarrollo en el que los distintos procedimientos puedan ser reutilizados en distintos problemas.

Capítulo 6

Conclusiones y trabajos futuros

En este último capítulo resumimos las principales conclusiones que se pueden extraer del trabajo realizado con el fin de tener una visión general de los resultados obtenidos. También se enumeran los documentos generados durante la realización de la tesis, junto con los proyectos de investigación en los que se ha desarrollado. Por último, se explican las diferentes vías de investigación que se proponen como trabajos futuros.

6.1. Conclusiones

Como se ha visto en el capítulo de introducción, el enorme desarrollo de los sistemas informáticos, tanto en su vertiente hardware como software, ha venido motivado por la necesidad de emplearlos para resolver problemas con alta complejidad computacional. Se ha buscado lograr la **optimización** del software conforme a las características particulares del sistema informático a emplear en la resolución de estos problemas. El proceso se hace complicado cuando los usuarios finales de estos sistemas no tienen grandes conocimientos de los mismos ni están interesados en su manejo, sino que pretenden usarlos como una herramienta más en su trabajo de investigación. Surge así la necesidad de lograr que esta optimización se convierta en **autooptimización**. Es decir, se trata de conseguir que el proceso de optimización y adaptación del software a las diferentes arquitecturas y al problema concreto a resolver se haga de forma transparente al usuario, o que, al menos, no suponga un esfuerzo significativo para éste. Además, la autooptimización de algoritmos puede suponer una reducción del trabajo de mantenimiento y adaptación del software, verdadero caballo de batalla de su ciclo de vida.

A lo largo de la tesis se ha analizado un tipo particular de algoritmos, los que siguen el esquema iterativo, que consiste en la ejecución repetida de un conjunto

de instrucciones hasta que se cumple un criterio de convergencia. Estos esquemas se utilizan habitualmente en la resolución de problemas científicos, por lo que tiene interés incluir en ellos técnicas de autooptimización. La finalidad del estudio no ha sido sólo la aplicación a esquemas iterativos, sino que supone un primer paso para lograr la aplicación a otros esquemas algorítmicos de la metodología empleada. Dado el interés del uso de grandes sistemas computacionales en la resolución de estos problemas científicos, se han estudiado tanto esquemas secuenciales como paralelos, y también se han estudiado algunos aspectos particulares de los esquemas según el sistema informático a emplear sea homogéneo o heterogéneo.

Se ha partido de la hipótesis de que es posible hacer uso del modelado del tiempo de ejecución del software con el fin de estimar el tiempo de ejecución de los algoritmos paralelos. Una vez estimado el tiempo de ejecución, el propio software podría ser capaz de decidir por sí mismo y sin intervención del usuario cómo ajustar los parámetros adecuados para lograr la reducción de su tiempo de ejecución en cualquier sistema informático. De esta manera se evitaría rediseñar las rutinas para obtener versiones eficientes para diferentes sistemas actuales o futuros, así como liberar al usuario del mantenimiento y optimización del software permitiéndole trabajar sólo en su ámbito de investigación.

Aunque la autooptimización podría abordase desde distintas perspectivas, nos hemos centrado en la propuesta que considera modelos matemáticos parametrizados del tiempo de ejecución de los algoritmos. Los modelos incluyen parámetros que reflejan las características del sistema y otros que pueden variarse para obtener distintas versiones del algoritmo en cuestión. La autooptimización consistiría en obtener valores de los parámetros del algoritmo con los que se consigue un tiempo teórico de ejecución mínimo sobre la plataforma considerada (teniendo en cuenta los valores de los parámetros que reflejan las características de la plataforma). En el caso de la adaptación de software homogéneo a un entorno heterogéneo, el modelo reasignará las tareas o procesos (que originariamente se repartían de forma equitativa entre los elementos del entorno homogéneo) sobre el entorno heterogéneo usando técnicas exhaustivas, aproximadas usando heurísticas o aplicando de manera sistemática técnicas metaheurísticas.

Se ha hecho un estudio sobre la aplicación de técnicas de autooptimización a esquemas paralelos iterativos de programación dinámica en un entorno homogéneo, en cuatro sistemas diferentes, con lo que se ha mostrado la ventaja que tiene la paralelización de los esquemas algorítmicos elegidos. Además, por los resultados obtenidos en los experimentos sobre dos de los sistemas anteriores, se puede deducir que con la metodología propuesta es posible la no intervención del usuario para conseguir ejecuciones

cercanas a la óptima, en tiempo de ejecución, en sistemas homogéneos. En este caso la decisión a tomar es el número de procesadores que intervienen en la resolución del problema. En la figura 6.1 se muestra una comparativa entre los diferentes usuarios y el mejor método de estimación que confirma lo anterior.

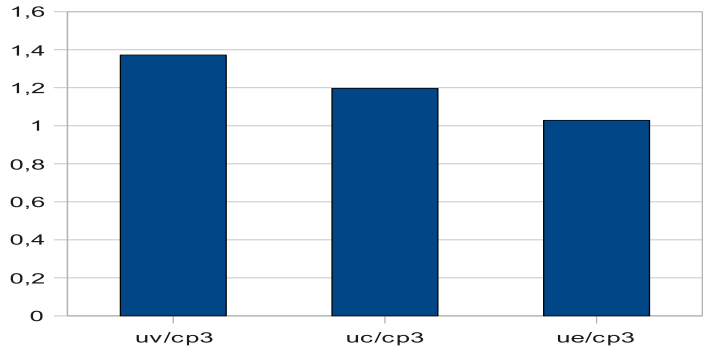

Figura 6.1: Cocientes de los tiempos de ejecución obtenidos por los distintos usuarios modelados y con respecto al mejor método de estimación de los parámetros (cp3).

Puesto que los sistemas informáticos suelen estar compuestos por elementos con características computacionales diferentes, se ha analizado la posibilidad de disponer de optimización automática en el diseño de algoritmos paralelos iterativos sobre sistemas heterogéneos. Se ha adaptado a sistemas heterogéneos la metodología de optimización basada en modelos parametrizados aplicada a sistemas homogéneos. En este caso, habría que decidir el número y tipos de procesadores a usar, el número de procesos a emplear y su distribución a los diferentes procesadores. A pesar de que el problema a optimizar es mucho más complejo que en el caso homogéneo, los experimentos realizados confirman que se pueden obtener tiempos de ejecución reducidos sin intervención del usuario haciendo uso de técnicas de autooptimización, pero también se ha hecho patente la necesidad de desarrollar métodos de selección de los parámetros (número de procesos y asignación de procesos a procesadores) más sofisticados en el caso de sistemas heterogéneos muy complejos.

El problema de asignación de procesos de esquemas paralelos iterativos a sistemas de procesadores se puede resolver buscando a través de un árbol de asignación que incluye todas las posibilidades de asignación de procesos a procesadores, y donde cada nodo del árbol tiene asociado el tiempo de ejecución teórico correspondiente a la asignación representada por el nodo. Resolver el problema de optimización por medio de técnicas exhaustivas de recorrido del árbol supone un tiempo de ejecución excesivo incluso para sistemas con un número de procesadores no demasiado elevado. Este

tiempo de recorrido del árbol no es asumible por suponer una sobrecarga al problema que se quiere resolver. Se pueden utilizar técnicas heurísticas para reducir el número de nodos recorridos y consecuentemente el tiempo de obtención de la configuración con que se resolverá el problema. Se ha analizado el recorrido de estos árboles con métodos de avance rápido y de búsqueda exhaustiva y se ha comprobado que en sistemas de tamaño reducido el uso de un método de recorrido exhaustivo del árbol de asignaciones puede ser satisfactorio al no suponer un tiempo de ejecución muy elevado y obtenerse una asignación cercana a la óptima. Sin embargo, en sistemas compuestos de muchos elementos computacionales muy variados, el recorrido exhaustivo del árbol es imposible en un tiempo razonable, aunque se use alguna estrategia de poda de nodos, por lo que la utilización de un método de avance rápido con un tiempo aceptable suele proporcionar asignaciones alejadas de la óptima. En la figura 6.2 se muestra una comparativa de diferentes simulaciones entre los cocientes de los tiempos de ejecución de los distintos métodos y usuarios con respecto a los obtenidos con $GrPe$ a) y $L = Gr$, $G = Gr$ b). Cuando el valor es menor que uno el método o usuario correspondiente toma una mejor decisión que el método aproximado correspondiente. Se puede ver que casi siempre las decisiones tomadas con $GrPe$ y con $L = Gr$, $G = Gr$ son mejores que las tomadas por los distintos usuarios.

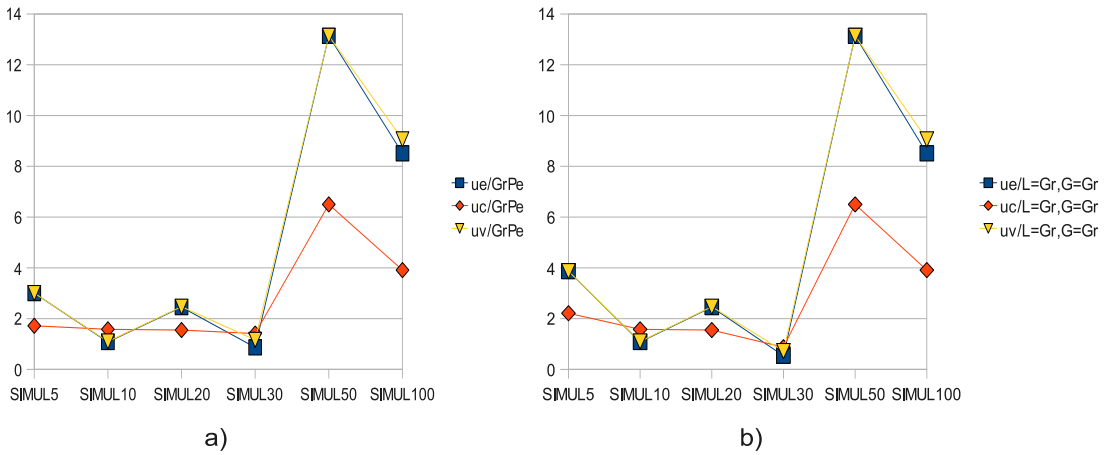

Figura 6.2: Cocientes de los tiempos obtenidos con diferentes métodos y usuarios con respecto a $GrPe$ a) y $L = Gr$, $G = Gr$ b).

Una alternativa es el empleo sistemático de técnicas metaheurísticas. Se trataría de aplicar técnicas que permiten obtener buenas soluciones para el problema del mapeo

de procesos a procesadores con unos tiempos de decisión reducidos. Las técnicas metaheurísticas han sido aplicadas con éxito en ámbitos de investigación muy variados, por lo que se ha estudiado su aplicación en combinación con la técnica de optimización de esquemas paralelos iterativos basada en el modelado parametrizado del tiempo de ejecución. En la búsqueda a través del árbol de asignaciones (o del espacio de posibles asignaciones) se pueden utilizar diferentes metaheurísticas, como búsqueda dispersa, algorítmos genéticos, búsqueda tabú, temple simulado... Se plantea la utilización de un esquema general válido para distintas metaheurísticas, lo que permite desarrollar fácilmente distintos métodos y variantes de ellos, reutilizando funciones básicas. De esta manera se facilita la experimentación para encontrar una metaheurística adecuada al problema con que se está trabajando y ajustar al problema los parámetros y funciones de la metaheurística. Se ha experimentado con la técnica de búsqueda dispersa, con la que se obtienen tiempos de resolución reducidos y asignaciones cercanas a la óptima, haciendo así posible la autooptimización de esquemas iterativos en sistemas heterogéneos complejos tal y como se aprecia en la figura 6.3.

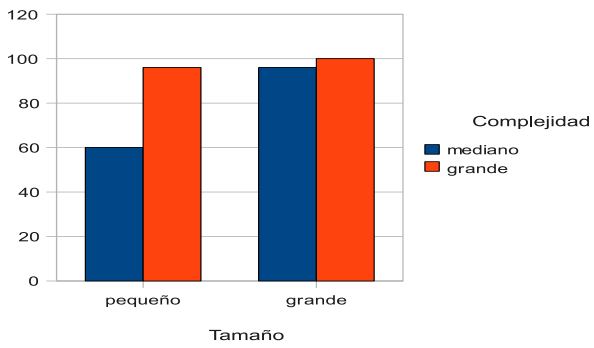

Figura 6.3: Porcentaje de ejecuciones en que la búsqueda dispersa obtiene mejor tiempo total (tiempo modelado más tiempo de decisión) que *backtracking* con eliminación de nodos para un sistema simulado con 60 procesadores.

Por último, cabe destacar que se ha conseguido el objetivo fundamental de esta tesis, que ha sido demostrar que es posible conseguir la autooptimización de códigos de esquemas algorítmicos paralelos iterativos usando técnicas exactas, aproximadas y/o metaheurísticas. Así, se han alcanzado objetivos que suponen una evolución dentro del actual contexto de técnicas de optimización de software en sistemas tanto homogéneos como heterogéneos.

6.2. Resultados y entorno de trabajo

El trabajo de esta tesis se ha realizado en el marco de varios proyectos de investigación regionales y nacionales:

- Proyecto de la Fundación Séneca, Consejería de Cultura y Educación de la Región de Murcia: "Optimización automática de rutinas paralelas" (PI-34/00788/FS/01), de 1 de enero de 2002 a 31 de diciembre de 2003.
- Proyecto CICYT, Ministerio de Educación y Ciencia: "Desarrollo y optimización de código paralelo para sistemas de audio 3D" (TIC2003-08238-C02-02), de 1 de enero de 2004 a 31 de diciembre de 2006.
- Proyecto CICYT, Ministerio de Educación y Ciencia: "COMPARHE: Computación en Paralelo y sistemas heterogéneos" (TIN2005-09037-C02-01), de 31 de diciembre de 2005 a 30 de diciembre de 2008.
- Proyecto de la Fundación Séneca, Consejería de Cultura y Educación de la Región de Murcia: "Técnicas de optimización de rutinas paralelas y aplicaciones" (02973/PI/05), de 1 de enero de 2006 a 31 de diciembre de 2008.
- Proyecto de la Fundación Séneca, Consejería de Cultura y Educación de la Región de Murcia: "Adaptación y Optimización de Código Científico en Sistemas Computacionales Jerárquicos" (08763/PI/08), de 1 de enero de 2009 a 31 de diciembre de 2011.
- Proyecto CICYT, Ministerio de Educación y Ciencia: "Construcción y optimización automáticas de bibliotecas paralelas de computación científica" (TIN2008-06570-C04-02, TIN2008-06570-C04-03), de 1 de enero de 2009 a 31 de diciembre de 2011.

Durante la realización de la tesis se han realizado varias publicaciones y comunicaciones que se enumeran y comentan a continuación:

- Automatic Optimization in Parallel Dynamic Programming Schemes. Domingo Giménez and Juan Pedro Martínez Gallar. VECPAR2004. Valencia. June 28-30, 2004. [59]

  En esta comunicación se realiza un estudio de la utilización de técnicas de autooptimización en sistemas homogéneos. Su contenido se corresponde con parte del capítulo tres de esta tesis.

- Heuristics for Work Distribution of a Homogeneous Parallel Dynamic Programming Scheme on Heterogeneous Systems. Javier Cuenca, Domingo Giménez and Juan Pedro Martínez Gallar. Proceedings de ISPDC/HeteroPar, pp 354-361. Cork, Irlanda. 2004. [38]

  Se estudia la optimización de esquemas iterativos homogéneos (con una serie de tareas idénticas) en sistemas heterogéneos por medio del modelado parametrizado del tiempo de ejecución y la resolución del problema de optimización asociado. Su contenido se encuentra en el capítulo cuatro de la tesis.

- Heuristics for Work Distribution of a Homogeneous Parallel Dynamic Programming Scheme on Heterogeneous Systems. Javier Cuenca, Domingo Giménez and Juan Pedro Martínez Gallar. Parallel Computing, 31(7):711-735, 2005. [39]

  Este artículo es una versión extendida de la comunicación anterior. Incluye un estudio más detallado del problema de optimización y su aproximación por medio de técnicas exhaustivas combinadas con heurísticas de exploración de un árbol de asignaciones. Se realiza un estudio experimental en sistemas reales y simulados. Su contenido se corresponde con parte del capítulo cuatro de la tesis.

- Parametrización de esquemas algorítmicos paralelos para autooptimización. Francisco Almeida, Juan Manuel Beltrán, Murilo Boratto, Domingo Giménez, Javier Cuenca, Luis-Pedro García, Juan-Pedro Martínez-Gallar and Antonio M. Vidal. Proceedings XVII Jornadas de Paralelismo. Albacete, 18-20 September 2006. [4]

  Se analiza la técnica de autooptimización por parametrización del modelo del tiempo de ejecución, y se muestran ejemplos con esquemas de programación dinámica (incluidos en esta tesis), divide y vencerás, *backtracking* y factorizaciones matriciales.

- Mapping in heterogeneous systems with heuristics methods. Juan Pedro Martínez Gallar, Francisco Almeida and Domingo Giménez. Proceedings de PARA 2006, Applied Parallel Computing, State-of-the-art in Scientific and Parallel Computing, Lecture Notes in Computer Science 4699, 1084-1093, 2006. [99]

  Se estudia la aplicación de técnicas metaheurísticas al problema de mapeo de procesos a procesadores para esquemas paralelos iterativos. El contenido de este trabajo se incluye en el capítulo cinco de la tesis.

- Un algoritmo de búsqueda dispersa para el problema de la asignación de procesos en arquitecturas paralelas heterogéneas. Juan Pedro Martínez Gallar, Fran-

cisco Almeida y Domingo Giménez. V Congreso de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados, 385-392. Universidad de La Laguna. 2007. [100]

El contenido de esta comunicación coincide básicamente con el del artículo anterior, aunque incluye algunos resultados adicionales.

- Application of metaheuristics to tasks to processors assignation problems in heterogeneous systems. Francisco Almeida, Domingo Giménez, Javier Cuenca and Juan Pedro Martínez Gallar. XIX Jornadas de Paralelismo. Castellón, 17-19 septiembre 2008. [6]

  Resume el trabajo de las dos comunicaciones anteriores y extiende la propuesta de aplicación de metaheurísticas de manera sistemática a otro problema de asignación de tareas con restricciones de memoria en un esquema maestro-esclavo. La aplicación a este nuevo problema no se incluye en esta tesis por no ser parte central del trabajo realizado en ella, aunque sí es una clara extensión de él.

- A framework for the application of metaheuristics to tasks-to-processors assignation problems. Francisco Almeida, Javier Cuenca, Domingo Giménez, Antonio Llanes-Castro and Juan-Pedro Martínez-Gallar. Admitido en Journal of Supercomputing. Publicado online en junio de 2009. [5]

  Este artículo contiene el estudio de la comunicación anterior ampliándolo, y propone la construcción de un entorno unificado de desarrollo y evaluación de metaheurísticas para problemas de mapeo. Esta nueva propuesta no se incluye en la tesis aunque es continuación del trabajo iniciado en ella.

- Application of metaheuristics to task-to-processors assignation problems. Domingo Giménez. Workshop Scheduling for large-scale systems. University of Tennessee, Knoxville, Mayo 2009. [57]

  Es una presentación invitada donde se analizan problemas de asignación de tareas a procesadores, incluidos los problemas que se incluyen en esta tesis y otros que están relacionados con ellos.

6.3. Trabajos futuros

Como se ha indicado, el trabajo realizado para esta tesis ha dado lugar a continuaciones y ampliaciones en distintas direcciones:

- La aplicación de técnicas heurísticas en combinación con algoritmos de recorrido exhaustivo de árboles de asignación de procesos a procesadores [38, 39] se ha

extendido y modificado para descomposiciones matriciales [32].

- La metodología de aplicación sistemática de metaheurísticas [99] se ha aplicado a problemas de asignación de tareas con restricciones de memoria en un esquema maestro-esclavo [6, 33, 37, 57].

- Teniendo en cuenta la experiencia con problemas de asignación de distinto tipo (los de esquemas iterativos objeto de esta tesis, esquemas maestro-esclavo...), tal como se ha comentado, se ha propuesto en [5] la construcción de un entorno para desarrollo y evaluación de metaheurísticas para este tipo de problemas.

- Los problemas de mapeo aquí planteados y el desarrollo de técnicas metaheurísticas para su resolución se han utilizado en un curso de programación paralela basado en problemas donde se han aplicado metaheurísticas a problemas de mapeo y se ha estudiado la paralelización de estas metaheurísticas [55, 56].

A la vista de los resultados obtenidos, se proponen algunas líneas de investigación (en algunas ya se ha empezado a trabajar) relacionadas y complementarias con el trabajo realizado en esta tesis, que se consideran interesantes a corto y medio plazo y que se podrían agrupar en tres direcciones:

- Modelado del tiempo de ejecución. Se trataría de mejorar los modelos teóricos que representan los tiempos de ejecución de los algoritmos iterativos sobre sistemas tanto homogéneos como, sobre todo, heterogéneos. Como se ha comprobado, a veces los resultados de las asignaciones de procesos a procesadores que se obtienen no son buenos al no serlos los modelos que los representan. La combinación de modelos más precisos con las técnicas propuestas facilitaría la toma de decisiones acertadas.

- Esquemas algorítmicos. Hemos usado esquemas paralelos iterativos, pero la metodología propuesta es válida para otros esquemas. En algunos casos se ha empezado a utilizar el modelado parametrizado del tiempo de ejecución de esquemas como divide y vencerás, *backtracking*, descomposiciones matriciales, maestro-esclavo..., y la investigación se puede extender a otros esquemas.

- Metaheurísticas. Vistos los buenos resultados obtenidos con el uso de la técnica de búsqueda dispersa, ha quedado demostrado que las metaheurísticas son útiles también en el caso de búsqueda de buenas soluciones para sistemas grandes y complejos cuyos árboles de asignación son muy grandes. Por ello, se propone

la continuación de esta investigación para diferentes técnicas metaheurísticas como búsqueda tabú, temple simulado, algoritmos genéticos... y la aplicación de estas técnicas a otros esquemas, y en particular el desarrollo de un entorno de trabajo común a diversas técnicas y esquemas.

Bibliografía

[1] Enrique Alba. *Parallel Metaheuristics: A New Class of Algorithms*. Wiley-Interscience, 2005.

[2] Francisco Almeida, Rumen Andonov, Daniel González, Luz Marina Moreno, Vincent Poirriez, and Casiano Rodríguez. Optimal tiling for the RNA base pairing problem. In *SPAA*, pages 173–182, 2002.

[3] Francisco Almeida, Rumen Andonov, Luz Marina Moreno, Vincent Poirriez, M. Pérez, and Casiano Rodríguez. On the parallel prediction of the RNA secondary structure. In *PARCO*, pages 525–532, 2003.

[4] Francisco Almeida, Juan Manuel Beltrán, Murilo Boratto, Domingo Giménez, Javier Cuenca, Luis-Pedro García, Juan-Pedro Martínez-Gallar, and Antonio M. Vidal. Parametrización de esquemas algorítmicos paralelos para autooptimización. In *Proceedings XVII Jornadas de Paralelismo*, 2006.

[5] Francisco Almeida, Javier Cuenca, Domingo Giménez, Antonio Llanes-Castro, and Juan-Pedro Martínez-Gallar. A framework for the application of metaheuristics to tasks-to-processors assignation problems. *Journal of Supercomputing*, publicado online, junio 2009.

[6] Francisco Almeida, Javier Cuenca, Domingo Giménez, and Juan-Pedro Martínez-Gallar. Application of metaheuristics to processors assignation problems in heterogeneous systems. In *Proceedings XIX Jornadas de Paralelismo*, 2008.

[7] Francisco Almeida, Domingo Giménez, José Miguel Mantas, and Antonio M. Vidal. *Introducción a la programación paralela*. Paraninfo Cengage Learning, 2008.

[8] Francisco Almeida, Daniel González, and Luz Marina Moreno. The master-slave paradigm on heterogeneous systems: A dynamic programming approach for the optimal mapping. *Journal of Systems Architecture*, 52(2):105–116, 2006.

[9] Francisco Almeida, Daniel González, Luz Marina Moreno, and Casiano Rodríguez. Pipelines on heterogeneous systems: models and tools. *Concurrency - Practice and Experience*, 17(9):1173–1195, 2005.

[10] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. J. Dongarra, J. Du Croz, A. Grenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1995.

[11] R. Andonov and S. Rajopadhye. Optimal Orthogonal Tiling of 2-D Iterations. *Journal of Parallel and Distributed Computing*, 45:159–165, September 1997.

[12] Rumen Andonov, Stephan Balev, Sanjay V. Rajopadhye, and Nicola Yanev. Optimal semi-oblique tiling. In *13th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 153–162, 2001.

[13] Jacques Mohcine Bahi, Sylvain Contassot-Vivier, and Raphaël Couturier. *Parallel Iterative Algorithms*. Chapman & Hall, 2007.

[14] Rashmi Bajaj and Dharma P. Agrawal. Improving scheduling of tasks in a heterogeneous environment. *IEEE Trans. Parallel Distrib. Syst.*, 15(2):107–118, 2004.

[15] Cyril Banino, Olivier Beaumont, Larry Carter, Jeanne Ferrante, Arnaud Legrand, and Yves Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. *IEEE Trans. Parallel Distrib. Syst.*, 15(4):319–330, 2004.

[16] Cristina Barrado, Eduard Ayguadé, and Jesús Labarta. Graph traverse software pipelining. Technical report, Departamento de Arquitectura de Ordenadores, UPC.

[17] Olivier Beaumont, Vincent Boudet, Antoine Petitet, Fabrice Rastello, and Yves Robert. A proposal for a heterogeneous cluster ScaLAPACK (Dense Linear Solvers). *IEEE Trans. Computers*, 50(10):1052–1070, 2001.

[18] Olivier Beaumont, Arnaud Legrand, Fabrice Rastello, and Yves Robert. Dense linear algebra kernels on heterogeneous platforms: Redistribution issues. *Parallel Computing*, 28(2):155–185, 2002.

[19] Brett A. Becker and Alexey L. Lastovetsky. Matrix multiplication on two interconnected processors. In *CLUSTER*, 2006.

[20] F. Bitz and H.T. Kung. Path planning on the Warp computer using a linear systolic array in dynamic programming. *Inter. J. Computer Math.*, 25:173–188, 1988.

[21] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[22] Jacek Blazewicz, Fred Glover, and Marta Kasprzak. DNA sequencing - tabu and scatter search combined. *INFORMS Journal on Computing*, 16(3):232–240, 2004.

[23] G. Brassard and P. Bratley. *Fundamentals of Algorithms*. Prentice-Hall, 1996.

[24] Gilles Brassard and Paul Bratley. *Fundamentos de algoritmia*. Prentice-Hall, 1997.

[25] E. A. Brewer. Portable high-performance supercomputing: High-level platform-dependent optimization, Ph.D. Thesis, Massachusetts Institute of Technology, 1994.

[26] Ángel-Luis Calvo, Ana Cortés, Domingo Giménez, and Carmela Pozuelo. Using metaheuristics in a parallel computing course. In Marian Bubak, G. Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *ICCS (2)*, volume 5102 of *Lecture Notes in Computer Science*, pages 659–668. Springer, 2008.

[27] Zizhong Chen, Jack Dongarra, Piotr Luszczek, and Kenneth Roche. Self-adapting software for numerical linear algebra and LAPACK for clusters. *Parallel Computing*, 29(11-12):1723–1743, 2003.

[28] Andrea Clematis, Gabriella Dodero, and Vittoria Gianuzzi. A practical approach to efficient use of heterogeneous PC network for parallel mathematical computation. In *HPCN Europe*, pages 464–473, 2001.

[29] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. An investigation of some properties of an "Ant Algorithm". In *PPSN*, pages 515–526, 1992.

[30] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[31] Javier Cuenca. Optimización automática de software paralelo de álgebra lineal. Departamento de Ingeniería y Tecnología de los Computadores de la Universidad de Murcia, Ph.D. Thesis, 2004.

[32] Javier Cuenca, Luis-Pedro García, Domingo Giménez, and Jack Dongarra. Processes distribution of homogeneous parallel linear algebra routines on heterogeneous clusters. In *CLUSTER*, 2005.

[33] Javier Cuenca and Domingo Giménez. Improving metaheuristics for mapping independent tasks into heterogeneous memory-constrained systems. In Marian Bubak, G. Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *ICCS (1)*, volume 5101 of *Lecture Notes in Computer Science*, pages 236–245. Springer, 2008.

[34] Javier Cuenca, Domingo Giménez, and José González. Towards the design of an automatically tuned linear algebra library. In *PDP*, pages 201–208, 2002.

[35] Javier Cuenca, Domingo Giménez, and José González. Architecture of an automatically tuned linear algebra library. *Parallel Computing*, 30(2):187–210, 2004.

[36] Javier Cuenca, Domingo Giménez, José González, Jack Dongarra, and Kenneth Roche. Automatic optimisation of parallel linear algebra routines in systems with variable load. In *PDP*, pages 409–416, 2003.

[37] Javier Cuenca, Domingo Giménez, Juan-José López-Espín, and Juan-Pedro Martínez-Gallar. A proposal of metaheuristics to schedule independent tasks in heterogeneous memory-constrained systems. In *Proc. IEEE Int. Conf. on Cluster Computing*. IEEE Computer Society, September 2007.

[38] Javier Cuenca, Domingo Giménez, and Juan-Pedro Martínez-Gallar. Heuristics for work distribution of a homogeneous parallel dynamic programming scheme on heterogeneous systems. In *ISPDC/HeteroPar*, pages 354–361, 2004.

[39] Javier Cuenca, Domingo Giménez, and Juan-Pedro Martínez-Gallar. Heuristics for work distribution of a homogeneous parallel dynamic programming scheme on heterogeneous systems. *Parallel Computing*, 31(7):711–735, 2005.

[40] O. de Moor. Dynamic programming as a software component. In N. Mastorakis, editor, *Proc. 3rd WSEAS Int. Conf. Circuits, Systems, Communications and Computers*, 1999.

[41] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subroutines. *ACM Transactions on Mathematical Software*, 14:1–17, 1988.

[42] Jack Dongarra. Self Adapting Numerical Software (SANS) for Grid Computing. Innovative Computing Laboratory, University of Tennessee, `http://icl.utk.edu/iclprojects/pages/sans.html`, 2002.

[43] Jack Dongarra, George Bosilca, Zizhong Chen, Victor Eijkhout, Graham E. Fagg, Erika Fuentes, Julien Langou, Piotr Luszczek, Jelena Pjesivac-Grbovic, Keith Seymour, Haihang You, and Sathish S. Vadhiyar. Self-adapting numerical software (SANS) effort. *IBM Journal of Research and Development*, 50(2-3):223–238, 2006.

[44] Jack Dongarra, Jean-Francois Pineau, Yves Robert, and Frederic Vivien. Matrix product on heterogeneous master-worker platforms. *In PPoPP*, pages 53–62, 2008.

[45] Antonio J. Dorta, Jesús A. González, Casiano Rodríguez, and Francisco de Sande. llc: A parallel skeletal language. *Parallel Processing Letters*, 13(3):437–448, sep 2003.

[46] Kathryn A. Dowsland and Belarmino Adenso Díaz. Diseño de heurísticas y fundamentos del recocido simulado. In *Revista Iberoamericana de Inteligencia Artificial. No.19*, pages 93–102, 2003.

[47] J. Dréo, A. Pétrowski, P. Siarry, and E. Taillard. *Metaheuristics for Hard Optimization*. Springer, 2005.

[48] Leah Epstein and Jiri Sgall. Approximation schemes for scheduling on uniformly related and identical parallel machines. *Algorithmica*, 39(1):43–57, 2004.

[49] M. Frigo. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the ICASSP Conference*, volume 3, page 1381, 1998.

[50] M. Frigo and S. Kral. The advanced FFT program generator GENFFT. Report AURORA TR 2001-03, Vienna University of Technology, Vienna Austria, 2001.

[51] Satoshi Fujita, Masayuki Masukawa, and Shigeaki Tagashira. A fast branch-and-bound scheme for the multiprocessor scheduling problem with communication time. In *ICPP Workshops*, pages 104–111, 2003.

[52] Zvi Galil and Kunsoo Park. Parallel algorithms for dynamic programming recurrences with more than O(1) dependency. *Journal of Parallel and Distributed Computing*, 21:213–222, 1994.

[53] Ismael Galindo, Francisco Almeida, Vicente Blanco, and José Manuel Badía-Contelles. Dynamic load balancing on dedicated heterogeneous systems. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 64–74, Berlin, Heidelberg, 2008. Springer-Verlag.

[54] A. Gibbons and W. Rytter. *Efficient parallel algorithms*. Cambridge University Press, 1988.

[55] D. Giménez, A.-L. Calvo, A. Cortés, and C. Pozuelo. Using metaheuristics in a parallel computing course. In *ICCS08, Proceedings International Conference on Computational Science, Vol II, LNCS*, volume 5102, pages 659–668. Springer, 2008.

[56] Domingo Giménez. Una experiencia de combinación de heurísticas y programación paralela en un curso guiado por problemas. In *JENUI*, 2008.

[57] Domingo Giménez. Application of metaheuristics to task-to-processors assignation problems, Worksop on Scheduling for large-scale systems, University of Tennessee, Knoxville, USA, 2009.

[58] Domingo Giménez, Ginés García, Joaquín Cervera, and Norberto Marín. *Algoritmos y estructuras de datos. Volumen II: Algoritmos*. Texto guía Universidad de Murcia, Diego Marín, 2003.

[59] Domingo Giménez and Juan-Pedro Martínez. Automatic optimization in parallel dynamic programming schemes. In *Proceedings of the 6th International Meeting VECPAR04*, pages 639–650, 2004.

[60] F. Glover and G. Kochenberger. *Handbook of Metaheuristics*. Springer, 2003.

[61] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[62] Fred Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8(1):156–166, 1977.

[63] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Comput. Oper. Res.*, 13(5):533–549, May 1986.

[64] Fred Glover. Tabu Search - Part I. *INFORMS Journal on Computing*, 1(3):190–206, 1989.

[65] Fred Glover. Tabu Search - Part II. *INFORMS Journal on Computing*, 2(1):4–32, 1990.

[66] Fred Glover and Manuel Laguna. General purpose heuristics for integer programming-part II. *J. Heuristics*, 3(2):161–179, 1997.

[67] Alfonso González and Martín Lara. Aplicación de la programación paralela a problemas de cálculo de órbitas. Meeting on Parallel Routines Optimization and Applications, `http://dis.um.es/~domingo/08/reunion/reunion.html`, May 2008.

[68] D. González, F. Almeida, L. Moreno, and C. Rodríguez. Towards the automatic optimal mapping of pipeline algorithms. *Parallel Computing*, 29:241–254, 2003.

[69] Daniel González, Francisco Almeida, Luz Marina Moreno, and Casiano Rodríguez. Towards the automatic optimal mapping of pipeline algorithms. *Parallel Computing*, 29(2):241–254, 2003.

[70] Daniel González, Francisco Almeida, José L. Roda, and Casiano Rodríguez. From the theory to the tools: Parallel dynamic programming. *Concurrency - Practice and Experience*, 12(1):21–34, 2000.

[71] Daniel González-Morales, Francisco Almeida, F. García, J. Gonzalez, José L. Roda, and Casiano Rodríguez. A skeleton for parallel dynamic programming. In *Euro-Par*, pages 877–887, 1999.

[72] Daniel González-Morales, José L. Roda, Francisco Almeida, Casiano Rodríguez, and F. García. Integral knapsack problems: Parallel algorithms and their implementations on distributed systems. In *International Conference on Supercomputing*, pages 218–226, 1995.

[73] Jean-Baptiste Gotteland and Nicolas Durand. Genetic algorithms applied to airport ground traffic optimization. In *IEEE Congress on Evolutionary Computation (1)*, pages 544–551, 2003.

[74] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, second edition, 2003.

[75] W. D. Harvey. Nonsystematic backtracking search. Ph.D Thesis. University of Oregon, 1995.

[76] P. Helman. A common schema for dynamic programming and branch and bound algorithms. *Journal of the ACM*, 36:97–128, 1989.

[77] J. Hromkovič. *Algorithmics for Hard Problems*. Springer, second edition, 2003.

[78] W. N. N. Hung and X. Song. BDD Variable Ordering By Scatter Search. In *International Conference on Computer Design*, pages 368–373. IEEE Computer Society, 2001.

[79] T. Ibaraki. *Enumerative Approaches to Combinatorial Optimization, Part II*. Annals of Operations Research. Volume 11, 1-4, 1988.

[80] Klaus Jansen. Scheduling malleable parallel tasks: An asymptotic fully polynomial time approximation scheme. *Algorithmica*, 39(1):59–81, 2004.

[81] Yves Robert Jean-Francois Pineau, Frédéric Vivien. The impact of heterogeneity on master-slave scheduling. pages 158–176. Parallel Computing 34(3), (2008).

[82] Sonia Jerez, Juan-Pedro Montávez, and Domingo Giménez. Optimizing the execution of a parallel meteorology simulation code. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*. IEEE, May 2009.

[83] Alexey Kalinov and Alexey L. Lastovetsky. Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers. *J. Parallel Distrib. Comput.*, 61(4):520–535, 2001.

[84] R. M. Karp and M. Held. Finite state process and dynamic programming. *SIAM Journal in Applied Mathematics*, 15:693–718, 1967.

[85] Takahiro Katagiri, Kenji Kise, Hiroki Honda, and Toshitsugu Yuba. Effect of auto-tuning with user's knowledge for numerical software. In *Conf. Computing Frontiers*, pages 12–25, 2004.

[86] H. Kuroda, T. Katagiri, and Y. Kanada. Parallel numerical library project, how ILB is to be developed. In *Workshop on Scalable Solver Software (SSS2001)*, 2001.

[87] Alexey L. Lastovetsky and Ravi Reddy. A novel algorithm of optimal matrix partitioning for parallel dense factorization on heterogeneous processors. In *PaCT*, pages 261–275, 2007.

[88] Alexey L. Lastovetsky and Ravi Reddy. A novel algortihm of optimal matrix partitioning for parallel dense factorization on heterogeneous processors. *In PaCT*, pages 261–275, 2007.

[89] Arnaud Legrand, Hélène Renard, Yves Robert, and Frédéric Vivien. Mapping and load-balancing iterative computations. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):546–558, 2004.

[90] H. Lennerstad and L. Lundberg. Optimal scheduling results for parallel computing. *SIAM News*, pages 16–18, 1994.

[91] G. Li and B. Wah. Parallel processing of serial dynamic programming programs. In *Proc. of COMPSAC 85*, pages 81–89, 1985.

[92] Chun-Yuan Lin, Chen Tai Huang, Yeh-Ching Chung, and Chuan Yi Tang. Efficient parallel algorithm for optimal three-sequences alignment. In *ICPP*, page 14, 2007.

[93] Francisco López. Paralelización del modelo hidrodinámico secuencial COHERENS para sistemas multicore mediante OpenMP. Meeting on Parallel Routines Optimization and Applications, `http://dis.um.es/~domingo/08/reunion/reunion.html`, May 2008.

[94] B. Louka and M. Tchuente. Dynamic programming on two-dimensional systolic arrays. *Information Processing Letters*, 29:97–104, 1988.

[95] G. Luque, E. Alba, and B. Dorronsoro. Parallel genetic algorithms. *Parallel Metaheuristics, E. Alba (ed.)*, 2005.

[96] R. Martí and L. Laguna. Búsqueda dispersa (scatter search). In *Actas del Primer Congreso de Algoritmos Evolutivos*, pages 302–307, 2002.

[97] R. Martí and L. Laguna. Scatter search: Methodology and implementations in C. Kluwer Academic Publishers, 2003.

[98] Rafael Martí and J. Marcos Moreno Vega. MultiStart Methods. *Revista Iberoamericana de Inteligencia Artificial*, (19):49–60, 2003.

[99] Juan-Pedro Martínez-Gallar, Francisco Almeida, and Domingo Giménez. Mapping in heterogeneous systems with heuristic methods. In *PARA*, pages 1084–1093, 2006.

[100] Juan-Pedro Martínez-Gallar, Francisco Almeida, and Domingo Giménez. Un algoritmo de búsqueda dispersa para el problema de la asignación de procesos en arquitecturas paralelas heterogéneas. In *V Congreso de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados*, pages 385–392, 2007.

[101] S. Miguet and Y. Robert. Dynamic programming on a ring of processors. *Hypercube and Distributed Computers*, pages 19–33, 1989.

[102] Serge Miguet and Yves Robert. Path planning on a ring of processors. *Intern. Journal Computer Math*, 32:61–74, 1990.

[103] Nenad Mladenovic and Pierre Hansen. Variable neighborhood search. *Computers & OR*, 24(11):1097–1100, 1997.

[104] Dan I. Moldovan. *Parallel Processing: From Applications to Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[105] D. Morales, F. Almeida, C. Rodríguez, J. Roda, I. Coloma, and A. Delgado. Parallel dynamic programming and automata theory. *Parallel Computing*, 26:113–134, 2000.

[106] L. M. Moreno. Computación en paralelo y entornos heterogéneos. Ph. D Thesis. Universidad de La Laguna, 2005.

[107] Pablo Moscato and Carlos Cotta. Una introducción a los Algoritmos Memeticos. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial*, 19:131–148, 2003.

[108] Network Weather Service. `http://nws.cs.ucsb.edu./ewiki/`.

[109] S. Pelagatti. *Structured development of parallel programs*. Taylor & Francis, 1997.

[110] Susanna Pelagatti. Data parallelism in P3L. *Parallel and distributed computing. Springer*, pages 155–186, 2002.

[111] Susanna Pelagatti, Roberto Di Cosmo, Zheng Li, and Pierre Weis. Skeletal parallel programming with OcamlP3. *Parallel Processing Letters*, 18:149–164, 2008.

[112] M. Pérez. Estudios computacionales para problemas tipo p-hub (in Spanish). Ph.D Thesis. Universidad de La Laguna, 2009.

[113] Melquíades Pérez Pérez, Francisco Almeida, and J. Marcos Moreno-Vega. Genetic algorithm with multistart search for the p-Hub median problem. In *EUROMICRO*, pages 20702–20707, 1998.

[114] Manuel Quesada. Técnicas de autooptimización en recorridos de árboles por medio de backtracking. Proyecto fin de carrera, Facultad de Informática, Universidad de Murcia, `http://www.um.es/pcgum/PFCs_y_TMs/index.html`, February 2009.

[115] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. van de Geijn, Field G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*, 36(3), 2009.

[116] Fethi A. Rabhi and Sergei Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[117] Günther R. Raidl. A unified view on hybrid metaheuristics. In *Hybrid Metaheuristics*, pages 1–12, 2006.

[118] M. Resende. Greedy randomized adaptative search procedures (GRASP). Technical Report TR 98.41.1, AT&T Labs-Research, 2000.

[119] C. Rodríguez, D. González, F. Almeida, J. Roda, and F. García. Parallel algorithms for polyadic problems. In *Proceedings of the 5th Euromicro Workshop on Parallel and Distributed Processing*, pages 394–400, 1997.

[120] Wojciech Rytter. On efficient parallel computations for some dynamic programming problems. *Theor. Comput. Sci.*, 59:297–307, 1988.

[121] Gerald Sabin, Rajkumar Kettimuthu, Arun Rajan, and P. Sadayappan. Scheduling of parallel jobs in a heterogeneous multi-site environement. In *JSSPP*, pages 87–104, 2003.

[122] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high-perfomance scientific simulations in: J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon and A. White. *Sourcebook of parallel computing (Morgan Kaufmann Publishers*, pages 491–452, 2003.

[123] Marc Snir and William Gropp. *MPI. The Complete Reference. 2nd edition.* The MIT Press, 1998.

[124] TOP500. `http://www.top500.org/`.

[125] TORC, página del sistema. `http://icl.cs.utk.edu/projects/torc/`.

[126] Michel Toulouse, Teodor Gabriel Crainic, and Brunilde Sansò. Systemic behavior of cooperative search algorithms. *Parallel Computing*, 30(1):57–79, 2004.

[127] Rob J. M. Vaessens, Emile H. L. Aarts, and Jan Karel Lenstra. A local search template. In *PPSN*, pages 67–76, 1992.

[128] Deo Prakash Vidyarthi, Anil Kumar Tripathi, Biplab Kumer Sarker, Abhishek Dhawan, and Laurence Tianruo Yang. Cluster-based multiple task allocation in distributed computing system. In *IPDPS*, 2004.

[129] Stefan Voß. Meta-heuristics: The state of the art. In *Local Search for Planning and Scheduling*, pages 1–23, 2000.

[130] B. Wah, G. Li, and C. Fen. Multiprocessing of combinatorial search problems. volume 18, pages 93–108, 1985.

[131] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[132] Barry Wilkinson and Michael Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[133] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[134] Wei Zhao and Krithi Ramamritham. Simple and integrated heuristic algorithms for scheduling tasks with time and resource constraints. *Journal of Systems and Software*, 7(3):195–205, 1987.