Curso 2004/05
CIENCIAS Y TECNOLOGÍAS/23
I.S.B.N.: 84-7756-662-3

LUZ MARINA MORENO DE ANTONIO

Computación paralela y entornos heterogéneos

Director
FRANCISCO ALMEIDA RODRÍGUEZ

SERVICIO DE PUBLICACIONES
UNIVERSIDAD DE LA LAGUNA

SOPORTES AUDIOVISUALES E INFORMÁTICOS
Serie Tesis Doctorales

Agradecimientos

Quiero expresar mi profundo agradecimiento a todos aquellos que de una forma u otra me han ayudado a realizar este trabajo a lo largo de los años.

A Francisco Almeida Rodríguez, director de esta tesis, por su constante ayuda e interés, así como por la confianza depositada en todo mi trabajo.

A Rumen Andonov (INRIA/IRISA, Rennes, Francia) y Vicent Poirriez (LAMIH/ROI, Valenciennes, Francia) por su colaboración en el desarrollo del último capítulo de esta tesis.

A todos mis compañeros del Grupo de Paralelismo, del área de Lenguajes y Sistemas Informáticos y del Departamento de Estadística, Investigación Operativa y Computación por la ayuda y el apoyo facilitado en todo momento.

A mis compañeros de la Torre de Química, con los que he compartido cafés, comidas y muchas, muchas horas de trabajo, por su apoyo constante y sus consejos, sobre todo en los malos momentos. Especialmente, a mi compañero de despacho, Roberto y a Isa S., Vicky, Isa D., Macu, Bea, Jesús, Andrés, Jordi, José Luis y Félix.

Y, por supuesto, a mi familia, que ha estado conmigo siempre.

Prólogo

Este trabajo se enmarca en el contexto de la computación en paralelo. Es un hecho conocido que el paralelismo constituye una alternativa real para reducir el tiempo de ejecución de las aplicaciones. Arquitecturas paralelas homogéneas con elementos de proceso de características similares están siendo usadas con éxito en distintos ámbitos de la ciencia y de la industria. De forma natural, la ampliación y la potenciación de un sistema homogéno con nuevos elementos, deriva en un sistema de naturaleza heterogénea en el que los nuevos componentes presentan características diferenciales con los anteriores. Así pues, es muy frecuente encontrar arquitecturas paralelas donde las características de los elementos que componen el sistema (los procesadores, la memoria, la red, ...) pueden ser diferentes.

A finales del siglo XX, en la decada de los 90, con la consolidación de estándares para la programación en arquitecturas de paso de mensaje como PVM y MPI, la programación de sistemas heterogéneos se convierte en un problema tan sencillo o complicado como pueda serlo la programación de arquitecturas homogéneas. Sin embargo, aunque la portabilidad de los códigos está garantizada, no ocurre lo mismo con el rendimiento observado en las aplicaciones. Aparecen nuevas situaciones y cuellos de botella que no pueden ser resueltos mediante la aplicación directa de los modelos y las técnicas conocidas para el caso homogeneo. Se hace necesario adaptar los métodos conocidos y en muchos casos diseñar nuevas estrategias comenzando desde cero. Nos encontramos ante un conjunto importante de problemas abiertos que están siendo intensamente estudiados y analizados. Los objetivos de este memoria se centran en el desarrollo de modelos y herramientas que faciliten el trabajo en este último tipo de entornos.

El capítulo 1, auténtico prólogo de esta memoria, consiste en una breve introducción que permite situarnos en el estado actual y en la problemática asociada a la computación en paralelo sobre los sistemas heterogéneos. Una aproximación formal a conceptos y métricas que además ilustra el contexto específico del desarrollo de nuestra propuesta y metodología.

Dos paradigmas de programación que presentan interés desde la perspectiva de la computación en paralelo son el paradigma *maestro-esclavo* y el paradigma *pipeline*. Es cierto que ambos paradigmas han sido profusamente estudiados sobre arquitecturas homogéneas; su análisis y desarrollo sobre plataformas heterogéneas constituye en estos momentos un auténtico reto. Los modelos formulados para arquitecturas homogéneas pueden considerarse como el punto de partida con el que abordar el caso heterogéneo, sin embargo, tales modelos no son directamente aplicables a este nuevo contexto. En ambas técnicas, partiendo de las propuestas clásicas para el caso homogéneo, estudiamos su comportamiento sobre arquitecturas heterogéneas. La heterogeneidad viene dada por diferencias en los elementos de procesamiento y en las capacidades de comunicación. Derivamos modelos analíticos que predicen su rendimiento y proponemos estrategias para la distribución óptima de tareas desarrollando las herramientas y utilidades necesarias en cada caso.

El capítulo 2 contempla el paradigma *maestro-esclavo*, modelizamos las estrategias FIFO y LIFO validando el modelo analítico sobre aplicaciones bien conocidas: Productos de Matrices y Transformadas de Fourier bidimensionales. Hemos resuelto el problema de la asignación óptima de los recursos (los procesadores) a las tareas, minimizando la función objetivo obtenida desde el modelo análitico, mediante algoritmos de programación dinámica. La utilidad propuesta permite no sólo establecer los beneficios de las aproximaciones FIFO y LIFO, sino que además deriva el conjunto óptimo de procesadores para una ejecución eficiente.

El capítulo 3 alude al método *pipeline*. Desarrollamos herramientas que facilitan la instanciación de algoritmos *pipeline* en los que el número de etapas puede ser muy superior al número de procesadores reales. Admite distribuciones cíclicas y cíclicas por bloques de las etapas del *pipeline* virtual al *pipeline* real. Considerado el caracter heterogéneo de la plataforma, las distribuciones cíclicas por bloques admiten asignaciones vectoriales en las que los procesadores reciben tamaños de bloques acordes a su capacidad computacional. El tamaño del *buffer* de comunicación es también un parámetro de entrada. La im-

plementación de algoritmos de programación dinámica para Problemas de la Mochila 0/1, Problemas de Asignación de Recursos, Problemas de Subsecuencias de Cadenas y Problemas de Planificación de Caminos ilustra el uso y los beneficios de la librería. La modelización analítica del método ha sido validada sobre problemas sintéticos proponiendo técnicas con la que obtener parámetros (tamaños de bloques y buffers) para su ejecución óptima. Algunos autores encuentran analogías entre el comportamiento de las aplicaciones en plataformas heterogéneas y el de determinados códigos irregulares sobre plataformas homogéneas. Este hecho admite la consideración de la heterogeneidad debida al programa en plataformas homogéneas. El capítulo 4 presenta una de tales aplicaciones, la predicción de la estructura secundaria del RNA. El paquete de Viena es un software de libre distribución que resuelve este problema, además de otros asociados a la biología molecular. Implementa un conocido algoritmo de Programación Dinámica que presenta un particular tipo de dependencias en el dominio de ejecución. El espacio de iteraciones presenta un comportamiento irregular en el que el coste de cada iteración es variable, condicionando un particionado no regular. Proponemos modelos analíticos con los que obtener las dimensiones óptimas en esta división del trabajo. Dada la dificultad del problema, el modelo introduce simplificaciones en el sistema para facilitar su manipulación mediante el aparato matemático. Presentamos un modelo estadístico que prueba su efectividad más allá del ámbito de aplicabilidad del modelo analítico. Ambas propuestas han sido validadas sobre móleculas reales obtenidas de bases de datos públicas, generadas por diversos proyectos del contexto de la biología. La modelización estadística se concibe en esta ocasión como un instrumento valioso en la predicción y análisis del rendimiento de aplicaciones paralelas. En el apéndice A se hace un ligero recorrido sobre algunos conceptos y términos de la biología computacional que han sido utilizados de forma natural en el capítulo 4. Con este apartado, se pretende además contextualizar el problema de la predicción de la estructura secundaria del RNA y su relevancia.

Índice general

1. Sistemas Heterogéneos: Conceptos, Objetivos y Metodología **5**

1.1. Introducción 5

1.2. El concepto de heterogeneidad en sistemas paralelos 8

1.3. Plataformas heterogéneas 9

1.4. Métricas de rendimiento en sistemas heterogéneos 11

1.5. Ejemplos de sistemas heterogéneos 14

1.5.1. Un ejemplo de plataforma heterogenea 14

1.5.2. Un ejemplo de programa heterogéneo 17

1.6. Objetivos y metodología 18

2. El Paradigma Maestro-Esclavo **21**

2.1. Resumen 21

2.2. Introducción 21

2.3. El paradigma maestro-esclavo 22

2.4. Estrategia FIFO de resolución en sistemas homogéneos 25

2.5. Estrategia FIFO de resolución en sistemas heterogéneos 26

2.6. Estrategia LIFO de resolución del problema 30

2.7. Validación de los modelos analíticos 31

2.7.1. Algoritmo Secuencial 31

2.7.2. Algoritmo Paralelo 32

2.7.3. Distribución óptima de trabajos en los sistemas homogéneos 34

2.7.4. Validación del modelo analítico FIFO en un sistema heterogéneo con dos tipos de procesadores 38

2.7.5. Validación del modelo analítico FIFO en el sistema heterogéneo con 3 tipos de procesadores 41

2.8. Un método exacto para obtener la distribución óptima 44

2.9. Metodología aplicada 48

2.10. Predicción de la distribución óptima: Producto de matrices 49

2.10.1. Análisis de sensitividad . . . 49
2.10.2. Predicción de la distribución óptima de trabajos con 2 tipos de procesadores . . . 50
2.10.3. Predicción de la distribución óptima de trabajos con 3 tipos de procesadores . . . 54
2.10.4. Herramienta de predicción de esquema y distribución óptima de trabajos . . . 61
2.11. Predicción de la distribución óptima: FFT-2D . . . 63
2.11.1. Algoritmo Secuencial . . . 64
2.11.2. Algoritmo Paralelo . . . 65
2.11.3. Caracterización de la plataforma heterogénea . . . 68
2.11.4. Distribución óptima de trabajos en los sistemas homogéneos . . . 70
2.11.5. Predicción de la distribución óptima de trabajos con 2 tipos de procesadores . . . 73
2.11.6. Predicción de la distribución óptima de trabajos con 3 tipos de procesadores . . . 76
2.11.7. Herramienta de predicción de esquema y distribución óptima de trabajos . . . 81
2.12. Conclusiones . . . 82

3. El Paradigma *Pipeline* 85

3.1. Resumen . . . 85
3.2. Introducción . . . 85
3.3. El paradigma *pipeline* . . . 87
3.4. *llp*: Una herramienta en entornos homogéneos . . . 90
3.5. *llpW*: Una herramienta en entornos heterogéneos . . . 93
3.6. Comparativa *llp* vs *llpW* . . . 95
3.6.1. Problema de la mochila unidimensional . . . 96
3.6.2. El problema de caminos mínimos . . . 101
3.6.3. El problema de la asignación de recursos . . . 106
3.6.4. Problema de la subsecuencia común más larga . . . 111
3.7. El modelo analítico . . . 115
3.7.1. El número de etapas es igual al número de procesadores . . . 116
3.7.2. El número de etapas es superior al número de procesadores: *mapping* cíclico puro . 118
3.7.3. El número de etapas es superior al número de procesadores: *mapping* cíclico por bloques . . . 119
3.8. Validación del modelo analítico . . . 125
3.8.1. El número de etapas es igual al número de procesadores . . . 126
3.8.2. El número de etapas es superior al número de procesadores: *mapping* cíclico puro . 126
3.8.3. El número de etapas es superior al número de procesadores: *mapping* cíclico por bloques . . . 128
3.9. Métodos de predicción de los parámetros óptimos . . . 132
3.9.1. Método exacto enumerativo . . . 132

3.9.2. Un método heurístico . . . 133
3.10. Predicción de los parámetros óptimos . . . 134
3.11. Conclusiones . . . 136

4. La Predicción de la Estructura Secundaria del RNA 137
4.1. Resumen . . . 137
4.2. Introducción . . . 138
4.3. El problema de la predicción de la estructura secundaria del RNA . . . 141
4.4. El paquete de Viena . . . 145
4.5. Estrategias de paralelización . . . 147
4.6. La paralelización del paquete de Viena: tamaño constante del *tile* . . . 149
4.6.1. Paralelización de los triángulos . . . 150
4.6.2. Paralelización de los cuadrados utilizando 2 procesadores . . . 151
4.6.3. Paralelización de los cuadrados utilizando p procesadores . . . 153
4.7. Resultados computacionales: tamaño constante del *tile* . . . 156
4.8. Modelo analítico . . . 182
4.8.1. Desarrollo del modelo sobre una máquina ideal . . . 182
4.8.2. Desarrollo del modelo sobre una máquina real . . . 183
4.8.3. Optimizando el tamaño del *tile* . . . 184
4.9. Validación del modelo analítico . . . 187
4.10. Modelo estadístico . . . 189
4.11. Comparativa modelo analítico vs modelo estadístico . . . 191
4.11.1. Casos reales . . . 191
4.12. La paralelización del paquete de Viena: tamaño variable del *tile* . . . 193
4.13. Resultados computacionales: tamaño variable del *tile* . . . 194
4.14. Validación de los modelos: tamaño variable de *tile* . . . 196
4.14.1. Casos reales . . . 197
4.15. Conclusiones . . . 197

A. Introducción a la biología molecular 212
A.1. Las proteínas . . . 212
A.2. Funciones de las proteínas . . . 213
A.3. Estructura de las proteínas . . . 214
A.4. El problema del plegado . . . 215
A.5. El déficit secuencia/estructura . . . 215
A.6. Ácidos nucleicos . . . 217
A.7. Síntesis de proteínas . . . 220

A.8. Virus y bacterias 221
A.9. El proyecto genoma humano 223
A.10.Importancia de la bioinformática 227

Capítulo 1

Computación Paralela en Sistemas Heterogéneos: Conceptos Básicos, Objetivos y Metodología

1.1. Introducción

Los computadores secuenciales tradicionales han sido diseñados durante muchos años basándose en el modelo introducido por John Von Newman [151, 179, 180]. Este modelo consiste en una unidad central de proceso (CPU) y una memoria. Las máquinas ejecutan una única secuencia de instrucciones, operan con una única secuencia de datos y su velocidad está limitada por dos factores: la capacidad de ejecución de instrucciones y la velocidad de intercambio entre memoria y CPU. La primera máquina que se construyó utilizando está tecnología fue la *EDVAC* [108, 121, 181], en 1947; desde entonces, este modelo se convirtió en el más utilizado en el diseño de ordenadores. Sin embargo, actualmente existe un amplio rango de aplicaciones (científicas, técnicas, médicas, etc.) que requieren procesar grandes cantidades de datos o realizar un alto número de operaciones, pero no existen procesadores con capacidad para resolverlas de forma independiente en un período de tiempo razonable [1, 102, 113, 185]. Esto indica que para ciertas aplicaciones, no es posible emplear el modelo Von Newman y es necesario buscar alternativas tecnológicas.

Una de estas alternativas consiste en emplear múltiples procesadores que cooperen para resolver un problema en una fracción del tiempo que requiere un procesador secuencial. Un problema cualquiera se divide en un número arbitrario de subproblemas y estos subproblemas pueden resolverse simultáneamente sobre diferentes procesadores. Por último, los procesadores se comunican entre sí para intercambiar y combinar los resultados parciales obtenidos. Este modelo tecnológico alternativo es denominado **computación en paralelo** y pretende conseguir que p procesadores, trabajando de forma conjunta, resuelvan un problema p veces más rápido que un único procesador; aunque esta situación ideal raras veces se alcanza en la práctica.

Durante años se han propuesto una amplia variedad de estrategias para abordar la computación en paralelo. Estos nuevos modelos difieren en la forma en la que se comunican los procesadores (memoria compartida o distribuida, red de interconexión en forma de árbol, estrella, etc.), pueden ejecutar el mismo algoritmo o no, operar de forma síncrona o asíncrona, etc. Esto significa que existen diversas formas de concebir la programación en paralelo y de construir máquinas paralelas para resolver un problema mediante esta técnica de computación. Estas máquinas presentan distintas características de red y arquitectura. Podemos encontrar sistemas formados por una única máquina con varios procesadores que colaboran en la resolución de un problema. Estos sistemas han sido denominados **multiprocesadores**

o **multicomputadores**; se caracterizan por una gran capacidad de almacenamiento y una alta velocidad de cómputo, pero resultan muy caros y difíciles de amortizar. Normalmente, se considera que estos dos sistemas se diferencian en el acceso a memoria, la mayoría de las definiciones establecen que un multiprocesador es un sistema de memoria compartida, mientras que un multicomputador trabaja con memoria distribuida [93, 141, 169]. Un multiprocesador puede a su vez clasificarse en grupos basados en la implementación de la memoria compartida [31, 164, 171]: puede existir un acceso uniforme a memoria (*UMA*), donde el tiempo necesario para acceder a un dato es independiente de su localización; o un acceso no uniforme (*NUMA*), donde la memoria compartida está físicamente distribuida entre los nodos de procesamiento.

Una alternativa de bajo coste para los sistemas paralelos la encontramos en sistemas que denominamos ***clusters***. Estos sistemas se componen de múltiples máquinas, conectadas mediante una red, que el usuario utiliza como si dispusiera de un único computador paralelo [42, 43, 44]. Los *clusters* de PCs están jugando un papel importante en la Computación de Alto Rendimiento (*High Performance Computing*). Esto es debido al bajo coste de estos sistemas en relación a su rendimiento. Nosotros hemos centrado gran parte de nuestro trabajo en la resolución de problemas sobre este tipo de recursos. Otro factor, que también ha contribuido de forma notable a la expansión de estos sistemas, se encuentra en la aparición de librerías estándar de procesamiento paralelo que han facilitado la portabilidad de programas paralelos a diferentes arquitecturas (PVM (*Parallel Virtual Machine*) [63, 78] y MPI (*Message passing interface*) [89, 134] para el paradigma de paso de mensajes y OpenMP [142] para trabajar con sistemas de memoria compartida).

Actualmente, la importancia de los *clusters* en la computación paralela queda reflejada en la inclusión de estos sistemas en la lista de los 500 sistemas más potentes (la lista del *top500* [125]). En esta lista se encuentran, a fecha de junio de 2004, 291 *clusters*. Esta cifra supone un 58 % de los 500 sistemas. La proporción es aún mayor si consideramos únicamente los 10 primeros sistemas, donde aparecen 6 *clusters*. Los dos sistemas más potentes de este tipo se encuentran en las posiciones 2 y 3 de la lista y ambos se encuentran instalados en Estados Unidos. El primero de ellos en el *Lawrence Livermore National Laboratory* y está compuesto por 4096 procesadores *Thunder* e *Intel Itanium2 Tiger4* de 1,4 GHz. El segundo está localizado en *Los Alamos National Laboratory*, compuesto por 8192 procesadores *ASCI Q - AlphaServer SC45* de 1.25 GHz.

En muchos casos, los *clusters* están compuestos por procesadores de características similares conectados mediante una única arquitectura de red. Los tiempos de cómputo para todos los procesadores son iguales, así como también lo son los tiempos de comunicación entre ellos. A estos sistemas se los conoce como **sistemas homogéneos**. Sin embargo, debido a la arquitectura inherente a este tipo de sistema, es muy probable que al incrementar el número de procesadores en el *cluster* o al sustituir algún elemento de la máquina (procesador, memoria, interfaz de red, etc.) las características de la nueva máquina no coincidan con las características de los componentes originales, dando lugar a un sistema de naturaleza heterogénea.

El concepto de **sistema heterogéneo** se aplica, en ocasiones, a sistemas compuestos por diferentes tipos de PCs y máquinas con múltiples procesadores conectados mediante redes. Debido a las diferencias entre las máquinas que forman el sistema, es probable que las velocidades de cómputo de los procesadores sean distintas y los tiempos de transferencia de datos también pueden ser diferentes en las comunicaciones entre cada par de procesadores. La naturaleza de esta red es inherentemente dinámica y depende de qué máquinas se utilicen en cada momento para resolver un problema y cuáles sean sus características (capacidades de cómputo, memoria, comunicaciones, etc.). La programación dependiente de la arquitectura de la máquina supone una dificultad adicional en este tipo de sistemas.

Otra de las desventajas que aparecen en los sistemas heterogéneos se debe al comportamiento de los tiempos de ejecución de las aplicaciones. El rendimiento conseguido con un multiprocesador o multicomputador paralelo suele ser más elevado que el obtenido en un entorno de máquinas heterogéneas; los usuarios pueden verse obligados a aceptar en algunos casos una reducción del rendimiento de sus aplicaciones a favor de una gran reducción en el coste del sistema. Una de las razones para que esta situación se produzca, es que la mayoría de los programas paralelos han sido desarrollados bajo la hipótesis de trabajar sobre

una arquitectura homogénea. Esta hipótesis también ha sido considerada de manera habitual al desarrollar modelos analíticos con los que predecir el rendimiento de un sistema paralelo [24, 57, 73, 87, 154, 178]; generalmente, esos modelos no pueden ser aplicados directamente a entornos de naturaleza heterogénea. Se hace necesario, por tanto, revisar los conceptos y modelos utilizados hasta el momento en los sistemas homogéneos para adaptarlos a entornos heterogéneos [145, 191]. En los últimos años se han publicado diversos trabajos que analizan los entornos heterogéneos y los parámetros que deben considerarse en el uso de estos sistemas: las distintas velocidades de cómputo de los procesadores que trabajan en la resolución del problema; las capacidades de memoria; la arquitectura de la red de interconexión entre las máquinas; la planificación de la distribución de los subproblemas a los procesadores; el análisis del rendimiento del sistema, etc. Algunos de los mejores trabajos realizados sobre sistemas heterogéneos en los últimos años podemos encontrarlos en [14, 15, 16, 17, 18, 37, 39, 40, 116, 118]. En [37] los autores describen diferentes tipos de heterogeneidad que pueden encontrarse en un sistema y estudian la mayoría de las situaciones que surgen en las plataformas homogéneas y heterogéneas, para cómputos regulares e irregulares. Debido a que existen diferentes tipos de heterogeneidad, algunos autores [16] consideran una simplificación del problema al tener en cuenta, únicamente, la heterogeneidad producida por la diferencia entre las velocidades de procesamiento de las estaciones de trabajo. Sin embargo, es un hecho probado que incluso la heterogeneidad en la velocidad de los procesadores puede tener un impacto significativo en la sobrecarga de las comunicaciones [14], aún cuando las capacidades de comunicación en la red sean las mismas para todos los procesadores. En [18] los autores dan una definición formal de una plataforma heterogénea, como un grafo dirigido donde cada nodo es un recurso de computación y las aristas son los enlaces de comunicación entre ellos; en [17] se pueden encontrar algoritmos (*round robin* y *multiple round robin*), que permiten calcular la distribución de trabajo en plataformas homogéneas y heterogéneas, considerando la existencia o no de latencia en el sistema; en [15] los autores realizan un estudio sobre estrategias de planificación teniendo en cuenta distintas plataformas heterogéneas en estrella o en árbol, considerando además diferentes niveles de paralelismo interno en los procesadores. En [116] se presentan modelos de programación eficientes en entornos heterogéneos comparando el rendimiento de un *cluster* Linux (14 procesadores *Intel Pentium II*), con el de una máquina *Cray-T3E* con 512 procesadores *DEC Alpha*. En [118] se puede encontrar una heurística que proporciona la distribución de la carga óptima para los procesadores organizados mediante un anillo virtual, de modo que cada procesador sólo puede comunicarse con otros dos procesadores: el predecesor y el posterior en el anillo. En [39, 40] presentan un sistema denominado *PINCO* para monitorizar la carga de las máquinas del sistema, de manera que pueden realizar una distribución dinámica de la carga. En estos trabajos los autores plantean las tres posibles distribuciones de tareas más utilizadas en los sistemas heterogéneos.

- Realizar un particionamiento regular de los datos entre los procesos y asignar más de un proceso por procesador.
- Realizar un particionamiento irregular de los datos a los procesos y asignar únicamente un proceso a cada procesador.
- Seguir un modelo maestro-esclavo, donde existe un conjunto de tareas simétricas y un proceso por procesador. Es utilizado cuando el número de procesos es significativamente menor que el número de tareas.

La ventaja de los dos primeros modelos es que ofrecen un soporte eficiente para resolver problemas, capaz de mantener la eficiencia cuando la plataforma es altamente dinámica; aunque los sistemas de carga dinámica se encuentran fuera del ámbito de este trabajo.

El resto del capítulo lo hemos estructurado de la siguiente manera. En 1.2 se introduce el concepto de heterogeneidad en sistemas paralelos. En la sección 1.3 se describen las plataformas heterogéneas y se comentan algunas de sus características. En 1.4 se detallan algunas de las métricas utilizadas para establecer el rendimiento obtenido en estos sistemas. En 1.5 presentamos dos ejemplos de sistemas heterogéneos: el *cluster* donde hemos realizado la mayoría de las ejecuciones presentadas en este trabajo (subsección 1.5.1)

y un ejemplo de programa heterogéneo (subsección 1.5.2). Por último, en 1.6 presentamos los objetivos de nuestra investigación y la metodología seguida para alcanzarlos.

1.2. El concepto de heterogeneidad en sistemas paralelos

Los elementos fundamentales que intervienen en un sistema paralelo son los procesadores, la arquitectura de red, la memoria, la forma de acceso a memoria (memoria compartida / memoria distribuida) y el tipo del problema a resolver. Cada uno de los anteriores elementos constituyen una posible fuente de heterogeneidad en el sistema.

- **Los procesadores:** La velocidad de un procesador puede definirse en función del número de operaciones por segundo, así como del tiempo que necesita el procesador para realizar accesos a memoria. Se produce heterogeneidad debida a los procesadores cuando la velocidad de las máquinas disponibles en el sistema no es la misma para todos. Aunque también se puede producir heterogeneidad cuando procesadores con las mismas características ejecutan un mismo programa a distinta velocidad, debido a la influencia de otros elementos de la arquitectura. La velocidad de un procesador no está sólo en función del número de operaciones por segundo.

- **La memoria:** Es posible jerarquizar la memoria de las máquinas realizando una clasificación por niveles (figura 1.1), donde el nivel más cercano al procesador es más rápido y reducido por razones de coste.

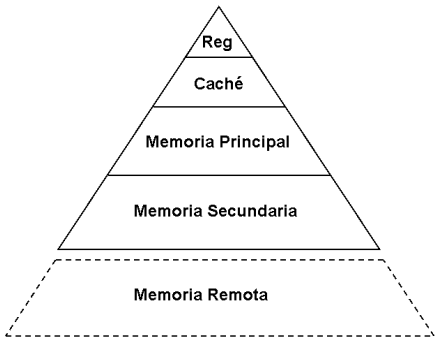

Figura 1.1: Jerarquía de memoria: el nivel más cercano al procesador es más rápido y reducido por razones de coste.

  La heterogeneidad debida a la memoria se produce cuando los procesadores disponen de jerarquías distintas de memoria o de diferentes cantidades de memoria en cada una de las categorías existentes. El tamaño de los problemas que pueden resolverse en una máquina está limitado por la cantidad de memoria disponible en esa máquina. Considerando un sistema heterogéneo, el tamaño de los problemas que pueden resolverse está limitado por la cantidad de memoria combinada que exista en el sistema. Si no se tienen en cuenta las diferencias de memoria se podrían estar infrautilizando las capacidades de ejecución del sistema.

- **La red:** En un sistema paralelo se debe considerar también el coste de las comunicaciones entre dos máquinas cualesquiera. Por simplicidad, suele asumirse que el coste de las comunicaciones punto a punto entre las máquinas de una red se puede caracterizar mediante el modelo lineal clásico: $\beta + \tau * w$, donde β es la latencia de la red, que es independiente del tamaño del mensaje enviado; τ es el tiempo de transferencia por byte, que es la inversa del ancho de banda del enlace entre los procesadores; y w representa el número de bytes que se transmiten. El coste de comunicación entre

dos procesadores modelado de esta forma incluye el tiempo que tarda un procesador en enviar datos y el que invierte el segundo procesador en recibirlos.

Estos dos parámetros son fundamentales en las comunicaciones en una red, debido a que una latencia alta puede producir un alto coste en las comunicaciones y reducir el rendimiento del sistema; mientras que el ancho de banda puede convertirse en un cuello de botella si la cantidad de información que se quiere transmitir es superior a la capacidad del canal.

Sin embargo, en redes heterogéneas donde las capacidades de comunicación pueden ser diferentes en los distintos procesadores, los parámetros β y τ deben ser obtenidos para cada par de procesadores de forma independiente. De acuerdo con esta situación, para establecer una comunicación de tamaño w entre dos procesadores p_i y p_j se necesita invertir un tiempo $\beta_{ij} + \tau_{ij} * w$, donde β_{ij} y τ_{ij} son la latencia y tiempo de transferencia por byte, respectivamente, que se obtienen entre los procesadores p_i y p_j.

- **El programa:** Los procesadores pueden ejecutar programas compuestos por códigos regulares o irregulares. Considerando estos programas, puede asumirse que existen dos tipos de iteraciones: homogéneas y heterogéneas [37]. Se conocen como **iteraciones homogéneas** aquellas en las que los procesadores realizan siempre la misma cantidad de trabajo. Las **iteraciones heterogéneas** son aquellas en las que varía la cantidad de trabajo que realizan los procesadores en cada una de las pasadas del algoritmo. Un programa que contenga iteraciones heterogéneas puede ser considerado como un **programa heterogéneo**.

  Desde esta perspectiva, se puede considerar como sistema heterogéneo, un programa heterogéneo ejecutado de forma paralela sobre procesadores homogéneos conectados mediante una red homogénea.

1.3. Plataformas heterogéneas

Una plataforma heterogénea puede ser definida como un infraestructura computacional donde están disponibles procesadores de diferentes características conectados a través de una red. Esta plataforma puede ser representada por un grafo no dirigido $HN(P, IN)$ [192], donde:

- $P = \{p_0 \dots p_{p-1}\}$ representa al conjunto de procesadores heterogéneos disponibles en el sistema (p procesadores). La capacidad de cómputo de cada uno de estos procesadores está determinada por la velocidad de CPU, la entrada/salida y velocidad de acceso a memoria. Los procesadores constituyen los nodos del grafo.
- IN es la red de interconexión entre los procesadores. Cada arista del grafo representa una conexión física entre un par de procesadores y puede caracterizarse por una latencia y un ancho de banda diferente, lo que significa que cada arista podría etiquetarse con dos valores que especifiquen el tiempo necesario para transferir un mensaje entre los dos procesadores que se encuentran en los extremos de la arista. Se asume que una arista entre dos nodos, p_i y p_j, es bidireccional y simétrica; es decir, es posible transmitir el mensaje en cualquiera de las dos direcciones posibles y el tiempo para enviar el mensaje de p_i a p_j es el mismo que el tiempo necesario para transmitirlo de p_j a p_i.

El conjunto de aristas del sistema, IN, puede llegar a ser muy complejo e incluir múltiples caminos y ciclos. El grafo puede ser completo o no (figuras 1.2-*a* y 1.2-*b* respectivamente) o estar organizado en una estructura de estrella (figura 1.2-*c*), árbol (figura 1.2-*d*), etc. Si consideramos un procesador cualquiera p_i, se denominan **vecinos** de p_i a todos aquellos procesadores interconectados a p_i a través de una arista del grafo. En el caso de disponer de un conjunto de procesadores interconectados a través de un grafo completo, todos los procesadores son vecinos entre sí, debido a que siempre existe un enlace entre cada par de nodos (p_i, p_j).

En el ejemplo de la figura 1.2-*a*, el grafo completo, puede considerarse que existe un enlace físico de comunicación independiente entre cada par de nodos, o bien, considerar la plataforma donde todos los

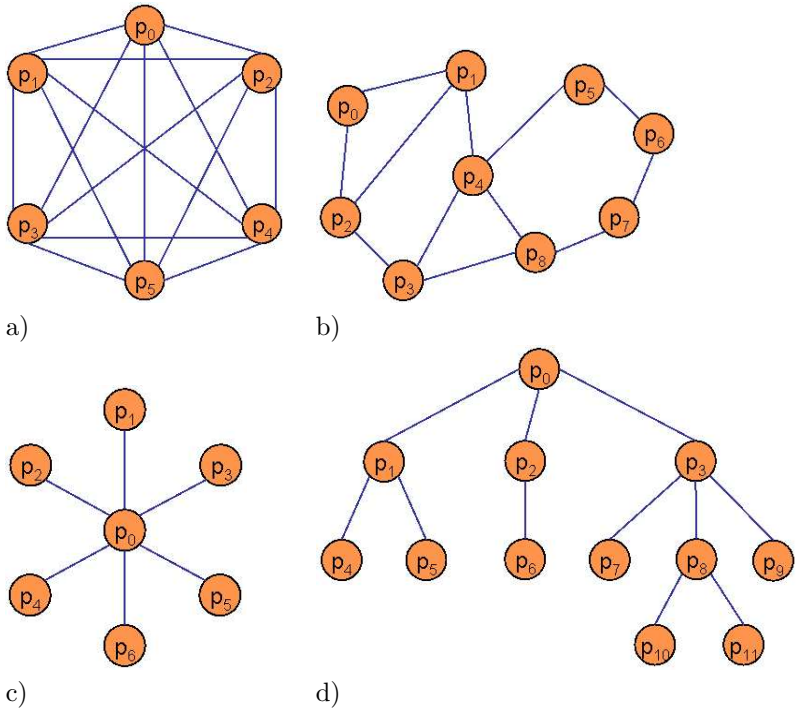

Figura 1.2: Ejemplos de plataformas heterogéneas: *a*) Grafo completo con 6 procesadores: existe un enlace directo entre cada par de procesadores heterogéneos de la plataforma. *b*) Grafo no completo con 9 procesadores. *c*) Plataforma en forma de estrella con 7 procesadores. *d*) Plataforma en forma de árbol con 12 procesadores.

procesadores están conectados a través de un único bus (figura 1.3). En este último tipo de plataforma el canal de comunicación disponible entre las máquinas suele ser exclusivo; es decir, en un instante de tiempo dado solamente un procesador puede utilizar la red de interconexión. Esto supone que si otro procesador quiere utilizar el canal en el momento en que un procesador ya está transmitiendo, debe esperar a que el canal se libere, lo que puede significar un retraso importante en las comunicaciones.

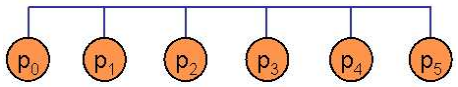

Figura 1.3: Ejemplo de plataforma heterogénea con 6 procesadores: Todos los procesadores están conectados a través de un único bus.

En todos estos ejemplos hemos considerado que un enlace entre dos procesadores cualesquiera es bidireccional; sin embargo, una variante a este tipo de plataforma puede asumir que en lugar de disponer de aristas bidireccionales, existen dos enlaces unidireccionales entre cada par de procesadores, con etiquetas que pueden ser diferentes. En este caso, la plataforma heterogénea se representa por un grafo dirigido y el tiempo de transmisión de un mensaje de p_i a p_j puede no coincidir con el tiempo empleado para transmitirlo de p_j a p_i.

Asimismo, los procesadores del sistema pueden presentar diferentes comportamientos, en función de sus capacidades para solapar cómputo y comunicaciones [15]: realizar simultáneamente múltiples recepciones de datos de todos sus vecinos, alguna operación de cómputo y múltiples operaciones de envío de información a todos sus vecinos; pueden tener limitado el número de recepciones y envíos a k vecinos ($k = 1, \ldots, p-2$); pueden realizar dos acciones en paralelo (recepción + cómputo, recepción + envío o

cómputo + envío); o pueden llevar a cabo únicamente una operación en cada instante de tiempo. Esta última opción elimina el paralelismo interno de los procesadores, ya que todas las operaciones que ejecuta un procesador las realiza de forma secuencial.

1.4. Métricas de rendimiento en sistemas heterogéneos

En esta sección se especifican algunas de las métricas empleadas para analizar el rendimiento en sistemas heterogéneos. Algunas de ellas se definen de forma natural por extensión del caso homogéneo; otras, por contra, requieren una formulación específica. En la definición de estas medidas se utiliza la siguiente notación:

- M: Es el tamaño del problema, expresado como el número de operaciones necesarias para resolverlo.
- T_{seq_i}: Es el tiempo de ejecución del algoritmo secuencial en el procesador p_i.
- T_i: Es el tiempo empleado por la ejecución paralela desde su inicio hasta la finalización del procesador p_i.

Considerando las definiciones anteriores, se denomina **tiempo de ejecución paralelo**, T_{par}, al tiempo transcurrido desde el comienzo de la ejecución paralela hasta el momento en que finaliza el último procesador [113]: $T_{par} = \text{máx}_{i=0}^{p-1} T_i$.

La **potencia computacional** de un procesador en un sistema heterogéneo puede ser definida como la cantidad de trabajo realizada en el procesador durante una unidad de tiempo [145, 192]. La potencia computacional depende de las características físicas del procesador, pero también de la tarea que se esté ejecutando. Esto significa que este valor puede variar para diferentes aplicaciones y, debido a las limitaciones del tamaño de memoria, caché y otros componentes hardware, la velocidad de cómputo puede cambiar también al modificar el tamaño del problema de la aplicación.

La potencia computacional del procesador p_i para una carga de trabajo M puede ser expresada como $CP_i = \frac{M}{T_{seq_i}}$. Desde un punto de vista práctico, la potencia computacional media, en un conjunto de procesadores, puede calcularse ejecutando una versión secuencial del algoritmo en cada uno de ellos, usando un problema de tamaño adecuado para prevenir errores debido a efectos en la memoria caché.

En los sistemas homogéneos la potencia computacional es constante para todos los procesadores $CP_i = CP$, $i = 0, \ldots, p-1$.

La **potencia computacional total** de un sistema heterogéneo, compuesto por p procesadores ($CP_t(p)$), puede definirse como la suma de la potencia computacional de todos los procesadores que componen el sistema [145]. Este parámetro refleja la cantidad de trabajo que puede realizar el sistema en una unidad de tiempo, al ejecutar un algoritmo específico. Puede expresarse como $CP_t(p) = \sum_{i=0}^{p-1} CP_i$.

En los sistemas homogéneos, al permanecer constante la potencia computacional para todos los procesadores, la potencia computacional total puede expresarse como $CP_t(p) = p * CP$.

Otra de las métricas que pueden ser utilizadas en un sistema es su nivel de **heterogeneidad**. Este parámetro ofrece un valor de la similitud o diversidad de las máquinas del sistema; por ejemplo, cuando un número pequeño de procesadores es mucho más rápido o más lento que la mayoría se considera que es un sistema con una heterogeneidad alta. Se calcula en función de la potencia computacional de sus máquinas: escalando los valores obtenidos de las potencias computacionales para los procesadores, de modo que a aquellos que sean más rápidos se les asigne una potencia computacional de 1. Denominando CP_i^E a las potencias computacionales escaladas de las máquinas, se define la heterogeneidad del sistema como $H = \frac{\sum_{i=0}^{p-1}(1-CP_i^E)}{p}$ [192]. Esta métrica varía entre 0 cuando el sistema es homogéneo (la potencia computacional es la misma para todos los procesadores) y un valor máximo próximo a 1.

Independientemente de las características de los procesadores, también es necesario considerar el **coste de las comunicaciones**. Tal como se ha descrito en la sección 1.2, un modelo habitual utiliza β_{ij} y τ_{ij} (latencia y tiempo de transferencia respectivamente entre dos procesadores de la red, p_i y p_j) para representar el coste de las comunicaciones mediante el modelo lineal clásico $\beta_{ij} + \tau_{ij} * w$.

La **aceleración** (Sp) obtenida en un sistema paralelo es también una métrica importante del rendimiento de un algoritmo en el sistema, puesto que muestra el beneficio relativo de resolver un problema en paralelo frente a su resolución en una máquina secuencial.

En sistemas homogéneos se define la aceleración como la relación entre el tiempo secuencial obtenido en cualquiera de las máquinas y el tiempo de ejecución paralelo: $Sp = \frac{T_{seq}}{T_{par}}$, donde $T_{seq} = T_{seq_i}, i = 0, \ldots, p-1$ [1, 185].

Si $Sp > 1$ entonces el sistema presenta un rendimiento superior al de las máquinas individuales del sistema. Teóricamente, la aceleración nunca puede exceder al número de procesadores p que intervienen en la resolución del problema. En muchos casos, la aceleración conseguida es mucho menor que la aceleración máxima ($Sp << p$). Esta situación puede producirse cuando no es posible descomponer el problema en un número apropiado de tareas independientes a ser ejecutadas simultáneamente; o bien, si el tiempo requerido en la fase de comunicación entre los procesadores es demasiado alto.

Sin embargo, en algunos algoritmos y para algunos casos, se puede obtener una aceleración superior al máximo ($Sp > p$). Este fenómeno se conoce como **aceleración superlineal** [20, 91, 96]. Puede aparecer debido a algoritmos secuenciales no óptimos o a características del *hardware* que ralentizan la ejecución del algoritmo secuencial (por ejemplo, el tamaño de la memoria caché). La primera de estas situaciones se produce cuando el programa paralelo implementado realiza un número inferior de operaciones, para resolver el problema, que el algoritmo secuencial. En este caso, la superlinealidad puede resolverse fácilmente si es posible diseñar un nuevo algoritmo secuencial que sea equivalente al algoritmo paralelo empleado. La segunda de las situaciones es debida al tamaño de la memoria caché de un procesador, que puede permitir resolver el subproblema asignado en una ejecución paralela, realizando pocos intercambios de información entre memoria caché y memoria principal; mientras que el tamaño del problema a resolver en el caso secuencial es muy superior y son necesarios un número excesivo de intercambios de datos entre memoria principal y caché, ralentizando su ejecución. En este caso, el problema es más difícil de resolver, es necesario modificar las características *hardware* de la máquina.

En el caso heterogéneo no todos los procesadores invierten el mismo tiempo en resolver el problema de forma secuencial, por lo que es necesario adaptar el concepto de aceleración. Diferentes autores proporcionan definiciones de la aceleración en una red heterogénea [2, 32, 47, 62, 191, 192]. La definición más extendida es la que considera la aceleración como la relación entre el tiempo necesario para completar una ejecución en el procesador de referencia de la red y el tiempo empleado para realizar la misma ejecución sobre el sistema heterogéneo. Es habitual considerar como procesador de referencia el procesador más rápido del sistema; en este caso, la definición de la aceleración se convierte en $Sp = \frac{\text{mín}_{i=0}^{p} T_{seq_i}}{T_{par}}$. Esta definición es la que hemos empleado a lo largo de la memoria.

En estos sistemas la acelaración máxima que se puede alcanzar es menor que el número de procesadores que intervienen en la ejecución, puesto que no todas las máquinas disponen de las mismas características. En [47] los autores definen la **potencia relativa** del sistema como el número de procesadores equivalentes al procesador considerado como referencia, que deberían componer un sistema homogéneo, para que los dos sistemas tuvieran la misma potencia de cómputo total. Para calcular este valor es necesario conocer el número de máquinas de cada tipo disponibles en el sistema y las relaciones entre sus velocidades. Considerando como procesador de referencia el correspondiente a las máquinas más rápidas, utilizamos las potencias computacionales escaladas para obtener la heterogeneidad del sistema. Suponiendo que tenemos un sistema con dos tipos de máquinas, la aceleración máxima que podemos obtener sería: *Sp_max = número de máquinas rápidas + número de máquinas lentas * potencia computacional escalada de las máquinas lentas*. Considerando un sistema heterogéneo general, donde existen m tipos de máquinas diferentes, la aceleración máxima viene dada por la siguiente ecuación: $Sp\_max = n_R + \sum_{i=1}^{m-1} n_i * CP_i^E$;

donde n_R es el número de procesadores rápidos (tipo de máquina 0) y se suma el número de procesadores de cada uno de los tipos restantes ($i = 1, \ldots, m-1$) por su potencia computacional escalada. Un ejemplo de aplicación de esta definición lo encontramos en [88], aunque en este caso los autores utilizan como máquina de referencia a los procesadores más lentos.

La eficiencia y la escalabilidad son también dos conceptos fundamentales para definir el rendimiento de los sistemas paralelos, particularmente en el caso de los *clusters*.

La **eficiencia** es una medida del porcentaje de tiempo por máquina que es empleado para resolver un cómputo paralelo. En los sistemas homogéneos la eficiencia se define como la aceleración dividida entre el número de procesadores Sp/p y se han desarrollado numerosos trabajos, tanto teóricos como prácticos, para caracterizar la eficiencia en estos sistemas [1, 58, 68, 87, 115, 185]. En un sistema ideal, el valor de la eficiencia es 1, pero en sistemas reales varía entre 0 y 1.

La **escalabilidad** de un sistema homogéneo es un término impreciso, puede definirse como la capacidad de un sistema paralelo para mejorar su rendimiento cuando aumenta el tamaño del problema y el número de procesadores. Se basa en la selección de una métrica empleada para caracterizar el comportamiento del sistema. Las métricas más utilizadas han sido la aceleración, la eficiencia, la velocidad media y la latencia. El grado de escalabilidad de un sistema se calcula mediante la ecuación: *crecimiento del problema / crecimiento del sistema* [190, 192].

Sin embargo, como sucede con las restantes medidas de rendimiento vistas en esta sección, no es posible aplicar estos conceptos del ámbito homogéneo directamente al caso heterogéneo, debido a las diferencias de características y rendimiento de los procesadores que conforman el sistema. Para poder establecer definiciones que se adapten a los sistemas heterogéneos partimos del concepto anterior de potencia computacional total y concluimos que su valor en un sistema depende tanto del número de procesadores que componen el sistema como de la potencia computacional de cada uno de ellos. Esto significa que dos sistemas con el mismo número de procesadores no necesariamente deben tener la misma potencia computacional total y, por lo tanto, es posible incrementar su valor aumentando el número de procesadores (**escalabilidad física**) o la potencia de alguno de ellos (**escalabilidad de potencia**) [145, 192].

Como consecuencia de este hecho no es posible definir la eficiencia de un sistema heterogéneo únicamente como una función del número de procesadores, como se hace en el caso de los sistemas homogéneos, debido a que el rendimiento del sistema depende de cuáles son las máquinas utilizadas en cada momento. La eficiencia en un sistema heterogéneo puede definirse como la relación entre el menor tiempo de ejecución necesario para resolver un problema específico en el sistema y el tiempo real obtenido durante la ejecución del algoritmo: *EF = Tiempo de ejecución óptimo / Tiempo de ejecución real* [145].

Formalmente, en [145] se establece la siguiente definición de escalabilidad: Dado un sistema paralelo heterogéneo HN con p procesadores, con CP_t para un trabajo M, y un sistema HN', con p' procesadores y con CP'_t para un trabajo M', con $CP'_t > CP_t$; se puede decir que HN es un sistema escalable si, cuando el sistema es ampliado desde HN a HN', es posible seleccionar un problema M' tal que la eficiencia de HN y HN' permanezcan constantes.

A efectos prácticos consideraremos que el sistema es escalable cuando sea posible incrementar la eficiencia del sistema aumentando la potencia de cómputo a través de alguno de los dos elementos definidos.

Por último, el **número de procesadores** utilizados para resolver un problema también es una medida a tener en cuenta en el caso homogéneo. El número total de procesadores disponibles en un entorno puede ser superior al número óptimo necesario para ejecutar un algoritmo, lo que puede producir un uso ineficiente de la arquitectura paralela. Esto significa que hay que seleccionar un subconjunto de procesadores que minimice el tiempo de ejecución del problema. En el caso heterogéneo, es necesario calcular el número óptimo de procesadores de cada uno de los tipos disponibles de máquinas, para minimizar el tiempo de ejecución.

1.5. Ejemplos de sistemas heterogéneos

1.5.1. Un ejemplo de plataforma heterogenea

Consideramos la plataforma heterogénea donde hemos realizado las pruebas experimentales de los capítulos 2 y 3, como el grafo no dirigido $HN(P, IN)$ (sección 1.3), donde el conjunto de procesadores disponibles en el sistema, P, está formado por los siguientes grupos de máquinas:

- El primer grupo está compuesto por una máquina de memoria compartida con cuatro procesadores Intel (R) Xeon^{TM} de 1.40 GHz con 3Gb de memoria. Los procesadores que forman esta máquina se denominarán a lo largo de toda la memoria como máquinas rápidas (R).
- El segundo grupo de máquinas lo forman cuatro PCs AMD Duron^{TM} de 800 MHz con 256 MB de memoria. Estas máquinas se denominarán máquinas intermedias (I).
- El último grupo, las máquinas lentas (L), está formado por seis PCs AMD-K6^{TM} de 501 MHz con 256 MB de memoria.

Todas las máquinas de estos tres grupos trabajan con el sistema operativo Debian Linux y están conectadas mediante un *switch Fast Ethernet* a 100 Mbit/s en un esquema de bus (figura 1.3) formando la red de inteconexión (IN). En esta plataforma vamos a considerar la heterogeneidad debida a diferencias en las velocidades de cómputo de los procesadores y a las diferencias en las velocidades de comunicación (sección 1.2). La heterogeneidad debida a las diferencias en las capacidades de memoria no la hemos tenido en cuenta al realizar este trabajo.

A continuación, caracterizamos esta plataforma (HN) mediante algunas de las métricas de rendimiento vistas en la sección 1.4. La primera métrica calculada en HN es la potencia de cómputo de cada uno de los grupos de procesadores. Definíamos esta métrica como la cantidad de trabajo realizado en el procesador durante una unidad de tiempo. En la práctica utilizamos como medida el tiempo que tarda un procesador p_i en computar una cantidad de trabajo. Podemos determinar este valor ejecutando el algoritmo secuencial en los tres tipos de máquinas que componen HN, y posteriormente usar la relación entre estos valores para establecer las cantidades de trabajo que se asignan a cada procesador. Esta medida depende del tipo de trabajo a realizar, por lo que es necesario calcularla para cada problema de forma independiente; en los capítulos siguientes se presentan los resultados obtenidos para cada caso. La tabla 1.1 muestra un ejemplo de los resultados obtenidos al resolver el problema del producto de matrices sobre los tres tipos de máquinas. Los valores que aparecen en la tabla son las relaciones obtenidas entre los tiempos secuenciales de las máquinas rápidas y lentas (columna L / R), entre los tiempo secuenciales de las máquinas rápidas e intermedias (columna I / R) y entre intermedias y lentas (columna I / L). Este problema lo resolvemos de forma paralela mediante un algoritmo maestro-esclavo en el capítulo 2.

Tabla 1.1: Relaciones obtenidas entre los tiempos secuenciales para los tres tipos de máquinas que componen HN con diferentes tamaños de problemas: lentas / rápidas (L / R), intermedias / rápidas (I / R) y lentas / intermedias (L / I).

Filas	Columnas	L / R	I / R	L / I
700	700	4.89	1.63	3.00
1000	1000	5.03	1.67	3.01
1500	1500	4.85	2.99	1.62
2000	2000	3.94	3.70	1.06

Utilizando el mismo problema también podemos calcular el valor de otras dos métricas definidas en la sección 1.4: la heterogeneidad y la aceleración máxima que puede alcanzarse. Para calcular el valor de la

heterogenidad considerábamos en la sección 1.4 que asignábamos un 1 a la potencia computacional de la máquina más rápida y escalábamos los valores para el resto de las máquinas. A continuación aplicábamos la fórmula: $H = \frac{\sum_{i=0}^{p-1}(1-CP_i^E)}{p}$. La tabla 1.2 contiene los valores escalados de la potencia computacional para los tres tipos de máquinas que componen el *cluster* (columnas CP_R^E para las máquinas rápidas, CP_I^E para las intermedias y CP_L^E para las lentas) y el valor total de la heterogeneidad para el problema del producto de matrices, con los cuatro tamaños diferentes de problemas que presentamos en la tabla 1.1, cuando ejecutamos los problemas sobre el *cluster* completo de 14 máquinas.

Tabla 1.2: Potencia computacional escalada para los tres tipos de máquinas del cluster (rápidas, intermedias y lentas) y valor de la heterogeneidad al resolver cuatro problemas de distintos tamaños del producto de matrices.

Filas	Columnas	CP_R^E	CP_I^E	CP_L^E	Heterogeneidad
700	700	1	0.61	0.20	0.45
1000	1000	1	0.60	0.20	0.46
1500	1500	1	0.33	0.21	0.53
2000	2000	1	0.27	0.25	0.53

Como se observa en la tabla 1.2, disponemos de un sistema bastante heterogéneo, el valor de la heterogeneidad se mantiene alrededor de 0.5. La heterogeneidad se incrementa al aumentar el tamaño del problema, puesto que la diferencia entre las velocidades de las máquinas rápidas e intermedias también aumenta al resolver los problemas mayores (tabla 1.1).

La tercera métrica que vamos a aplicar utilizando este problema es la aceleración máxima del sistema. Adaptamos la fórmula definida en la sección 1.4 a tres tipos de máquinas y obtenemos la siguiente ecuación: $Sp_{max} = n_R + n_I * CP_I^E + n_L * CP_L^E$, donde n_R, n_I y n_L son el número de procesadores rápidos, intermedios y lentos respectivamente; CP_I^E y CP_L^E son los valores de las potencias de cómputo escaladas que obtuvimos en el cálculo de la heterogeneidad (tabla 1.2). La figura 1.4 muestra cuáles son las aceleraciones máximas posibles para el sistema, cuando resolvemos los problemas utilizando los 14 procesadores disponibles.

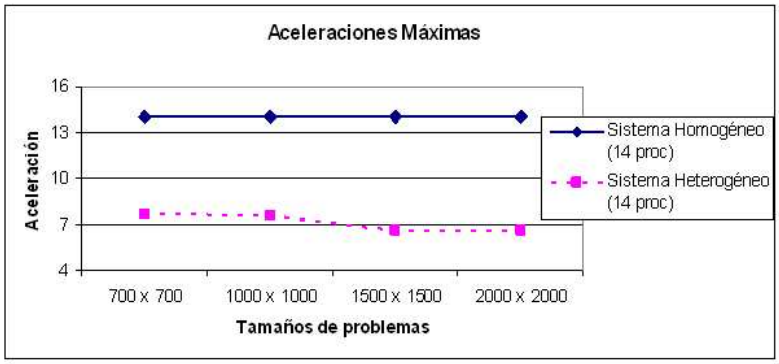

Figura 1.4: Aceleraciones máximas posibles para los cuatro problemas del producto de matrices utilizando las 14 máquinas disponibles en el sistema, comparadas con la aceleración máxima en un sistema homogéneo del mismo tamaño.

Como se observa en la figura 1.4, las aceleraciones máximas que podemos conseguir en este sistema están muy lejos de la $Sp_{max} = 14$ que tendríamos si ejecutásemos los problemas sobre un sistema homogéneo con el mismo número de máquinas que nuestro sistema heterogéneo.

Otra de las medidas importantes que debemos considerar en nuestra plataforma es el coste de las comunicaciones entre las máquinas. Para modelizar estas comunicaciones se han medido los parámetros

Tabla 1.3: Latencia y tiempo de transferencia por byte en nuestra plataforma para todas las combinaciones de dos procesadores.

Tipo Emisor	Tipo Receptor	β	τ
Rápido	Rápido	6.52452E-06	8.7862E-09
Rápido	Intermedio	0.000129947	9.45607E-08
Rápido	Lento	0.000218896	1.0238E-07
Intermedio	Intermedio	0.000139594	9.93899E-08
Intermedio	Lento	0.000236513	1.07065E-07
Lento	Lento	0.000306578	1.15404E-07

β y τ mediante experimentos de tipo *ping-pong*. Se han utilizado las funciones *MPI_Send* y *MPI_Receive* con bloqueo definidas en el estándar de la librería de paso de mensajes MPI [89, 134] y se realiza un ajuste lineal de los tiempos de envío, como función del tamaño del mensaje, para combinaciones de pares de máquinas lentas (L), intermedias (I) y rápidas (R) (tabla 1.3). Como los parámetros se han medido utilizando un *ping-pong*, los valores obtenidos por las combinaciones rápida-lenta y lenta-rápida son iguales, así como los obtenidos para las combinaciones rápida-intermedia, intermedia-rápida y lenta-intermedia, intermedia-lenta. La figura 1.5 muestra el efecto de los parámetros β y τ.

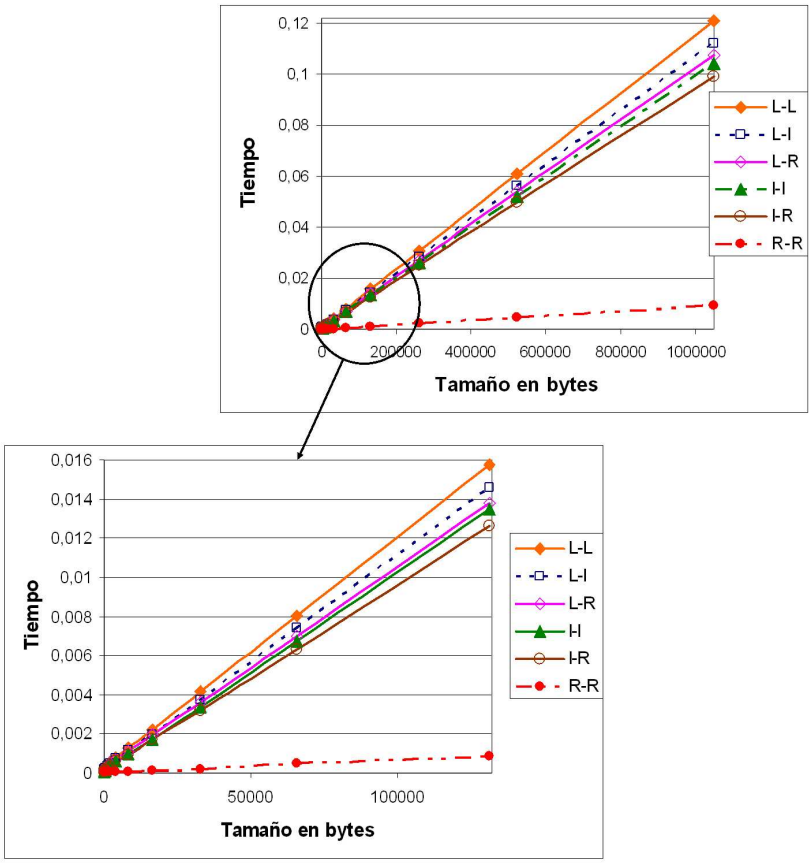

Figura 1.5: Modelo lineal clásico para el tiempo de transferencia usando diferentes tipos de procesadores. R = Procesador Rápido, I = Procesador Intermedio y L = Procesador Lento.

1.5.2. Un ejemplo de programa heterogéneo

En el capítulo 4, sin embargo, presentamos un ejemplo de heterogeneidad debida al programa. El problema que hemos utilizado como ejemplo en este capítulo es la predicción de la estructura secundaria del *RNA*. Este problema consiste en utilizar la secuencia de una molécula de *RNA* para obtener su estructura secundaria. La secuencia es considerada como una cadena de n caracteres: $R = r_1 r_2 \dots r_n, r_i \in \{A, C, G, U\}$, donde cada caracter r_i se denomina **base** y la estructura secundaria se caracteriza por una colección S de pares (r_i, r_j) de bases. Cada par de bases que se une libera energía y la estructura secundaria se obtiene minimizando la energía liberada empleando la siguiente aproximación de programación dinámica [103]:

$$E(S_{i,j}) = \text{mín} \begin{cases} E(S_{(i+1,j)}) \\ E(S_{(i,j-1)}) \\ \text{mín}\{E(S_{(i,k)}) + E(S_{(k+1,j)})\} \quad para \;\; i < k < j \\ E(L_{i,j}) \end{cases}$$

sobre un dominio del problema definido como una matriz $n \times n$ triangular: $T = \{(i,j) | 0 \leq i \leq j \leq n\}$; y donde $L_{i,j}$ especifica la situación de las bases incluidas entre r_i y r_j cuando $(r_i, r_j) \in S$, incluyendo la existencia de bucles en la estructura secundaria.

De acuerdo con esta recurrencia, calcular un elemento (i, j) implica la necesidad de disponer de los valores de los elementos anteriores (figura 1.6). Como se observa en la figura, el número de elementos que es necesario calcular previamente, depende de los valores de i y j.

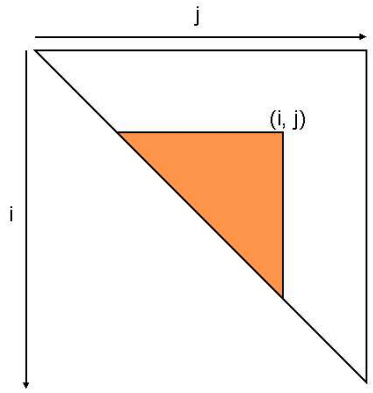

Figura 1.6: Dominio del problema. La parte coloreada representa las dependencias de un elemento (i, j). Son los elementos que deben estar calculados previamente para obtener el valor de (i, j).

En el capítulo 4 resolvemos este problema sobre un multiprocesador, con procesadores homogéneos y en una red de comunicaciones homogénea. En la paralelización de este algoritmo podemos optar por distribuir la misma cantidad de elementos durante toda su ejecución, o bien modificar esa distribución de datos en cada una de las iteraciones del algoritmo. En este caso nos encontramos con un sistema heterogéneo según la definición de la sección 1.2, puesto que el tiempo de ejecución para calcular cada uno de los elementos no es constante, varía dependiendo del número de valores que es necesario tener calculados previamente.

1.6. Objetivos y metodología

Los factores que determinan el rendimiento de un programa paralelo son numerosos (velocidad de CPU, arquitectura del sistema, entorno de la programación paralela, algoritmos, paradigmas de programación, etc.). En este trabajo nos proponemos como objetivo **predecir el rendimiento de un sistema heterogéneo** para diferentes paradigmas de programación (en el capítulo 2 trabajamos con el paradigma maestro-esclavo y en el capítulo 3 con los algoritmos *pipelines*). Pretendemos obtener los parámetros necesarios (número de procesadores y distribución de trabajo) para una ejecución eficiente de los programas sobre sistemas heterogéneos. Sin embargo, no solamente deseamos disponer de modelos analíticos de rendimiento, sino que también pretendemos **desarrollar herramientas** que faciliten el diseño y la programación paralela y su **aplicación a diversos problemas**: problemas de optimización mediante programación dinámica; operaciones matriciales, como el producto de matrices o la *FFT* bidimensional; o problemas que surgen en el ámbito de la biología, como la predicción de la estructura secundaria del *RNA*.

En general, podemos afirmar que las dos grandes aproximaciones a la predicción del rendimiento de sistemas paralelos son el modelado analítico y el *profiling* (simulación/experimentación) de rendimiento. Sin embargo, en los últimos años podemos ver como muchos autores aplican con éxito técnicas mixtas resultantes de la combinación de ambas. Asimismo, en algunos trabajos se considera como una alternativa viable la predicción del rendimiento mediante modelos estadísticos.

El modelado analítico utiliza modelos de la arquitectura y de los algoritmos para predecir el tiempo de ejecución de un programa. El proceso de modelado consiste en el estudio del diseño del algoritmo, la extracción de una serie de parámetros (tanto de la arquitectura de red como del algoritmo a resolver) y una formulación matemática que los relaciona. De ella se obtiene una aproximación al valor real que se pretende estimar. El grado de aproximación de los resultados a la realidad depende del número de suposiciones y simplificaciones realizadas en la definición del modelo. Cuantas más situaciones contemple el modelo, mejores resultados se podrán obtener, pero más complejo será su uso. Las ventajas de los modelos estriban en que definen un proceso estructurado que permite entender los problemas de rendimiento que pueden surgir y que facilita la búsqueda de un diseño óptimo del programa. Los principales inconvenientes están relacionados con el hecho de que, en ocasiones, no son lo suficientemente precisos y a medida que crece el sistema aumenta la complejidad del modelo. Históricamente se han propuesto muchos modelos analíticos (*PRAM* [73], *LogP* [57, 58], *BSP* [178], *BSPWB* [154], etc.) asumiendo distintos paradigmas de cómputo / comunicación que han sido aplicados con distintos grados de éxito en el análisis de algoritmos y arquitecturas. Para el caso heterogéneo nos encontramos con el modelo *HBSP* (*Heterogeneous BSP*) [186] que es una actualización del modelo clásico *BSP* para que incluya parámetros que reflejen las distintas velocidades de las máquinas que componen el sistema. Sin embargo, en la mayoría de los casos que aparecen en la literatura, se hacen análisis *ad-hoc* de los problemas o paradigmas concretos, adaptando los modelos clásicos a las arquitecturas específicas. En [38] los autores modelan el rendimiento de un sistema heterogéneo, siguiendo esta última estrategia, pero consideran únicamente sistemas con las mismas velocidades de comunicaciones entre cada par de máquinas.

Por otro lado, el *profiling* puede ayudar a detectar cuellos de botella en el rendimiento de un sistema paralelo concreto, corregirlos e identificar y prevenir los problemas de rendimiento futuros. Las métricas de rendimiento requieren *hardware* y *software* de propósito específico. Los estudios de estas medidas se realizan a menudo para determinar parámetros del sistema, que se usarán posteriormente para estudiar o validar un modelo. En [71] se presenta una herramienta de predicción que forma parte del *Viena Fortran Compilation System* cuyo objetivo es mejorar el rendimiento de diferentes versiones del programa paralelo sin ejecutarlas. También se han desarrollado muchos proyectos para crear ficheros de trazas de eventos con información temporal asociada, para examinarlos posteriormente, interpretándolos gráficamente en una estación de trabajo. La capacidad para generar ficheros de trazas de forma automática es un componente importante en muchas de estas herramientas como *MPICL* [189], *Dimemas* [80, 114], *Kpi* [69], etc. Los programas de presentación de ficheros de trazas como *ParaGraph* [94, 95], *Vampir* [25, 137], *Paraver* [115], etc. ayudan a analizar la información contenida en ellos.

En el ámbito de las herramientas mixtas, habitualmente se emplean modelos analíticos soportados o combinados con herramientas de *profiling*. Una de estas herramientas es *CALL* [86, 156], una herramienta de uso muy sencillo, simplemente se introducen *pragmas* en el código para definir la sección que se va a ejecutar en paralelo, la librería de comunicaciones a utilizar (por ejemplo OpenMP [142] o *MPI* [89, 134]), implementar bucles, establecer sincronizaciones, etc. Otra de estas herramientas es *POETRIES* [28, 29, 30]: está basada en esqueletos, y contiene un módulo, *MATE* [128], que permite utilizar el modelo analítico para mejorar de forma dinámica, en tiempo de ejecución, el rendimiento de una aplicación.

Por último, mencionar también el modelado estadístico como un instrumento de predicción de rendimiento. Se pretende obtener un modelo mediante una reducción en los datos originales, estableciendo una relación entre las variables del sistema [41, 127, 159]. Se intenta establecer también la medida de hasta que punto esta nueva información, o modelo reducido, es válido y representa correctamente al original. La idea de estos modelos se basa en estudiar el problema para una pequeña parte de los posibles valores que pueden tomar los parámetros y predecir cuál será el comportamiento del sistema para el rango completo de valores.

Como metodología de trabajo en nuestra investigación, optamos por emplear modelos analíticos para predecir el rendimiento de un sistema heterogéneo, siguiendo la metodología representada en la figura 1.7: desarrollamos modelos analíticos basados en los parámetros obtenidos para el algoritmo y la arquitectura de la plataforma que predicen los valores adecuados para una ejecución óptima. En la figura 1.7 representamos por un círculo los pasos de la metodología que hay que realizar manualmente y por un cuadrado aquellos pasos que pueden automatizarse.

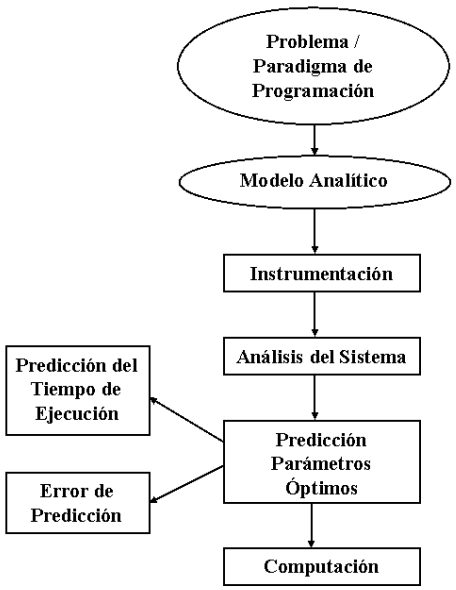

Figura 1.7: Metodología general aplicada. Representamos por un círculo los pasos de la metodología que hay que realizar manualmente y por un cuadrado aquellos pasos que pueden automatizarse.

Inicialmente, se define el problema o paradigma de programación que vamos a estudiar y desarrollamos un modelo analítico para predecir el tiempo invertido en la resolución de un problema. Estos dos pasos hay que realizarlos de forma manual para cada uno de los paradigmas de programación que queramos modelar. A continuación, generamos simuladores del modelo analítico. Estos simuladores nos permiten calcular, de forma automática, el tiempo que tardará en resolverse un problema cuando se utiliza una distribución específica de tareas entre los procesadores. El siguiente paso a llevar a cabo es el análisis

de nuestro sistema, obteniendo los valores que caracterizan nuestra plataforma (latencia y ancho de banda) y el algoritmo (velocidad de cómputo para el problema de cada tipo de máquina disponible). Este paso es posible automatizarlo para conseguir fácilmente todos los valores. Una vez calculados los datos anteriores, aplicamos métodos exactos o heurísticos para calcular el tiempo mínimo necesario para resolver el problema, así como los valores óptimos de los parámetros (número de procesadores y distribución de tareas entre los procesadores). Por último, ejecutamos el programa que resuelve el problema utilizando la configuración de valores obtenidos en el paso anterior.

También hemos desarrollado herramientas que faciliten el diseño y la implementación de programas paralelos para el caso del paradigma *pipeline*. Estas herramientas permiten escribir un código paralelo de forma que el paralelismo sea transparente al programador. Además, es posible introducir en estas herramientas el modelo analítico definido para el paradigma, para que calcule, de forma automática, los valores óptimos de los parámetros con los que debe resolverse un problema y realice su ejecución.

En trabajos publicados recientemente ([7, 9, 10]) demostramos que la modelización analítica puede ser aplicada, satisfactoriamente, para predecir el tiempo de ejecución de aplicaciones paralelas en sistemas heterogéneos empleando los paradigmas de programación maestro-esclavo y *pipelines*. Esos modelos no solamente generan una distribución óptima de tareas sino que, en algunos casos, nos permiten también determinar el conjunto más apropiado de procesadores que debemos utilizar para resolver el problema de forma paralela. Además de su aplicación a paradigmas de programación genéricos como los mencionados anteriormente, la metodología es aplicable a problemas específicos. En particular, en el capítulo 4 aplicamos esta metodología, comparando los resultados de los modelos analíticos con los de modelos estadísticos desarrollados.

Capítulo 2

El Paradigma Maestro-Esclavo en Sistemas Hetererogéneos

2.1. Resumen

En este capítulo estudiamos los algoritmos maestro-esclavo y proponemos modelos analíticos para predecir su rendimiento sobre sistemas heterogéneos. Estos modelos en particular han demostrado su eficacia en diferentes esquemas de resolución, tal y como hemos comprobado haciendo uso de casos de estudio sobre un cluster heterogéneo de PCs (sección 1.5.1). Mediante algoritmos maestro-esclavo hemos abordado productos de matrices y Transformadas Rápidas de Fourier bidimensionales. Los resultados obtenidos, en la resolución de estos dos problemas, demuestran la precisión de la aproximación numérica y los beneficios de nuestra propuesta.

Los modelos desarrollados constituyen un instrumento con el que sintonizar los algoritmos paralelos. Consideramos dos importantes elementos a sintonizar en una computación en paralelo: la asignación de datos a las unidades de cómputo y el número óptimo de procesadores a utilizar. La cuestión de decidir entre varias estrategias de paralelización prometedoras se resuelve también aquí, de manera natural, utilizando las capacidades predictivas de estos modelos en una herramienta integrada.

2.2. Introducción

El problema maestro-esclavo ha sido extensamente estudiado y modelado en arquitecturas homogéneas [81, 92, 153, 176] y ya se han propuesto numerosas técnicas de planificación para cómputos irregulares y para sistemas heterogéneos [14, 16, 66, 160]. En [66] ha sido presentado un modelo analítico y una estrategia para encontrar una distribución óptima de las tareas sobre un paradigma maestro-esclavo que consiste en resolver un sistema de $p-1$ ecuaciones; sin embargo, consideramos que la solución presentada tiene la desventaja de no analizar las colisiones de comunicaciones en una plataforma heterogénea. De esta estrategia hablaremos con más detalle en la sección 2.5. En [16] los autores proporcionan algoritmos eficientes en el contexto de la computación heterogénea para el paradigma maestro-esclavo. Formulan el problema considerando diferentes variantes, pero consideran la heterogenidad debida únicamente a la velocidad de procesamiento de las máquinas. En [160] se presenta un modelo estructural para predecir el rendimiento de los algoritmos maestro-esclavo sobre sistemas heterogéneos. El modelo es bastante general y cubre la mayoría de las situaciones que aparecen en los problemas prácticos. Este modelo estructural puede considerarse como un esqueleto donde pueden ser instanciados los modelos para aplicaciones maestro-esclavo con diferentes implementaciones o entornos. Un inconveniente de esta aproximación es

que la instanciación del esqueleto a un caso particular no es una tarea trivial. En su propuesta, no consideran la posibilidad de encontrar la distribución óptima de trabajo entre los procesadores.

Como se observa en estas referencias y ha sido comentado en el capítulo 1, algunos autores [16] consideran la heterogeneidad debida exclusivamente a las diferentes velocidades de procesamiento de las máquinas. Sin embargo, se ha demostrado que las diferencias en las comunicaciones también son significativas [14], aún cuando las capacidades de comunicación de los procesadores sean las mismas. Por este motivo, nosotros consideramos la heterogeneidad en el sistema debida a la velocidad de cómputo de los procesadores y a las diferencias en el tiempo de transmisión entre cada par de máquinas, lo que nos proporciona una visión más general del sistema.

En este capítulo presentamos dos modelos analíticos que hemos desarrollado para predecir el tiempo de ejecución de los algoritmos maestro-esclavo sobre sistemas heterogéneos [6, 7, 10]. Los modelos consideran heterogeneidad debida a los procesadores y a la red (sección 1.2) e incluyen algunas de las variantes estudiadas en [16, 160]; en particular, consideramos las estrategias denominadas FIFO y LIFO. Haciendo uso de estos modelos hemos desarrollado una estrategia para obtener la asignación óptima de tareas a los procesadores que mejora la aproximación clásica, que asigna las tareas de forma proporcional a la velocidad de procesamiento de cada máquina. Los resultados computacionales demuestran la eficiencia de este esquema y la eficacia de sus predicciones.

La mayoría de los trabajos mencionados anteriormente resuelven el problema de la distribución del trabajo sobre un conjunto fijo de procesadores. Un trabajo donde sí pueden determinar el número óptimo de procesadores se encuentra en [176]. Los autores desarrollan dos esquemas de particionado óptimo de trabajo para el procesamiento de secuencias de video en paralelo en sistemas de tiempo real; sin embargo, consideran únicamente sistemas homogéneos. Nosotros también consideramos la resolución de este problema pero bajo un entorno heterogéneo, en el que la propia naturaleza de la arquitectura introduce un grado de complejidad adicional. La formulación como problema de optimización de recursos (RAP) [104] nos ha proporcionado el contexto adecuado para obtener soluciones eficientes.

El resto del capítulo lo hemos estructurado en las siguientes secciones: en la sección 2.3 definimos el paradigma maestro-esclavo y las posibles estrategias para resolver un problema; en las secciones 2.4, 2.5 y 2.6 presentamos los modelos analíticos que hemos desarrollado para resolver este tipo de problema, para dos de los esquemas de resolución (FIFO y LIFO), tanto en entornos homogéneos como heterogéneos y en la sección 2.7 se encuentran los resultados experimentales que hemos obtenido al validar nuestros modelos. Un método exacto que nos permite obtener la mejor distribución de trabajo para realizar la ejecución en un tiempo mínimo ha sido presentado en la sección 2.8; y en la sección 2.9 describimos la metodología que hemos empleado. En las secciones 2.10 y 2.11 se encuentran los resultados obtenidos al ejecutar con nuestra estrategia, comparando estos resultados con los obtenidos con otras distribuciones y, por último, en la sección 2.12 se encuentran las conclusiones de este trabajo.

2.3. El paradigma maestro-esclavo

En el paradigma **maestro-esclavo** se pretende resolver un problema de tamaño m, asumiendo que el trabajo puede ser dividido en un conjunto de p tareas independientes: $tarea_1, \ldots, tarea_p$, de tamaños arbitrarios $m_1, \ldots, m_p$, donde $\sum_{i=1}^{p} m_i = m$. Al no existir dependencias entre los datos, cada una de estas tareas pueden ser procesadas en paralelo por p procesadores, denominados **esclavos** $(p_1, \ldots, p_p)$.

Se considera que inicialmente la carga total de trabajo m reside en un único procesador denominado **maestro** (p_0). El procesador maestro, de acuerdo con alguna política de asignación, divide el trabajo m en p partes y distribuye el trabajo a los demás procesadores, de manera que la $tarea_i$ le corresponde al procesador p_i. Una vez recibida su parte de trabajo cada procesador comienza inmediatamente a computarlo obteniendo un $resultado_i$. Finalmente, todos los esclavos devuelven su resultado al maestro. El orden de recepción de los resultados en el maestro depende del esquema utilizado en la resolución del problema.

Asumimos en nuestro trabajo que todos los procesadores están conectados a través de un canal de comunicación, normalmente un bus (figura 2.1). A este canal sólo es posible acceder de forma exclusiva; es decir, en un instante de tiempo dado, solamente un esclavo puede comunicar con el maestro para recibir sus datos o enviarle el resultado.

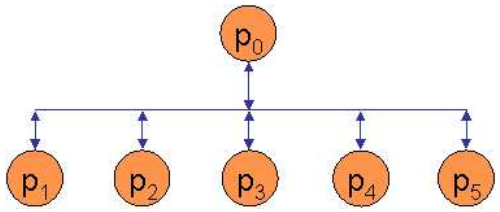

Figura 2.1: Arquitectura maestro-esclavo. p_0 es el procesador maestro mientras que los procesadores $p_1, \ldots, p_p$ se denominan esclavos. Todos los procesadores están conectados a través de un canal de comunicación exclusivo.

Para resolver este problema existen varios esquemas de ejecución:

1. Estrategia **FIFO**: Los esclavos finalizan su ejecución en el mismo orden en que la comenzaron; es decir, el maestro recibe los resultados de los esclavos en el mismo orden en el que realizó los envíos de las tareas (figura 2.2-a).

2. Estrategia **LIFO**: Los esclavos finalizan su ejecución en el orden inverso al de comienzo; es decir, el orden en el que el maestro recibe los resultados de los esclavos está invertido respecto al orden de envío de los trabajos (figura 2.2-b).

3. El maestro recibe los resultados de los esclavos a medida que estos están disponibles, sin que exista un orden preestablecido. La figura 2.2-c muestra un caso particular de esta estrategia cuando sólo existe una etapa; es decir, los esclavos realizan el proceso de recibir su tarea, computarla y enviar al maestro su resultado una única vez. La figura 2.2-d muestra un caso particular de esta estrategia cuando la resolución del problema requiere más de una etapa. Para cada esclavo se muestra con una línea vertical gruesa el inicio de la segunda etapa.

En la figura 2.2 se muestran los diagramas de tiempo para las tres estrategias de resolución de un problema maestro-esclavo. En estos diagramas cada una de las barras horizontales representa la ejecución de un esclavo (p_i) y para cada uno de ellos representamos por un rectángulo con puntos azules el tiempo recepción del trabajo, por un rectángulo con líneas discontínuas verticales naranjas el tiempo que el esclavo invierte en realizar el cómputo, y por un rectángulo con líneas diagonales verdes el tiempo necesario para devolver el resultado al maestro. Las restantes imágenes de este capítulo que representan diagramas de tiempo mantienen este mismo formato de colores.

En este trabajo hemos abordado las estrategias FIFO y LIFO (figura 2.2). Los algoritmos que ejecutan el maestro y los esclavos para resolver el problema, mediante estas dos estrategias, se presentan a continuación:

Estrategia FIFO

```
    /* Procesador: Maestro */
for (proc = 1; proc <= p; proc++)
    enviar(proc, trabajo[proc]);

for (proc = 1; proc <= p; proc++)
    recibir(proc, resultado[proc]);
```

```
  /* Procesador: Esclavo */
recibir(proc, trabajo);
computar(trabajo, resultado);
enviar(proc, resultado);
```

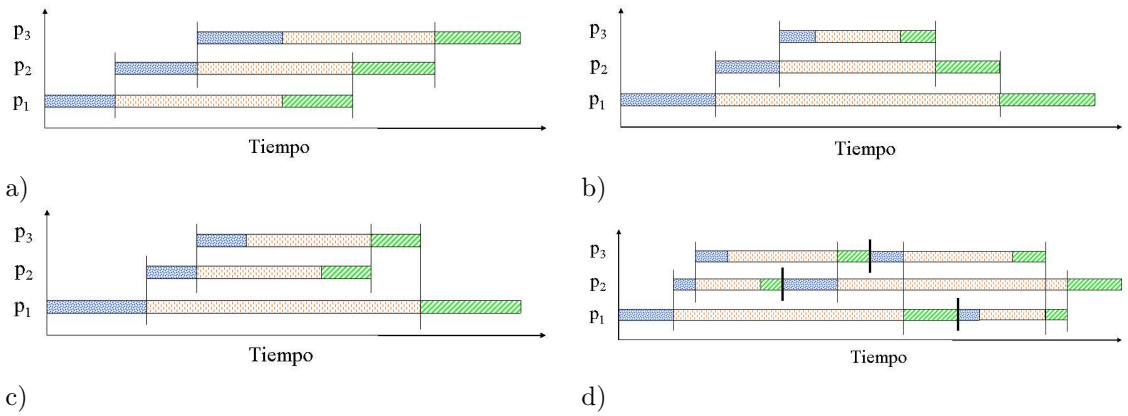

Figura 2.2: Esquemas de resolución del paradigma maestro-esclavo. a) Estrategia FIFO: los esclavos finalizan su ejecución en el mismo orden en que comenzaron. b) Estrategia LIFO: los esclavos finalizan su ejecución en el orden inverso al de comienzo. c) Caso particular de la tercera estrategia, resolviendo el problema en una única etapa. d) Caso particular de la tercera estrategia, resolviendo el problema en dos etapas.

Estrategia LIFO

```
    /* Procesador: Maestro */
for (proc = 1; proc <= p; proc++)
    enviar(proc, trabajo[proc]);

for (proc = p - 1; proc >= 1; proc--)
    recibir(proc, resultado[proc]);
```

```
  /* Procesador: Esclavo */
recibir(proc, trabajo);
computar(trabajo, resultado);
enviar(proc, resultado);
```

Se observa que en ambos casos el proceso realizado en los esclavos es el mismo: reciben una tarea, computan ese trabajo y devuelven el resultado al maestro. La diferencia entre los dos esquemas radica únicamente en el orden de recepción de los resultados de los esclavos en el maestro (figura 2.2).

Dependiendo de la relación entre el tamaño del trabajo enviado y el tamaño del resultado generado por los esclavos, pueden darse diferentes situaciones al resolver este problema, empleando cualquiera de las dos estrategias consideradas (FIFO y LIFO). La primera de estas situaciones se produce cuando los dos tamaños son iguales y, por lo tanto, el tiempo que tarda un esclavo en recibir su tarea coincide con el tiempo que tarda en enviar su resultado (figura 2.2). Sin embargo, esto no sucede en todas las aplicaciones. Cuando los tamaños de los datos de entrada y del resultado no coinciden, la distribución uniforme entre todos los procesadores puede no ser la distribución óptima. Puede ocurrir que el tamaño de la tarea sea superior al tamaño del resultado. Esta situación se puede encontrar en problemas donde cada esclavo recibe un conjunto de datos (normalmente una matriz o una parte de ella) y el resultado a devolver es un único valor obtenido al operar con los datos iniciales. Un caso particular se produce cuando el esclavo no genera ningún resultado (el tamaño del resultado es cero). Esto podría deberse, por ejemplo, a que se esté resolviendo el problema en un sistema de memoria compartida o bien porque el esclavo almacene el resultado obtenido directamente en un fichero. La figura 2.3 muestra gráficamente estos dos casos para la estrategia de resolución FIFO.

Debido a la exclusividad del canal de comunicación, cuando se resuelve un problema hay ocasiones en las que todos los esclavos están computando por lo que el maestro permanece ocioso. Sin embargo, también puede suceder que un procesador termine de computar su trabajo y quiera enviar su resultado al maestro pero el canal se encuentre ocupado por otro esclavo, en este caso el esclavo que ya ha finalizado su cómputo debe esperar sin trabajar hasta que el canal sea liberado. En el diagrama de tiempo esto lo representamos como un hueco entre el cómputo (rectángulo de líneas verticales naranjas) y el envío

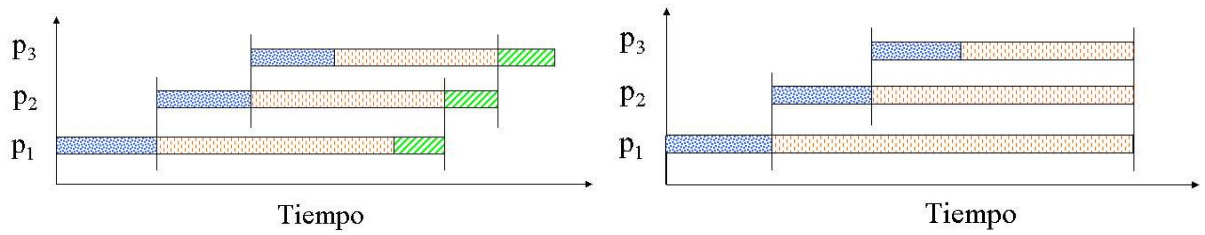

Figura 2.3: Ejemplos de las situaciones particulares para la estrategia FIFO. Izquierda: El tamaño de las tareas es mayor que el tamaño de los resultados. Derecha: El tamaño del resultado es 0.

del resultado (rectángulo con líneas diagonales verdes). De aquí en adelante, denominaremos **hueco** al período de tiempo en el que un esclavo permanece ocioso durante el proceso de la resolución del problema. En la figura 2.3 se observa que para evitar la existencia de huecos que retrasen la resolución del problema se asignan a los esclavos cantidades distintas de trabajo, de forma que el tiempo de cómputo de cada esclavo disminuya progresivamente.

La tercera situación se produce cuando el tamaño del resultado es mayor que el tamaño de la tarea (figura 2.4-Izquierda). Como caso particular, también nos encontramos la posibilidad de que el esclavo no tenga que recibir inicialmente ninguna información, en cuyo caso el tamaño de la tarea es 0 (figura 2.4-Derecha). Esto puede deberse a que los esclavos leen la información de entrada directamente de un fichero y trabajan sobre los datos leídos. En la figura 2.4 el tiempo de cómputo del primer esclavo es menor que el tiempo de cómputo del último para evitar la aparición de huecos en los diagramas de tiempo.

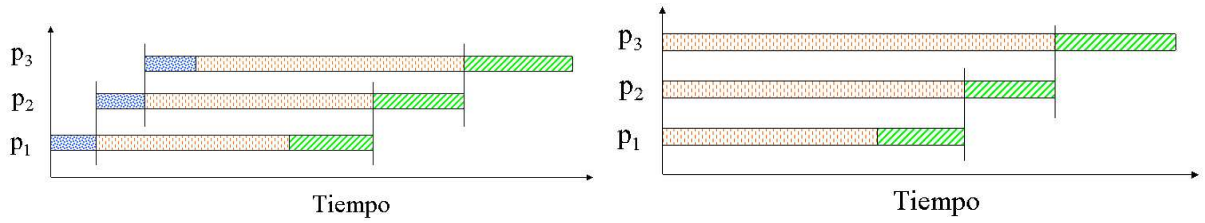

Figura 2.4: Ejemplos de las situaciones particulares para la estrategia FIFO. Izquierda: El tamaño de la tarea es menor que el tamaño del resultado. Derecha: El tamaño del tarea es 0.

Los modelos, que presentamos en las secciones siguientes, consideran el caso general donde tenemos un tiempo de recepción de tareas y un tiempo de envío de resultados para cada esclavo; y estos dos valores pueden ser iguales o diferentes, e incluso tomar el valor 0. Así mismo, los tiempos de cómputo no tienen que ser iguales para todos los esclavos. Todas estas consideraciones permiten a nuestros modelos reflejar cualquiera de las situaciones vistas anteriormente, sin necesidad de establecer casos particulares.

2.4. Estrategia FIFO de resolución del problema en sistemas homogéneos

Un sistema homógeneo es aquel donde todos los procesadores tienen las mismas características (sección 1.1). En estos sistemas y en la situación donde el tamaño de la tarea es igual al tamaño del resultado, el particionamiento óptimo de los datos se obtiene cuando se asigna a todos los procesadores la misma cantidad de trabajo (m/p), de forma que el tiempo de cómputo es el mismo para todos los esclavos. Este tiempo de cómputo depende del tamaño total del problema y de la cantidad de trabajo asignada a cada

procesador, por lo que definimos una función que nos devuelva el tiempo de cómputo de cada esclavo. Debido a que en un sistema homogéneo a todos los procesadores se les asigna la misma cantidad de trabajo, la función la representamos como $C_i(m/p)$ y nos devuelve un valor que es constante para todos los procesadores. La figura 2.5 representa un diagrama de tiempo típico para un problema maestro-esclavo homogéneo donde el tamaño de la tarea coincide con el tamaño del resultado.

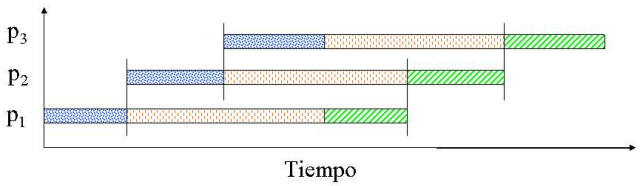

Figura 2.5: Diagrama de tiempo para un problema maestro-esclavo homogéneo donde el tamaño de la tarea coincide con el tamaño del resultado.

En la figura 2.5, los rectángulos con líneas verticales naranjas representan el tiempo, $C(m/p)$, que los esclavos emplean en computar el trabajo asignado; los rectángulos con puntos azules representan el tiempo de comunicación entre el maestro y los esclavos para recibir las tareas $(\beta + \tau * w)$, donde w (el tamaño en bytes de la tarea que el maestro envía a cada esclavo) es un valor fijo para todos los procesadores; y los rectángulos con líneas diagonales verdes representan el tiempo de comunicación entre los esclavos y el maestro para enviar los resultados $(\beta + \tau * s)$, donde s es el número de bytes que ocupa el resultado que los esclavos devuelven al maestro. También s es un valor fijo para todos los procesadores, pero puede ser un valor igual o diferente al tamaño de las tareas. Estos dos valores normalmente dependen del tipo de datos, del tamaño total del problema y de la cantidad de trabajo asignada a cada procesador.

El tiempo total de ejecución viene dado por el tiempo empleado hasta que finaliza el último esclavo (procesador p_p) y se puede obtener mediante la fórmula:

$T_{par} = T_p = (p * (\beta + \tau * w)) + C(m/p) + (\beta + \tau * s)$

Es decir, el tiempo total se calcula como la suma de los tiempos de recepción de todos los esclavos más el tiempo que tarda el último esclavo en computar su tarea y devolver su resultado al maestro (figura 2.5).

2.5. Estrategia FIFO de resolución del problema en sistemas heterogéneos

A diferencia de los sistemas homogéneos, en los entornos heterogéneos consideramos que los tiempos de transmisión entre el maestro y los diferentes esclavos pueden ser distintos, así como el tiempo de cómputo que necesitan para realizar su tarea (heterogeneidad debida a los procesadores y heterogeneidad debida a la red (sección 1.2)).

En la sección 1.5.1 presentamos las características de la plataforma heterogénea sobre la que realizamos las pruebas. Utilizando estas características definimos tres funciones de tiempo que vamos a utilizar en el desarrollo de los modelos analíticos. La primera de ellas la denominamos **tiempo de cómputo** $(C_i(m_i))$ y nos indica el tiempo que tarda un procesador p_i en computar una tarea de tamaño m_i. Esta función nos permite determinar las diferencias en las velocidades de cómputo de las máquinas (heterogeneidad debida a los procesadores). Al ser una función dependiente del tipo de tarea a procesar, en cada uno de los problemas que se resuelven en este capítulo se presentan tablas y gráficas de la función obtenida.

Las otras dos funciones hacen referencia a los tiempos de transmisión entre dos procesadores (heterogeneidad debida a la red). Sea un procesador cualquiera, p_i, vamos a denominar **tiempo de recepción**

de datos al tiempo que tarda el procesador maestro en transmitir los datos y el que necesita el procesador p_i en recibirlos. Este tiempo lo denotamos por $R_i(w_i) = \beta_i + \tau_i * w_i$, donde w_i representa el tamaño en bytes de la información que recibe el procesador p_i y β_i y τ_i es la latencia y el tiempo de tranferencia por byte respectivamente entre el maestro y p_i. El segundo tiempo que consideramos para el procesador p_i es el **tiempo de envío** de datos. Se obtiene como $S_i(s_i) = \beta_i + \tau_i * s_i$, donde s_i es el tamaño en bytes de los datos transmitidos. Como ocurre en el caso anterior, la comunicación incluye el tiempo que tarda el procesador en enviar la información como el tiempo que tarda el procesador maestro en recibirla.

Al definir las funciones anteriores estamos considerando que el tamaño de la tarea y el tamaño de la solución pueden ser iguales ($w_i = s_i$) o diferentes. Esto nos permite cubrir todas las posibles situaciones especificadas en la sección 2.3.

Debido a estas circunstancias, en el entorno heterogéneo pueden aparecer situaciones anómalas que tenemos que considerar dependiendo de la posición que ocupe cada procesador en el conjunto total de procesadores.

El tiempo total que necesita el esclavo p_i para finalizar su trabajo lo denominamos T_i, este valor incluye no sólo el tiempo que el procesador tarda en computar su tarea ($C_i(m_i)$), sino también el tiempo necesario para recibir los datos desde el maestro ($R_i(w_i)$) y el tiempo para devolver el resultado ($S_i(s_i)$). Por tanto, el tiempo total de un problema maestro-esclavo con p esclavos viene dado por el tiempo que tarda el último esclavo en completar su trabajo ($T_{par} = T_p$). En lo que resta de capítulo y, sin pérdida de generalidad, se simplificará la notación utilizando únicamente R_i, C_i y S_i en lugar de $R_i(w_i)$, $C_i(m_i)$ y $S_i(s_i)$ respectivamente.

Como comentábamos en la sección 2.2, se han propuesto muchas técnicas de planificación de tareas en entornos heterogéneos. En [66] se presenta una estrategia para resolver este caso, que puede verse como la extensión natural del trabajo desarrollado en [37]. Ellos consideran que la distribución óptima en la estrategia FIFO se obtiene cuando $C_i + S_i = R_{i+1} + C_{i+1}$ (figura 2.6). La idea es solapar la computación y la comunicación y conseguir una distribución equilibrada en carga entre todos los procesadores. La distribución se obtiene resolviendo un sistema lineal de $p-1$ ecuaciones de la forma anterior. Sin embargo, en este trabajo no se analizan las colisiones en un plataforma heterogénea.

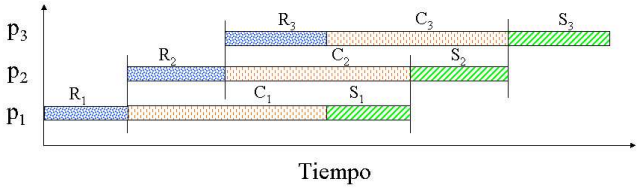

Figura 2.6: Diagrama de tiempo para la estrategia de distribución $C_i + S_i = R_{i+1} + C_{i+1}$. Se consigue una distribución perfectamente equilibrada en carga, pero no se realiza un análisis de colisiones en las comunicaciones.

Si el número de procesadores es demasiado grande, la resolución del sistema podría no generar una distribución óptima, tal y como veremos a continuación. Consideremos un ejemplo en el que se resuelve un problema pequeño de tamaño $m = 38$ empleando 5 esclavos, en una situación donde el tamaño de la tarea es igual al tamaño del resultado que genera, por lo que tenemos $w_i = s_i$ y por lo tanto $R_i = S_i$. En la figura 2.7-a se puede ver una solución para el sistema de ecuaciones que no produce una distribución óptima debido a que existe un solapamiento de las comunicaciones (la zona sombreada de color amarillo). Este solapamiento no es posible puesto que el canal de comunicación disponible es un canal exclusivo. En esta estrategia se está suponiendo que el primer procesador finalizará su cómputo después de que el último esclavo haya comenzado a trabajar ($\sum_{i=2}^{p} R_i \geq C_1$) y no se comprueba que esta suposición sea correcta. Cuando se observa el diagrama real de los tiempos de ejecución para esta distribución, sin solapamiento en las comunicaciones, se producen huecos en los que los procesadores se encuentran ociosos durante la ejecución. Esto incrementa el tiempo necesario para resolver el problema (figura 2.7-b).

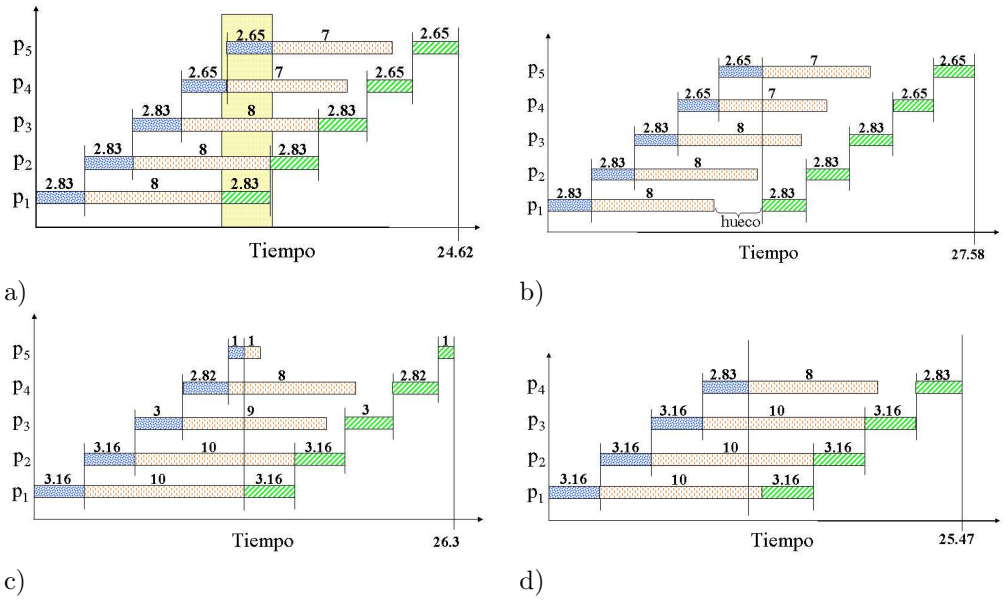

Figura 2.7: Comparación de los diagramas de tiempo correspondientes a diferentes distribuciones de tareas para resolver un problema de tamaño $m = 38$ con 5 procesadores. a) Diagrama de tiempo erróneo, la distribución está equilibrada en carga, pero en la zona sombreada S_i se solapa con R_i; esto no es posible puesto que el canal de comunicación es exclusivo. b) Diagrama de tiempo correcto para la distribución obtenida como solución del sistema de ecuaciones, donde los huecos aumentan el tiempo de resolución del problema c) Una distribución que no es solución del sistema de ecuaciones reduce el tiempo de ejecución con 5 procesadores. d) La distribución óptima para este problema se consigue utilizando únicamente 4 procesadores.

El tiempo total de ejecución del problema se puede reducir utilizando los 5 procesadores pero con una distribución de tareas diferente, que no es solución del sistema de ecuaciones anterior (figura 2.7-c). Esto significa que para un número de procesadores fijo, la mejor distribución de trabajos no se obtiene utilizando esta estrategia. Además, la distribución óptima para este problema se consigue cuando se reduce el número de procesadores y se emplean únicamente 4 de los 5 procesadores disponibles (figura 2.7-d). Como se puede observar en la figura 2.7 el mejor tiempo obtenido con cinco procesadores es peor que el tiempo obtenido con la distribución óptima con cuatro de ellos.

En este capítulo mostramos que para obtener la distribución óptima de trabajos para un problema maestro-esclavo no sólo debemos tener en cuenta el tamaño del problema a resolver, sino que el número de procesadores empleados para su resolución es también un parámetro importante en la minimización del tiempo total de ejecución.

El modelo analítico que presentamos puede ser considerado como una instanciación particular del modelo estructural descrito en [160]. De acuerdo con nuestro modelo, el tiempo total de ejecución del problema es obtenido como $T_{par} = T_p$; es decir, el tiempo necesario para que finalice la ejecución del último procesador.

Si consideramos que existe un único esclavo el tiempo total de ejecución es $T_{par} = T_1 = R_1 + C_1 + S_1$.

Cuando disponemos de dos esclavos para resolver un problema maestro-esclavo heterogéneo encontramos dos posibles situaciones que determinan el tiempo de ejecución (figura 2.8). Si el esclavo p_2 no ha terminado su cómputo cuando el procesador p_1 finaliza el envío del resultado al maestro, la resolución del problema se retrasa porque el maestro debe esperar a que el segundo esclavo termine su cómputo y devuelva el resultado (figura 2.8-Izquierda); en este caso se cumple la expresión

$d_2 = g_2 = (R_2 + C_2) - (C_1 + S_1) > 0$. Por el contrario, si el procesador p_2 finaliza el cómputo cuando p_1 aún no ha terminado el envío del resultado (figura 2.8-Derecha), p_2 tiene que permanecer ocioso hasta que el canal de comunicación se libere; en este segundo caso se cumple que $d_2 = g_2 = (R_2+C_2)-(C_1+S_1) < 0$. El tiempo total de ejecución que obtenemos es: $T_2 = T_1 + S_2 + max(0, g_2)$. De esta forma, en el primer caso (figura 2.8-Izquierda) el tiempo total es $T_{par} = T_2 = T_1 + S_2 + g_2$, mientras que en el segundo caso (figura 2.8-Derecha) el tiempo necesario para la ejecución es $T_{par} = T_2 = T_1 + S_2$.

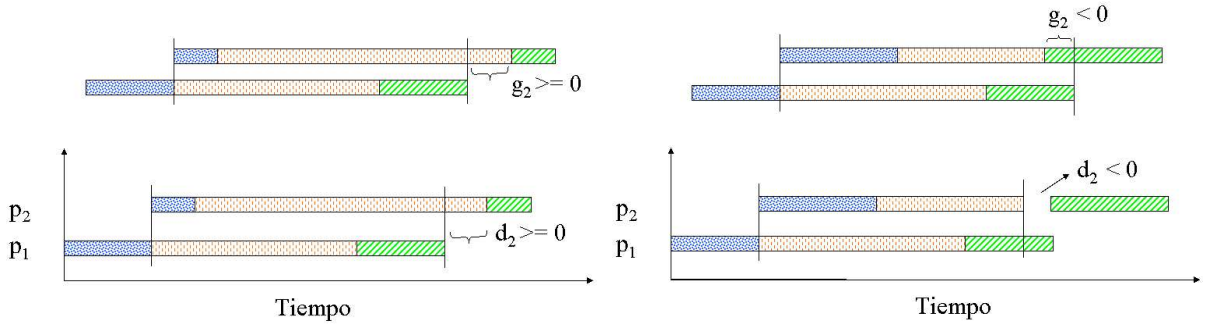

Figura 2.8: Diagramas de tiempo para el problema del maestro-esclavo heterogéneo con 2 esclavos. Izquierda: $g_2 = (R_2 + C_2) - (C_1 + S_1) > 0$, entre la recepción de los resultado de los esclavos p_1 y p_2 el maestro se encuentra ocioso. Derecha: $g_2 = (R_2 + C_2) - (C_1 + S_1) < 0$, una vez que el procesador p_2 termina de computar la tarea, debe esperar a que p_1 termine de enviar su resultado al maestro para poder realizar su propio envío.

Denotamos por $g_i = (R_i + C_i) - (C_{i-1} + S_{i-1})$ al hueco (*gap*) de tiempo que se produce entre el procesador p_{i-1} y el procesador p_i. Si es positivo, el tiempo de recepción del esclavo p_i más su tiempo de cómputo es superior a la suma de los tiempos de cómputo y envío del resultado del procesador p_{i-1}, lo que significa que el procesador p_i se encuentra todavía computando su tarea cuando el procesador anterior ya ha finalizado su envío. En caso contrario el valor de g_i es menor que 0.

Denominamos d_i al retraso (*delay*) acumulado hasta el procesador p_i, donde

$$d_i = \begin{cases} g_i & si\ d_{i-1} \geq 0 \\ g_i - \mid d_{i-1} \mid & en\ otro\ caso \end{cases}$$

Un retraso negativo para el procesador p_i ($d_i < 0$) indica que el cómputo del esclavo p_i se solapa con el envío del resultado del procesador anterior, de forma que solamente debemos considerar el tiempo que tarda el esclavo en enviar el resultado al maestro. Un retardo positivo d_i para el procesador p_i significa que el cómputo del procesador p_i aún continúa cuando el procesador anterior ha terminado el envío de su resultado. Como no es posible aplicar esta fórmula al primer esclavo asignamos $d_1 = 0$ y, debido a esta asignación inicial, se cumple siempre que para el segundo procesador $d_2 = g_2$. Por lo tanto, el tiempo total de ejecución necesario para resolver este problema se puede formular como: $T_{par} = T_2 = T_1 + S_2 + max(0, d_2)$.

Si incrementamos el número de procesadores observamos que aumentan las situaciones que nos podemos encontrar en la ejecución de un problema maestro-esclavo. La figura 2.9 muestra los diagramas de tiempo de dos de las posibles situaciones en la ejecución de un problema cuando utilizamos 3 esclavos. En la figura, g_2 denota el hueco que se produce entre los esclavos p_1 y p_2, y g_3 denota el hueco que se produce entre p_2 y p_3. La figura 2.9-Izquierda considera la situación en la que p_2 es el procesador más rápido y p_3 es el procesador más lento. Se produce un hueco entre los dos primeros esclavos ($d_2 < 0$) porque p_2 necesita un tiempo de $R_2 + C_2$ para recibir y computar su tarea, y este tiempo es menor que el tiempo empleado por p_1 en computar su tarea y enviar el resultado ($C_1 + S_1$), por lo que p_2 debe esperar a que p_1 termine para enviar su propio resultado. Sin embargo, cuando p_2 finaliza, p_3 aún está trabajando

($g_3 > 0$). El retraso acumulado para el procesador p_3 es positivo y tiene un valor de $d_3 = g_3 - |d_2|$. El maestro va a pasar un período de tiempo ocioso esperando por el resultado del último esclavo.

En la figura 2.9-Derecha se muestra otro posible caso, en el que p_3 es el procesador más rápido y el esclavo colocado en la posición intermedia (p_2) es el más lento. En esta figura se observa que $g_2 > 0$ y $g_3 < 0$. El retraso acumulado aquí resulta ser negativo ($d_3 < 0$), por lo que p_3 no puede enviar su resultado hasta que p_2 haya finalizado.

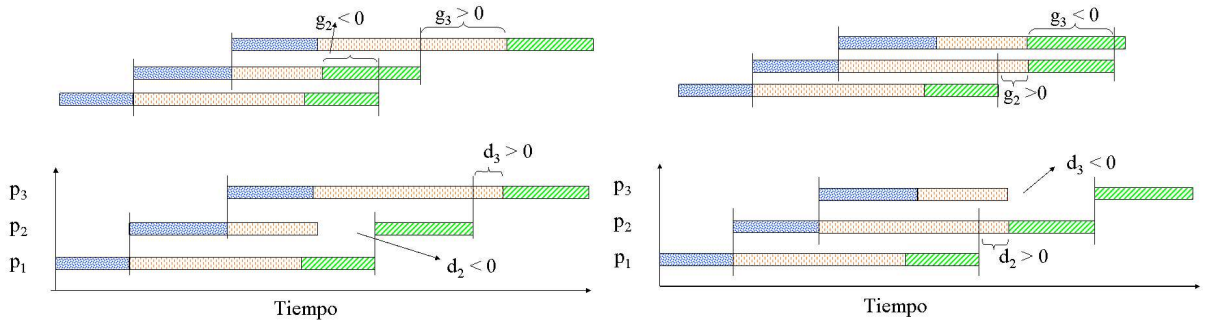

Figura 2.9: Diagramas de tiempo para el problema maestro-esclavo heterogéneo con 3 esclavos. Izquierda: $d_3 > 0$, el maestro se mantiene ocioso entre el envío del resultado de p_2 y el de p_3. Derecha: $d_3 < 0$, el procesador p_3 tiene que esperar para poder comunicarse con el maestro.

De acuerdo con este modelo, el tiempo total de ejecución para el programa paralelo empleando p esclavos lo podemos establecer utilizando la siguiente ecuación de recurrencia:

$$T_{par} = T_p = T_{p-1} + S_p + \text{máx}(0, d_p) = T_{p-2} + S_{p-1} + S_p + \text{máx}(0, d_{p-1}) + \text{máx}(0, d_p)$$

$$= T_1 + \sum_{i=2}^{p} S_i + \sum_{i=2}^{p} \text{máx}(0, d_i) = R_1 + C_1 + \sum_{i=1}^{p} S_i + \sum_{i=1}^{p} \text{máx}(0, d_i)$$

donde $d_1 = 0$ y para $i > 1$,

$$d_i = \begin{cases} g_i & \textit{si } d_{i-1} \geq 0 \\ g_i - \mid d_{i-1} \mid & \textit{en otro caso} \end{cases} \quad \text{con}$$

$$g_i = (R_i + C_i) - (C_{i-1} + S_{i-1}) \quad \text{y}$$

$$\sum_{i=2}^{p} R_i \leq C_1$$

Esta última restricción del modelo nos permite asegurar que en ningún caso se producirá una colisión de las comunicaciones, puesto que la suma de los tiempos de trasmisión de las tareas para los esclavos $p_2, \ldots, p_p$ no puede superar el tiempo de cómputo del primer esclavo.

2.6. Estrategia LIFO de resolución del problema

La estrategia LIFO para resolver un problema maestro-esclavo se ilustra en la figura 2.2-b. En ella se observa que los esclavos devuelven los resultados en el orden inverso al que reciben sus tareas. Para evitar que se produzcan huecos y se incremente el tiempo total de resolución del problema, el tiempo que un procesador tarda en computar su trabajo debe ser igual al tiempo que tarda el siguiente procesador

en recibir su trabajo, realizar su cómputo y devolver su resultado al maestro. Observando la figura se comprueba que en este caso el tiempo total de ejecución viene determinado por el tiempo de ejecución del primer esclavo ($T_{par} = T_1$).

Para simplificar las fórmulas del modelo introducimos una nueva notación: Denotamos por ϕ_i al tiempo empleado por el procesador p_i hasta la finalización de su ejecución. Este tiempo incluye el tiempo que transcurre desde el instante en que el procesador p_i recibe los datos hasta que devuelve su resultado al maestro, y excluye el tiempo inicial de espera para recibir su trabajo.

El procesador p_p, al ser el último de los esclavos, no tiene que esperar a que finalice el resto de los procesadores, sino que es el primero en enviar el resultado al maestro. Esto significa que puede realizar el envío en el momento en que termine su cómputo. El tiempo total que tarda este procesador en finalizar su ejecución viene dado por la suma del tiempo que permanece esperando a que le lleguen los datos de entrada, más el tiempo necesario para recibir esos datos, procesarlos y devolver el resultado al maestro. Este tiempo podemos modelizarlo mediante la siguiente ecuación:

$$T_p = \sum_{i=1}^{p-1} R_i + \phi_p, \qquad \text{con} \qquad \phi_p = R_p + C_p + S_p$$

El resto de los procesadores, desde p_1 hasta p_{p-1}, tiene que esperar a que el procesador siguiente finalice el envío de su resultado al maestro, por lo que el tiempo de ejecución para los restantes procesadores lo podemos modelar utilizando la ecuación:

$$T_i = \sum_{k=1}^{i-1} R_k + \phi_i, \qquad \text{con} \qquad \phi_i = R_i + \text{máx}(C_i, \phi_{i+1}) + S_i$$

Como el tiempo total de resolución del problema es el tiempo que tarda el primer esclavo, p_1, en finalizar la ejecución, la fórmula que determina el tiempo paralelo es:

$$T_{par} = T_1 = \phi_1, \qquad \text{con} \qquad \phi_1 = R_1 + \text{máx}(C_1, \phi_2) + S_1$$

2.7. Validación de los modelos analíticos

Una vez presentados los modelos analíticos, que predicen el tiempo de ejecución paralelo de los algoritmos maestro-esclavo (secciones 2.4, 2.5 y 2.6), el primer paso consiste en validar estos modelos para comprobar que sus predicciones son correctas. Para realizar esta prueba hemos seleccionado un problema muy conocido: el problema del producto de matrices [37, 191, 192].

Consideramos dos matrices cuadradas A y B de tamaño $m \times m$ y resolvemos el problema mediante la siguiente ecuación:

$$C[i][j] = \sum_{k=0}^{m-1} A[i][k] * B[k][j]$$

2.7.1. Algoritmo Secuencial

La función *ProductoMatrices* (figura 2.10) es la que calcula la matriz resultado del producto. Se pasan como parámetros las dos matrices iniciales, A y B, y la matriz donde se va a almacenar el producto de ambas, C. Todas estas matrices son de tamaño $m \times m$, por lo que el valor de m también se le pasa como parámetro a la función.

En las líneas 4 a 6 del código se realizan los tres bucles que permiten recorrer todos los elementos de las matrices para realizar el producto. Con el primero se recorren las filas de A y con el segundo las columnas de B. Estos dos bucles determinan el elemento a calcular en la matriz producto. El tercer bucle (línea 6 del código) permite realizar el sumatorio de la fórmula que resuelve el problema.

```
1 void ProductoMatrices(int **A, int **B, int **C, int m) {
2    int i, j, k;              // Variables índices para los bucles
3
4    for (i = 0; i < m; i++)               // fila matriz producto
5        for (j = 0; j < m; j++)              // columna matriz producto
6            for (k = 0; k < m; k++)              // sumatorio
7                C[i][j] += A[i][k]*B[k][j];
8 }
```

Figura 2.10: Función que resuelve el problema del producto de matrices de forma secuencial.

Hemos ejecutado este programa secuencial en las diferentes máquinas disponibles en el *cluster* (sección 1.5.1). Realizamos diferentes pruebas variando el tamaño de la matriz cuadrada entre 700 y 2000, los resultados obtenidos en estas ejecuciones se muestran en la tabla 2.1. La columna *Tamaño* especifica el tamaño de la matriz cuadrada (número de filas y de columnas de la matriz); las restantes tres columnas representan el tiempo de resolución del problema (expresados en segundos) para las máquinas rápidas (R), intermedias (I) y lentas (L) respectivamente.

Tabla 2.1: Tiempos de ejecución del algoritmo secuencial del producto de matrices en los tres tipos de máquinas que componen el *cluster*.

Tamaño	Máquinas rápidas (R) Tiempo (seg.)	Máquinas Intermedias (I) Tiempo (seg.)	Máquinas Lentas (L) Tiempo (seg.)
700	20.13	32.84	98.49
1000	59.37	99.15	298.56
1500	205.77	615.89	998.93
2000	593.70	2198.24	2337.66

2.7.2. Algoritmo Paralelo

El algoritmo secuencial visto en la sección 2.7.1 puede ser fácilmente paralelizado: la matriz B es enviada previamente a todos los esclavos y, posteriormente, cada esclavo recibe m_i filas de la matriz A, donde $\sum_{i=1}^{p} m_i = m$.

Hemos implementado el algoritmo paralelo según las dos versiones vistas en las secciones 2.5 y 2.6. La figura 2.11 muestra el código que resuelve este problema de forma paralela mediante la estrategia FIFO.

La función *ProductoMatrices* recibe como parámetros las dos matrices originales (A y B) y la matriz donde se almacena el resultado (C); el número de filas y columnas de las matrices (m); el identificador del procesador (id), que toma el valor 0 cuando el procesador es el maestro y un valor entre 1 y p para los esclavos y un vector con la cantidad de trabajo que le corresponde a cada esclavo (*trabajo*).

Previamente a la llamada a esta función, en el programa principal, el procesador maestro ha enviado la matriz B completa a todos los esclavos mediante una operación de *broadcast*. Entre las líneas 7 y 14 del código el maestro envía a cada esclavo el trabajo (número de filas de la matriz A) que le corresponde según el valor del vector *trabajo*. En las líneas 15 a 18 los esclavos reciben el trabajo a realizar: primero

un mensaje con el número de filas de la matriz A que tienen que computar y después otro mensaje con las filas correspondientes de la matriz.

En el siguiente bloque de código (líneas 20 a 25), los esclavos calculan su parte de la matriz resultado. Hay que tener en cuenta que el primer bucle (que recorre las filas de la matriz A y de la matriz resultado) sólo varía entre 0 y el número de filas que le asignó el maestro, mientras que los otros dos bucles varían entre 0 y el tamaño total de la matriz. De esta forma, cada esclavo calcula, solamente, la parte de la matriz resultado correspondiente a las filas de la matriz A asignadas.

En la última sección del código los esclavos devuelven su parte de la matriz resultado al maestro. Entre las líneas 27 y 29 los esclavos envían su resultado, mientras que en las líneas 30 a 36 el maestro recibe los datos de los esclavos en el mismo orden en que realizó los envíos, combinando todos los resultados parciales para obtener la matriz producto completa.

```
 1 void ProductoMatrices(int **A, int **B, int**C, int m, int id, int *trabajo) {
 2     int i, j, k;        // Variables índices para los bucles
 3     int n;              // Trabajo que le corresponde a un esclavo
 4
 5          // El maestro ha enviado la matriz B completa a todos los procesadores previamente
 6
 7     if (id == 0) {                      //  MAESTRO
 8        j = 0; // Envía a los esclavos las filas correspondientes de la matriz A
 9        for (i = 1; i < numprocs; i++) {
10            MPI_Send(&trabajo[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD);
11            MPI_Send(&A[j][0], (m * trabajo[i]), MPI_INT, i, 0, MPI_COMM_WORLD);
12            j += trabajo[i];
13        }
14     }
15     else {                              // ESCLAVOS
16         MPI_Recv(&nf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
17         MPI_Recv(&A[0][0],(m * nf),MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
18     }
19
20     if (id != 0) {                      // ESCLAVOS
21         for (i = 0; i < nf; i++)
22            for (j = 0; j < m; j++)
23               for (k = 0; k < m; k++)
24                  C[i][j] += A[i][k] * B[k][j];
25     }
26
27     if (id != 0) {                      // ESCLAVOS
28        MPI_Send(&C[0][0], m * nf, MPI_INT, 0, 0, MPI_COMM_WORLD);
29     }
30     else {                             // MAESTRO
31         j = 0;           // Recibe los resultados: las filas de la matriz producto
32         for (i = 1; i < numprocs; i++) {
33            MPI_Recv(&C[j][0], m * trabajo[i], MPI_INT, i, 0, MPI_COMM_WORLD, &status);
34            j += trabajo[i];
35         }
36     }
37 }
```

Figura 2.11: Función que resuelve el problema del producto de matrices de forma paralela mediante la estrategia FIFO.

Para resolver el problema mediante el esquema LIFO sólo es necesario modificar las líneas 31 a 34 para invertir el orden de recepción de los resultados, empezando por el último esclavo. La figura 2.12 muestra únicamente esta parte del código, puesto que el resto de la función es idéntica a la del esquema FIFO presentado en la figura 2.11.

```
30     else {                              // MAESTRO
31         j = m - trabajo[numprocs - 1];           // Recibe los resultados:
32         for (i = numprocs - 1; i >= 1; i--) {    //          las filas de la matriz producto
33            MPI_Recv(&C[j][0], m * trabajo[i], MPI_INT, i, 0, MPI_COMM_WORLD, &status);
34            j -= trabajo[i - 1];
35         }
36     }
```

Figura 2.12: Código paralelo que resuelve el problema del producto de matrices mediante la estrategia LIFO.

Hemos modelizado este algoritmo según los dos esquemas presentados (secciones 2.5 y 2.6), por lo que es posible predecir el tiempo que va a tardar el programa en resolver el problema. En las subsecciones siguientes, comparamos los resultados de los dos modelos con los tiempos reales de ejecución que hemos obtenido en nuestro *cluster* (sección 1.5.1).

2.7.3. Distribución óptima de trabajos en los sistemas homogéneos

Las primeras ejecuciones paralelas realizadas las hemos llevado a cabo sobre los subsistemas homogéneos existentes en el *cluster* (sección 1.5.1): ejecutamos los problemas en el subsistema formado únicamente por los 4 procesadores rápidos del *cluster*; en el subsistema de los 4 procesadores intermedios; y también en el subsistema de los 6 procesadores lentos. Los problemas utilizados son los mismos que han sido resueltos con el algoritmo secuencial (sección 2.7.1): multiplicar dos matrices cuadradas de tamaños que varían entre 700 y 2000. En el caso paralelo hemos realizado las pruebas utilizando los dos esquemas de resolución (FIFO y LIFO).

Teniendo en cuenta los modelos presentados en las secciones 2.4, 2.5 y 2.6 es necesario conocer inicialmente las funciones de cómputo ($C_i(m)$) para este problema específico, antes de poder predecir cuáles son las distribuciones óptimas para cada modelo y el tiempo que se tarda en resolverlo. En la figura 2.13 se pueden ver, de forma gráfica, estas funciones para los 4 problemas resueltos con los tres tipos de procesadores disponibles.

Se observa en la figura 2.13 el cambio que se produce en la capacidad de las máquinas intermedias del *cluster*: en los problemas pequeños la velocidad de los procesadores intermedios está próxima a la velocidad de los procesadores rápidos, pero su capacidad se va reduciendo, al incrementar el tamaño del problema, hasta que para el problema de tamaño 2000 su velocidad es prácticamente idéntica a la de los procesadores lentos.

El tamaño de los subsistemas homogéneos para cada tipo de máquina varía entre un sistema donde sólo existan el procesador maestro con dos esclavos y el mayor sistema homogéneo disponible, donde trabajen el mayor número posible de máquinas del mismo tipo. La tabla 2.2 muestra el resultado obtenido al ejecutar cada uno de los problemas en los correspondientes subsistemas homogéneos del *cluster* para los procesadores rápidos (R), intermedios (I) y lentos (L) respectivamente. La columna *Tamaño* de estas tablas especifica el número de filas y columnas de las matrices cuadradas; la columna *Número de esclavos* es el tamaño del subsistema homogéneo; y por último, existen dos macrocolumnas con los resultados obtenidos para los dos esquemas de resolución (FIFO y LIFO). Para cada macrocolumna se especifica el tiempo real en segundos que se tarda en resolver el problema (columna *Tiempo*), el tiempo estimado por el modelo para la ejecución del programa (columna *Modelo*), el error relativo entre estas dos cantidades

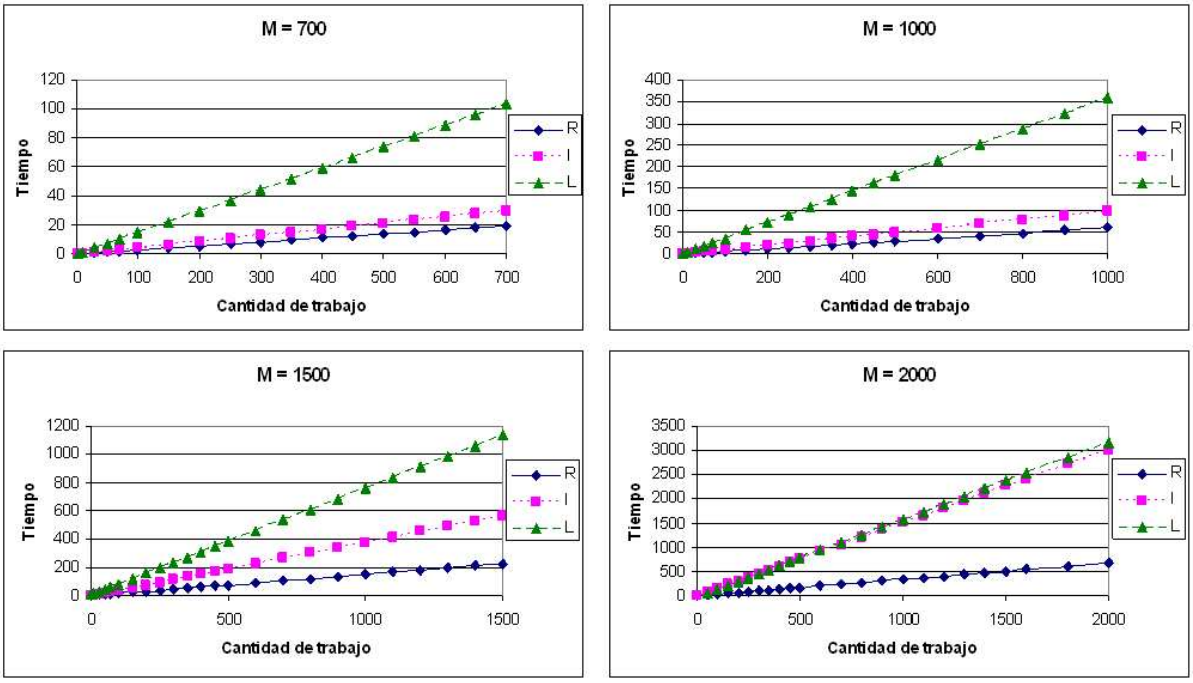

Figura 2.13: Funciones para el cálculo de las funciones $C_i(m)$ del tiempo de cómputo para los 4 tamaños de problemas considerados en las pruebas y para los tres tipos de máquinas existentes en el *cluster*.

obtenido mediante la fórmula $\frac{Tiempo-Modelo}{Tiempo}$ (columna *Error*)y, en la última columna, la aceleración (columna *Acelerac.*) correspondiente al tiempo real de ejecución sobre el tiempo secuencial obtenido al resolver el problema en una de las máquinas del subsistema (tabla 2.1).

Como se observa en la tabla 2.2 la aceleración conseguida en todos los casos es aceptable, aunque disminuye al incrementarse el tamaño de los problemas. Se comprueba la precisión del modelo analítico, que presenta un error de predicción bastante bajo en todas las pruebas: únicamente en 7 de las 64 ejecuciones el error supera el 6 %; de las cuales, sólo en un caso llega a alcanzar un valor del 10 % y en más de la mitad de las ejecuciones realizadas el error es menor al 2 %.

En la tabla 2.3 puede consultarse las distribuciones de trabajo realizadas para cada tamaño de problema y cada subsistema homogéneo. Las columnas *Tamaño* y *Núm. esclavos* tienen el mismo significado que en la tabla 2.2: el número de filas y columnas de la matriz y el tamaño del sistema homogéneo respectivamente. Las dos últimas columnas muestran la distribución entre los esclavos para los dos esquemas ejecutados. Para el esquema FIFO se ha asignado a todos los procesadores la misma cantidad de trabajo (el mismo número de filas) o el número más próximo posible. Hemos establecido esta asignación debido a que, en un sistema homogéneo con este esquema de resolución, la mejor distribución posible es repartir el trabajo de forma uniforme entre todos los procesadores (sección 2.5). En el esquema LIFO, por la forma en la que se resuelve el problema, se le asignan cantidades distintas a cada máquina: un procesador recibe menos trabajo que el procesador anterior. Las cantidades correspondientes a cada procesador se obtienen al aplicar el método exacto que se verá en la sección 2.8. Se utiliza la nomenclatura m_R, m_I y m_L para especificar el número de filas asignado a los procesadores rápidos, intermedios y lentos respectivamente.

En la tabla 2.3, para simplificar los datos, se muestra solamente un valor cuando la cantidad de trabajo asignada es la misma para todos los esclavos y, en caso contrario, se anotan todos los valores en orden ascendente de procesadores, comenzando en la cantidad de trabajo asignada a p_1 y finalizando en la asignada a p_p.

Tabla 2.2: Tiempos de ejecución del algoritmo paralelo del producto de matrices al ejecutar sobre los distintos subsistemas homogéneos del *cluster* con diferentes tamaños de problemas utilizando los dos esquemas de resolución (FIFO y LIFO).

Máquinas rápidas del *cluster* (R)									
	Número de	Esquema FIFO				Esquema LIFO			
Tamaño	esclavos	Tiempo	Modelo	Error	Acelerac.	Tiempo	Modelo	Error	Acelerac.
700	2	9.64	9.65	0.19 %	2.09	9.64	9.66	0.28 %	2.09
700	3	6.47	6.46	0.23 %	3.11	6.47	6.45	0.28 %	3.11
1000	2	29.52	29.93	1.39 %	2.01	30.15	29.95	0.67 %	1.97
1000	3	19.98	20.01	0.16 %	2.97	19.92	19.98	0.30 %	2.98
1500	2	107.02	113.35	5.91 %	1.92	118.25	113.39	4.11 %	1.74
1500	3	78.43	75.58	3.63 %	2.62	81.68	75.63	7.40 %	2.52
2000	2	318.85	335.06	5.08 %	1.86	345.87	335.13	3.11 %	1.72
2000	3	236.09	221.85	6.03 %	2.51	246.55	221.85	10.02 %	2.41
Máquinas intermedias del *cluster* (I)									
	Número de	Esquema FIFO				Esquema LIFO			
Tamaño	esclavos	Tiempo	Modelo	Error	Acelerac.	Tiempo	Modelo	Error	Acelerac.
700	2	15.01	15.05	0.30 %	2.19	15.01	15.07	0.39 %	2.19
700	3	10.11	10.13	0.19 %	3.25	10.11	10.13	0.20 %	3.25
1000	2	49.40	49.68	0.57 %	2.01	49.42	49.68	0.53 %	2.01
1000	3	33.13	33.32	0.57 %	2.99	33.06	33.29	0.70 %	3.00
1500	2	281.31	284.65	1.19 %	2.19	281.25	284.72	1.23 %	2.19
1500	3	188.09	189.73	0.87 %	3.27	188.09	189.89	0.96 %	3.27
2000	2	1503.85	1509.09	0.35 %	1.46	1503.46	1509.81	0.42 %	1.46
2000	3	1004.42	1006.33	0.19 %	2.19	1004.96	1006.33	0.14 %	2.19
Máquinas lentas del *cluster* (L)									
	Número de	Esquema FIFO				Esquema LIFO			
Tamaño	esclavos	Tiempo	Modelo	Error	Acelerac.	Tiempo	Modelo	Error	Acelerac.
700	2	49.57	51.76	4.41 %	1.99	49.74	51.80	4.14 %	1.98
700	3	32.99	34.59	4.86 %	2.98	32.92	34.59	5.06 %	2.99
700	4	25.10	25.86	3.04 %	3.92	25.16	25.92	3.02 %	3.92
700	5	20.82	20.68	0.69 %	4.72	20.93	30.74	0.91 %	4.71
1000	2	165.84	180.21	8.66 %	1.80	175.44	180.33	2.79 %	1.70
1000	3	119.14	120.63	1.26 %	2.51	116.43	120.58	3.57 %	2.56
1000	4	92.54	90.49	2.21 %	3.23	92.23	90.61	1.76 %	3.24
1000	5	75.22	72.55	3.55 %	3.97	75.24	72.72	3.34 %	3.97
1500	2	560.59	572.72	2.16 %	1.78	552.44	572.96	3.71 %	1.81
1500	3	388.75	384.35	1.13 %	2.57	392.22	384.41	1.99 %	2.55
1500	4	305.42	290.17	5.00 %	3.27	300.01	290.43	3.19 %	3.33
1500	5	249.74	233.65	6.44 %	4.00	239.76	233.99	2.40 %	4.17
2000	2	1464.71	1573.16	7.40 %	1.60	1530.43	1573.83	2.84 %	1.53
2000	3	1035.86	1042.39	0.63 %	2.26	971.95	1042.39	7.25 %	2.41
2000	4	780.13	776.20	0.50 %	3.00	734.71	776.67	5.71 %	3.18
2000	5	627.95	616.81	1.77 %	3.72	651.33	617.55	5.19 %	3.59

Tabla 2.3: Distribuciones óptimas de trabajo para los distintos subsistemas homogéneos del *cluster* con los dos esquemas de resolución de los algoritmos maestro-esclavo para el problema del producto de matrices.

Máquinas rápidas del *cluster* (R)			
Tamaño	Núm. esclavos	Esquema FIFO	Esquema LIFO
700	2	$m_R = 350$	$m_R = 350$
700	3	$m_R = 234\ 233\ 233$	$m_R = 243\ 233\ 233$
1000	2	$m_R = 500$	$m_R = 500$
1000	3	$m_R = 334\ 333\ 333$	$m_R = 334\ 333\ 333$
1500	2	$m_R = 750$	$m_R = 750$
1500	3	$m_R = 500$	$m_R = 500$
2000	2	$m_R = 1000$	$m_R = 1000$
2000	3	$m_R = 667\ 667\ 666$	$m_R = 667\ 667\ 666$
Máquinas intermedias del *cluster* (I)			
Tamaño	Núm. esclavos	Esquema FIFO	Esquema LIFO
700	2	$m_I = 350$	$m_I = 352\ 348$
700	3	$m_I = 234\ 233\ 233$	$m_I = 237\ 233\ 230$
1000	2	$m_I = 500$	$m_I = 502\ 498$
1000	3	$m_I = 334\ 333\ 333$	$m_I = 336\ 333\ 331$
1500	2	$m_I = 750$	$m_I = 751\ 749$
1500	3	$m_I = 500$	$m_I = 502\ 500\ 498$
2000	2	$m_I = 1000$	$m_I = 1001\ 999$
2000	3	$m_I = 667\ 667\ 666$	$m_I = 667\ 667\ 666$
Máquinas lentas del *cluster* (L)			
Tamaño	Núm. esclavos	Esquema FIFO	Esquema LIFO
700	2	$m_L = 350$	$m_L = 351\ 349$
700	3	$m_L = 234\ 233\ 233$	$m_L = 235\ 233\ 232$
700	4	$m_L = 175$	$m_L = 176\ 175\ 175\ 174$
700	5	$m_L = 140$	$m_L = 141\ 141\ 140\ 139\ 139$
1000	2	$m_L = 500$	$m_L = 501\ 499$
1000	3	$m_L = 334\ 333\ 333$	$m_L = 334\ 333\ 333$
1000	4	$m_L = 250$	$m_L = 251\ 250\ 250\ 249$
1000	5	$m_L = 200$	$m_L = 201\ 201\ 200\ 199\ 199$
1500	2	$m_L = 750$	$m_L = 751\ 749$
1500	3	$m_L = 500$	$m_L = 501\ 500\ 499$
1500	4	$m_L = 375$	$m_L = 376\ 375\ 375\ 374$
1500	5	$m_L = 300$	$m_L = 301\ 301\ 300\ 299\ 299$
2000	2	$m_L = 1000$	$m_L = 1001\ 999$
2000	3	$m_L = 667\ 667\ 666$	$m_L = 667\ 667\ 666$
2000	4	$m_L = 500$	$m_L = 501\ 500\ 500\ 499$
2000	5	$m_L = 400$	$m_L = 401\ 400\ 400\ 400\ 399$

En el bloque correspondiente a las máquinas rápidas de la tabla 2.3, se comprueba que las cantidades asignadas a los procesadores coinciden en los dos esquemas de resolución. Esta situación se debe a la velocidad en las comunicaciones internas entre los procesadores de la máquina de memoria compartida (sección 1.5.1), que convierte los valores de las funciones R_i y S_i en cantidades despreciables frente a los tiempos de cómputo del problema. En los subsistemas de los procesadores intermedios y lentos no sucede lo mismo, las distribuciones para los dos esquemas no son exactamente iguales, aunque, en la mayoría de los casos, la diferencia es reducida. Esto es debido a que, para este problema, la influencia del tiempo de cómputo en el tiempo total de ejecución es superior a la influencia del tiempo empleado en las comunicaciones entre las máquinas.

2.7.4. Validación del modelo analítico FIFO en un sistema heterogéneo con dos tipos de procesadores

Una vez comprobado que los modelos presentados han sido validados y que sus predicciones son aceptables para los sistemas homogéneos, hemos realizado las mismas pruebas para los sistemas heterogéneos. Inicialmente validamos los modelos en un subconjunto del *cluster* completo, utilizando únicamente las máquinas intermedias y lentas del *cluster*, reservando los procesadores rápidos para pruebas posteriores. Esta decisición está motivada por el hecho de que procesadores rápidos forman parte de una máquina de memoria compartida, mientras que los procesadores intermedios y lentos constituyen un *cluster* de PCs independientes.

En la sección 2.7.3 se consideraba que todas las máquinas de los subsistemas homogéneos tardaban el mismo tiempo en realizar el trabajo asignado, esa condición ha cambiado al considerar un sistema heterogéneo. En esta sección validamos el modelo desarrollando dos series de experimentos. En la primera asignamos la misma cantidad de trabajo a todos los procesadores ($m_I = m_L$) sin tener en cuenta las diferencias entre los tipos de máquinas (**distribución uniforme**). La segunda asume que la cantidad de trabajo asignada (m_i) debe ser proporcional a la velocidad de procesamiento de la máquina (**distribución proporcional**); para esta última distribución hemos elegido una proporción fija de $m_I = 3 * m_L$ aunque, como se puede observar en la tabla 1.1, esta diferencia disminuye al aumentar el tamaño del problema.

Para el esquema LIFO en el caso homogéneo ya se asigna a cada procesador una cantidad distinta de trabajo; se está considerando que el tiempo total empleado por cada procesador en finalizar su ejecución es diferente. El estudio en el caso homogéneo, para esta estrategia, es directamente aplicable a los sistemas heterogéneos. Por otra parte, debido a que los tiempos totales de ejecución para los procesadores deben ser distintos, no tiene sentido aplicar una distribución uniforme o proporcional a esta estrategia. Por estos motivos, en esta sección, vamos a concentrarnos en comprobar la validez del modelo FIFO y en las subsecciones 2.10 y 2.11, al presentar los resultados de las predicciones del tiempo de ejecución mínimo, también realizaremos una validación al comparar el tiempo estimado por el modelo, frente al tiempo real de ejecución.

Para comprobar la precisión del modelo hemos elegido distintos subsistemas heterogéneos con 2, 4, 7 y 9 esclavos. En primer lugar calculamos la relación entre las velocidades de las máquinas. Estos valores los calculamos a partir de los tiempos secuenciales obtenidos para cada procesador (tabla 2.1) y obtenemos la tabla que hemos presentado como ejemplo en la sección 1.5.1 (tabla 1.1).

En la tabla 2.4 presentamos las configuraciones de procesadores utilizadas para realizar las pruebas y les asignamos un nombre que vamos a utilizar posteriormente en el resto de las tablas de este capítulo (columna *Nombre Configuración*). Se muestran en dos columnas diferentes el tipo de procesador que actúa como maestro y los tipos de los procesadores que actúan como esclavos, donde I representa un procesador intermedio y L un procesador lento. En esta tabla, también mostramos la aceleración máxima posible que podremos obtener para este problema, con las distintas configuraciones de procesadores. Estos valores los hemos calculado aplicando la fórmula vista en las secciones 1.4 y 1.5.1, tomando como referencia el tiempo secuencial del procesador que actúa como maestro.

Tabla 2.4: Configuraciones de los sistemas heterogéneos utilizados en las pruebas computacionales de este problema. En todas estas configuraciones se emplean únicamente dos tipos de máquinas del *cluster*: intermedias (I) y lentas (L).

Nombre Configuración	Maestro	Esclavos	Aceleraciones Máximas $m = 700$	$m = 1000$	$m = 1500$	$m = 2000$
c1-2t-3p	I	I L	1.33	1.33	1.62	1.94
c2-2t-5p	I	I I L L	2.66	2.66	3.23	3.88
c3-2t-5p	I	I I I L	3.33	3.33	3.61	3.94
c4-2t-8p	I	I I I L L L L	4.33	4.33	5.47	6.76
c5-2t-10p	I	I I I L L L L L L	5.00	5.00	6.70	8.64

En la tabla 2.4 se observa, también, que la aceleración máxima que se puede alcanzar está por debajo de lo que sería la aceleración máxima, en un sistema homogéneo con el mismo número de procesadores; por ejemplo, para la configuración *c5-2t-10p*, la aceleración máxima para el problema más pequeño es casi la mitad de la aceleración máxima si el sistema fuese homogéneo ($Sp = 9$). Sin embargo, en los tamaños de problemas grandes la aceleración máxima aumenta debido al peor comportamiento de las máquinas intermedias, lo que reduce la diferencia entre los dos tipos de máquinas con las que estamos ejecutando. Esto conlleva a que la heterogeneidad del sistema se reduce y se convierte en un entorno prácticamente homogéneo.

En la tabla 2.5 presentamos los resultados obtenidos al ejecutar los problemas con las dos distribuciones utilizadas (uniforme y proporcional). El formato de esta tabla es muy similar al de la tabla 2.2 donde presentábamos los resultados de los subsistemas homogéneos. En este caso, la primera columna indica el nombre de la configuración de procesadores utilizada (tabla 2.4). Las dos macrocolumnas muestran los resultados obtenidos con la distribución uniforme y proporcional respectivamente: tiempo real de ejecución en segundos, tiempo estimado por el modelo analítico, el error cometido por el modelo y la aceleración obtenida en la ejecución, si utilizamos como tiempo secuencial de referencia, el tiempo de ejecución en el maestro. El número de filas y columnas de las matrices cuadradas se especifica como cabecera de cada uno de los bloques de las tablas.

Como se observa en la tabla 2.5, el error relativo obtenido en estas pruebas es bastante bajo, solamente hay 3 casos en los que el error supera el 5 % y en la mayoría de las pruebas está por debajo del 2 %. Esto supone un porcentaje de error asumible y, por lo tanto, podemos considerar que el modelo está validado cuando utilizamos el susbsistema de las máquinas intermedias y lentas.

En los tiempos reales de la tabla 2.5 comprobamos que, al introducir un mayor número de procesadores, el tiempo necesario para resolver el problema disminuye, como también disminuye si sustituimos un procesador lento por otro intermedio (configuraciones *c2-2t-5p* y *c3-3t-5p*). Además, la distribución proporcional resulta mejor que la distribución uniforme al tener en cuenta las diferencias en las velocidades de cómputo de las máquinas, obligando a que los procesadores más rápidos realicen una mayor cantidad de trabajo. Esto reduce el tiempo total de ejecución paralelo. La única excepción se encuentra en el problema de mayor tamaño ($m = 2000$) debido, como ya hemos mencionado, a la reducción de velocidad que nos encontramos en las máquinas intermedias del *cluster* al resolver problemas de gran tamaño y a que estamos considerando como fijo el porcentaje utilizado para calcular los valores m_I y m_L en la distribución proporcional. Sin embargo, obteniendo valores adecuados de la función C_i estamos teniendo en cuenta también esta situación al aplicar el modelo. En la tabla también mostramos la aceleración obtenida por el sistema heterogéneo frente al tiempo secuencial cuando ejecutamos en un procesador intermedio (el procesador maestro). En la figura 2.14 mostramos la relación entre las aceleraciones reales y las máximas para dos de los problemas ($m = 700$ y $m = 1500$) y las 5 configuraciones utilizadas. En las gráficas presentadas se observa que la distribución proporcional es siempre mejor que la distribución uniforme. La aceleración de la distribución proporcional se encuentra próxima al límite de la aceleración, excepto para los subsistemas grandes con el problema de tamaño 1500.

Tabla 2.5: Tiempos de ejecución del algoritmo paralelo del producto de matrices al ejecutar sobre subsistemas heterogéneos del *cluster* utilizando únicamente las máquinas intermedias y lentas, con diferentes tamaños de problemas utilizando dos tipos de distribución: uniforme y proporcional.

Tamaño de problema: **700**								
	Distribución Uniforme				Distribución Proporcional			
Configuración	Tiempo	Modelo	Error	Aceleración	Tiempo	Modelo	Error	Aceleración
c1-2t-3p	52.28	51.73	1.06 %	0.63	26.48	25.83	2.48 %	1.24
c2-2t-5p	26.20	25.83	1.39 %	1.25	12.89	13.10	1.65 %	2.55
c3-2t-5p	26.71	25.83	3.30 %	1.23	11.55	10.28	10.96 %	2.84
c4-2t-8p	14.58	14.73	1.03 %	2.25	7.85	7.92	0.95 %	4.19
c5-2t-10p	11.67	11.48	1.60 %	2.82	7.13	6.89	3.47 %	4.60
Tamaño de problema: **1000**								
	Distribución Uniforme				Distribución Proporcional			
Configuración	Tiempo	Modelo	Error	Aceleración	Tiempo	Modelo	Error	Aceleración
c1-2t-3p	167.38	180.14	7.62 %	0.59	88.19	90.42	2.53 %	1.12
c2-2t-5p	86.28	90.43	4.82 %	1.15	43.83	45.57	3.96 %	2.26
c3-2t-5p	85.48	90.43	5.79 %	1.16	36.70	36.60	0.29 %	2.70
c4-2t-8p	52.53	52.04	0.94 %	1.89	27.29	28.35	3.89 %	3.63
c5-2t-10p	42.70	40.66	4.77 %	2.32	24.88	24.77	0.46 %	3.98
Tamaño de problema: **1500**								
	Distribución Uniforme				Distribución Proporcional			
Configuración	Tiempo	Modelo	Error	Aceleración	Tiempo	Modelo	Error	Aceleración
c1-2t-3p	557.37	572.58	2.73 %	1.10	421.98	427.05	1.20 %	1.46
c2-2t-5p	279.85	290.04	3.64 %	2.20	210.73	214.79	1.93 %	2.92
c3-2t-5p	297.11	290.02	2.39 %	2.07	168.86	170.75	1.12 %	3.65
c4-2t-8p	165.84	169.12	1.98 %	3.71	132.32	132.78	0.35 %	4.65
c5-2t-10p	134.98	133.33	1.22 %	4.56	112.80	113.82	0.90 %	5.46
Tamaño de problema: **2000**								
	Distribución Uniforme				Distribución Proporcional			
Configuración	Tiempo	Modelo	Error	Aceleración	Tiempo	Modelo	Error	Aceleración
c1-2t-3p	1535.10	1572.90	2.46 %	1.43	2257.71	2264.02	0.28 %	0.97
c2-2t-5p	753.50	775.97	2.98 %	2.92	1130.33	1131.67	0.12 %	1.94
c3-2t-5p	760.33	775.94	2.05 %	2.89	903.95	905.19	0.14 %	2.43
c4-2t-8p	431.84	434.90	0.71 %	5.09	696.21	696.86	0.09 %	3.16
c5-2t-10p	336.73	336.06	0.20 %	6.53	603.69	604.45	0.46 %	3.64

En las tablas 2.6 y 2.7 pueden consultarse las distribuciones de trabajo empleadas en cada una de las ejecuciones de la tabla 2.5. Para cada ejecución se diferencia entre la cantidad de trabajo asignada a los procesadores intermedios (m_I) y a los procesadores lentos (m_L). Para simplificar la tabla, cuando la cantidad asignada a todos los procesadores del mismo tipo sea la misma, se escribe el valor una única vez. Por el contrario, cuando a procesadores del mismo tipo se le asignen cantidades diferentes, debido a problemas de redondeo en la distribución del trabajo, se mostrarán las distintas cantidades indicando, en cada caso, a qué procesadores se asigna cada una de ellas mediante la expresión i = *índices de los procesadores*.

En las tablas 2.6 y 2.7 se observan las diferencias en la asignación de las tareas para las dos distribuciones. Mientras que para la distribución uniforme se asigna siempre la misma cantidad de trabajo a todos los procesadores, en la distribución proporcional esa asignación varía, asignándole mayor número de filas a computar a los procesadores con una velocidad de cómputo superior (los procesadores intermedios).

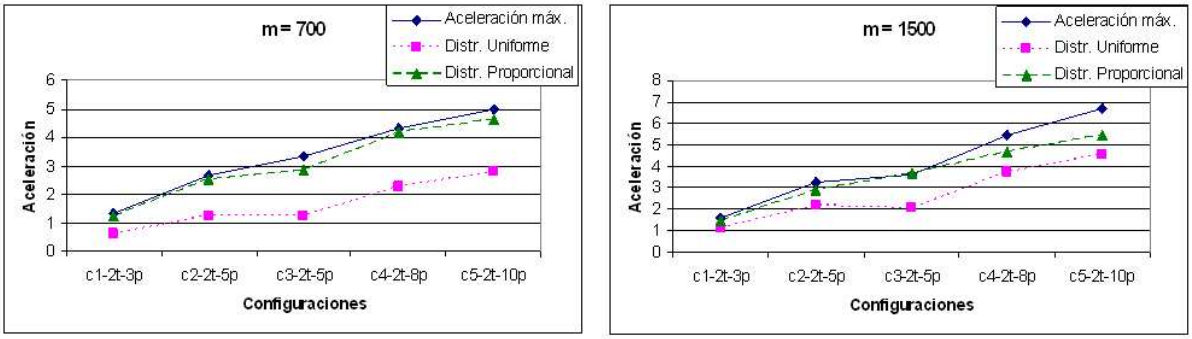

Figura 2.14: Relación entre las aceleraciones reales obtenidas en las ejecuciones de los programas y las aceleraciones máximas, para dos de los problemas ($m = 700$ y $m = 1500$) y las 5 configuraciones utilizadas.

Tabla 2.6: Distribuciones uniformes y proporcionales de trabajo para los subsistemas heterogéneos del *cluster*, utilizando diferentes configuraciones de máquinas intermedias y lentas para el problema del producto de matrices, utilizando matrices cuadradas de tamaños 700 y 1000.

	Tamaño de problema: **700**		Tamaño de problema: **1000**	
Configuración	Distribución Uniforme	Distribución Proporcional	Distribución Uniforme	Distribución Proporcional
c1-2t-3p	$m_I = 350$ $m_L = 350$	$m_I = 525$ $m_L = 175$	$m_I = 500$ $m_L = 500$	$m_I = 750$ $m_L = 250$
c2-2t-5p	$m_I = 175$ $m_L = 175$	$m_I = 262$ $m_L = 87$, i = 3 $m_L = 89$, i = 4	$m_I = 250$ $m_L = 250$	$m_I = 375$ $m_L = 125$
c3-2t-5p	$m_I = 175$ $m_L = 175$	$m_I = 210$ $m_L = 70$	$m_I = 250$ $m_L = 250$	$m_I = 300$ $m_L = 100$
c4-2t-8p	$m_I = 100$ $m_L = 100$	$m_I = 162$ $m_L = 54$, i = 4..6 $m_L = 52$, i = 7	$m_I = 143$ $m_L = 143$, i = 4..6 $m_L = 142$, i = 7	$m_I = 231$ $m_L = 77$, i = 4..6 $m_L = 76$, i = 7
c5-2t-10p	$m_I = 78$ $m_L = 78$, i = 4..7 $m_L = 78$, i = 8..9	$m_I = 140$ $m_L = 47$, i = 4..8 $m_L = 45$, i = 9	$m_I = 112$, i = 1 $m_I = 111$, i = 2..3 $m_L = 111$	$m_I = 200$ $m_L = 67$, i = 4..8 $m_L = 55$, i = 9

2.7.5. Validación del modelo analítico FIFO en el sistema heterogéneo con 3 tipos de procesadores

La última prueba realizada para validar nuestro modelo consiste en incluir la máquina de memoria compartida (los cuatro procesadores rápidos) en nuestra plataforma de pruebas. En la tabla 2.8 presentamos las configuraciones de procesadores que hemos empleado; en este caso todas las configuraciones tienen a un procesador rápido como procesador maestro. Esta tabla mantiene la misma estructura que la tabla 2.4, donde se encontraban las configuraciones para los subsistemas con máquinas intermedias y lentas únicamente (sección 2.7.4).

Las aceleraciones máximas que se pueden obtener con estos sistemas están muy lejos, en algunos casos, de lo que correspondería a un entorno homogéneo con el mismo número de procesadores. También se observa que al incrementar el tamaño del problema se produce una reducción en la aceleración máxima alcanzable en el sistema. Esto es debido al comportamiento de las máquinas intermedias que, como ya hemos comentado, ralentizan su capacidad de cómputo al trabajar con problemas grandes. Esta circunstancia aumenta la heterogeneidad del sistema.

Tabla 2.7: Distribuciones uniformes y proporcionales de trabajo para los subsistemas heterogéneos del *cluster* utilizando diferentes configuraciones de máquinas intermedias y lentas para el problema del producto de matrices, utilizando matrices cuadradas de tamaños 1500 y 2000.

	Tamaño de problema: **1500**		Tamaño de problema: **2000**	
Configuración	Distribución Uniforme	Distribución Proporcional	Distribución Uniforme	Distribución Proporcional
c1-2t-3p	$m_I = 750$ $m_L = 750$	$m_I = 1125$ $m_L = 375$	$m_I = 1000$ $m_L = 1000$	$m_I = 1500$ $m_L = 500$
c2-2t-5p	$m_I = 375$ $m_L = 375$	$m_I = 562$ $m_L = 187$, i = 3 $m_L = 189$, i = 4	$m_I = 500$ $m_L = 500$	$m_I = 750$ $m_L = 250$
c3-2t-5p	$m_I = 375$ $m_L = 375$	$m_I = 450$ $m_L = 150$	$m_I = 500$ $m_L = 500$	$m_I = 600$ $m_L = 200$
c4-2t-8p	$m_I = 215$, i = 1..2 $m_I = 214$, i = 3 $m_L = 214$	$m_I = 346$ $m_L = 115$, i = 4..6 $m_L = 117$, i = 7	$m_I = 286$ $m_L = 286$, i = 4..5 $m_L = 285$, i = 6..7	$m_I = 462$ $m_L = 154$, i = 4..6 $m_L = 152$, i = 7
c5-2t-10p	$m_I = 167$ $m_L = 167$, i = 4..6 $m_L = 166$, i = 7..9	$m_I = 300$ $m_L = 100$	$m_I = 223$, i = 1..2 $m_I = 222$, i = 3 $m_L = 222$	$m_I = 400$ $m_L = 133$, i = 4..8 $m_L = 135$, i = 9

Tabla 2.8: Configuraciones de los sistemas heterogéneos utilizados en las pruebas computacionales de este problema y las aceleraciones máximas que se pueden alcanzar en cada subsistema. En estas configuraciones se emplean todos los tipos de máquinas diferentes existentes en el *cluster*: rápidas (R), intermedias (I) y lentas (L).

Nombre			Aceleraciones Máximas			
Configuración	Maestro	Esclavos	$m = 700$	$m = 1000$	$m = 1500$	$m = 2000$
c6-2t-5p	R	R R I I	3.23	3.20	2.67	2.54
c7-2t-8p	R	R R R I I I I	5.45	5.40	4.34	4.08
c8-2t-8p	R	R R R L L L L	3.82	3.80	3.82	4.02
c1-3t-8p	R	I I I I L L L	3.07	2.99	1.95	1.84
c2-3t-8p	R	R R I I L L L	3.84	3.79	3.29	3.30
c3-3t-9p	R	R R R I I I I L	5.66	5.59	4.54	4.33
c4-3t-13p	R	R R R I I I I L L L L L	6.47	6.39	5.37	5.35
c5-3t-14p	R	R R R I I I I L L L L L L	6.68	6.59	5.57	5.60

Las pruebas que hemos realizado son las mismas que presentamos en la sección 2.7.4: ejecutar los 4 tamaños de problemas con las distintas configuraciones seleccionadas (tabla 2.8) en dos experimentos (distribuciones uniforme y proporcional). La tabla 2.9 muestra los resultados de estas pruebas siguiendo el mismo formato que la tabla 2.5. Para la distribución proporcional, hemos calculado una proporción fija de $m_R \approx \frac{5}{3} m_I$ y $m_I = 3 * m_L$, aunque como se observa en la tabla 1.1 estas diferencias no se mantienen constantes para todos los tamaños de problemas.

Observando los resultados presentados en la tabla 2.9, comprobamos que solamente existen ocho casos (de 64 ejecuciones) donde el error supera el 7 %, mientras que en la mitad de las ejecuciones el error se sitúa por debajo del 2 %. Con este resultado podemos considerar nuestro modelo validado ya en todas las situaciones. Como vimos también en la sección 2.7.4, el tiempo de ejecución real se reduce si la distribución de trabajo entre los procesadores la realizamos en función de las capacidades de las máquinas, excepto en algunos casos al resolver el problema de mayor tamaño. Por otro lado, la aceleración conseguida en estas ejecuciones es bastante alta considerando la aceleración máxima que se podía alcanzar (tabla 2.8). La relación entre ambas, para los problemas $m = 700$ y $m = 1500$ con las 8 configuraciones empleadas, se puede ver gráficamente en la figura 2.15. En esta figura, igual que ocurría en la figura 2.14, la aceleración

Tabla 2.9: Tiempos de ejecución del algoritmo paralelo del producto de matrices al ejecutar sobre subsistemas heterogéneos del *cluster* utilizando los tres tipos de máquinas con diferentes tamaños de problemas utilizando dos tipos de distribución: uniforme y proporcional.

Tamaño de problema: **700**								
	Distribución Uniforme				Distribución Proporcional			
Configuración	Tiempo	Modelo	Error	Aceleración	Tiempo	Modelo	Error	Aceleración
6-2t-5p	7.52	7.53	0.10 %	2.68	6.09	6.11	0.35 %	3.31
c7-2t-8p	4.44	4.36	1.69 %	4.54	3.77	3.67	2.51 %	5.35
c8-2t-8p	14.75	14.65	0.66 %	1.36	5.21	5.26	0.96 %	3.86
c1-3t-8p	13.17	14.72	11.73 %	1.53	6.25	6.87	9.96 %	3.22
c2-3t-8p	15.02	14.67	2.30 %	1.34	5.87	5.66	3.71 %	3.43
c3-3t-9p	12.26	12.77	4.20 %	1.64	3.62	3.78	4.45 %	5.56
c4-3t-13p	9.23	8.51	7.83 %	2.18	3.56	3.29	7.45 %	5.66
c5-3t-14p	8.54	7.87	7.81 %	2.36	3.37	3.15	6.42 %	5.98
Tamaño de problema: **1000**								
	Distribución Uniforme				Distribución Proporcional			
Configuración	Tiempo	Modelo	Error	Aceleración	Tiempo	Modelo	Error	Aceleración
c6-2t-5p	24.75	24.85	0.42 %	2.40	19.01	19.02	0.07 %	3.12
c7-2t-8p	14.31	14.33	0.18 %	4.15	11.34	11.36	0.09 %	5.23
c8-2t-8p	53.37	51.87	2.81 %	1.11	18.59	19.44	4.59 %	3.19
c1-3t-8p	53.08	52.02	2.00 %	1.12	24.22	24.74	2.12 %	2.45
c2-3t-8p	49.77	51.92	4.32 %	1.19	19.24	19.68	2.31 %	3.09
c3-3t-9p	44.46	45.42	2.16 %	1.34	12.63	13.22	4.68 %	4.70
c4-3t-13p	29.86	30.50	2.14 %	1.99	11.27	12.08	7.17 %	5.27
c5-3t-14p	28.05	28.26	0.65 %	2.12	12.78	11.42	10.61 %	4.65
Tamaño de problema: **1500**								
	Distribución Uniforme				Distribución Proporcional			
Configuración	Tiempo	Modelo	Error	Aceleración	Tiempo	Modelo	Error	Aceleración
c6-2t-5p	140.55	141.82	0.91 %	1.46	105.38	106.04	0.63 %	1.95
c7-2t-8p	80.50	80.90	0.50 %	2.56	62.86	62.81	0.08 %	3.27
c8-2t-8p	162.42	168.51	3.75 %	1.27	69.26	66.34	4.22 %	2.97
c1-3t-8p	160.61	169.06	5.26 %	1.28	113.53	113.75	0.20 %	1.81
c2-3t-8p	160.73	168.46	4.81 %	1.28	89.18	89.42	0.26 %	2.31
c3-3t-9p	137.18	148.40	8.18 %	1.50	60.61	60.55	0.10 %	3.39
c4-3t-13p	98.49	101.43	2.99 %	2.09	53.23	53.02	0.40 %	3.87
c5-3t-14p	89.88	94.30	4.91 %	2.29	51.35	54.90	6.92 %	4.01
Tamaño de problema: **2000**								
	Distribución Uniforme				Distribución Proporcional			
Configuración	Tiempo	Modelo	Error	Aceleración	Tiempo	Modelo	Error	Aceleración
c6-2t-5p	750.91	753.41	0.33 %	0.79	564.79	564.52	0.05 %	1.05
c7-2t-8p	430.07	430.42	0.08 %	1.38	341.21	336.64	1.34 %	1.74
c8-2t-8p	437.72	434.22	0.80 %	1.36	190.65	177.42	6.94 %	3.11
c1-3t-8p	431.72	434.79	0.71 %	1.38	601.38	604.74	0.56 %	0.99
c2-3t-8p	430.59	434.40	0.89 %	1.38	475.72	477.19	0.31 %	1.28
c3-3t-9p	376.99	376.86	0.03 %	1.58	322.77	321.58	0.37 %	1.84
c4-3t-13p	252.34	251.06	0.51 %	2.35	282.24	290.49	2.92 %	2.10
c5-3t-14p	232.97	231.46	0.65 %	2.55	274.68	273.40	0.47 %	2.16

para la distribución proporcional es superior a la alcanzada con la distribución uniforme para cualquiera de los problemas. Para el problema de tamaño 700 la aceleración de la distribución proporcional está próxima a la aceleración máxima. Sin embargo, para el problema de mayor tamaño ($m = 1500$), se encuentra más lejos de su valor máximo. Por otro lado, también se observa, en las dos gráficas, una reducción de las aceleraciones para la configuración *c1-3t-8p*, debido a que en esta configuración no hay ninguna máquina rápida entre los esclavos y estamos comparando con el tiempo del algoritmo secuencial ejecutado sobre una máquina rápida.

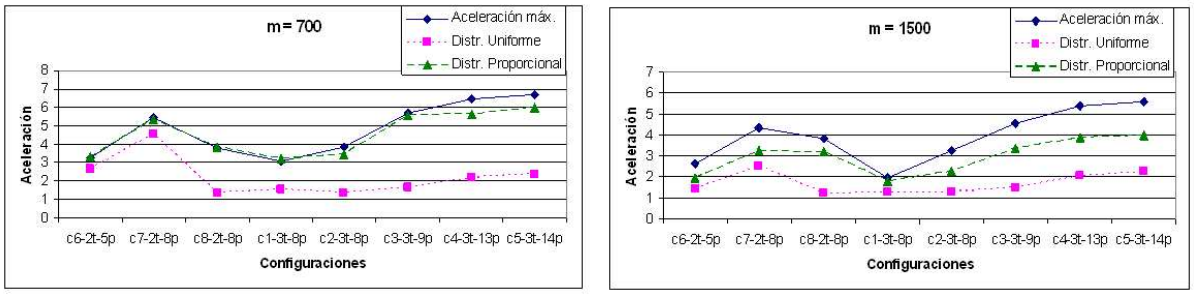

Figura 2.15: Relación entre las aceleraciones reales obtenidas en las ejecuciones de los programas y las aceleraciones máximas, para dos tamaños de problemas ($m = 700$ y $m = 1500$) y las 8 configuraciones de procesadores utilizadas.

En las tablas 2.10 y 2.11 pueden consultarse las distribuciones de trabajo con las que se han realizado las ejecuciones de la tabla 2.9. Para cada ejecución se especifica el número de filas que deben calcular los procesadores rápidos (m_R), los procesadores intermedios (m_I) y los procesadores lentos (m_L). La estructura de estas tablas es la misma que la que presentamos en las tablas 2.6 y 2.7.

En las tablas 2.10 y 2.11 se observa la diferencia en el trabajo asignado a las máquinas intermedias y lentas, cuando realizamos la distribución de trabajo acorde con la capacidad de cómputo de cada una de ellas (distribución proporcional).

2.8. Un método exacto para obtener la distribución óptima

Las capacidades predictivas de los modelos analíticos, que presentamos en las secciones 2.5 y 2.6, podemos utilizarlas para predecir los valores de los parámetros para una ejecución óptima, es decir, para calcular la cantidad óptima de trabajo que debemos asignar a cada esclavo para conseguir el mínimo tiempo de ejecución posible. Siendo más precisos, tenemos que encontrar los tamaños $m_1, \ldots, m_p$ de las tareas $trabajo_1, \ldots, trabajo_p$ donde las funciones de complejidad analítica (T_{par}) alcanzan el valor mínimo.

Una aproximación analítica que minimice esta función puede resultar un proceso extremadamente complejo. En su lugar proponemos métodos algorítmicos que nos permitan obtener el mínimo. En esta sección se describe el método exacto para resolver el problema de optimización general.

Podemos modelizar el problema de encontrar la distribución óptima de trabajo como un problema de asignación de recursos (*RAP*, *Resource Allocation Problem*). Un problema de asignación de recursos clásico ([85, 104]) consiste en asignar una cantidad de recursos limitados a un conjunto de actividades, para maximizar su efectividad o para minimizar el costo. El problema más simple incluye un único tipo de recurso discreto. Se asume que se dispone de M unidades de un recurso indivisible y de N actividades que consumen el recurso. Para cada actividad i, la función $f_i(x)$ proporciona el beneficio (o coste) obtenido al asignar a la actividad i una cantidad x del recurso. Matemáticamente este problema se formula de la siguiente manera:

Tabla 2.10: Distribuciones uniformes y proporcionales de trabajo para los subsistemas heterogéneos del *cluster* utilizando diferentes configuraciones para el problema del producto de matrices, utilizando matrices cuadradas de tamaños 700 y de 1000.

	Tamaño de problema: **700**		Tamaño de problema: **1000**	
Configuración	Distribución Uniforme	Distribución Proporcional	Distribución Uniforme	Distribución Proporcional
c6-2t-5p	$m_R = 175$ $m_I = 175$	$m_R = 219$ $m_I = 131$	$m_R = 250$ $m_I = 250$	$m_R = 312$ $m_I = 187$, i = 3 $m_I = 189$, i = 4
c7-2t-8p	$m_R = 100$ $m_I = 100$	$m_R = 130$ $m_I = 78$, i = 4..6 $m_I = 76$, i = 7	$m_R = 143$ $m_I = 143$, i = 4..6 $m_I = 142$, i = 7	$m_R = 185$ $m_I = 111$, i = 4..6 $m_I = 112$, i = 7
c8-2t-8p	$m_R = 100$ $m_L = 100$	$m_R = 184$ $m_L = 37$	$m_R = 143$ $m_L = 143$, i = 4..6 $m_L = 142$, i = 7	$m_R = 263$ $m_L = 53$, i = 4..6 $m_L = 52$, i = 7
c1-3t-8p	$m_I = 100$ $m_L = 100$	$m_I = 140$ $m_L = 47$, i = 5..6 $m_L = 46$, i = 7	$m_I = 143$ $m_L = 143$, i = 4..6 $m_L = 142$, i = 7	$m_I = 200$ $m_L = 67$, i = 5..6 $m_L = 66$, i = 7
c2-3t-8p	$m_R = 100$ $m_I = 100$ $m_L = 100$	$m_R = 184$ $m_I = 111$ $m_L = 37$, i = 5..6 $m_L = 36$, i = 7	$m_R = 143$ $m_I = 143$ $m_L = 143$, i = 5..6 $m_L = 142$, i = 7	$m_R = 263$ $m_I = 158$ $m_L = 53$, i = 5..6 $m_L = 52$, i = 7
c3-3t-9p	$m_R = 88$ $m_I = 88$, i = 4..5 $m_I = 87$, i = 6..7 $m_I = 87$	$m_R = 125$ $m_I = 75$ $m_L = 25$	$m_R = 125$ $m_I = 125$ $m_L = 125$	$m_R = 179$ $m_I = 107$ $m_L = 35$
c4-3t-13p	$m_R = 59$ $m_I = 59$, i = 4 $m_I = 59$, i = 5..7 $m_L = 58$	$m_R = 109$ $m_I = 66$ $m_L = 22$, i = 8..11 $m_L = 21$, i = 12	$m_R = 84$ $m_I = 84$, i = 4 $m_I = 83$, i = 5..7 $m_L = 83$	$m_R = 156$ $m_I = 94$ $m_L = 31$, i = 8..11 $m_L = 32$, i = 12
c5-3t-14p	$m_R = 54$ $m_I = 54$ $m_L = 54$, i = 8..11	$m_R = 106$ $m_I = 64$ $m_L = 21$	$m_R = 77$ $m_I = 77$ $m_L = 77$, i = 8..12 $m_L = 76$, i = 13	$m_R = 152$ $m_I = 91$ $m_L = 30$

$$\text{mín} \sum_{i=1}^{N} f_i(x_i)$$

$$sujeto\ a \sum_{i=1}^{N} x_i = M$$

$$y\ a \leq x_i$$

donde a es el número mínimo de unidades de recurso que pueden ser asignados a cualquier actividad.

Una forma eficiente de encontrar soluciones óptimas a este problema de optimización consiste en utilizar un algoritmo de programación dinámica.

Un esquema maestro-esclavo puede ser formulado como un problema de asignación de recursos. Se considera que las unidades de recurso en el problema son, ahora, los tamaños de los subproblemas (m_i) que pueden ser asignados a un procesador p_i y las actividades están representadas por los procesadores ($N = p$). La asignación de una unidad de recurso a un procesador significa asignarle una tarea de tamaño uno. La función de costo puede ser expresada en términos de complejidad analítica.

Tabla 2.11: Distribuciones uniformes y proporcionales de trabajo para los subsistemas heterogéneos del *cluster* utilizando diferentes configuraciones para distintos tamaños del problema del producto de matrices, utilizando matrices cuadradas de tamaños 1500 y 2000.

	Tamaño de problema: **1500**		Tamaño de problema: **2000**	
Configuración	Distribución Uniforme	Distribución Proporcional	Distribución Uniforme	Distribución Proporcional
c6-2t-5p	$m_R = 375$ $m_I = 375$	$m_R = 469$ $m_I = 281$	$m_R = 500$ $m_I = 500$	$m_R = 625$ $m_I = 375$
c7-2t-8p	$m_R = 215$, i = 1..2 $m_R = 214$, i = 3 $m_I = 214$	$m_R = 278$ $m_I = 167$, i = 4..6 $m_I = 165$, i = 7	$m_R = 286$ $m_I = 286$, i = 4..5 $m_I = 285$, i = 6..7	$m_R = 370$ $m_I = 222$, i = 4..6 $m_I = 224$, i = 7
c8-2t-8p	$m_R = 215$, i = 1..2 $m_R = 214$, i = 3 $m_L = 214$	$m_R = 395$ $m_L = 79$, i = 4..6 $m_L = 78$, i = 7	$m_R = 286$ $m_L = 286$, i = 4..5 $m_L = 285$, i = 6..7	$m_R = 526$ $m_L = 105$, i = 4..6 $m_L = 107$, i = 7
c1-3t-8p	$m_I = 215$, i = 1..2 $m_I = 214$, i = 3..4 $m_L = 214$	$m_I = 300$ $m_L = 100$	$m_I = 286$ $m_L = 286$, i = 5 $m_L = 285$, i = 6..7	$m_I = 400$ $m_L = 133$, i = 5..6 $m_L = 134$, i = 7
c2-3t-8p	$m_R = 215$ $m_I = 214$ $m_L = 214$	$m_R = 395$ $m_I = 237$ $m_L = 79$, i = 5..6 $m_L = 78$, i = 7	$m_R = 286$ $m_I = 286$ $m_L = 286$, i = 5 $m_L = 285$, i = 6..7	$m_R = 526$ $m_I = 316$ $m_L = 105$, i = 5..6 $m_L = 106$, i = 7
c3-3t-9p	$m_R = 188$ $m_I = 188$, i = 4 $m_I = 187$, i = 5..7 $m_L = 187$	$m_R = 268$ $m_I = 161$ $m_L = 52$	$m_R = 250$ $m_I = 250$ $m_L = 250$	$m_R = 357$ $m_I = 214$ $m_L = 73$
c4-3t-13p	$m_R = 125$ $m_I = 125$ $m_L = 125$	$m_R = 234$ $m_I = 141$ $m_L = 47$, i = 8..11 $m_L = 46$, i = 12	$m_R = 167$ $m_I = 167$ $m_L = 167$, i = 8 $m_L = 166$, i = 9..12	$m_R = 312$ $m_I = 187$ $m_L = 62$, i = 8..11 $m_L = 68$, i = 12
c5-3t-14p	$m_R = 116$ $m_I = 116$, i = 4..5 $m_I = 115$, i = 6..7 $m_L = 115$	$m_R = 227$ $m_I = 136$ $m_I = 45$, i = 8..12 $m_L = 50$, i = 13	$m_R = 154$ $m_I = 154$ $m_L = 154$, i = 8..11 $m_L = 153$, i = 12..13	$m_R = 303$ $m_I = 182$ $m_L = 61$, i = 8..12 $m_L = 58$, i = 13

Para el esquema FIFO (figura 2.2-a y sección 2.5) la función de coste se puede expresar como $f_1 = T_1$ y $f_i = S_i + \text{máx}(0, d_i)$. Esto significa que $\sum_{i=1}^{p} f_i = T_p = T_{par}$. El problema de optimización es:

$$\text{mín}\, T_{par} = T_p$$

$$sujeto\ a \sum_{i=1}^{p} m_i = m$$

$$y\ 0 \leq m_i$$

Esto es, minimizar el tiempo total de cómputo sujeto a que la suma de los tamaños de los subproblemas sea el tamaño del total del problema; es decir, se haya resuelto el problema completo.

Para la estrategia LIFO (figura 2.2-b y sección 2.6), la función de coste se define como $f_1 = \phi_p$ y $f_i = R_i + \text{máx}(C_i, f_{i-1}) + S_i$, con $f_p = \phi_1 = T_1$. El problema de optimización lo expresamos como:

$$\text{mín}\, T_{par} = T_1$$

$$sujeto\ a \sum_{i=1}^{p} m_i = m$$

$$y\ 0 \leq m_i$$

Modelizando el problema de este modo, minimizamos el tiempo total de ejecución del algoritmo paralelo sobre p procesadores (T_p para la estrategia FIFO y T_1 para LIFO). También aseguramos que la cantidad total de trabajo ha sido distribuida entre todo el conjunto de procesadores ($\sum_{i=1}^{p} m_i = m$). Una solución de este problema de optimización nos proporciona una distribución óptima para un problema maestro-esclavo. La asignación de un valor $m_i = 0$ al procesador p_i implica que utilizar el procesador p_i no reduce el tiempo de ejecución, por lo que se elimina del conjunto de procesadores a utilizar.

Para resolver este problema de optimización hemos utilizado una estrategia de programación dinámica. La **Programación Dinámica** es una importante técnica de resolución de problemas que puede ser aplicada a muchos problemas de optimización combinatoria. Se define como un método de diseño de algoritmos que puede ser empleado cuando la solución a un problema se puede obtener como el resultado de una secuencia de decisiones. El método comprueba todas las secuencias de decisiones y selecciona la mejor de ellas. El enfoque de la programación dinámica [103] consiste en evitar el cálculo de la misma solución más de una vez, para ello todas las soluciones de los diferentes subproblemas se almacenan en una estructura de datos para su uso posterior (se sacrifica espacio para ganar tiempo). La programación dinámica es una técnica ascendente: se comienza con los subproblemas más pequeños y, combinando sus soluciones, se obtienen los resultados de subproblemas cada vez mayores, hasta resolver el problema completo. Es necesario elegir correctamente el orden en que se resuelven, puesto que al calcular un subproblema los resultados de aquellos que sean necesarios deben estar ya almacenados. Este método está soportado por un conjunto de ecuaciones de recurrencia, de los que se deriva la solución al problema. En el caso que nos ocupa podemos obtener las ecuaciones de recurrencia del siguiente modo: denotamos por $G[i][x]$ al mínimo tiempo de ejecución obtenido cuando se utilizan los primeros i procesadores para resolver el subproblema de tamaño x, $i = 1, \ldots, p$ y $x = 1, \ldots, m$. El resultado óptimo de un problema maestro-esclavo de tamaño m con p procesadores se obtiene en $G[p][m]$.

Las ecuaciones de recurrencia utilizadas para calcular los valores de $G[i][x]$ para la estrategia FIFO son:

$$G[i][x] = \text{mín}\{G[i-1][x-j] + f_i(j)/0 < j \leq x\}, i = 2, \ldots, p,$$

$$G[1][x] = f_1(x), 0 < x \leq m,$$

$$G[i][x] = 0, i = 1, \ldots, p; x = 0.$$

Mientras que para la estrategia LIFO, las fórmulas que aplicamos son:

$$G[i][x] = \text{mín}\{f_i(j, G[i-1][x-j])/0 < j \leq x\}, i = 2, \ldots, p,$$

$$G[1][x] = f_1(x, 0), 0 < x \leq m,$$

$$G[i][x] = 0, i = 1, \ldots, p; x = 0.$$

Estas ecuaciones de recurrencia permiten obtener un algoritmo de programación dinámica de complejidad $O(pm^2)$. Utilizando la aproximación presentada en [85] este algoritmo puede ser fácilmente paralelizado usando p procesadores con una complejidad $O(p + m^2)$.

La formulación que presentamos es general, puesto que no se está haciendo ninguna consideración especial sobre los diferentes tipos de procesadores a utilizar para resolver el problema. La complejidad es la misma si el sistema es homogéneo o si es un sistema heterogéneo formado por procesadores de distintas capacidades de cómputo o de comunicación. El usuario únicamente necesita ejecutar este algoritmo proporcionando como parámetros de entrada las características del conjunto de procesadores y del problema considerado.

Una ventaja importante de la programación dinámica es que permite realizar un análisis de sensitividad, lo que permite comprobar el efecto de asignar una unidad de recurso a un procesador. Este hecho

permite que el número de procesadores para una ejecución óptima sea obtenido como una aplicación directa de este análisis. El conjunto de procesadores a ser utilizados en la resolución de un problema está determinado por la cantidad de tarea asignada a cada uno. Los procesadores que reciban una cantidad 0 de trabajo para procesar son excluidos del cómputo. El número óptimo de procesadores, p_{optimo}, puede ser definido como el número de procesadores que se mantienen en este conjunto. En un entorno heterogéneo podemos considerar el conjunto total de procesadores en el sistema cuando ejecutamos el algoritmo de programación dinámica; el resultado que obtenemos nos indica cuáles de los procesadores debemos utilizar para resolver el problema.

Una característica importante de esta aproximación es que, a partir de la resolución de un problema de tamaño m con p procesadores, disponemos de toda la tabla de resultados. Esto nos permite determinar la distribución y el número de procesadores óptimos para cualquier problema de tamaño inferior a m con un máximo de p procesadores.

2.9. Metodología aplicada

Realizamos una experiencia computacional para verificar que nuestra metodología puede ser aplicada satisfactoriamente e introduce una mejora en la resolución de problemas mediante algoritmos maestro-esclavo. La figura 2.16 muestra la metodología aplicada para la paralelización de estos algoritmos.

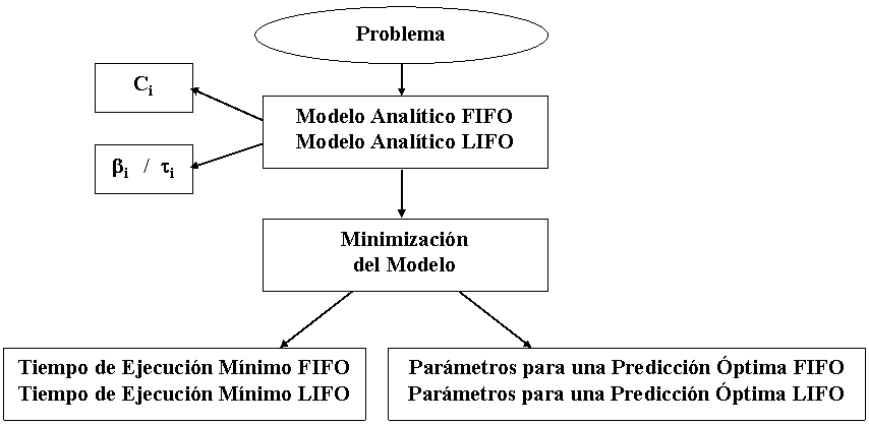

Figura 2.16: Metodología aplicada en la paralelización de algoritmos maestro-esclavo.

Hemos desarrollado una herramienta que encuentra los parámetros óptimos para una ejecución óptima, usando las dos estrategias (FIFO y LIFO) en nuestro *cluster* (sección 1.5.1). Los modelos analíticos que predicen los tiempos de ejecución de nuestras aplicaciones (secciones 2.5 y 2.6) son minimizados usando el procedimiento de programación dinámica descrito en la sección 2.8. La herramienta es alimentada con todos los parámetros de la arquitectura necesarios para el modelo analítico.

A continuación presentamos las experiencias computacionales que hemos realizado para validar las estrategias propuestas y obtener las distribuciones óptimas para dos problemas diferentes: el producto de matrices y la Transformada Rápida de Fourier bidimensional. En la sección 2.7 ya ha sido verificado que las predicciones de tiempo de los modelos son correctas y ahora utilizamos el algoritmo de programación dinámica para obtener las distribuciones óptimas para cada una de las estrategias (FIFO y LIFO).

2.10. Predicción de la distribución óptima: Producto de matrices

En la sección 2.7 describimos el problema del producto de matrices, presentamos los códigos para resolverlo y lo utilizamos para validar nuestros modelos en entornos homogéneos y heterogéneos, comprobando que el tiempo que el modelo predice se corresponde con el tiempo real de ejecución del programa. En esta sección vamos a usar el mismo problema para comprobar que con nuestros modelos también podemos obtener los valores de los parámetros que hacen que el tiempo de ejecución necesario para resolver el problema sea mínimo; y presentamos los resultados obtenidos al predecir la distribución de trabajo óptima entre los procesadores según los dos esquemas de resolución (FIFO y LIFO).

Siguiendo el mismo procedimiento que utilizamos en la sección 2.7, vamos a utilizar inicialmente como plataforma de pruebas el subsistema formado únicamente por las máquinas intermedias y lentas del *cluster* (sección 1.5.1). Posteriormente incluiremos la máquina de memoria compartida, disponiendo entonces en nuestra plataforma también de los 4 procesadores rápidos.

Sin embargo, antes de comenzar con las ejecuciones para predecir las distribuciones óptimas con las que debemos resolver los problemas, hemos realizado un pequeño experimento para demostrar que, mediante el análisis de sensitividad, no solamente obtenemos la distribución óptima, sino que también vamos a determinar el número óptimo de procesadores que debemos emplear en la resolución del problema. Esto nos permite ejecutar el método exacto que minimiza nuestros modelos analíticos considerando el *cluster* completo y que el método seleccione los procesadores a utilizar.

2.10.1. Análisis de sensitividad

Realizamos el análisis de sensitividad utilizando como ejemplo un problema de tamaño pequeño ($m = 100$) ejecutado mediante la estrategia FIFO. Suponiendo que disponemos de un *cluster* con 10 procesadores intermedios y 10 procesadores lentos, vamos a aplicar el método exacto para saber cuántos procesadores son realmente necesarios para resolver el problema en el menor tiempo posible, si consideramos distintos subsistemas. En la tabla 2.12 se encuentran los resultados que hemos obtenido en este análisis de sensitividad, incluyendo la información sobre el subsistema, la distribución y el tiempo de ejecución estimado por el modelo.

Tabla 2.12: Análisis de sensitividad para un problema pequeño: $m = 100$.

Procesadores	Distribución	Tiempo Predecido
10 Intermedios	$m_I = 10$	0.03891
10 Intermedios 1 Lento	$m_I = 10$ $m_L = 0$	0.03891
10 Intermedios 10 Lentos	$m_I = 9$ $m_S = 5$, i = 1..2 $m_S = 0$, i = 3..10	0.03869

Inicialmente consideramos el subsistema formado únicamente por los 10 procesadores intermedios (primera fila de la tabla). En este caso, al ser un sistema homogéneo, se asigna a cada procesador la misma cantidad de trabajo (10 filas de la matriz A). A continuación, en la segunda fila, añadimos al sistema homogéneo anterior un único procesador lento. Como se observa en la tabla, el procesador lento no recibe ninguna carga de trabajo, esto significa que utilizarlo no reduce el tiempo de ejecución del programa; por lo que resolvemos el problema sobre el sistema homogéneo de las máquinas intermedias. Por último, en la tercera fila, se considera el cluster completo de procesadores. Ahora el algoritmo de

programación dinámica asigna trabajo a todos los procesadores intermedios y solamente a dos de los procesadores lentos. Esto significa que la mejor opción para resolver el problema con el *cluster* disponible consiste en no utilizar 8 de los 10 procesadores lentos

Con este pequeño ejemplo ilustramos que el método empleado en el cálculo de la mejor distribución de tareas para ejecutar un problema, nos permite también conocer el número óptimo de procesadores. Aquellos procesadores a los que se les asigna una cantidad de trabajo igual a 0, deben ser eliminados de la ejecución porque su uso no aporta ningún beneficio.

2.10.2. Predicción de la distribución óptima de trabajos con 2 tipos de procesadores

En esta sección vamos a comparar los resultados presentados en la sección 2.7 de las distribuciones uniforme y proporcional, con los tiempos de ejecución obtenidos al resolver los problemas utilizando las distribuciones calculadas al aplicar el método exacto (sección 2.8) para las estrategias FIFO y LIFO. En estas distribuciones hemos tenido en cuenta las diferencias entre las máquinas, considerando valores distintos de velocidades de cómputo y de las comunicaciones entre cada par de procesadores. Al ejecutar el programa con las dos distribuciones, podemos concluir cuál de las dos estrategias es mejor para resolver este problema en cada caso.

Primero validamos el tiempo que predice el modelo (tabla 2.13) con las dos estrategias. Las ejecuciones se realizan sobre las 5 configuraciones de procesadores definidas en la tabla 2.4 de la sección 2.7.4 (columna *Configuración*). Comparamos el tiempo estimado con el tiempo real empleado para resolver el problema y calculamos el error cometido por los modelos. El formato de la tabla 2.13 es idéntico al de la tabla 2.5, únicamente hemos eliminado la columna *Aceleración*, que se presentará en una tabla posterior.

En la tabla 2.13 demostramos que nuestros modelos predicen perfectamente el tiempo real de resolución de este problema tanto para la estrategia FIFO como para la estrategia LIFO. Si observamos los valores de los errores cometidos, solamente en 2 de las 40 ejecuciones el error supera el 2 % (para la configuración *c5-2t-10p* en el problema más pequeño para las dos estrategias).

A continuación, comparamos los tiempos reales de ejecución (en segundos) y las aceleraciones obtenidas para las cuatro distribuciones (uniforme, proporcional, óptima para el esquema FIFO y óptima para el esquema LIFO) Los tiempos de ejecución de las cuatro distribuciones ya han sido presentados en tablas anteriores: los tiempos correspondientes a las distribuciones uniforme y proporcional en la tabla 2.5 de la sección 2.7.4; y los tiempos de las distribuciones óptimas FIFO y LIFO en la tabla 2.13 en esta misma sección. Sin embargo, la tabla 2.14 considera nuevamente dichos valores para facilitar el estudio comparativo de los resultados. El tiempo secuencial de referencia para calcular la aceleración alcanzada es siempre el correspondiente al procesador maestro (en este caso el tiempo secuencial de un procesador intermedio).

En la tabla 2.14 no se observan diferencias significativas entre los resultados obtenidos con las distribuciones óptimas para las dos estrategias, aunque en la mayoría de los casos el tiempo empleado es ligeramente inferior para el esquema LIFO. Por otro lado, comparando los resultados para la estrategia FIFO comprobamos que las ejecuciones para las distribuciones óptimas son mucho más eficientes que las ejecuciones para las distribuciones uniforme y proporcional, excepto para el problema de tamaño 2000. Para este problema, la distribución óptima FIFO sigue siendo mejor que la distribución uniforme, pero la diferencia entre ellas es mínima. Este hecho se debe al comportamiento de las máquinas intermedias para este tamaño de problema (figura 2.13), que convierte al sistema prácticamente en un entorno homogéneo, de modo que la distribución uniforme está muy cerca de la distribución óptima. La figura 2.17 muestra la mejora relativa conseguida al ejecutar con nuestra distribución óptima FIFO frente a los resultados correspondientes a la distribución proporcional para dos problemas ($m = 700$ y $m = 1500$). No se incluye la relación con el tiempo de la distribución LIFO debido a la similitud entre los tiempos de las dos distribuciones óptimas. En la figura se comprueba gráficamente que en todos los casos es mejor considerar la heterogeneidad en las comunicaciones.

Tabla 2.13: Validación del modelo: comparación del tiempo estimado por el modelo para las distribuciones óptimas en las dos estrategias de resolución con los tiempos reales de ejecución, sobre subsistemas heterogéneos del *cluster*, utilizando únicamente las máquinas intermedias y lentas.

Tamaño de problema: **700**						
	Óptima FIFO			Óptima LIFO		
Configuración	Tiempo	Modelo	Error	Tiempo	Modelo	Error
c1-2t-3p	23.37	23.25	0.53 %	23.29	23.24	0.20 %
c2-2t-5p	11.80	11.70	0.86 %	11.78	11.71	0.58 %
c3-2t-5p	9.30	9.24	0.65 %	9.25	9.23	0.18 %
c4-2t-8p	7.41	7.33	1.16 %	7.34	7.32	0.24 %
c5-2t-10p	7.00	6.44	7.98 %	6.87	6.43	6.31 %
Tamaño de problema: **1000**						
	Óptima FIFO			Óptima LIFO		
Configuración	Tiempo	Modelo	Error	Tiempo	Modelo	Error
c1-2t-3p	78.79	77.86	1.18 %	78.31	77.82	0.63 %
c2-2t-5p	38.93	39.14	0.54 %	39.53	39.19	0.87 %
c3-2t-5p	30.41	30.57	0.52 %	30.58	30.58	0.00 %
c4-2t-8p	25.11	24.66	1.78 %	24.96	24.65	1.27 %
c5-2t-10p	22.17	21.89	1.25 %	21.97	21.83	0.67 %
Tamaño de problema: **1500**						
	Óptima FIFO			Óptima LIFO		
Configuración	Tiempo	Modelo	Error	Tiempo	Modelo	Error
c1-2t-3p	382.27	381.19	0.28 %	379.82	381.18	0.36 %
c2-2t-5p	195.72	192.03	1.89 %	190.10	192.02	1.01 %
c3-2t-5p	164.76	163.54	0.74 %	163.19	163.65	0.28 %
c4-2t-8p	116.73	116.74	0.01 %	116.75	116.81	0.06 %
c5-2t-10p	99.61	98.63	0.98 %	98.03	98.55	0.53 %
Tamaño de problema: **2000**						
	Óptima FIFO			Óptima LIFO		
Configuración	Tiempo	Modelo	Error	Tiempo	Modelo	Error
c1-2t-3p	1533.83	1540.86	0.46 %	1533.23	1540.24	0.46 %
c2-2t-5p	753.23	764.82	1.54 %	754.85	765.20	1.37 %
c3-2t-5p	757.89	760.26	0.31 %	759.15	759.84	0.09 %
c4-2t-8p	431.11	433.30	0.51 %	431.21	433.26	0.48 %
c5-2t-10p	334.12	334.50	0.11 %	334.69	334.46	0.07 %

La tabla 2.14 también muestra las aceleraciones obtenidas. En la figura 2.18 comparamos gráficamente la aceleración que se obtiene con las distribuciones proporcional y óptima de la estrategia FIFO y la distribución óptima para la estrategia LIFO con las distribuciones máximas alcanzables en el sistema (tabla 2.4) para dos de los problemas ejecutados ($m = 700$ y $m = 1500$).

Las aceleraciones reales obtenidas con las distribuciones óptimas, se acercan mucho a las aceleraciones máximas que se pueden alcanzar, llegando incluso a la superlinealidad en algunos casos. Esta superlinealidad se debe a efectos debidos a la relación entre los tamaños de problemas y el tamaño de la memoria caché de las máquinas al ejecutar el algoritmo secuencial (sección 1.4). En la figura 2.18 la línea correspondiente a la distribución proporcional se encuentra en todo momento por debajo de las distribuciones óptimas.

Las asignaciones de trabajo que hemos utilizado en las distribuciones óptimas FIFO y LIFO se muestran en las tablas 2.15 y 2.16. Las asignaciones correspondientes a las distribuciones uniforme y proporcional se pueden consultar en la sección 2.7.4 (tablas 2.6 y 2.7). En las tablas se representan por m_I y

Tabla 2.14: Resultados del algoritmo paralelo para el producto de matrices al ejecutar distintas distribuciones de trabajo: uniforme, proporcional y óptimo para la estrategia FIFO y óptimo para la estrategia LIFO. Todas las ejecuciones se realizan sobre subsistemas heterogéneos del *cluster* utilizando únicamente las máquinas intermedias y lentas.

Tamaño de problema: **700**								
	Uniforme		Proporcional		Óptima FIFO		Óptima LIFO	
Configuración	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.
c1-2t-3p	52.28	0.63	26.48	1.24	23.37	1.41	23.29	1.41
c2-2t-5p	26.20	1.25	12.89	2.55	11.80	2.79	11.78	2.79
c3-2t-5p	26.71	1.23	11.55	2.84	9.30	3.53	9.25	3.55
c4-2t-8p	14.58	2.25	7.85	4.19	7.41	4.43	7.34	4.48
c5-2t-10p	11.67	2.82	7.13	4.60	7.00	4.69	6.86	4.78
Tamaño de problema: **1000**								
	Uniforme		Proporcional		Óptima FIFO		Óptima LIFO	
Configuración	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.
c1-2t-3p	167.38	0.59	88.19	1.12	78.79	1.26	78.31	1.27
c2-2t-5p	86.28	1.15	43.83	2.26	38.93	2.55	39.53	2.51
c3-2t-5p	85.48	1.16	36.70	2.70	30.41	3.26	30.58	3.24
c4-2t-8p	52.53	1.89	27.29	3.63	25.11	3.95	24.96	3.97
c5-2t-10p	42.70	2.32	24.88	3.98	22.17	4.47	21.97	4.51
Tamaño de problema: **1500**								
	Uniforme		Proporcional		Óptima FIFO		Óptima LIFO	
Configuración	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.
c1-2t-3p	557.37	1.10	421.98	1.46	382.27	1.61	379.82	1.62
c2-2t-5p	279.85	2.20	210.73	2.92	195.72	3.15	190.10	3.24
c3-2t-5p	297.11	2.07	168.86	3.65	164.76	3.74	163.19	3.77
c4-2t-8p	165.84	3.71	132.32	4.65	116.73	5.28	116.75	5.28
c5-2t-10p	134.98	4.56	112.80	5.46	99.61	6.18	98.03	6.28
Tamaño de problema: **2000**								
	Uniforme		Proporcional		Óptima FIFO		Óptima LIFO	
Configuración	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Aceleración
c1-2t-3p	1535.10	1.43	2257.71	0.97	1534.32	1.43	1533.23	1.43
c2-2t-5p	753.50	2.92	1130.33	1.94	753.23	2.92	754.85	2.91
c3-2t-5p	760.33	2.89	903.95	2.43	757.89	2.90	759.15	2.90
c4-2t-8p	431.84	5.09	696.21	3.16	431.11	5.10	431.21	5.10
c5-2t-10p	336.73	6.53	603.69	3.64	334.12	6.58	334.69	6.57

m_L la cantidad de filas asignadas a los procesadores intermedios y lentos respectivamente. Cuando la cantidad asignada a todos los procesadores del mismo tipo es la misma, incluimos en la tabla un único valor. Si existen valores distintos en la distribución FIFO en la tabla aparece una fila por cada valor y se indica a qué procesadores se les asigna el valor mediante la expresión i = *índice de los procesadores*. En el caso de la estrategia LIFO, al poder ser todas las cantidades asignadas diferentes, en la tabla se especifican escribiendo una única vez el tipo de procesador (m_I o m_L) y a continuación una lista con los valores asignados a todos los procesadores de ese tipo. Por ejemplo, para el problema de tamaño 700 y la configuración *c4-2t-5p*, las cantidades que aparecen en la distribución óptima FIFO significan que se le asignan 167 filas de la matriz A a los tres procesadores intermedios; y de los 4 procesadores lentos de la configuración, los tres primeros reciben 50 filas, mientras que al último procesador se le asignan solamente 49. En la distribución para el esquema LIFO y la misma configuración aparece m_I = *171 169 166* y m_L = *49 49 48 48*; esto indica que las cantidades asignadas son: $m_1 = 171, m_2 = 169$ y $m_3 = 166$ (los tres esclavos intermedios), y para los esclavos lentos: $m_4 = 49, m_5 = 49, m_6 = 48$ y $m_9 = 48$.

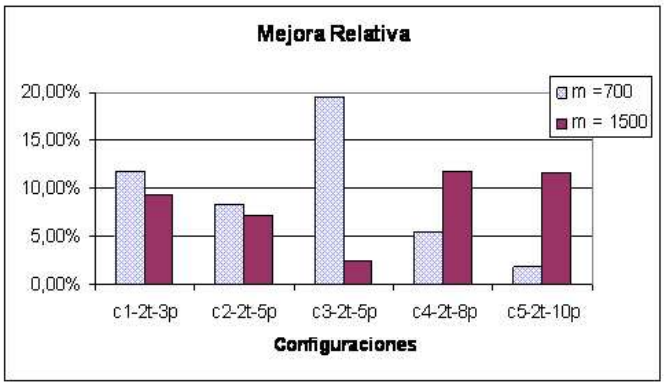

Figura 2.17: Mejora relativa obtenida al resolver el problema con la distribución óptima FIFO frente a la distribución proporcional, para dos problemas ($m = 700$ y $m = 1500$) y las 5 configuraciones utilizadas.

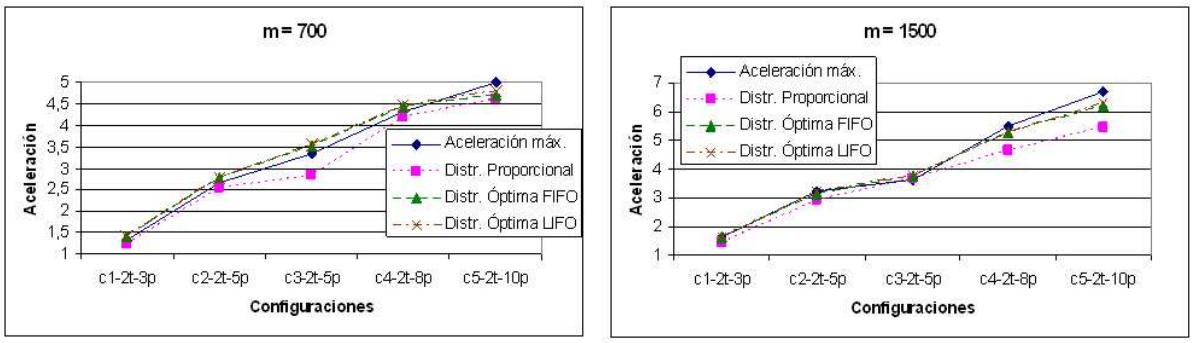

Figura 2.18: Comparativa de las aceleraciones conseguidas al ejecutar el problema con la distribución proporcional, la distribución óptima FIFO y la distribución óptima LIFO con las aceleraciones máximas, para dos problemas ($m = 700$ y $m = 1500$) y las 5 configuraciones utilizadas.

En las tablas 2.15 y 2.16 el número de filas asignadas a los procesadores intermedios, en cada caso, es muy superior al trabajo asignado a los procesadores lentos. Para el problema de mayor tamaño, nuestros modelos tienen en cuenta la reducción del rendimiento en las máquinas intermedias, y la diferencia entre el número de filas que tienen que computar los procesadores disminuye, o incluso llega a ser superior para los procesadores lentos (configuración *c5-2t-10p*). En estas tablas también incluimos el tiempo que se tarda en ejecutar el algoritmo exacto que calcula las distribuciones óptimas. Teniendo en cuenta la reducción que se obtiene en los tiempos de ejecución real, el tiempo empleado en la obtención de la distribución óptima es asumible para la resolución del problema, puesto que considerando la suma de estos dos tiempos, en la mayoría de los casos, aún se reduce el tiempo empleado al ejecutar utilizando la distribución proporcional. Existen algunos casos donde esto no sucede: las configuraciones *c4-2t-8p* y *c5-2t-10p* para el problema más pequeño. Esto es debido a que al incrementar el número de procesadores, el tiempo necesario para ejecutar el método exacto también aumenta, aunque en estos dos casos siga siendo menor de un segundo. Sin embargo, el problema es muy pequeño, por lo que un segundo supone un porcentaje importante respecto al tiempo total de ejecución del programa. Debido al bajo tiempo de ejecución en los problemas pequeños no puede haber una reducción importante; mientras que en los problemas grandes sí es posible reducir el tiempo invertido en la resolución del problema.

Tabla 2.15: Distribuciones óptimas FIFO y LIFO calculadas por el modelo y tiempo empleado para ejecutar el método exacto que las calcula para los subsistemas heterogéneos del *cluster*, utilizando diferentes configuraciones de máquinas intermedias y lentas para el problema del producto de matrices, utilizando matrices cuadradas de tamaños 700 y 1000.

Tamaño de problema: **700**				
	Óptima FIFO		Óptima LIFO	
Configuración	Distribución	Tiempo	Distribución	Tiempo
c1-2t-3p	$m_I = 543$ $m_L = 157$	0.11	$m_I = 544$ $m_L = 156$	0.10
c2-2t-5p	$m_I = 271$ $m_L = 79$	0.40	$m_I = 274\ 270$ $m_L = 78$	0.31
c3-2t-5p	$m_I = 213$, i = 1..2 $m_I = 212$, i = 3 $m_L = 62$	0.36	$m_I = 216\ 213\ 210$ $m_L = 61$	0.30
c4-2t-8p	$m_I = 167$ $m_L = 50$, i = 4..6 $m_L = 49$, i = 7	0.71	$m_I = 171\ 169\ 166$ $m_L = 49\ 49\ 48\ 48$	0.61
c5-2t-10p	$m_I = 146$ $m_L = 44$, i = 4..7 $m_L = 43$, i = 8..9	0.94	$m_I = 150\ 148\ 146$ $m_L = 43\ 43\ 43\ 43\ 42\ 42$	0.81
Tamaño de problema: **1000**				
	Óptima FIFO		Óptima LIFO	
Configuración	Distribución	Tiempo	Distribución	Tiempo
c1-2t-3p	$m_I = 785$ $m_L = 215$	0.24	$m_I = 786$ $m_L = 214$	0.21
c2-2t-5p	$m_I = 393$ $m_L = 107$	0.72	$m_I = 396\ 392$ $m_L = 106$	0.62
c3-2t-5p	$m_I = 306$, i = 1..2 $m_I = 305$, i = 3 $m_L = 83$	0.72	$m_I = 308\ 306\ 304$ $m_L = 61$	0.62
c4-2t-8p	$m_I = 246$, i = 1 $m_I = 245$, i = 2..3 $m_L = 66$	1.43	$m_I = 248\ 246\ 245$ $m_L = 66\ 65\ 65\ 65$	1.23
c5-2t-10p	$m_I = 217$ $m_L = 59$, i = 4 $m_L = 58$, i =5..9	1.92	$m_I = 220\ 218\ 216$ $m_L = 58\ 58\ 58\ 58\ 57\ 57$	1.68

2.10.3. Predicción de la distribución óptima de trabajos con 3 tipos de procesadores

Por último, vamos a calcular las distribuciones óptimas que obtenemos al aplicar el método exacto a nuestros modelos, cuando introducimos en el *cluster* los procesadores rápidos. Las pruebas realizadas han sido las mismas que cuando consideramos el subsistema de las máquinas intermedias y lentas únicamente (sección 2.10.2): calculamos las distribuciones óptimas para las estrategias FIFO y LIFO, ejecutamos el programa paralelo utilizando estas distribuciones, comprobamos que el tiempo estimado por el modelo coincide con el tiempo real de ejecución, y comparamos los resultados con los presentados en la sección 2.7.5 (distribuciones uniforme y proporcional). Las configuraciones de procesadores, con las que hemos realizado las pruebas, son las mismas de la sección 2.7 y se encuentran especificadas en la tabla 2.8. La tabla 2.17 compara el tiempo que predice el modelo con el tiempo real empleado en la ejecución, calculando el error cometido.

Los resultados que se muestran en la tabla 2.17 nos permiten considerar los modelos validados para

Tabla 2.16: Distribuciones óptimas FIFO y LIFO calculadas por el modelo y tiempo empleado para ejecutar el método exacto que las calcula para los subsistemas heterogéneos del *cluster*, utilizando diferentes configuraciones de máquinas intermedias y lentas para el problema del producto de matrices, utilizando matrices cuadradas de tamaños 1500 y 2000.

Tamaño de problema: **1500**				
	Óptima FIFO		Óptima LIFO	
Configuración	Distribución	Tiempo	Distribución	Tiempo
c1-2t-3p	$m_I = 1004$ $m_L = 496$	0.54	$m_I = 1005$ $m_L = 495$	0.47
c2-2t-5p	$m_I = 506$ $m_L = 244$	1.62	$m_I = 507\ 506$ $m_L = 244\ 243$	1.40
c3-2t-5p	$m_I = 431$ $m_L = 207$	1.64	$m_I = 433\ 431\ 430$ $m_L = 206$	1.42
c4-2t-8p	$m_I = 307$ $m_L = 145$, i = 4..6 $m_L = 144$, i = 7	2.28	$m_I = 309\ 308\ 307$ $m_L = 144$	2.78
c5-2t-10p	$m_I = 260$ $m_L = 120$	4.27	$m_I = 261\ 261\ 260$ $m_L = 120\ 120\ 120\ 120\ 119\ 119$	3.73
Tamaño de problema: **2000**				
	Óptima FIFO		Óptima LIFO	
Configuración	Distribución	Tiempo	Distribución	Tiempo
c1-2t-3p	$m_I = 1021$ $m_L = 979$	0.95	$m_I = 1021$ $m_L = 979$	0.82
c2-2t-5p	$m_I = 507$ $m_L = 493$	2.91	$m_I =508\ 507$ $m_L = 493\ 492$	2.44
c3-2t-5p	$m_I = 504$, i = 1 $m_I = 503$, i = 2..3 $m_L = 490$	2.92	$m_I = 504\ 504\ 503$ $m_L = 489$	2.48
c4-2t-8p	$m_I = 287$ $m_L = 285$, i = 4..6 $m_L = 284$, i = 7	5.86	$m_I = 288\ 288\ 287$ $m_L = 285\ 284\ 284\ 284$	4.89
c5-2t-10p	$m_I = 221$ $m_L = 223$, i = 4..8 $m_L = 222$, i = 9	7.87	$m_I = 222$ $m_L = 223\ 223\ 222\ 222\ 222\ 222$	6.65

estas distribuciones: en 26 de las 64 ejecuciones realizadas, el error cometido es inferior al 2 % y sólo en 12 supera el 8 %. Los resultados para el problema $m = 2000$ ofrecen los peores resultados, ya que el error se sitúa en torno al 11 %.

La tabla 2.18 muestra los tiempos de ejecución reales expresados en segundos y las aceleraciones para las diferentes distribuciones de trabajo. Los datos de las distribuciones uniforme y proporcional ya han sido presentados en la tabla 2.9 (sección 2.7.5), y los resultados correspondientes a las distribuciones óptimas FIFO y LIFO en la tabla 2.17. Volvemos a incluirlos en la tabla 2.18 para poder comparar todos los resultados obtenidos. En estas ejecuciones, el tiempo de referencia para calcular las aceleraciones es el tiempo secuencial de los procesadores rápidos, puesto que en todas las configuraciones es el tipo del procesador que actúa como maestro.

Observamos en la tabla 2.18 que al introducir los procesadores rápidos la diferencia entre las distribuciones óptimas y las distribuciones uniforme y proporcional ha aumentado, siendo menor el tiempo de ejecución al utilizar nuestras distribuciones. También comprobamos que sólo hay 3 casos en los que la distribución óptima FIFO es mejor que la distribución óptima LIFO, aunque en la mayoría de las ejecuciones, se trate de diferencias reducidas.

Tabla 2.17: Validación del modelo: comparación del tiempo estimado por el modelo, para las distribuciones óptimas en las dos estrategias de resolución con los tiempos reales de ejecución, al trabajar con subsistemas heterogéneos del *cluster* con diferentes tamaños de problemas.

Tamaño de problema: **700**						
	Óptima FIFO			Óptima LIFO		
Configuración	Tiempo	Modelo	Error	Tiempo	Modelo	Error
c6-2t-5p	5.94	5.94	0.04 %	5.90	5.90	0.14 %
c7-2t-8p	3.64	3.56	2.19 %	3.60	3.52	2.11 %
c8-2t-8p	5.21	5.18	0.63 %	5.20	5.14	1.31 %
c1-3t-8p	6.24	6.28	0.61 %	6.28	6.28	0.08 %
c2-3t-8p	5.16	5.08	1.37 %	5.14	5.04	1.82 %
c3-3t-9p	3.52	3.45	1.83 %	3.47	3.39	2.12 %
c4-3t-13p	3.19	3.06	4.09 %	3.11	3.01	3.38 %
c5-3t-14p	3.28	2.96	9.60 %	3.14	2.92	7.21 %
Tamaño de problema: **1000**						
	Óptima FIFO			Óptima LIFO		
Configuración	Tiempo	Modelo	Error	Tiempo	Modelo	Error
c6-2t-5p	18.71	18.80	0.47 %	18.70	18.73	0.13 %
c7-2t-8p	11.33	11.24	0.82 %	11.30	11.16	1.23 %
c8-2t-8p	16.97	16.55	2.47 %	16.86	16.48	2.29 %
c1-3t-8p	20.74	20.89	0.73 %	21.04	20.94	0.48 %
c2-3t-8p	16.41	16.37	0.25 %	16.40	16.30	0.66 %
c3-3t-9p	11.08	10.95	1.24 %	11.06	10.85	1.95 %
c4-3t-13p	9.94	9.86	0.77 %	9.87	9.77	1.02 %
c5-3t-14p	9.94	9.62	3.24 %	9.75	9.56	2.00 %
Tamaño de problema: **1500**						
	Óptima FIFO			Óptima LIFO		
Configuración	Tiempo	Modelo	Error	Tiempo	Modelo	Error
c6-2t-5p	86.42	81.09	6.17 %	83.11	80.97	2.58 %
c7-2t-8p	53.46	49.45	7.50 %	52.84	49.26	6.77 %
c8-2t-8p	62.53	61.24	2.06 %	62.14	61.17	1.57 %
c1-3t-8p	105.20	105.40	0.19 %	104.81	105.42	0.58 %
c2-3t-8p	73.55	68.10	7.40 %	71.36	67.97	4.75 %
c3-3t-9p	51.01	47.62	6.65 %	50.92	47.48	6.75 %
c4-3t-13p	45.27	41.95	7.35 %	44.09	41.82	5.15 %
c5-3t-14p	44.89	40.82	9.07 %	43.82	40.54	7.49 %
Tamaño de problema: **2000**						
	Óptima FIFO			Óptima LIFO		
Configuración	Tiempo	Modelo	Error	Tiempo	Modelo	Error
c6-2t-5p	287.58	273.44	4.92 %	285.76	273.22	4.39 %
c7-2t-8p	191.46	170.49	10.95 %	191.32	170.41	10.93 %
c8-2t-8p	188.68	168.15	10.88 %	188.54	167.96	10.92 %
c1-3t-8p	433.28	432.57	0.16 %	433.31	432.73	0.13 %
c2-3t-8p	224.96	212.76	5.42 %	224.32	212.41	5.31 %
c3-3t-9p	178.67	160.69	10.06 %	178.28	160.35	10.06 %
c4-3t-13p	144.41	129.63	10.24 %	143.96	129.38	10.13 %
c5-3t-14p	138.93	123.55	11.07 %	139.07	123.24	11.38 %

Tabla 2.18: Comparación de los tiempos de ejecución del algoritmo paralelo del producto de matrices al ejecutar con las distintas distribuciones de trabajo: uniforme, proporcional y óptima para la estrategia FIFO y óptima para la estrategia LIFO.

Tamaño de problema: **700**								
	Uniforme		Proporcional		Óptima FIFO		Óptima LIFO	
Configuración	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.
6-2t-5p	7.52	2.68	6.09	3.31	5.94	3.39	5.89	3.41
c7-2t-8p	4.44	4.54	3.77	5.35	3.64	5.52	3.60	5.59
c8-2t-8p	14.75	1.36	5.21	3.86	5.21	3.86	5.20	3.87
c1-3t-8p	13.17	1.53	6.25	3.22	6.24	3.22	6.28	3.20
c2-3t-8p	15.02	1.34	5.87	3.43	5.16	3.90	5.14	3.92
c3-3t-9p	12.26	1.64	3.62	5.56	3.52	5.72	3.47	5.80
c4-3t-13p	9.23	2.18	3.56	5.66	3.19	6.31	3.11	6.47
c5-3t-14p	8.54	2.36	3.37	5.98	3.28	6.14	3.14	6.41
Tamaño de problema: **1000**								
	Uniforme		Proporcional		Óptima FIFO		Óptima LIFO	
Configuración	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.
c6-2t-5p	24.75	2.40	19.01	3.12	18.71	3.17	18.70	3.17
c7-2t-8p	14.31	4.15	11.34	5.23	11.33	5.24	11.30	5.25
c8-2t-8p	53.37	1.11	18.59	3.19	16.97	3.50	16.86	3.52
c1-3t-8p	53.08	1.12	24.22	2.45	20.74	2.86	21.04	2.82
c2-3t-8p	49.77	1.19	19.24	3.09	16.41	3.62	16.40	3.62
c3-3t-9p	44.46	1.34	12.63	4.70	11.08	5.36	11.06	5.37
c4-3t-13p	29.86	1.99	11.27	5.27	9.94	5.97	9.87	6.01
c5-3t-14p	28.05	2.12	12.78	4.65	9.94	5.97	9.75	6.09
Tamaño de problema: **1500**								
	Uniforme		Proporcional		Óptima FIFO		Óptima LIFO	
Configuración	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.
c6-2t-5p	140.55	1.46	105.38	1.95	86.42	2.38	83.11	2.48
c7-2t-8p	80.50	2.56	62.86	3.27	53.46	3.84	52.84	3.89
c8-2t-8p	162.42	1.27	64.36	3.20	62.53	3.29	62.14	3.31
c1-3t-8p	160.61	1.28	113.53	1.81	105.20	1.96	104.81	1.96
c2-3t-8p	160.73	1.28	89.18	2.31	73.55	2.80	71.36	2.88
c3-3t-9p	137.18	1.50	60.61	3.39	51.01	4.03	50.92	4.04
c4-3t-13p	98.49	2.09	53.23	3.87	45.27	4.55	44.09	4.67
c5-3t-14p	89.88	2.29	51.35	4.01	44.89	4.58	43.82	4.70
Tamaño de problema: **2000**								
	Uniforme		Proporcional		Óptima FIFO		Óptima LIFO	
Configuración	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.
c6-2t-5p	750.91	0.79	564.79	1.05	287.58	2.06	285.76	2.08
c7-2t-8p	430.07	1.38	341.21	1.74	191.46	3.10	191.32	3.10
c8-2t-8p	437.72	1.36	190.65	3.11	188.68	3.15	188.54	3.15
c1-3t-8p	431.72	1.38	601.38	0.99	433.28	1.37	433.31	1.37
c2-3t-8p	430.59	1.38	475.72	1.28	224.96	2.64	224.32	2.65
c3-3t-9p	376.99	1.58	322.77	1.84	178.67	3.32	178.28	3.3
c4-3t-13p	252.34	2.35	282.24	2.10	144.41	4.11	143.96	4.12
c5-3t-14p	232.97	2.55	274.68	2.16	138.93	4.27	139.07	4.27

La figura 2.19 muestra la mejora relativa conseguida al utilizar la distribución óptima FIFO frente a la distribución proporcional y la mejora que se obtiene con la distribución óptima LIFO frente a la distribución óptima FIFO.

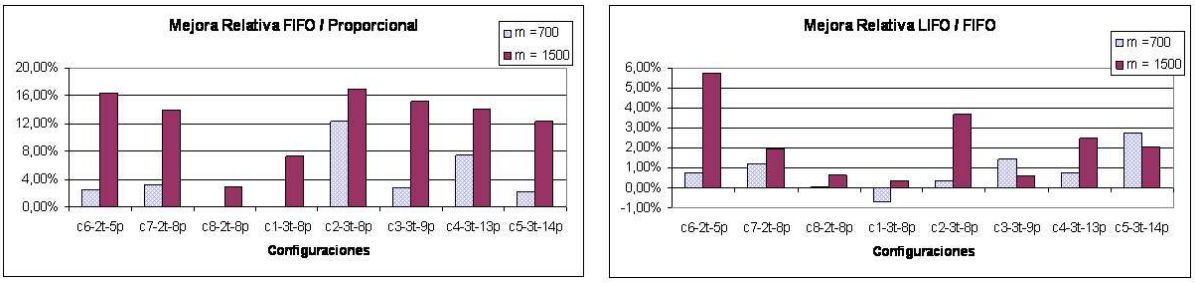

Figura 2.19: Mejoras relativas para dos problema ($m = 700$ y $m = 1500$) y las 8 configuraciones utilizadas. Izquierda: Mejora obtenida al resolver el problema con la distribución óptima FIFO frente a la ejecución del programa con la distribución proporcional. Derecha: Mejora conseguida por la distribución óptima LIFO frente a la distribución óptima FIFO.

En la figura 2.19-Izquierda se observa que la distribución óptima FIFO es mejor que la distribución proporcional: obtenemos un mejora relativa mayor que 0 al utilizar nuestra distribución; excepto para las configuraciones *c8-2t-8p* y *c1-3t-8p* con el tamaño 700, donde las diferencias son mínimas. La figura 2.19-Derecha muestra la relación entre las dos distribuciones óptimas. Calculamos la mejora relativa de la distribución LIFO frente a la distribución FIFO: observamos que los resultados en las 2 estrategias son muy similares (únicamente en una ejecución la mejora obtenida supera el 4 %); y solamente en un caso (configuración *c1-3t-8p* y el problema de tamaño 700) la distribución FIFO es la mejor estrategia (la mejora relativa en la gráfica aparece negativa).

En la figura 2.20 se muestra gráficamente la relación entre las aceleraciones máximas y las que obtenemos al ejecutar con la distribución proporcional y las distribuciones óptimas para los dos esquemas de resolución, para dos tamaños de problemas ($m = 700$ y $m = 1500$).

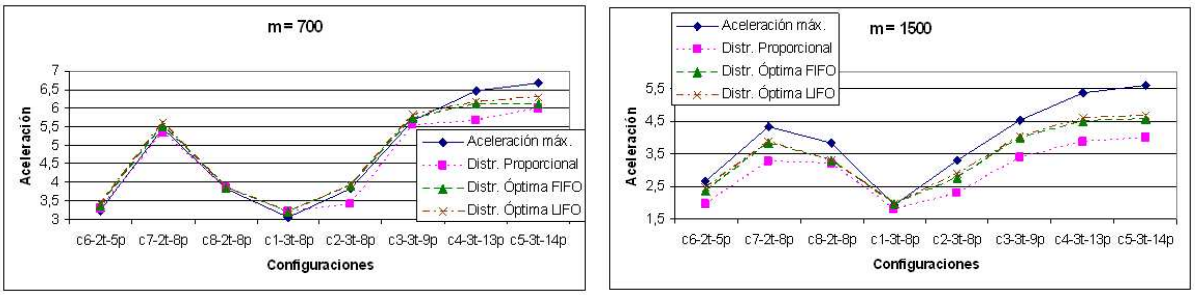

Figura 2.20: Aceleraciones conseguidas al ejecutar el problema con la distribución proporcional, la distribución óptima FIFO y la distribución óptima LIFO y su comparación con las aceleraciones máximas, para dos problemas ($m = 700$ y $m = 1500$) y todas las configuraciones utilizadas.

Comparando las aceleraciones reales con las aceleraciones máximas que aparecen en la tabla 2.18 y en la figura 2.20 podemos ver que para el problema más pequeño las aceleraciones obtenidas están muy cerca de la aceleración máxima, e incluso la superan (superlinealidad) en algunos casos. Como ya comentábamos en la sección 2.10.2, la superlinealidad puede deberse a los efectos producidos por el tamaño de la memoria caché, frente a los tamaños de problemas en las ejecuciones secuenciales. Para el problema de tamaño 1500 las aceleraciones alcanzadas se encuentran más lejos de los valores máximos, aunque es mejor para las dos distribuciones óptimas que para la proporcional.

Las distribuciones de trabajo uniforme y proporcional se encuentran en las tablas 2.15 y 2.16 de la sección 2.10.3. En las tablas 2.19, 2.20, 2.21 y 2.22 incluimos únicamente el número de filas que se asignan a los procesadores en las distribuciones óptimas FIFO y LIFO, así como el tiempo necesario para la ejecución del método exacto. El formato de la tabla es idéntico al utilizado en la tabla 2.15 y 2.16, donde m_R, m_I y m_L son las cantidades de trabajo asignadas a los procesadores rápidos, intermedios y lentos respectivamente.

Tabla 2.19: Distribuciones óptimas FIFO y LIFO calculadas por el modelo y tiempo empleado para ejecutar el método exacto que los calcula para los subsistemas heterogéneos del *cluster* utilizando diferentes configuraciones, para el problema del producto de matrices de tamaño 700.

Tamaño de problema: **700**				
	Óptima FIFO		Óptima LIFO	
Configuración	Distribución	Tiempo	Distribución	Tiempo
c6-2t-5p	$m_R = 212$ $m_I = 138$	0.52	$m_R = 214\ 213$ $m_I = 137\ 136$	0.32
c7-2t-8p	$m_R = 126$ $m_I = 81$, i = 4..6 $m_I = 79$, i = 7	1.05	$m_R = 127$ $m_I = 82\ 80\ 79\ 78$	0.60
c8-2t-8p	$m_R = 186$ $m_L = 36$, i = 4..6 $m_L = 34$, i = 7	1.05	$m_R = 186\ 186\ 185$ $m_L = 36\ 36\ 36\ 35$	0.62
c1-3t-5p	$m_I = 143$ $m_L = 43$, i = 5..6 $m_L = 42$, i = 7	1.05	$m_I = 147\ 145\ 143\ 141$ $m_L = 42\ 41\ 41$	0.60
c2-3t-8p	$m_R = 181$ $m_I = 117$ $m_L = 35$, i = 5..6 $m_L = 34$, i = 7	1.04	$m_R = 183\ 182$ $m_I = 117\ 116$ $m_L = 34$	0.60
c3-3t-9p	$m_R = 117$ $m_I = 76$, i = 4..6 $m_I = 75$, i = 7 $m_L = 23$	1.22	$m_R = 123\ 122\ 122$ $m_I = 79\ 78\ 77\ 76$ $m_L = 23$	0.75
c4-3t-13p	$m_R = 107$ $m_I = 69$ $m_L = 21$, i = 8..11 $m_L = 19$, i = 12	1.92	$m_R = 109\ 108\ 108$ $m_I = 70\ 69\ 68\ 67$ $m_L = 21\ 20\ 20\ 20\ 20$	1.25
c5-3t-14p	$m_R = 103$ $m_I = 67$, i = 4 $m_I = 66$, i = 5..7 $m_L = 21$	2.06	$m_R = 105$ $m_I = 67\ 67\ 66\ 65$ $m_L = 20$	1.29

En las distribuciones de las tablas 2.19, 2.20, 2.21 y 2.22, el método no ha asignado una cantidad de trabajo 0 a ningún procesador, lo que significa que, en ningún caso, el número óptimo de procesadores es menor que el número de procesadores disponibles en los subsistemas. Por otro lado, el tiempo invertido por la ejecución del método exacto aumenta considerablemente al incrementar el número de procesadores en el sistema y el tamaño del problema. Para problemas pequeños, el tiempo necesario para calcular las distribuciones óptimas no compensa las mejoras obtenidas, debido a que el tiempo de ejecución paralelo es muy reducido. Sin embargo, para problemas grandes, donde el tiempo total de ejecución paralelo supera un minuto, el tiempo necesario para obtener las distribuciones óptimas es menor que la mejora obtenida al ejecutar con la distribución proporcional o uniforme, por lo que aplicar este método sí resulta ser rentable.

Tabla 2.20: Distribuciones óptimas FIFO y LIFO calculadas por el modelo y tiempo empleado para ejecutar el método exacto que los calcula para los subsistemas heterogéneos del *cluster* utilizando diferentes configuraciones, para el problema del producto de matrices de tamaño 1000.

Tamaño de problema: **1000**				
	Óptima FIFO		Óptima LIFO	
Configuración	Distribución	Tiempo	Distribución	Tiempo
c6-2t-5p	$m_R = 311$ $m_I = 189$	1.07	$m_R = 313\ 312$ $m_I = 188\ 187$	0.63
c7-2t-8p	$m_R = 184$ $m_I = 112$	2.15	$m_R = 186$ $m_I = 112\ 111\ 110\ 109$	1.22
c8-2t-8p	$m_R = 275$ $m_L = 44$, i = 4..6 $m_L = 43$, i = 7	2.11	$m_R = 275\ 275\ 274$ $m_I = 44$	1.24
c1-3t-5p	$m_I = 208$ $m_L = 56$	2.14	$m_I = 211\ 210\ 208\ 206$ $m_L =$	1.19 55
c2-3t-8p	$m_R = 270$ $m_I = 164$ $m_L = 44$	2.14	$m_R = 272$ $m_I = 164\ 163$ $m_L = 43$	1.22
c3-3t-9p	$m_R = 179$ $m_I = 109$, i = 4..5 $m_I = 108$, i = 6..7 $m_L = 29$	2.51	$m_R = 181\ 181\ 180$ $m_I = 109\ 108\ 107\ 106$ $m_L = 28$	1.42
c4-3t-13p	$m_R = 161$ $m_I = 97$ $m_L = 26$, i = 8..11 $m_L = 25$, i = 12	3.89	$m_R = 163\ 163\ 162$ $m_I = 98\ 98\ 97\ 96$ $m_L = 25$	2.30
c5-3t-14p	$m_R = 157$ $m_I = 95$ $m_L = 25$, i = 8..12 $m_L = 24$, i = 13	4.27	$m_R = 159$ $m_I = 96\ 95\ 94\ 94$ $m_L = 24$	2.44

Tabla 2.21: Distribuciones óptimas FIFO y LIFO calculadas por el modelo y tiempo empleado para ejecutar el método exacto que los calcula para los subsistemas heterogéneos del *cluster* utilizando diferentes configuraciones, para el problema del producto de matrices de tamaño 1500.

Tamaño de problema: **1500**				
	Óptima FIFO		Óptima LIFO	
Configuración	Distribución	Tiempo	Distribución	Tiempo
c6-2t-5p	$m_R = 535$ $m_I = 215$	2.37	$m_R = 536\ 535$ $m_I = 215\ 214$	1.42
c7-2t-8p	$m_R = 325$ $m_I = 132$, i = 4 $m_I = 131$, i = 5..7	4.80	$m_R = 326\ 326\ 325$ $m_I = 131\ 131\ 131\ 130$	2.87
c8-2t-8p	$m_R = 404$ $m_L = 72$	4.89	$m_R = 405\ 404\ 404$ $m_L = 72\ 72\ 72\ 71$	2.84
c1-3t-5p	$m_I = 278$, i = 1..2 $m_I = 277$, i = 3..4 $m_L = 130$	4.81	$m_I = 279\ 279\ 278\ 277$ $m_L = 129$	2.78
c2-3t-8p	$m_R = 448$ $m_I = 181$, i = 3 $m_I = 180$, i = 4 $m_L = 81$	4.72	$m_R = 450\ 449$ $m_I = 181\ 180$ $m_L = 80$	2.75
c3-3t-9p	$m_R = 313$ $m_I = 127$ $m_L = 53$	5.58	$m_R = 314$ $m_I = 127\ 126\ 126\ 126$ $m_L = 53$	3.17
c4-3t-13p	$m_R = 275$ $m_I = 112$ $m_L = 46$, i = 8..11 $m_L = 43$, i = 12	8.71	$m_R = 277\ 276\ 276$ $m_I = 112\ 112\ 111\ 111$ $m_L = 45$	5.10
c5-3t-14p	$m_R = 267$ $m_I = 109$, i = 4..6 $m_I = 108$, i = 7 $m_L = 44$	9.58	$m_R = 268$ $m_I = 109\ 108\ 108$ $m_I = 44\ 44\ 44\ 44\ 44\ 43$	5.59

2.10.4. Herramienta de predicción de esquema y distribución óptima de trabajos

Para aprovechar todas las ventajas de nuestros dos modelos, hemos desarrollado también una herramienta que permite determinar el mejor método para resolver un problema maestro-esclavo: secuencial o paralelo y, en este último caso, cuál de las dos estrategias debemos utilizar y con qué distribución. En la tabla 2.23 aparece el método con el que deberíamos resolver el problema (columna *Forma de Resolución*). Esta columna puede tomar uno de los tres valores siguientes: *Secuencial*, *FIFO* o *LIFO*.

Al comparar los resultados de la tabla 2.23 con las tablas de los tiempos de ejecución paralelo de las secciones anteriores (secciones 2.10.2 y 2.10.3), podemos comprobar el alto porcentaje de acierto de nuestra herramienta al predecir correctamente la estrategia adecuada de resolución. En las tablas encontramos que de las 52 ejecuciones realizadas del método exacto únicamente en 8 el resultado no es correcto; lo que supone un porcentaje de acierto del 84,62 %. En 5 de las ejecuciones erróneas las aceleraciones obtenidas al utilizar ambas estrategias es la misma, por lo que no puede considerarse realmente un error. Esto incrementa el porcentaje de acierto hasta el 94,23 %. En las otras tres ejecuciones la diferencia entre las aceleraciones no supera el 0,03, lo que en tiempo real de ejecución supone aproximadamente 1,5 segundos en un total de más de 160 (configuración *c3-2t-5p* para el problema de tamaño $m = 1500$). Este porcentaje resulta un valor aceptable del error. Por otro lado, en la tabla se comprueba que en la mayoría de los casos, lo más conveniente para resolver un problema del producto de matrices, es utilizar la estrategia

Tabla 2.22: Distribuciones óptimas FIFO y LIFO calculadas por el modelo y tiempo empleado para ejecutar el método exacto que los calcula para los subsistemas heterogéneos del *cluster* utilizando diferentes configuraciones, para el problema del producto de matrices de tamaño 2000.

Tamaño de problema: **2000**				
	Óptima FIFO		Óptima LIFO	
Configuración	Distribución	Tiempo	Distribución	Tiempo
c6-2t-5p	$m_R = 818$ $m_I = 182$	4.33	$m_R = 818$ $m_I = 182$	2.43
c7-2t-8p	$m_R = 515$ $m_I = 114$, i = 4..6 $m_I = 113$, i = 7	8.60	$m_R = 516\ 515\ 515$ $m_I = 114\ 114\ 113\ 113$	5.10
c8-2t-8p	$m_R = 508$ $m_L = 119$	8.64	$m_R = 508$ $m_L = 119$	4.94
c1-3t-5p	$m_I = 287$ $m_L = 284$	8.67	$m_I = 288\ 287\ 287\ 287$ $m_L = 284\ 284\ 283$	4.74
c2-3t-8p	$m_R = 638$ $m_I = 142$, i = 3 $m_I = 141$, i = 4 $m_L = 147$	8.62	$m_R = 639$ $m_I = 142\ 141$ $m_L = 147\ 146\ 146$	4.84
c3-3t-9p	$m_R = 486$ $m_I = 107$ $m_L = 114$	10.15	$m_R = 486$ $m_I = 107$ $m_L = 114$	5.70
c4-3t-13p	$m_R = 394$ $m_I = 86$ $m_L = 95$, i = 8..11 $m_L = 94$, i = 12	16.05	$m_R = 395$ $m_I = 87\ 86\ 86\ 86$ $m_L = 94$	9.11
c5-3t-14p	$m_R = 376$ $m_I = 82$ $m_L = 91$, i = 8..12 $m_L = 89$, i = 13	17.35	$m_R = 377\ 377\ 376$ $m_I = 83\ 82\ 82\ 82$ $m_L = 91\ 90\ 90\ 90\ 90\ 90$	9.95

Tabla 2.23: Resultados obtenidos con la herramienta: estrategia que debemos usar en la resolución de los problemas.

Configuración	$m = 700$	$m = 1000$	$m = 1500$	$m = 2000$
c1-2t-3p	LIFO	LIFO	LIFO	LIFO
c2-2t-5p	FIFO	FIFO	LIFO	FIFO
c3-2t-5p	LIFO	FIFO	FIFO	LIFO
c4-2t-8p	LIFO	LIFO	FIFO	LIFO
c5-2t-10p	LIFO	LIFO	LIFO	LIFO
c6-2t-5p	LIFO	LIFO	LIFO	LIFO
c7-2t-8p	LIFO	LIFO	LIFO	LIFO
c8-2t-8p	LIFO	LIFO	LIFO	LIFO
c1-3t-5p	LIFO	FIFO	FIFO	FIFO
c2-3t-8p	LIFO	LIFO	LIFO	LIFO
c3-3t-9p	LIFO	LIFO	LIFO	LIFO
c4-3t-13p	LIFO	LIFO	LIFO	LIFO
c5-3t-14p	LIFO	LIFO	LIFO	LIFO

LIFO; y en ningún caso una ejecución secuencial resulta ser la opción más acertada.

2.11. Predicción de la distribución óptima: Transformada Rápida de Fourier bidimensional (FFT-2D)

En esta sección aplicamos la metodología presentada en la sección 2.9 al problema de la **Transformada Rápida de Fourier bidimensional** (**FFT-2D**), para resolverlo de forma óptima en un sistema heterogéneo.

La Transformada Rápida de Fourier (FFT) juega un papel importante en numerosas aplicaciones científicas y técnicas: la física computacional, el procesamiento de señales digitales, el procesamiento de imágenes, etc. El estudio lo realizamos sobre las FFT-2D, que son particularmente importantes en la reconstrucción de imágenes tridimensionales y han sido estudiadas en [21, 22, 74, 122, 157, 177]. En la mayoría de las paralelizaciones de este algoritmo, los autores dividen el proceso en dos fases, aplicando sucesivamente una FFT unidimensional a cada una de las dos dimensiones (filas y columnas). En [122] los autores implementan dos versiones paralelas del cálculo de la Transformada de Fourier bidimensional, estudiando la cantidad de información desplazada entre procesadores y el rendimiento obtenido con los dos métodos. En [22] el autor implementa un filtro para procesar imágenes empleando una FFT-2D secuencial y en [21] se presenta un algoritmo paralelo para resolverla. En [157] los autores desarrollan una FFT-2D paralela sobre *threads* y en [177] se puede consultar una implementación de este algoritmo sobre un cluster de PCs. Uno de los trabajos más importantes sobre la FFT se encuentra en [74, 75], la FFTW (*Fast Fourier Transform in the West*), donde se presenta una librería de software libre, que permite realizar transformadas de Fourier para una o más dimensiones, con matrices de tamaños arbitrarios, tanto para datos reales como complejos, soportada por una amplia variedad de plataformas con un alto rendimiento.

El problema general de la FFT-2D consiste en operar sobre una matriz bidimensional de números complejos, de tamaño $M \times N$. Al tratarse de una matriz de números complejos, existen dos valores para cada una de las celdas de la matriz: uno para representar la parte real del número y otro que representa la parte imaginaria. Sin embargo, para representar una imagen, en ocasiones se asigna un valor 0 a la parte imaginaria y la parte real del número contiene el valor de un pixel de la imagen. La transformación a realizar viene dada por las siguientes ecuaciones, dependiendo de la dirección en la que se vaya a aplicar:

Directa:

$$F(u,v) = \frac{1}{N*M} \sum_{x=0}^{M} \sum_{y=0}^{N} f(x,y) * e^{-j*2*\pi*(u*x/M+v*y/N)}$$

Inversa:

$$F(u,v) = \sum_{x=0}^{M} \sum_{y=0}^{N} f(x,y) * e^{j*2*\pi*(u*x/M+v*y/N)}$$

Las fórmulas anteriores equivalen a aplicar una FFT unidimensional sobre cada una de las filas de la matriz que se quiere transformar y, a continuación, una FFT unidimensional sobre cada una de las columnas de la matriz resultante. Esto significa que puede implementarse el algoritmo mediante 4 pasos:

1. Ejecutar la FFT unidimensional sobre cada fila
2. Calcular la traspuesta de la matriz resultado
3. Ejecutar la FFT unidimensional sobre cada fila de la matriz resultado
4. Calcular la traspuesta de la matriz obtenida en el paso anterior

Los tres últimos pasos del algoritmo equivalen a realizar una FFT unidimensional por columnas sobre la matriz original.

2.11.1. Algoritmo Secuencial

El algoritmo secuencial que resuelve el problema de la FFT-2D transforma la matriz de entrada de acuerdo a las ecuaciones anteriores (sección 2.11). Asumiendo que se va a transformar una matriz de tamaño $M \times N$, los pasos 1 y 3 del algoritmo tienen una complejidad de $O(MNlog(MN))$, mientras que los pasos 2 y 4 (calcular las matrices traspuestas) requieren $O(MN)$ operaciones. En la figura 2.21 mostramos el código de la función que realiza la transformada [22]. La función recibe como parámetros de entrada la matriz sobre la que se va a realizar la transformada, c; el tamaño de la matriz (M, N), donde M y N denotan el número de filas y columnas respectivamente; y el parámetro *dir* que indica el tipo de transformada que se va a realizar (operación de tipo directa o inversa). La matriz es de un tipo definido llamado *COMPLEX*, que representa un número complejo e incluye dos valores flotantes: la parte real y la parte imaginaria del número complejo.

```
 1 int FFT2D(COMPLEX **c,int M, int N, int dir) {
 2   int i, m;
 3   double *real, *imag;
 4                                       // FFT por filas
 5   if (!AsignarMemoria(real, imag, N))
 6     return(FALSE);
 7
 8   if (!Potencia2(N, &m))   // Comprobar si el número de columnas es potencia de 2: N = 2^m
 9     return(FALSE);
10
11   for (i = 0; i < M; i++) {
12     Inicializar(N, real, imag, c);
13     FFT(dir, m, real, imag);           // FFT unidimensional para una fila
14     CopiarResultados(N, real, imag, c);
15   }
16   LiberarMemoria(real, imag);
17
18   CalcularTraspuesta(c, M, N);  // Cálculo de la traspuesta
19
20                                        // FFT por filas de la matriz traspuesta
21   if (!AsignarMemoria(real, imag, M))
22     return(FALSE);
23
24   if (!Potencia2(M, &m))   // Comprobar si el número de filas es potencia de 2: M = 2^m
25      return(FALSE);
26
27   for (i = 0; i < N; i++) {
28      Inicializar(M, real, imag, c);
29      FFT(dir, m, real, imag);           // FFT unidimensional para una fila de la traspuesta
30      CopiarResultados(M, real, imag, c);
31   }
32   LiberarMemoria(real, imag);
33
34   CalcularTraspuesta(c, N, M); // Cálculo de la traspuesta
35
36   return(TRUE);
37 }
```

Figura 2.21: Función que resuelve el problema de la FFT-2D secuencialmente: realiza llamadas a la función que calcula la FFT unidimensional

En primer lugar se realiza la transformada unidimensional en las filas de la matriz original: se asigna memoria a dos *arrays* auxiliares (*real* e *imag*) para almacenar la información de cada una de las columnas (línea 5 del código) y se comprueba que el número de columnas de la matriz sea una potencia de 2, puesto que es un requisito de la FFT unidimensional que se está aplicando (línea 8 del código). Si en alguno de estos dos pasos se retorna un error no se puede ejecutar la FFT-2D y finaliza la ejecución. Si no se produce ningún error, se copian los datos de la fila de la matriz inicial c a los *arrays* auxiliares (línea 12) y, a continuación, se aplica la FFT unidimensional sobre la fila (línea 13); por último, se copian nuevamente los resultados desde los *arrays* auxiliares a la matriz (línea 14). Una vez finalizado el cómputo de las FFT unidimensionales sobre todas las filas de la matriz, se libera la memoria utilizada en los *arrays* auxiliares y se calcula la matriz traspuesta (paso 2 del algoritmo de la sección 2.11) en la línea 18 del código. A continuación se repite el proceso, pero esta vez sobre cada una de las filas de la matriz traspuesta (líneas 21 a 32 de código). Por último, en la línea 35 se realiza el último paso del algoritmo, que consiste en volver a calcular la traspuesta de la matriz, con lo que obtenemos el resultado del problema.

La tabla 2.24 muestra los tiempos obtenidos en la ejecución de este algoritmo secuencial sobre los tres tipos de máquinas disponibles en el *cluster* (sección 1.5.1). Hemos realizado las pruebas para problemas de diferentes tamaños, donde el número de filas y el número de columnas son siempre potencias de 2, (requisito para aplicar la FFT unidimensional). Consideramos entradas donde las matrices son imágenes cuadradas, es decir, donde el número de filas es igual al número de columnas y el valor de la parte imaginaria es siempre 0. Los tamaños de las imágenes varían entre 1024×1024 y 8192×8192 para las máquinas rápidas, mientras que en los procesadores intermedios y lentos el mayor tamaño que se ha podido ejecutar es 4096×4096. Las columnas con etiquetas *Filas* y *Columnas* de la tabla 2.24 especifican el tamaño del problema a resolver; el resto de las columnas representan el tiempo, expresado en segundos, obtenido al ejecutar el algoritmo secuencial en las máquinas rápidas (R), intermedias (I) y lentas (L) del *cluster* respectivamente.

Tabla 2.24: Tiempos de ejecución del algoritmo secuencial de la FFT-2D en los tres tipos de máquinas que componen el cluster

Filas	Columnas	Máquinas rápidas (R) Tiempo (seg.)	Máquinas Intermedias (I) Tiempo (seg.)	Máquinas Lentas (L) Tiempo (seg.)
1024	1024	1.44	1.82	5.11
2048	2048	6.24	9.56	24.10
4096	4096	29.33	170.01	261.34
8192	8192	168.39	–	–

En la tabla 2.24 se observa que, al incrementar el tamaño del problema, los tiempos necesarios para resolverlo de forma secuencial aumentan considerablemente, especialmente para los procesadores intermedios y lentos.

2.11.2. Algoritmo Paralelo

Es un hecho conocido que la FFT-2D pertenece a una clase de aplicaciones que no escala bien en sistemas distribuidos, debido a la alta carga de comunicaciones entre los procesadores para el intercambio de los datos [177]. Por tratarse de un problema que no escala bien resulta de interés la aplicación de nuestra metodología para comprobar la mejora obtenida frente a las distribuciones uniforme y proporcional.

Aplicamos la paralelización únicamente sobre los pasos 1 y 3 del algoritmo presentado en la sección 2.11.1, mientras que los pasos 2 y 4 los ejecuta el maestro de forma secuencial. La figura 2.22 ilustra gráficamente el proceso para resolver este problema en un sistema con 4 esclavos: inicialmente el maestro almacena todos los datos de la matriz (1), agrupa las filas según alguna política de distribución y envía a cada esclavo la cantidad de trabajo que le corresponda. Los esclavos ejecutan una FFT unidimensional sobre cada fila que le haya correspondido (2) y devuelven el resultado al maestro. El maestro reagrupa

los resultados (3) y realiza la traspuesta sobre la matriz obtenida (4). A continuación se vuelve a repetir el proceso anterior: sobre esta matriz traspuesta el maestro vuelve a aplicar la política de distribución y envía de nuevo el trabajo a los esclavos. Éstos aplican una FFT unidimensional sobre cada una de sus filas (5) y devuelven el resultado. El maestro reagrupa la matriz (6) y calcula su traspuesta (7). La matriz obtenida en este último paso es la solución del problema. Si se realiza un particionado por bloques de los datos, la cantidad de información que se envía a cada esclavo es del orden $O(\frac{MN}{p})$ y para realizar un cómputo se invierte un tiempo del orden $O(\frac{MN}{p}log(MN))$. Si comparamos estos 2 valores, la diferencia entre ellos es únicamente del orden $O(log(MN))$. Esta cantidad tan reducida explica la falta de escalabilidad de este problema.

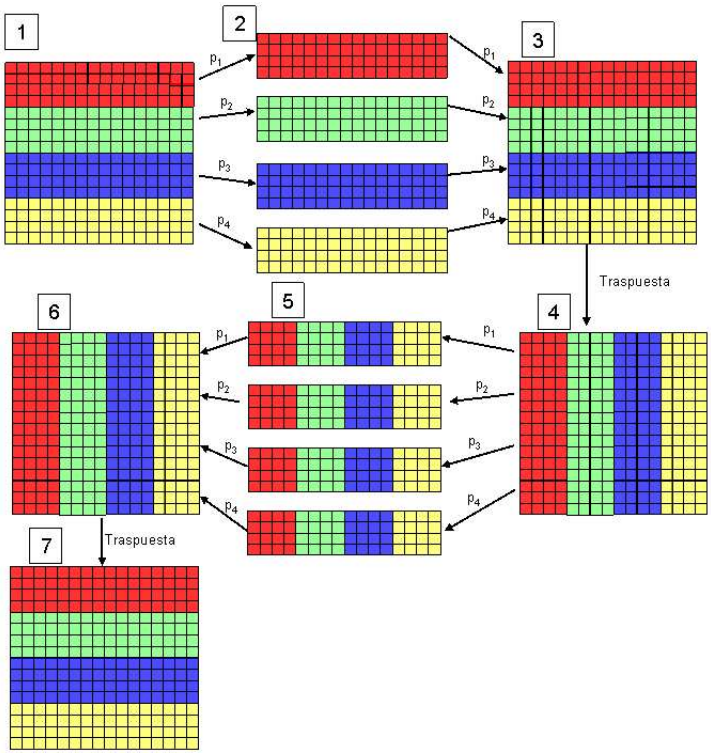

Figura 2.22: Proceso de resolución del problema de la FFT bidimensional de forma paralela.

En la versión paralela se han implementado dos códigos distintos, uno para cada estrategia de resolución presentado en las secciones 2.4, 2.5 y 2.6. La diferencia entre los dos códigos reside exclusivamente en el orden de recepción de los resultados por parte del maestro. En la figura 2.23 se presenta el código que resuelve el problema de la FFT-2D mediante la estrategia FIFO. La primera función se denomina *FFT2D* y se encarga de comprobar que tanto el número de filas como el número de columnas de la matriz son potencias de 2. Los parámetros que recibe son: la matriz original de datos (c); el tamaño de la matriz (M y N son el número de filas y de columnas respectivamente); la dirección de la ecuación para resolver el problema: directa o inversa (dir); el nombre del procesador (id), que valdrá 0 para el maestro y contendrá la posición que ocupa el procesador que está ejecutando el código dentro del conjunto de los esclavos, $1, \ldots, p$; y *vfilas*, que es un vector que contiene la distribución de datos que se va a realizar, es decir, el número de filas que le corresponde a cada esclavo.

En la función *FFT2D* primero se realiza la transformación por filas: se comprueba si el número de filas es potencia de 2 (línea 8) y llama a la función que se encarga de la distribución de tareas a los esclavos (línea 11). En las líneas 14 a 17 del código se repite el proceso para la transformación de las columnas.

```
1 /*************************************************************************
2  * FFT bidimensional paralela
3  * ************************************************************************/
4 boolean FFT2D(int *c, int M, int N, int dir, int id, int* vfilas) {
5   int i, j, m, *t, *t2;      // t y t2: matrices auxiliares
6
7               // Transformar las filas
8   if (!Potencia2 (M, &m))  // Comprueba si el número de filas es potencia de 2: M = 2^m
9      return(FALSE);
10
11  t = distribuir_trabajo (c, M, N, dir, num_proc, id, m, vfilas);
12
13              // Transformar las columnas
14  if (!Potencia2 (N, &m))  // Comprueba si el número de columnas es potencia de 2: N = 2^m
15    return (FALSE);
16
17  t2 = distribuir_trabajo (t, N, M, dir, num_proc, id, m, vfilas);
18  return(TRUE);
19 }
20
21 /*************************************************************************************
22  * Distribución de trabajo a los esclavos y cómputo de las filas de la matriz
23  * ************************************************************************************/
24 int *distribuir_tareas (int *c, int filas, int cols, int dir, int id, int m, int* vfilas) {
25   int i, nf;              // nf = número de filas que le corresponden a un esclavo
26   t_par par;             // parámetros de envío y recepción
27   int *mat, *res;         // matrices auxiliares
28
29   if (id == 0) {                 // MASTER
30      inic_param (&par, num_proc);          // Envía el trabajo a los esclavos
31      for (i = 1; i < num_proc; i++) {
32         enviar (&vfilas[i], 1, MPI_INT, i, TAGN);
33         enviar ((c + par.ini), (2 * vfilas[i] * cols), MPI_INT, i, TAG);
34         variar_par (&par, vfilas[i], cols);
35      }
36      inic_param (&par, num_proc);          // Recibe el trabajo de los esclavos:
37      res = inic_matrix(filas, cols);        // se utiliza una matriz auxiliar
38      for (i = 1; i < num_proc; i++) {
39         recibir ((res + par.ini), (2 * vfilas[i] * cols), MPI_INT, i, TAG);
40         variar_par (&par, vfilas[i], cols);
41      }
42      mat = traspuesta_matriz(res, filas, cols);        // Cálculo de la traspuesta
43      return mat;
44   }
45   else {                         // ESCLAVOS
46      recibir (&nf, 1, MPI_INT, 0, TAGN);                  // Recibe el trabajo a realizar
47      mat = inic_matrix (nf, cols);
48      recibir (mat, (2 * cols * nf), MPI_INT, 0, TAG);
49      mat = Realizar_FFT (mat, nf, cols, dir, m);          // FFT unidimensional
50      enviar (mat, (2 * nf * cols), MPI_INT, 0, TAG);      // Envía la solución al master
51   }
52 }
```

Figura 2.23: Código que resuelve el problema de la FFT-2D mediante la estrategia FIFO

La función *distribuir_tareas* recibe los mismos parámetros que la función *FFT2D*, añadiendo solamente un parámetro m, tal que $M = 2^m$. El maestro envía a cada esclavo el trabajo que le corresponde (líneas 30 a 35 del código) y, a continuación, recibe los resultados que devuelven los esclavos (líneas 36 a 41 del código). De acuerdo con la estrategia FIFO de resolución, se reciben los resultados de los esclavos en el mismo orden en el que se enviaron (sección 2.5). Para resolver el problema mediante la estrategia LIFO (sección 2.6) únicamente es necesario modificar estas líneas para recibir los resultados en el orden inverso. Por último, el maestro calcula la traspuesta de la matriz (línea 42 del código). Entre las líneas 45 a 51, los esclavos ejecutan su código: reciben las filas que le corresponden, realizan una llamada a la función que ejecuta la FFT unidimensional y devuelven su resultado al maestro.

Este algoritmo de la FFT-2D no permite la aplicación de nuestra metodología a toda la ejecución. Únicamente se utiliza sobre la transformación por filas de la matriz de entrada (paso 1 del algoritmo). El tiempo necesario para ejecutar el paso 3 es el mismo que para completar el paso 1: aquella distribución de trabajo que minimice el tiempo para la primera ejecución de la FFT unidimensional también minimiza el tiempo del conjunto de las dos ejecuciones. A pesar de no paralelizar el cálculo de la traspuesta, la ejecución seguirá siendo óptima, ya que, en ambos casos, son realizadas por el maestro y el tiempo invertido no depende de los esclavos que se utilicen, ni de la distribución de tareas asignada. Por lo tanto, aunque el tiempo que se obtiene con el modelo no es el tiempo real de ejecución del algoritmo, la distribución que minimiza el modelo sí es la distribución óptima para el problema completo bajo la estrategia de resolución considerada.

2.11.3. Caracterización de la plataforma heterogénea

En las secciones 1.5.1 y 2.7.3 caracterizamos nuestra plataforma heterogénea para el problema del producto de matrices, determinando la relación entre las velocidades de los distintos tipos de máquinas (tabla 1.1), las potencias computacionales escaladas para cada uno de ellos y el valor de la heterogeneidad del sistema (tabla 1.2), las aceleraciones máximas posibles para todos los tamaños de problemas, considerando la plataforma heterogénea (figura 1.4) y la definición de las funciones $C_i(m)$, que determinan el tiempo de cómputo (figura 2.13). En esta subsección vamos a realizar los mismos cálculos para nuestro sistema heterogéneo, cuando resolvemos el problema de la FFT bidimensional para los 4 tamaños de problemas que hemos resuelto con el algoritmo secuencial (tabla 2.24).

La tabla 2.25 presenta las relaciones obtenidas entre los tiempos secuenciales de las máquinas. El formato de esta tabla es idéntico al de la tabla 1.1: se incluye el tamaño de los problemas (columnas etiquetadas como *Filas* y *Columnas*) y la relación entre cada par de máquinas: lentas / rápidas (L / R), intermedias / rápidas (I / R) y lentas / intermedias (L / I).

Tabla 2.25: Relación para diferentes tamaño del problema de la FFT-2D entre los tiempos secuenciales obtenidos en los tres tipos de máquinas que componen HN: lentas / rápidas (L / R), intermedias / rápidas (I / R) y lentas / intermedias (L / I).

Filas	Columnas	L / R	I / R	L / I
1024	1024	3.54	2.81	1.26
2048	2048	3.86	2.52	1.53
4096	4096	8.91	1.54	5.80

En este caso, también comprobamos que estas relaciones entre las máquinas varían al incrementar el tamaño del problema. Como ya observamos en la tabla de los tiempos secuenciales (tabla 2.24), para el problema 4096×4096 las relaciones L / R y L / I sufren un aumento considerable frente a las obtenidas para los otros dos problemas. No hemos podido obtener la relación existente entre los tiempos para el problema de 8192×8192 puesto que no ha sido posible resolver el problema de forma secuencial para las máquinas intermedias y lentas.

A continuación calculamos la heterogeneidad que presenta nuestro *cluster* completo para este problema (sección 1.4): $H = \frac{\sum_{i=0}^{p-1}(1-CP_i^E)}{p}$. Para aplicar esta fórmula es necesario obtener previamente los valores escalados para las potencias computacionales de los tres tipos de máquinas, asignándole a los procesadores rápidos el valor más alto ($CP_R^E = 1$). La tabla 2.26 muestra los valores de las potencias computacionales escaladas para las máquinas rápidas (columna CP_R^E), intermedias (columna CP_I^E) y lentas (columna CP_L^E); así como el valor obtenido para la heterogeneidad con los 14 procesadores que componen el sistema.

Tabla 2.26: Potencia computacional escalada para los tres tipos de máquinas del cluster (rápidas, intermedias y lentas) y valor de la heterogeneidad al resolver cuatro problemas de distintos tamaños de la FFT bidimensional.

Filas	Columnas	CP_R^E	CP_I^E	CP_L^E	Heterogeneidad
1024	1024	1	0.79	0.28	0.37
2048	2048	1	0.65	0.26	0.42
4096	4096	1	0.17	0.11	0.62

En la tabla 2.26 se observa como la heterogeneidad del sistema aumenta al incrementar el tamaño del problema, pasando del 37 % de heterogeneidad para el problema de 1024×1024 al 62 % en el problema de 4096×4096. Analizando las potencias computacionales escaladas, podemos ver que la potencia computacional escalada para las máquinas lentas se reduce a algo más de la mitad entre el primer y el tercer problema, mientras que para las máquinas intermedias esta reducción supone más del 75 %. Para el problema de tamaño 8192×8192 no podemos calcular la heterogeneidad al no disponer de los datos secuenciales de los procesadores intermedios y lentos, pero es probable que la heterogeneidad del sistema fuese aún mayor que para el problema 4096×4096.

La tercera de las métricas aplicadas es el cálculo de las aceleraciones máximas que se pueden obtener en el sistema al utilizar las 14 máquinas disponibles en el *cluster* (sección 1.4). En la figura 2.24 mostramos la relación entre las aceleraciones máximas alcanzables en nuestro sistema, al compararla con la que se obtendría en un sistema homogéneo del mismo tamaño donde se cumple $Sp_{max} = p$.

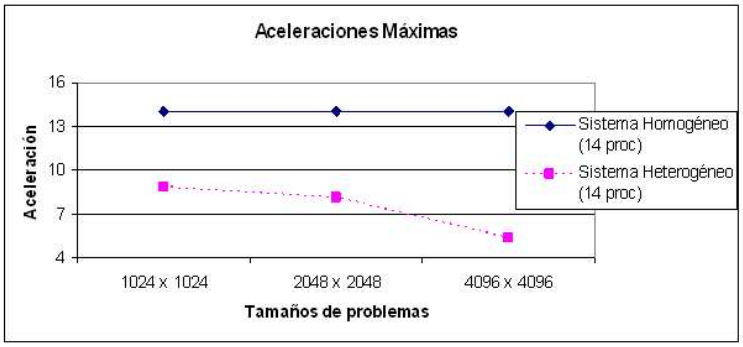

Figura 2.24: Aceleraciones máximas posibles para los tres problemas de la transformada rápida de Fourier bidimensional, utilizando las 14 máquinas disponibles en el sistema, comparadas con la aceleración máxima en un sistema homogéneo del mismo tamaño.

La aceleración máxima se encuentra bastante por debajo de la aceleración correspondiente en un sistema homogéneo. En la figura 2.24 se observa de forma gráfica como se produce un descenso en el rendimiento del sistema al resolver problemas grandes.

Por último, hemos calculado las funciones que determinan los tiempos de cómputo (funciones $C_i(m)$) para el problema de la FFT-2D, puesto que sin esta información no es posible aplicar el método exacto

para minimizar los modelos. La figura 2.25 muestra gráficamente estas funciones para el cómputo de los 4 tamaños de problemas ejecutados en los tres tipos de máquinas.

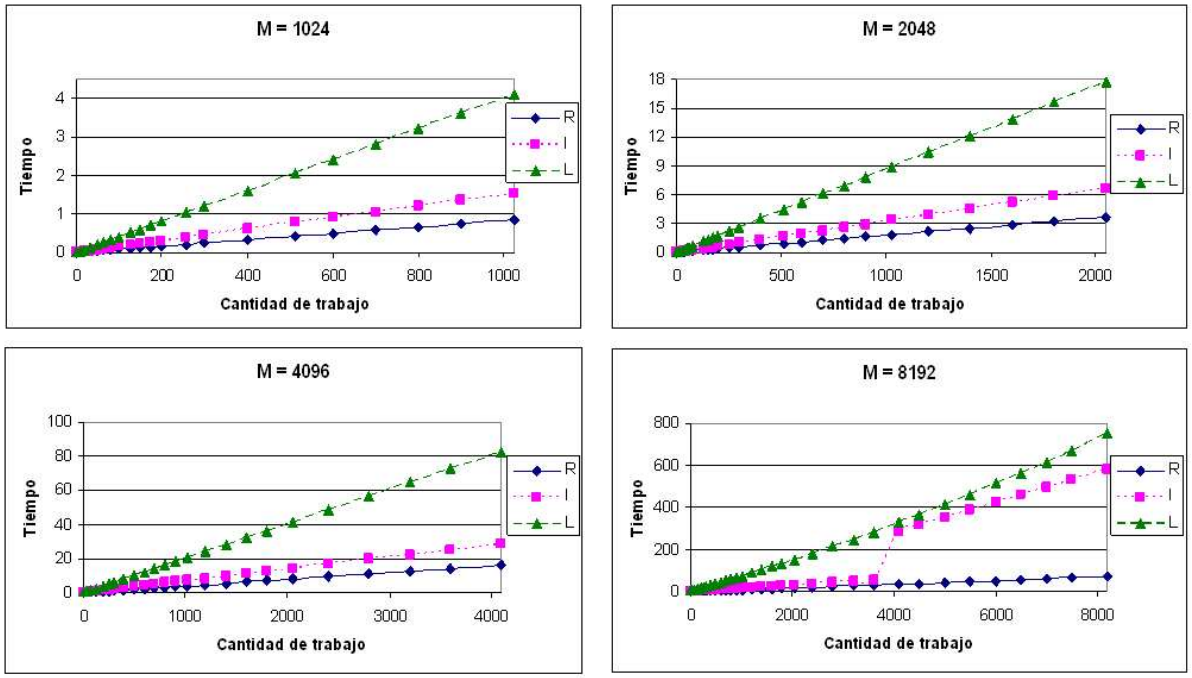

Figura 2.25: Funciones para el cálculo de las funciones $C_i(m)$ del tiempo de cómputo para los 4 tamaños de problemas considerados en las pruebas y los tres tipos de máquinas existentes en el *cluster*.

En la figura 2.25 se observa que el tiempo de cómputo de las máquinas intermedias se mantiene próximo al tiempo requerido por los procesadores más rápidos, mientras que los procesadores lentos tardan mucho más tiempo en realizar el mismo trabajo. Sin embargo, hay que destacar el efecto producido en las máquinas intermedias con el problema de mayor tamaño (8192×8192). Para este problema los procesadores intermedios presentan un comportamiento similar a las máquinas rápidas cuando se les asigna una cantidad de trabajo inferior a 4000 filas de la matriz, pero cuando el número de filas aumenta se produce un salto en el tiempo de cómputo, pasando a comportarse casi como una máquina lenta. Este efecto es parecido al que presentaban estas máquinas en el problema del producto de matrices, también con los problemas grandes, debido probablemente al efecto del tamaño de la memoria caché.

2.11.4. Distribución óptima de trabajos en los sistemas homogéneos

En primer lugar hemos ejecutado el algoritmo paralelo en tres diferentes subsistemas homogéneos, compuestos por máquinas rápidas (R), intermedias (I) y lentas (L). En estos subsistemas homogéneos variamos el número de esclavos entre 2 y el número máximo de procesadores disponibles según el tipo de máquina considerada.

La tabla 2.27 muestra los tiempos de ejecución sobre los sistemas paralelos de las máquinas rápidas, intermedias y lentas, expresados en segundos, y las aceleraciones obtenidas cuando resolvemos los mismos problemas utilizados en la ejecución del algoritmo secuencial (sección 2.24). En esta tabla, las columnas denominadas *Filas* y *Columnas* representan el tamaño del problema; la columna *Número de esclavos* especifica el tamaño del sistema homogéneo y dos macrocolumnas con los resultados obtenidos al ejecutar el problema con el primer y segundo esquema de resolución respectivamente (FIFO y LIFO). En cada

una de estas macrocolumnas se encuentran los tiempos y las aceleraciones conseguidas. Los tiempos secuenciales utilizados para calcular las aceleraciones son los que se muestran en la tabla 2.24.

Tabla 2.27: Tiempos de ejecución del algoritmo paralelo de la FFT-2D al ejecutar sobre los distintos subsistemas homogéneos del *cluster* con diferentes tamaños de problemas empleando las dos estrategias de resolución (FIFO y LIFO).

Máquinas rápidas del *cluster* (R)						
		Número de	Esquema FIFO		Esquema LIFO	
Filas	Columnas	esclavos	Tiempo	Aceleración	Tiempo	Aceleración
1024	1024	2	1.93	0.74	1.35	1.06
1024	1024	3	1.51	0.95	1.09	1.32
2048	2048	2	8.43	0.74	5.65	1.10
2048	2048	3	6.37	0.98	4.42	1.41
4096	4096	2	36.89	0.80	25.21	1.16
4096	4096	3	28.42	1.03	19.85	1.48
8192	8192	2	167.36	1.01	–	–
8192	8192	3	130.84	1.29	96.78	1.74
Máquinas intermedias del *cluster* (I)						
		Número de	Esquema FIFO		Esquema LIFO	
Filas	Columnas	esclavos	Tiempo	Aceleración	Tiempo	Aceleración
1024	1024	2	4.97	0.37	5.25	0.35
1024	1024	3	4.28	0.42	4.67	0.39
2048	2048	2	20.25	0.47	21.50	0.44
2048	2048	3	16.93	0.56	18.91	0.51
Máquinas lentas del *cluster* (L)						
		Número de	Esquema FIFO		Esquema LIFO	
Filas	Columnas	esclavos	Tiempo	Aceleración	Tiempo	Aceleración
1024	1024	2	8.56	0.60	9.01	0.57
1024	1024	3	6.84	0.78	7.47	0.68
1024	1024	4	6.03	0.85	6.81	0.75
1024	1024	5	5.54	0.92	6.44	0.79
2048	2048	2	36.22	0.67	37.97	0.63
2048	2048	3	28.88	0.83	31.29	0.77
2048	2048	4	25.35	0.95	28.51	0.84
2048	2048	5	23.28	1.03	27.42	0.88

En la tabla 2.27 se observa que únicamente utilizando los procesadores más rápidos se consigue ejecutar los 4 tamaños de problemas. Para los sistemas considerados, la estrategia LIFO muestra mejores tiempos de ejecución en el subsistema de las máquinas rápidas, mientras que la estrategia FIFO es siempre el mejor método para los procesadores intermedios y lentos. Las aceleraciones son muy reducidas en la mayoría de los casos: con las máquinas intermedias no se consigue mejorar el resultado de la ejecución secuencial en ninguna de las pruebas; mientras que para las máquinas lentas sólo existe un valor donde la aceleración sea superior a 1 (para el problema más grande, con el mayor número posible de esclavos utilizando la estrategia FIFO). Únicamente para los procesadores rápidos con la estrategia LIFO se mejora siempre el tiempo secuencial. Se observa también que al aumentar el número de procesadores empleados en la resolución del problema se mejora el rendimiento del sistema, reduciendo el tiempo ejecución.

Del mismo modo que en la tabla de tiempos de ejecución, en la tabla con las distribuciones (tabla 2.28) las columnas *Filas* y *Columnas* muestran el tamaño del problema y la columna *Número de esclavos* el tamaño del sistema homogéneo. Las dos últimas columnas especifican las distribuciones de trabajo entre los esclavos para las dos estrategias de resolución. En la estrategia FIFO, al ser un sistema homogéneo,

se asigna a todos los procesadores la misma carga de trabajo siempre que sea posible. Mientras que en la estrategia LIFO la cantidad de trabajo asignada a cada procesador viene determinada por la ejecución de nuestro método exacto (sección 2.8) sobre los subsistemas homogéneos. Las cantidades enviadas a los procesadores rápidos, intermedios y lentos se representan en la tabla por m_R, m_I y m_L respectivamente.

Tabla 2.28: Distribuciones óptimas de trabajo para los subsistemas homogéneos del *cluster* para los dos esquemas de resolución de los algoritmos maestro-esclavo con el problema de la FFT bidimensional.

Máquinas rápidas del *cluster* (R)				
Filas	Columnas	Número de esclavos	Estrategia FIFO	Estrategia LIFO
1024	1024	2	$m_R = 512$	$m_R = 553\ 471$
1024	1024	3	$m_R = 342\ 341\ 341$	$m_R\ 398\ 339\ 288$
2048	2048	2	$m_R = 1024$	$m_R = 1099\ 949$
2048	2048	3	$m_R = 683\ 683\ 682$	$m_R = 785\ 678\ 585$
4096	4096	2	$m_R = 2048$	$m_R = 2188\ 1908$
4096	4096	3	$m_R = 1366\ 1365\ 1365$	$m_R = 1556\ 1357\ 1183$
8192	8192	2	$m_R = 4096$	$m_R = 4357\ 3835$
8192	8192	3	$m_R = 2731\ 2731\ 2730$	$m_R = 3086\ 2716\ 2390$
Máquinas intermedias del *cluster* (I)				
Filas	Columnas	Número de esclavos	Estrategia FIFO	Estrategia LIFO
1024	1024	2	$m_I = 512$	$m_I = 691\ 333$
1024	1024	3	$m_I = 342\ 341\ 341$	$m_I = 597\ 288\ 139$
2048	2048	2	$m_I = 1024$	$m_I = 1365\ 683$
2048	2048	3	$m_I = 683\ 683\ 682$	$m_I = 1169\ 586\ 293$
Máquinas lentas del *cluster* (L)				
Filas	Columnas	Número de esclavos	Estrategia FIFO	Estrategia LIFO
1024	1024	2	$m_L = 512$	$m_L = 610\ 414$
1024	1024	3	$m_L = 342\ 341\ 341$	$m_L = 478\ 325\ 221$
1024	1024	4	$m_L = 256$	$m_L = 417\ 283\ 193\ 131$
1024	1024	5	$m_L = 205\ 205\ 205\ 205\ 204$	$m_L = 384\ 261\ 177\ 120\ 82$
2048	2048	2	$m_L = 1024$	$m_L = 1207\ 841$
2048	2048	3	$m_L = 683\ 683\ 682$	$m_L = 939\ 654\ 455$
2048	2048	4	$m_L = 512$	$m_L = 811\ 567\ 395\ 275$
2048	2048	5	$m_L = 410\ 410\ 410\ 409\ 409$	$m_L = 743\ 518\ 361\ 251\ 175$

Las diferencias existentes entre las cantidades de trabajo asignadas a los procesadores en la estrategia LIFO reflejan la importancia de las comunicaciones en el tiempo total de ejecución para este problema. En el problema del producto de matrices (tabla 2.3) comentábamos que la distribución realizada entre los procesadores para las estrategias FIFO y LIFO eran prácticamente idénticas, debido a que el costo de las comunicaciones era despreciable frente al tiempo invertido en el cómputo. En el problema de la FFT bidimensional encontramos el caso opuesto: existe una gran cantidad de información que se transmite entre los procesadores, lo que incrementa el tiempo de las comunicaciones y produce que el rendimiento en un *cluster* se vea reducido.

2.11.5. Predicción de la distribución óptima de trabajos con 2 tipos de procesadores

Hemos utilizado inicialmente el sistema heterogéneo donde sólo participan procesadores de dos tipos de máquinas distintas: las máquinas intermedias y lentas (el *cluster* de PCs independientes). En esta subsección vamos a comparar los resultados obtenidos para diferentes distribuciones de trabajo.

En la tabla 2.29 mostramos las diferentes configuraciones de máquinas con las que hemos realizado las ejecuciones, y las aceleraciones máximas que podremos obtener en cada caso para los distintos tamaños de problemas. Las configuraciones utilizadas figuran en la tabla 2.4; sin embargo, es necesario incluir la tabla 2.29 para actualizar las aceleraciones máximas en el nuevo problema. La columna *Nombre Configuración* asigna un nombre a las configuraciones. La columna *Maestro* especifica el tipo de máquina del procesador que trabaja como maestro y la columna *Esclavos* muestra los tipos de cada uno de los esclavos ordenados ascendentemente, de p_1 a p_p. Como se observa en la tabla, los procesadores se han ordenado de forma que todos los procesadores más rápidos estén situados en las primeras posiciones. En todas las configuraciones consideradas, el maestro siempre ha sido un procesador de las máquinas más rápidas incluidas en el cluster, en este caso, un procesador de tipo intermedio y se ha variado el número de esclavos del sistema y su tipo. La última de las configuraciones, *c5-2t-10p*, contiene el máximo número de procesadores posible en el *cluster* considerado: 4 máquinas intermedias (el maestro y 3 esclavos) y 6 máquinas lentas que se comportan como esclavos. Por último, en la tabla existe una macrocolumna con los datos de las aceleraciones máximas para cada tamaño del problema (1024×1024 y 2048×2048); los problemas más grandes no han podido ser ejecutados sobre estos sistemas heterogéneos.

Tabla 2.29: Configuraciones de los sistemas heterogéneos utilizados en las pruebas computacionales de este problema. En todas estas distribuciones se emplean únicamente dos tipos de nodos del cluster: intermedios (I) y lentos (L).

			Aceleraciones Máximas	
Nombre Configuración	Maestro	Esclavos	1024×1024	2048×2048
c1-2t-3p	I	I L	1.36	1.40
c2-2t-5p	I	I I L L	2.71	2.79
c3-2t-5p	I	I I I L	3.36	3.40
c4-2t-8p	I	I I I L L L L	4.42	4.59
c5-2t-10p	I	I I I L L L L L L	5.14	5.38

En la tabla 2.29 se observa que la aceleración máxima se incrementa al aumentar el tamaño del problema. Sin embargo, en todos los casos, la aceleración que se puede alcanzar en estos sistemas está bastante lejos del máximo en los sistemas homogéneos. Los valores calculados en esta tabla nos permitirán posteriormente conocer el rendimiento de nuestro sistema al resolver los problemas.

Hemos considerado 4 distribuciones distintas para resolver el problema (tabla 2.30): uniforme, proporcional y óptima para la estrategia FIFO y óptima para la estrategia LIFO. Del mismo modo que sucedió con los sistemas homogéneos de las máquinas intermedias y lentas, en el *cluster* formado exclusivamente por estos dos tipos de máquinas, no ha sido posible ejecutar problemas de tamaño superior a 2048×2048. El tiempo secuencial de referencia para calcular la aceleración es el tiempo de ejecución en el maestro, un procesador intermedio (tabla 2.24).

En la tabla 2.30 se comprueba nuevamente que este problema no escala bien sobre un cluster de memoria distribuida. En ningún caso el tiempo obtenido en el sistema heterogéneo mejora el resultado secuencial sobre una máquina intermedia. Sin embargo, se observa que el tiempo de ejecución se reduce al utilizar la distribución óptima FIFO frente a las distribuciones uniforme y proporcional. Analizando los resultados es evidente que no es eficiente resolver el problema con la estrategia LIFO, puesto que en ningún caso mejora el tiempo de la distribución óptima FIFO y, en la mayoría de las pruebas, el tiempo de ejecución es incluso peor que el que tarda el algoritmo con la distribución uniforme. En la figura 2.26

Tabla 2.30: Comparación de los tiempos de ejecución del algoritmo paralelo de la FFT-2D en un sistema heterogéneo para diferentes distribuciones empleando únicamente dos tipos de nodos del cluster: intermedios (I) y lentos (L).

Tamaño de problema: **1024**								
	Uniforme		Proporcional		Óptima FIFO		Óptima LIFO	
Configuración	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.
c1-2t-3p	7.63	0.24	5.97	0.30	5.81	0.31	6.35	0.29
c2-2t-5p	5.15	0.35	4.33	0.42	4.31	0.42	5.37	0.34
c3-2t-5p	5.13	0.35	4.33	0.42	4.33	0.42	5.25	0.35
c4-2t-8p	4.48	0.41	4.46	0.41	4.45	0.41	5.10	0.36
c5-2t-10p	4.55	0.40	4.55	0.40	4.53	0.40	5.05	0.36
Tamaño de problema: **2048**								
	Uniforme		Proporcional		Óptima FIFO		Óptima LIFO	
Configuración	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.
c1-2t-3p	31.93	0.30	24.53	0.39	23.95	0.40	26.09	0.37
c2-2t-5p	21.16	0.45	17.29	0.55	17.18	0.56	21.79	0.44
c3-2t-5p	20.99	0.46	17.00	0.56	16.99	0.56	21.21	0.45
c4-2t-8p	17.41	0.55	17.23	0.55	17.20	0.56	20,60	0.46
c5-2t-10p	17.48	0.55	17.30	0.55	17.29	0.55	20.44	0.47

se observa, de forma gáfica, las mismas conclusiones que ya obtuvimos al comparar los tiempos de la tabla 2.30.

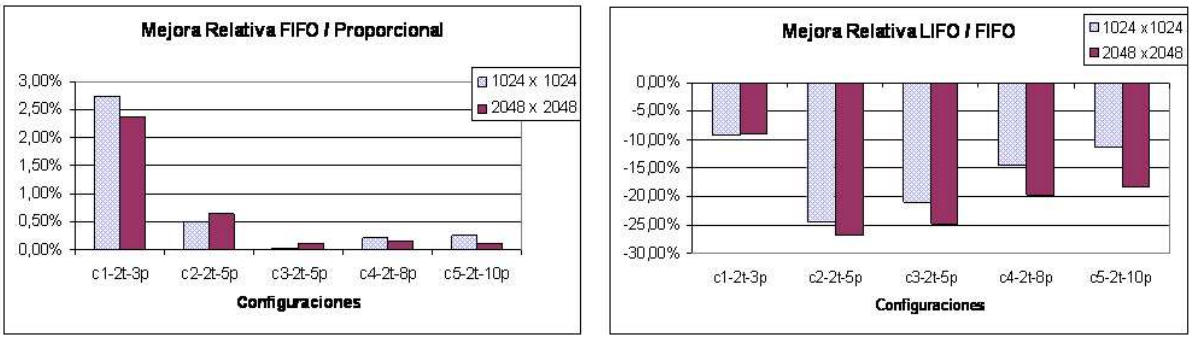

Figura 2.26: Mejora relativa al resolver los problemas (1024×1024 y 2048×2048) con las 5 configuraciones utilizadas. Izquierda: Mejora al resolver el problema con la distribución óptima FIFO frente a la ejecución del programa con la distribución proporcional. Derecha: Mejora obtenida al resolver el problema con la distribución óptima LIFO frente a la distribución óptima FIFO.

La figura 2.26-Izquierda muestra la mejora relativa que se consigue al ejecutar con nuestra distribución óptima FIFO frente a la distribución proporcional para los dos problemas. En todos los casos la distribución óptima FIFO es mejor que la distribución proporcional, aunque sea por un porcentaje de tiempo muy escaso. El bajo porcentaje de mejora se debe al reducido tiempo de ejecución total del programa para resolver el problema de mayor tamaño (2048×2048), solamente con la configuración *c1-2t-3p* el tiempo de ejecución supera los 20 segundos. Presentamos también la mejora relativa al ejecutar con la distribución óptima LIFO frente a la distribución óptima FIFO (figura 2.26-Derecha). Se observa en la gráfica la ventaja de utilizar la estrategia FIFO, puesto que, para los dos problemas, obtenemos valores negativos en la mejora relativa LIFO / FIFO. El beneficio de utilizar la estrategia FIFO llega en ocasiones a más del 25 %.

En la figura 2.27 se muestra gráficamente la relación obtenida entre las aceleraciones reales de las ejecuciones con las 4 distribuciones utilizadas. En esta figura no se incluye la aceleración máxima en el sistema para cada configuración, debido a que las aceleraciones reales se encuentran muy por debajo de ellas.

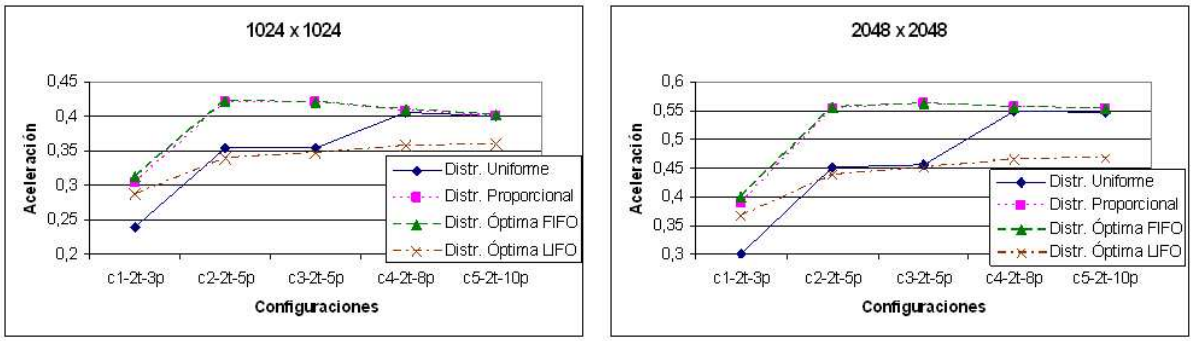

Figura 2.27: Comparativa de las aceleraciones conseguidas al ejecutar el problema con las 4 distribuciones, para los dos problemas (1024×1024 y 2048×2048) y las 5 configuraciones utilizadas.

Comparando las aceleraciones de forma gráfica (figura 2.27), comprobamos que la aceleración obtenida con la distribución óptima para la estrategia LIFO se encuentra en todo momento por debajo de las distribuciones proporcional y óptima para la estrategia FIFO. Para estas dos últimas distribuciones las aceleraciones son prácticamente idénticas para todas las configuraciones y los dos problemas. La aceleración conseguida con la distribución uniforme es la peor de todas para el sistema más pequeño (configuración *c1-2t-3p*), en las dos configuraciones intermedias se mantiene similar a la distribución óptima LIFO y para los sistemas de mayor tamaño se consigue una aceleración próxima a la aceleración óptima FIFO.

En la tabla 2.31 se pueden consultar las distribuciones realizadas para las pruebas mostradas de la tabla 2.30. La distribución uniforme no se incluye en la tabla puesto que la asignación es la misma para todos los procesadores. En la tabla se incluye también el tiempo que tarda el método exacto en calcular las distribuciones óptimas para las estrategias FIFO y LIFO.

Si comparamos las cantidades asignadas en las distribuciones proporcional y óptima FIFO, observamos que las diferencias entre el número de filas que corresponde a los procesadores intermedios y a los procesadores lentos se reduce en la distribución óptima. Esto se debe al efecto del costo de las comunicaciones en los tiempos de ejecución. El tiempo necesario para transferir los datos desde y hacia el maestro provoca que las ventajas obtenidas, por la mayor velocidad de cómputo de las máquinas intermedias, queden anuladas. De forma que la distribución óptima se obtiene cuando se reduce el número de filas asignado a los procesadores intermedios; equilibrando el tiempo de cómputo con el tiempo de comunicación.

En la tabla 2.31 se muestra el tiempo que tarda el algoritmo exacto en calcular las distribuciones óptimas para los dos modelos. Como se puede observar, el tiempo aumenta a medida que se incrementa el tamaño del problema o el número de esclavos en el sistema. Además se puede comprobar, igual que sucedió en el problema descrito en la sección 2.10, que el tiempo para obtener la distribución óptima para la estrategia LIFO es, aproximadamente, la mitad del tiempo necesario para la distribución de la estrategia FIFO en todos los casos.

En estas pruebas no resulta rentable utilizar el método exacto, puesto que si sumamos el tiempo invertido en el cálculo de la distribución óptima FIFO, con el tiempo necesario para ejecutar el programa con esta distribución, el valor obtenido es superior al invertido al utilizar la distribución proporcional. El tiempo total de ejecución del programa es muy pequeño (menos de 30 segundos) y el método exacto está pensado para problemas más grandes, que requieran más tiempo de ejecución, como ya se comentó para el

Tabla 2.31: Distribuciones de tareas a los procesadores calculadas por el modelo y tiempo empleado para ejecutar el método exacto que las calcula para los subsistemas heterogéneos del *cluster* utilizando diferentes configuraciones de máquinas intermedias y lentas para el problema de la FFT-2D.

Problema 1024×1024					
	Proporcional	Óptima FIFO		Óptima LIFO	
Conf.	Distribución	Distribución	Tiempo	Distribución	Tiempo
c1-2t-3p	$m_I = 731$ $m_L = 293$	$m_I = 694$ $m_L = 339$	0.22	$m_I = 811$ $m_L = 213$	0.11
c2-2t-5p	$m_I = 366$ $m_L = 146$	$m_I = 347$ $m_L = 165$	0.67	$m_I = 604\ 291$ $m_L = 76\ 53$	0.34
c3-2t-5p	$m_I = 301$ $m_L = 121$	$m_I = 295$, i = 1..2 $m_I = 294$, i = 3 $m_L = 140$	0.67	$m_I = 577\ 278$ $m_L = 134\ 35$	0.36
c4-2t-8p	$m_I = 223$ $m_L = 89$, i = 4..6 $m_L = 88$, i = 7	$m_I = 210$, i = 1 $m_I = 209$, i = 2,3 $m_L = 99$	1.36	$m_I = 549\ 265\ 127$ $m_L = 33\ 23$ $16\ 11$	0.67
c5-2t-10p	$m_I = 190$ $m_L = 76$, i = 4..8 $m_L = 74$, i = 9	$m_I = 176$, i = 1 $m_I = 175$, i = 2,3 $m_L = 83$	1.82	$m_I = 541\ 262\ 126$ $m_L = 33\ 23\ 16$ $11\ 7\ 5$	0.91
Problema 2048×2048					
	Proporcional	Óptima FIFO		Óptima LIFO	
Conf.	Distribución	Distribución	Tiempo	Distribución	Tiempo
c1-2t-3p	$m_I = 1463$ $m_L = 585$	$m_I = 1395$ $m_L = 653$	0.86	$m_I = 1616$ $m_L = 432$	0.50
c2-2t-5p	$m_L = 731$ $m_L = 293$	$m_I = 698$ $m_L = 326$	2.92	$m_I = 1185\ 593$ $m_L = 158\ 112$	1.48
c3-2t-5p	$m_I = 602$ $m_L = 242$	$m_I = 591$, i = 1,2 $m_L = 590$, i = 3 $m_L = 276$	2.68	$m_I = 1127\ 564\ 282$ $m_L = 75$	1.62
c4-2t-8p	$m_I = 445$ $m_L = 178$, i = 4..6 $m_L = 179$, i = 7	$m_I = 421$ $m_L = 197$, i = 4 $m_L = 196$, i = 5..7	6.27	$m_I = 1065\ 534\ 267$ $m_L = 71\ 50$ $36\ 25$	3.21
c5-2t-10p	$m_I = 379$ $m_L = 152$, i = 4..8 $m_L = 151$, i = 9	$m_I = 353$ $m_L = 165$, i = 4..8 $m_L = 164$, i = 9	8.05	$m_I = 1049\ 526\ 263$ $m_L = 69\ 50\ 35$ $25\ 18\ 13$	4.32

problema del producto de matrices (sección 2.10.2). Como en el *cluster* utilizado, con máquinas intermedias y lentas únicamente, no se pueden ejecutar problemas más grandes, comprobaremos la eficiencia del método exacto en la sección 2.11.6, donde trabajamos con todas las máquinas disponibles en el sistema.

2.11.6. Predicción de la distribución óptima de trabajos con 3 tipos de procesadores

El siguiente paso consiste en añadir la máquina de memoria compartida (los 4 procesadores más rápidos) en nuestro *cluster* de pruebas. Como se comprueba en los resultados presentados durante esta sección, la inclusión de esta nueva máquina modifica considerablemente los resultados obtenidos.

Utilizamos las mismas configuraciones de procesadores en las pruebas que las empleadas en el problema del producto de matrices. En la tabla 2.32 volvemos a mostrar las configuraciones elegidas para incluir las aceleraciones máximas, que es posible alcanzar en el sistema para cada una de estas configuraciones y para cada tamaño de problema al resolver la FFT-2D.

Tabla 2.32: Configuraciones de los sistemas heterogéneos utilizados en las pruebas computacionales de este problema y aceleraciones máximas que se pueden alcanzar en estos sistemas. En estas configuraciones se emplean todos los tipos de máquinas diferentes existentes en el *cluster*: rápidos (R), intermedios (I) y lentos (L).

Nombre Configuración	Maestro	Esclavos	Aceleraciones Máximas 1024×1024	2048×2048	4096×4096
c6-2t-5p	R	R R I I	3.58	3.31	2.35
c7-2t-8p	R	R R R I I I I	6.16	5.61	3.69
c8-2t-8p	R	R R R L L L L	4.13	4.04	3.45
c1-3t-8p	R	I I I I L L L	4.01	3.39	1.03
c2-3t-8p	R	R R I I L L L	4.43	4.08	2.68
c3-3t-9p	R	R R R I I I I L	6.45	5.87	3.80
c4-3t-13p	R	R R R I I I I L L L L L	7.57	6.91	4.25
c5-3t-14p	R	R R R I I I I L L L L L L	7.86	7.16	4.36

Como se observa en la tabla 2.32, la aceleración máxima que se puede alcanzar en estos subsistemas se reduce considerablemente al trabajar sobre el problema de 4096×4096, debido al incremento en el tiempo de ejecución para las máquinas intermedias y lentas (se puede comprobar en la tabla 2.24 los tiempos secuenciales para los distintos tipos de máquinas y tamaños de problemas). Para el problema de mayor tamaño, 8192×8192, no podemos obtener las aceleraciones máximas, puesto que este problema no fue posible ejecutarlo de forma secuencial en los dos tipos de máquinas más lentos.

La tabla 2.33 presenta los tiempos obtenidos para las pruebas realizadas. El formato de esta tabla es el mismo que el de la tabla 2.30: se muestra el tiempo obtenido al ejecutar, utilizando las distribuciones uniforme, proporcional, óptima para la estrategia FIFO y óptima para la estrategia LIFO. Al introducir las máquinas rápidas en las ejecuciones, se ha conseguido incrementar el tamaño de los problemas de prueba hasta 8192×8192 mientras que sin ellas, únicamente se podía llegar a 2048×2048.

Los tiempos de la tabla 2.33 se han reducido notablemente frente a los presentados en la tabla 2.30, lo que se refleja en el hecho de mejorar el resultado secuencial prácticamente con todas las configuraciones seleccionadas; la única configuración para la que no se consigue mejorar el resultado secuencial es *c1-3t-8p*, donde todos los esclavos son máquinas intermedias y lentas, y no hay ningún procesador rápido trabajando; el problema de mayor tamaño no ha podido ser ejecutado con la estrategia LIFO para esta distribución. Las diferencias entre los tiempos obtenidos con las configuraciones óptimas y las distribuciones proporcional y uniforme son muy superiores a las obtenidas con el cluster anterior. En la figura 2.28-Izquierda mostramos de forma gráfica esta mejora relativa, que obtenemos al utilizar la distribución óptima FIFO, frente a la distribución proporcional para dos de los problemas ejecutados: uno de los problemas pequeños (1024×1024) y uno de los dos problemas grandes (4096×4096). El beneficio obtenido al utilizar la distribución óptima alcanza incluso el 50 %.

Un aspecto que llama la atención es el cambio producido en la comparación entre los tiempos óptimos de las dos estrategias de resolución (figura 2.28-Derecha). Mientras que en el *cluster* de las máquinas intermedias y lentas, siempre es mejor resolver el problema mediante la estrategia FIFO, ahora la estrategia óptima de resolución es la estrategia LIFO, mejorando hasta en un 40 % los resultados FIFO. La única excepción la constituye la combinación *c1-3t-8p*. Para esta configuración se obtienen tiempos muy superiores al resto de las configuraciones, especialmente para las distribuciones óptimas; se observa también que la diferencia entre las distribuciones óptimas y las distribuciones uniforme y proporcional es mucho más reducida. Es decir, al utilizar una configuración donde sólo existan esclavos de los dos tipos de máquinas más lentos, el comportamiento del sistema es similar al obtenido al utilizar un *cluster* donde sólo existan estos dos tipos de procesadores. Con la ventaja de que, al utilizar un procesador rápido como maestro hemos podido ejecutar los problemas de mayor tamaño.

En la figura 2.29 mostramos también de forma gráfica la relación entre las aceleraciones obtenidas con las 4 distribuciones para dos problemas (1024×1024 y 4096×4096). En estas gráficas tampoco incluimos

Tabla 2.33: Comparación de los tiempos de ejecución del algoritmo paralelo de la FFT bidimensional al ejecutar con las distintas distribuciones de trabajo: uniforme, proporcional y óptima para la estrategia FIFO y óptima para la estrategia LIFO.

Tamaño de problema: 1024×1024								
	Uniforme		Proporcional		Óptima FIFO		Óptima LIFO	
Configuración	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.	Tiempo	Acelerac.
c6-2t-5p	2.54	0.57	2.21	0.65	1.93	0.74	1.25	1.15
c7-2t-8p	2.53	0.57	2.29	0.63	1.44	1.00	1.12	1.27
c8-2t-8p	3.01	0.48	1.83	0.78	1.51	0.95	1.09	1.32
c1-3t-8p	3.74	0.38	3.69	0.39	3.64	0.39	3.63	0.37
c2-3t-8p	3.19	0.45	2.43	0.59	1.93	0.74	1.27	1.13
c3-3t-9p	2.91	0.49	2.40	0.60	1.44	1.00	1.13	1.27
c4-3t-13p	3.22	0.45	2.75	0.52	1.44	1.00	1.13	1.27
c5-3t-14p	3.33	0.43	2.86	0.50	1.44	1.00	1.13	1.27
Tamaño de problema: 2048×2048								
c6-2t-5p	10.15	0.61	8.77	0.71	8.43	0.74	5.10	1.22
c7-2t-8p	9.48	0.66	8.55	0.73	6.37	0.98	4.34	1.44
c8-2t-8p	12.06	0.52	7.00	0.89	6.36	0.98	4.26	1.46
c1-3t-8p	14.57	0.43	14.32	0.44	14.04	0.44	14.38	0.43
c2-3t-8p	12.77	0.49	9.19	0.68	8.37	0.74	5.04	1.24
c3-3t-9p	11.33	0.55	8.85	0.70	6.36	0.98	4.34	1.44
c4-3t-13p	11.85	0.53	9.90	0.63	6.36	0.98	4.45	1.40
c5-3t-14p	12.17	0.51	10.30	0.60	6.36	0.98	4.45	1.40
Tamaño de problema: 4096×4096								
c6-2t-5p	43.30	0.68	37.83	0.78	36.80	0.80	22.57	1.30
c7-2t-8p	39.74	0.74	35.87	0.82	26.09	1.12	18.71	1.57
c8-2t-8p	52.80	0.56	31.13	0.94	26.02	1.13	19.11	1.53
c1-3t-8p	62.07	0.47	59.04	0.50	57.39	0.51	59.16	0.50
c2-3t-8p	55.97	0.52	38.47	0.76	36.80	0.80	21.98	1.33
c3-3t-9p	50.38	0.58	36.95	0.79	26.09	1.12	18.81	1.56
c4-3t-13p	48.91	0.60	40.93	0.72	26.09	1.12	19.19	1.53
c5-3t-14p	49.81	0.60	44.16	0.66	26.09	1.12	19.22	1.53
Tamaño de problema: 8192×8192								
c6-2t-5p	462.28	0.36	188.82	0.89	166.45	1.01	106.79	1.58
c7-2t-8p	172.21	0.98	157.21	1.07	130.84	1.29	89.50	1.88
c8-2t-8p	345.40	0.49	197.76	0.85	130.84	1.29	97.22	1.73
c1-3t-8p	352.71	0.48	267.47	0.63	241.41	0.70	–	–
c2-3t-8p	361.79	0.47	213.00	0.79	166.45	1.01	105.85	1.59
c3-3t-9p	259.19	0.65	167.28	1.01	130.84	1.29	89.77	1.88
c4-3t-13p	259.57	0.65	177.77	0.95	130.84	1.29	89.77	1.88
c5-3t-14p	255.69	0.66	183.14	0.92	130.84	1.29	89.77	1.88

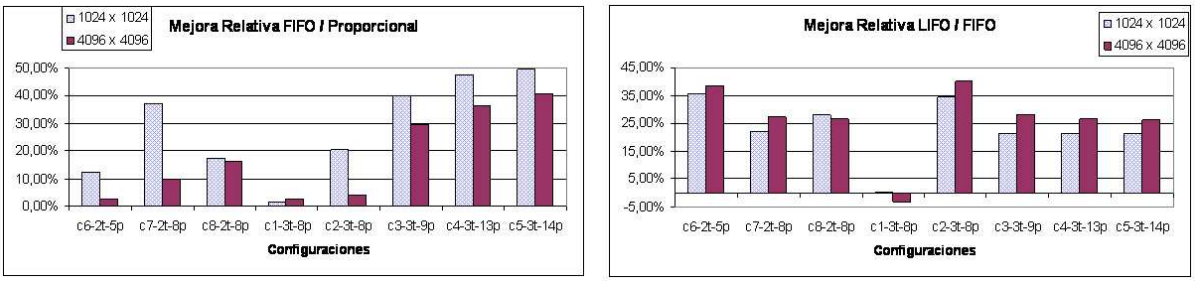

Figura 2.28: Mejora relativa al resolver los problemas 1024 × 1024 y 4096 × 4096 con las 8 configuraciones utilizadas. Izquierda: Mejora relativa al resolver el problema con la distribución óptima FIFO frente a la ejecución del programa con la distribución proporcional. Derecha: Mejora al emplear la distribución óptima de la estrategia LIFO frente a la distribución óptima en la estrategia FIFO.

las aceleraciones máximas posibles en el sistema, debido a que las aceleraciones son muy reducidas.

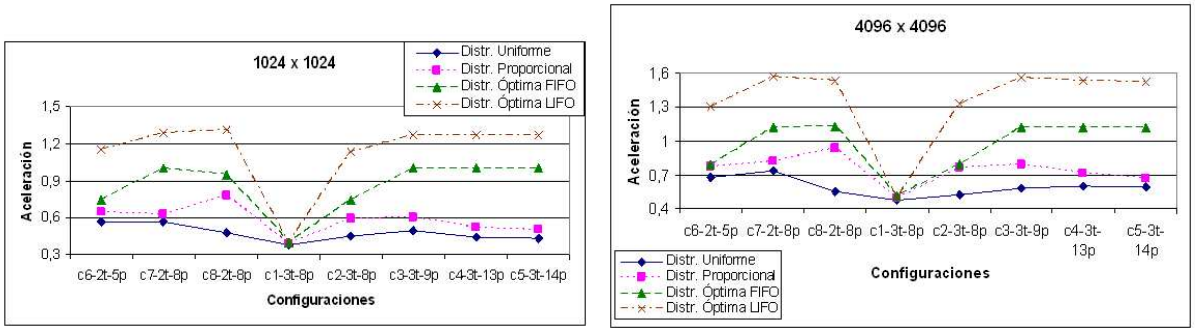

Figura 2.29: Aceleraciones conseguidas al ejecutar el problema con las 4 distribuciones para dos problemas (1024 × 1024 y 4096 × 4096) y todas las configuraciones utilizadas.

La forma de las gráficas de la figura 2.29 para los dos problemas es la misma. Se observan las 4 líneas de las aceleraciones separadas y manteniendo siempre el mismo orden; y el efecto de la distribución *c1-3t-8p*, donde las aceleraciones de las distribuciones óptimas se reducen considerablemente, debido a la ausencia de máquinas rápidas entre los esclavos.

En la tabla 2.33 podemos ver también que los tiempos de la distribución óptima FIFO en las configuraciones *c7-2t-8p*, *c8-2t-8p* *c3-3t-9p*, *c4-3t-13p* y *c5-3t-14p* coinciden en la mayoría de los tamaños de problemas, para las dos estrategias de resolución. Lo mismo sucede entre las configuraciones *c6-2t-5p* y *c2-3t-8p*. Esto ocurre porque el método exacto también permite determinar el número óptimo de procesadores, como ya comprobamos en la sección 2.10.1, por lo que en todos estos casos, realmente se ejecuta la misma distribución sobre los mismos procesadores y se asigna una cantidad de trabajo igual a cero a los restantes procesadores. Todas las distribuciones con las que se han realizado las pruebas proporcionales y óptimas, se pueden comprobar en la tabla 2.34 para el problema 1024 × 1024, en la tabla 2.35 para el problema 2048 × 2048, en la tabla 2.36 para 4096 × 4096 y en la tabla 2.37 para 8192 × 8192. La información sobre la distribución uniforme no está incluida en las tablas, puesto que se asigna a todos los procesadores la misma cantidad de trabajo, independiente de su velocidad de cómputo y comunicación.

Para la estrategia FIFO se observa que, la distribución óptima para la mayor parte de las configuraciones, se obtiene cuando se considera el subsistema homogéneo de los procesadores rápidos del *cluster*, sin introducir otros procesadores más lentos que no reducen el tiempo total, sino que retrasan la resolución del problema. Las distribuciones para las configuraciones *c6-2t-5p* y *c2-3t-8p* son iguales: se asigna

Tabla 2.34: Distribuciones de tareas a los procesadores calculadas por el modelo y tiempo empleado para ejecutar el método exacto que las calcula para los subsistemas heterogéneos del *cluster* para el problema de la FFT-2D, para el problema 1024×1024.

Conf.	Proporcional Distribución	Óptima FIFO Distribución	Óptima FIFO Tiempo	Óptima LIFO Distribución	Óptima LIFO Tiempo
c6-2t-5p	$m_R = 299$ $m_I = 213$	$m_R = 512$ $m_I = 0$	0.69	$m_R = 468\ 398$ $m_I = 106\ 52$	0.33
c7-2t-8p	$m_R = 175$ $m_I = 125$, i = 4..6 $m_I = 124$, i = 7	$m_R = 341$ $m_I = 1$, i = 4 $m_I = 0$, i = 5..7	1.37	$m_R = 350\ 297\ 253$ $m_I = 67\ 33\ 16\ 8$	0.66
c8-2t-8p	$m_R = 247$ $m_L = 71$, i = 4..6 $m_L = 70$, i = 70	$m_R = 342$, i = 1 $m_R = 341$, i = 2,3 $m_L = 0$	1.37	$m_R = 360\ 308\ 262$ $m_L = 37\ 26\ 18\ 13$	0.66
c1-3t-8p	$m_I = 197$ $m_L = 79$, i = 5,6 $m_L = 78$, i = 7	$m_I = 190$, i = 1 $m_I = 189$, i = 2..4 $m_L = 89$	1.44	$m_I = 531\ 262\ 130\ 64$ $m_L = 17\ 12\ 8$	0.71
c2-3t-8p	$m_R = 239$ $m_I = 171$ $m_L = 68$	$m_R = 512$ $m_I = 0$ $m_L = 0$	1.37	$m_R = 454\ 387$ $m_I = 103\ 51$ $m_L = 12\ 10\ 7$	0.68
c3-3t-9p	$m_R = 167$ $m_I = 119$ $m_L = 47$	$m_R = 341$ $m_I = 1$, i = 4 $m_I = 0$, i = 5..7 $m_L = 0$	1.62	$m_R = 349\ 297\ 252$ $m_I = 67\ 33\ 16\ 8$ $m_L = 2$	0.82
c4-3t-13p	$m_R = 141$ $m_I = 100$ $m_L = 40$, i = 8..11 $m_L = 41$, i = 12	$m_R = 341$ $m_I = 1$, i = 4 $m_I = 0$, i = 5..7 $m_L = 0$	2.51	$m_R = 348\ 296\ 252$ $m_I = 67\ 33\ 16\ 8$ $m_L = 2\ 1\ 1$ $m_L = 0$, i =11,12	1.25
c5-3t-14p	$m_R = 135$ $m_I = 97$ $m_L = 39$, i = 8..12 $m_L = 38$, i = 13	$m_R = 341$ $m_I = 1$, i = 4 $m_I = 0$, i = 5..7 $m_I = 0$	2.72	$m_R = 348\ 296\ 252$ $m_I = 67\ 33\ 16\ 8$ $m_L = 2\ 1\ 1$ $m_L = 0$, i = 11..13	1.34

una cantidad de filas a procesar a las máquinas rápidas y no se utilizan las máquinas intermedias y lentas. Lo mismo sucede con las configuraciones *c7-2t-8p*, *c8-2t-8p*, *c3-3t-9p*, *c4-3t-13p* y *c5-3t-14p* para los problemas 2048×2048 y 8192×8192. Para los problemas de tamaño 1024×1024 y 4096×4096, las distribuciones óptimas para las configuraciones *c7-2t-8p*, *c3-3t-9p*, *c4-3t-13p* y *c5-3t-14p* son también iguales, pero en este caso se asigna 1 unidad de trabajo a la primera máquina intermedia. Sin embargo, para estos dos problemas la distribución óptima en la configuración *c8-2t-8p* es diferente, debido a que no existen máquinas intermedias entre los esclavos, por lo que el trabajo de la máquina intermedia en las otras configuraciones se asigna aquí a un procesador rápido en lugar de a uno lento. También para el método LIFO, se observa que para el problema más grande no es eficiente utilizar todos los procesadores en la resolución del problema. En la estrategia LIFO y la configuración *c1-3t-8p* para el problema 8192×8192 observamos la distribución óptima que obtenemos con el método exacto. El motivo de que esta distribución no haya podido ser ejecutada está en la cantidad de filas que se asigna al primer procesador intermedio (3749 filas), ya que este procesador no tiene capacidad para poder recibirlas.

En las tablas 2.34, 2.35, 2.36 y 2.37 se presentan también los tiempos invertidos por el método exacto en minimizar los modelos presentados en la sección 2.5 y 2.6 y obtener las distribuciones óptimas para cada tamaño de problema y configuración. Como también se ha observado anteriormente, el tiempo que tarda el método exacto en obtener la distribución óptima para el segundo esquema de resolución, está en torno a la mitad del tiempo necesario para el primero. Calculando la suma entre el tiempo de ejecución del método exacto, más el tiempo de resolución del problema con la distribución óptima LIFO, comprobamos que este tiempo es menor que el tiempo de ejecución de la distribución proporcional FIFO en la mayoría

Tabla 2.35: Distribuciones de tareas a los procesadores calculadas por el modelo y tiempo empleado para ejecutar el método exacto que las calcula para los subsistemas heterogéneos del *cluster* para el problema de la FFT-2D, para el problema 2048×2048.

Conf.	Proporcional Distribución	Óptima FIFO Distribución	Tiempo	Óptima LIFO Distribución	Tiempo
c6-2t-5p	$m_R = 597$ $m_I = 427$	$m_R = 1024$ $m_I = 0$	2.81	$m_R = 917\ 791$ $m_I = 225\ 115$	1.33
c7-2t-8p	$m_R = 350$ $m_I = 250$, i = 4..6 $m_I = 248$, i = 7	$m_R = 683$, i = 1..2 $m_R = 682$, i = 3 $m_I = 0$	5.32	$m_R = 680\ 587\ 507$ $m_I = 143\ 74\ 38\ 19$	2.93
c8-2t-8p	$m_R = 494$ $m_L = 141$, i = 4..6 $m_L = 143$, i = 7	$m_R = 683$, i = 1,2 $m_R = 682$, i = 3 $m_L = 0$	5.32	$m_R = 707\ 610\ 526$ $m_L = 78\ 57\ 41\ 29$	2.82
c1-3t-8p	$m_I = 394$ $m_L = 158$, i = 5,6 $m_L = 156$, i = 7	$m_I = 380$ $m_L = 176$	5.85	$m_I = 1026\ 529\ 271\ 139$ $m_L = 37\ 27\ 19$	2.98
c2-3t-8p	$m_R = 478$ $m_I = 341$ $m_L = 137$, i = 5,6 $m_L = 136$, i = 7	$m_R = 1024$ $m_I = 0$ $m_L = 0$	5.67	$m_R = 887\ 766$ $m_I = 217\ 112$ $m_L = 30\ 21\ 15$	2.78
c3-3t-9p	$m_R = 333$ $m_I = 238$ $m_L = 97$	$m_R = 683$, i = 1,2 $m_R = 682$, i = 3 $m_I = m_L = 0$	6.19	$m_R = 678\ 586\ 505$ $m_I = 143\ 74\ 38\ 19$ $m_L = 5$	3.69
c4-3t-13p	$m_R = 281$ $m_I = 201$ $m_L = 80$, i = 8..11 $m_L = 81$, i = 12	$m_R = 683$, i = 1,2 $m_R = 682$, i = 3 $m_I = m_L = 0$	9.97	$m_R = 675\ 583\ 504$ $m_I = 142\ 74\ 38\ 19$ $m_L = 5\ 3\ 2\ 2\ 1$	5.63
c5-3t-14p	$m_R = 270$ $m_I = 193$ $m_L = 77$, i = 8..12 $m_L = 81$, i = 13	$m_R = 683$, i = 1,2 $m_R = 682$, i = 3 $m_I = m_L = 0$	11.23	$m_R = 676\ 583\ 503$ $m_I = 141\ 74\ 38\ 19$ $m_L = 5\ 3\ 2\ 2\ 1\ 1$	5.88

de las pruebas, exceptuando la configuración *c1-3t-8p*, donde no trabajan las máquinas rápidas.

2.11.7. Herramienta de predicción de esquema y distribución óptima de trabajos

Como ya se especificó en la sección 2.10 (el problema del producto de matrices), se ha desarrollado una herramienta que permite determinar la mejor forma de resolver el problema (de forma secuencial o paralela y, en este último caso, cuál de las dos estrategias y su distribución). En la tabla 2.38 aparece el método que debe utilizarse para resolver el problema para cada una de las distribuciones con sólo 2 tipos de máquinas y en el *cluster* con los 3 tipos de procesadores. La columna *Forma de Resolución* puede tomar uno de los tres siguiente valores: *Secuencial*, *FIFO* o *LIFO*.

Las distribuciones óptimas de trabajos a los esclavos, para cada caso, es la misma que se obtuvo en las secciones anteriores (2.11.5 y 2.11.6), por lo que puede consultarse en las tablas correspondientes.

En la tabla 2.38 podemos comprobar el alto porcentaje de acierto de nuestra herramienta, al comparar estos resultados con los presentados en las tablas de los tiempos de ejecución de esta sección (2.30 y 2.33). En las configuraciones con 2 tipos de procesadores, el mejor resultado se obtiene cuando se ejecuta el problema en un procesador intermedio de forma secuencial, para tres de las configuraciones la herramienta no selecciona el resultado secuencial, sino que elige la estrategia FIFO. En estas 6 ejecuciones de las 42

Tabla 2.36: Distribuciones de tareas a los procesadores calculadas por el modelo y tiempo empleado para ejecutar el método exacto que las calcula para los subsistemas heterogéneos del *cluster* para el problema de la FFT-2D, para el problema 4096×4096.

Conf.	Proporcional Distribución	Óptima FIFO Distribución	Óptima FIFO Tiempo	Óptima LIFO Distribución	Óptima LIFO Tiempo
c6-2t-5p	$m_R = 1195$ $m_I = 853$	$m_R = 2048$ $m_I = 0$	10.27	$m_R = 1805\ 1575$ $m_I = 468\ 248$	5.25
c7-2t-8p	$m_R = 699$ $m_I = 500$, i = 4..6 $m_I = 499$, i = 7	$m_R = 1365$ $m_I = 1$, i = 4 $m_I = 0$, i = 5..7	21.04	$m_R = 1332\ 1161\ 1013$ m_I 300 160 85 45	10.49
c8-2t-8p	$m_R = 989$ $m_L = 282$, i = 4..6 $m_L = 283$, i = 7 $m_I = 0$, i = 5..7	$m_R = 1366$, i = 1 $m_R = 1365$, i = 2,3 $m_I = 1$, i = 4	20.74	$m_R = 1393\ 1215\ 1060$ $m_L = 156\ 118\ 88\ 66$	10.72
c1-3t-8p	$m_I = 788$ $m_L = 315$, i = 5,6 $m_L = 314$, i =7	$m_I = 774$ $m_L = 334$	22.93	$m_I = 1191\ 1060\ 562\ 298$ $m_L = 80\ 60\ 45$	10.74
c2-3t-8p	$m_R = 1911$ $m_I = 1365$ $m_L = 546$, i = 5,6 $m_L = 548$, i = 7	$m_R = 4096$ $m_I = 0$ $m_L = 0$	21.26	$m_R = 1740\ 1518$ $m_I = 451\ 239$ $m_L = 64\ 48\ 36$	10.72
c3-3t-9p	$m_R = 667$ $m_I = 476$ $m_L = 191$	$m_R = 1365$ $m_I = 1$, i = 4 $m_I = 0$, i = 5..7 $m_L = 0$	25.08	$m_R = 1327\ 1158\ 1009$ $m_I = 300\ 159\ 84\ 45$ $m_L = 14$	12.44
c4-3t-13p	$m_R = 562$ $m_I = 402$ $m_L = 161$, i = 8..11 $m_L = 158$, i = 12	$m_R = 1365$ $m_I = 1$, i = 4 $m_I = 0$, i = 5..7 $m_L = 0$	39.45	$m_R = 1318\ 1150\ 1002$ $m_I = 298\ 158\ 84\ 44$ $m_L = 14\ 10\ 8\ 6\ 4$	19.91
c5-3t-14p	$m_R = 541$ $m_I = 386$ $m_L = 155$, i = 8..12 $m_L = 154$, i = 13	$m_R = 1365$ $m_I = 1$, i = 4 $m_I = 0$, i = 5..7 $m_L = 0$	42.91	$m_R = 1317\ 1149\ 1002$ $m_I = 297\ 158\ 84\ 44$ $m_L = 14\ 10\ 8\ 6\ 4\ 3$	21.57

ejecuciones totales, el método exacto no ofrece el resultado correcto; es decir, el error producido es sólo del 14 %. Además, a pesar de no ser la forma óptima de resolución, el método elige ejecutar con la mejor de las dos estrategias paralelas. En el resto de las ejecuciones, el método proporciona la estrategia correcta de resolución, la estrategia LIFO para todas las pruebas, excepto para la configuración *c1-3t-5p*, donde el mejor método para resolver el problema es la ejecución secuencial.

2.12. Conclusiones

En este capítulo hemos presentado una metodología general para una amplia clase de algoritmos maestro-esclavo, basada en modelos analíticos, que permite sintonizar aplicaciones paralelas en sistemas heterogéneos; considerando la heterogeneidad debida tanto a las diferencias en las velocidades de cómputo de los procesadores, como a las diferencias en las comunicaciones entre ellos.

Hemos propuesto un algoritmo de programación dinámica para resolver el problema de optimizar la asignación de recursos, basado en el modelo analítico anterior, que nos permite encontrar los parámetros de una ejecución óptima; incluyendo entre los parámetros, el conjunto óptimo de procesadores a utilizar en la resolución de un problema.

Tabla 2.37: Distribuciones de tareas a los procesadores calculadas por el modelo y tiempo empleado para ejecutar el método exacto que las calcula para los subsistemas heterogéneos del *cluster* para el problema de la FFT-2D, para el problema 8192×8192.

Conf.	Proporcional Distribución	Óptima FIFO Distribución	Tiempo	Óptima LIFO Distribución	Tiempo
c6-2t-5p	$m_R = 2389$ $m_I = 2389$	$m_R = 4096$ $m_I = 0$	42.51	$m_R = 1805\ 1575$ $m_I = 468\ 248$	20.96
c7-2t-8p	$m_R = 1399$ $m_I = 999$, i = 4..6 $m_I = 998$, i = 7	$m_R = 2731$, i = 1,2 $m_R = 2730$, i = 3 $m_I = 0$	85.41	$m_R = 2615\ 2301\ 2025$ $m_I = 616\ 342\ 189\ 104$	43.13
c8-2t-8p	$m_R = 989$ $m_L = 565$, i = 4..6 $m_L = 566$, i = 7	$m_R = 2731$, i = 1,2 $m_R = 2730$, i = 3 $m_L = 0$	88.48	$m_R = 2860\ 2518\ 2216$ $m_L = 192\ 160\ 134\ 112$	44.81
c1-3t-8p	$m_I = 1575$ $m_L = 630$, i = 5,6 $m_L = 632$, i = 7	$m_I = 1711$ $m_L = 450$, i = 5 $m_L = 449$, i = 6,7	94.81	$m_I = 3749\ 2267\ 1252\ 692$ $m_L = 92\ 76\ 64$	44.99
c2-3t-8p	$m_R = 1911$ $m_I = 1365$ $m_L = 546$, i = 5,6 $m_L = 548$, i = 7	$m_R = 4096$ $m_I = 0$ $m_L = 0$	88.13	$m_R = 3502\ 3082$ $m_I = 940\ 519$ $m_L = 59\ 49\ 41$	44.50
c3-3t-9p	$m_R = 1334$ $m_I = 953$ $m_L = 378$	$m_R = 2731$, i = 1,2 $m_R = 2730$, i = 3 $m_I = m_L = 0$	101.42	$m_R = 2615\ 2301\ 2025$ $m_I = 616\ 342\ 189\ 104$ $m_L = 0$	51.21
c4-3t-13p	$m_R = 1124$ $m_I = 803$ $m_L = 321$, i = 9..11 $m_L = 324$, i = 12	$m_R = 2731$, i = 1,2 $m_R = 2730$, i = 3 $m_I = m_L = 0$	162.05	$m_R = 2615\ 2301\ 2025$ $m_I = 616\ 342\ 189\ 104$ $m_L = 0$	85.05
c5-3t-14p	$m_R = 1082$ $m_I = 773$ $m_L = 309$	$m_R = 2731$, i = 1,2 $m_R = 2730$, i = 3 $m_I = m_L = 0$	178.46	$m_R = 2615\ 2301\ 2025$ $m_I = 616\ 342\ 189\ 104$ $m_L = 0$	91.21

Tabla 2.38: Resultados obtenidos con la herramienta: estrategia que debemos usar en la resolución de los problemas.

Configuración	1024×1024	2048×2048	4096×4096	8192×8192
c1-2t-3p	Secuencial	Secuencial	-	-
c2-2t-5p	Secuencial	Secuencial	-	-
c3-2t-5p	FIFO	FIFO	-	-
c4-2t-8p	FIFO	FIFO	-	-
c5-2t-10p	FIFO	FIFO	-	-
c6-2t-5p	LIFO	LIFO	LIFO	LIFO
c7-2t-8p	LIFO	LIFO	LIFO	LIFO
c8-2t-8p	LIFO	LIFO	LIFO	LIFO
c1-3t-5p	Secuencial	Secuencial	Secuencial	Secuencial
c2-3t-8p	LIFO	LIFO	LIFO	LIFO
c3-3t-9p	LIFO	LIFO	LIFO	LIFO
c4-3t-13p	LIFO	LIFO	LIFO	LIFO
c5-3t-14p	LIFO	LIFO	LIFO	LIFO

Esta metodología ha sido probada satisfactoriamente en un *cluster* heterogéneo con 2 problemas de prueba: el producto de matrices y la Transformada Rápida de Fourier bidimensional.

Hemos incluido esta metodología en una herramienta automática, que proporciona la estrategia a utilizar en la resolución del problema maestro-esclavo en una plataforma heterogénea y la distribución óptima de trabajos a los procesadores.

En cualquier caso, aún existen problemas abiertos que nos permitirán ampliar este trabajo: aunque las estrategias FIFO y LIFO representan a un amplio rango de problemas del tipo maestro-esclavo, el grupo de problemas donde los resultados de los esclavos son aceptados por el maestro tan pronto como estén disponibles, no ha sido considerado por nuestra herramienta. Nos planteamos ahora, como trabajo futuro, extender nuestra metodología para cubrir también esta última clase de problemas.

También pretendemos establecer una estrategia dinámica de distribución de tareas basada en la carga de las máquinas del sistema: al iniciar la ejecución, el programa comprueba la carga de cada máquina y predice la mejor distribución de trabajo a los procesadores, resolviendo el problema con los parámetros calculados.

Capítulo 3

El Paradigma *Pipeline* en Sistemas Heterogéneos

3.1. Resumen

En este capítulo estudiamos el rendimiento de una amplia clase de algoritmos *pipeline* en entornos heterogéneos. Del mismo modo que hicimos en el capítulo 2 para los algoritmos maestro-esclavo, no restringimos el concepto de heterogeneidad en el sistema únicamente a las diferencias en las velocidades de cómputo de las máquinas, sino que también consideramos las distintas capacidades en las comunicaciones (sección 1.2).

La asignación de tareas a los procesadores es un factor crucial que determina el rendimiento del algoritmo paralelo. Equilibrar la carga asignada a cada procesador, tomando en consideración las diferencias entre ellos, requiere una distribución de tareas que asigne un número diferente de procesos virtuales a cada procesador. En este capítulo abordamos el problema de encontrar la distribución óptima en algoritmos *pipeline* sobre un anillo de procesadores, desarrollamos un modelo analítico que predice el rendimiento del sistema para el caso general y proponemos estrategias que nos permiten estimar los parámetros necesarios para resolver el problema en el menor tiempo posible: número de procesos virtuales por procesador y tamaño de los paquetes enviados.

Hemos desarrollado también la herramienta *llpW*, que nos permite realizar de forma automática una distribución cíclica por bloques para los algoritmos *pipeline* en entornos heterogéneos. Esta herramienta soporta *pipelines* con un número de etapas muy superior al número de procesadores físicos disponibles en el sistema.

Hemos probado nuestra herramienta con varios algoritmos de programación dinámica. La programación dinámica es una técnica de optimización combinatoria, ampliamente utilizada en la resolución de problemas, fácilmente paralelizable mediante el paradigma *pipeline*. Los resultados computacionales, llevados a cabo sobre un *cluster* de PCs (sección 1.5.1) demuestran la utilidad de la herramienta y la precisión de las predicciones realizadas por el modelo analítico propuesto.

3.2. Introducción

El paradigma *pipeline* ha sido extensamente estudiado en arquitecturas homogéneas [11, 12, 85, 107, 133, 149] y constituye la base de una importante clase de aplicaciones; por ejemplo, de muchos algoritmos paralelos para problemas de programación dinámica [84, 126]. La mayoría de algoritmos *pipeline* parale-

los presentan un comportamiento óptimo si se consideran simplemente desde el punto de vista teórico, cuando se dispone de tantos procesadores como el tamaño de la entrada. Sin embargo, muchos de ellos muestran un mal rendimiento cuando se ejecutan sobre las arquitecturas paralelas actuales, debido a que la implementación de los algoritmos está fuertemente condicionada por la asignación de procesos virtuales a los procesadores físicos y su simulación, y por la granularidad del problema a resolver y de la arquitectura. Existen diversos estudios dedicados a caracterizar los parámetros de la arquitectura, especialmente la latencia de la red para determinar su valor óptimo [46, 167] y a caracterizar los parámetros del problema [11, 12, 85, 133]. Para conseguir una ejecución óptima de un algoritmo *pipeline*, debemos emplear una combinación apropiada de todos estos factores, como se deduce de numerosos trabajos publicados [36, 117, 148, 166, 168].

Se han proporcionado distintas aproximaciones *software* para intentar facilitar la programación de *pipelines*. *HPF* [65, 97] es un lenguaje orientado a paralelismo de datos y en las *approved extensions* de la versión 2.0 se introducen constructores para expresar el paralelismo *pipeline*. Sin embargo, la única implementación existente de *HPF* de acuerdo con algunas de estas extensiones, no considera la asignación óptima de estos recursos. Algo similar ocurre con P^3L [3, 59], un lenguaje orientado a esqueletos que permite expresar tales algoritmos y su combinación con otros paradigmas de programación. Se han desarrollado también librerías para facilitar el diseño de programas paralelos *pipeline*: en [110, 111] se encuentra la descripción de una librería de esqueletos que proporciona métodos para la paralelización de datos y de tareas; y en [182] encontramos una librería que permite disponer de un *pipeline* y *sub-pipelines*; es decir, un *pipeline* donde una de las etapas es, a su vez, otro *pipeline*. Sin embargo, ninguna de estas librerías resuelve el problema de la distribución óptima y en muy pocos casos se implementa un *pipeline* donde el número de etapas sea superior al número de procesadores.

llp (*La Laguna Pipeline*) [85, 129, 131] es una herramienta orientada a algoritmos *pipeline* en sistemas homogéneos que hemos desarrollado. *llp* intentaba cubrir la ausencia de este tipo de herramientas para simplificar la tarea del programador y facilitar el desarrollo de programas *pipeline* parametrizados y mecanismos para obtener la distribución óptima de trabajo a los procesadores. Está implementada utilizando las librerías de paso de mensajes PVM [63, 78] y MPI [89, 134]; soporta la asignación cíclica pura y cíclica por bloques de acuerdo con las especificaciones del usuario; y trabaja con *pipelines* donde el número de etapas puede ser mucho mayor que el número de procesadores físicos disponibles en el sistema. Asimismo, el usuario puede determinar el tamaño de los paquetes transmitidos entre los procesadores. Esta misma idea ha sido incorporada al lenguaje *llc* [64], donde el usuario puede expresar el paralelismo *pipeline* mediante el uso de *pragmas* de *OpenMP* [143, 144].

En entornos heterogéneos, el número de trabajos realizados sobre algoritmos *pipeline* es más reducido [33, 54, 56, 60, 112, 150, 170, 184]. En [54], por ejemplo, se compara la resolución de problemas sobre un anillo de procesadores en entornos homogéneos y heterogéneos. En [170] los autores buscan una buena distribución de tareas mediante heurísticas, buscando el equilibrio de la carga y un volumen de las comunicaciones adecuado; mientras que en [33] consideran la heterogeneidad debida al tipo de recurso y a las políticas de distribución; y en [150] emplean *pipelines* donde el cómputo y las comunicaciones se producen de forma solapada. Encontramos también sistemas para facilitar la distribución de aplicaciones paralelas en sistemas heterogéneos, en [184] presentan un sistema denominado *Prophet* (*Portable Resource management On Parallel HETerogeneous networks*) al que le han añadido un soporte para la distribución equilibrada de carga dinámica. Sin embargo, la mayoría de los trabajos [60, 112] están pensados para problemas donde el número de etapas es igual al número de procesadores disponibles y no se permite la virtualización de los procesos. En [56] se presenta un trabajo que estudia heurísticas para resolver el problema de optimizar la ejecución de algoritmos de programación dinámica sobre un entorno heterogéneo, considerando distinto número de procesos virtuales por procesador. Consideran que los parámetros del sistema deben ser funciones de los parámetros del algoritmo e implementan el algoritmo de programación dinámica realizando una fase de comunicación, donde se produce la sincronización de todos los procesadores.

Para poder trabajar en entornos heterogéneos, hemos introducido nuevas funcionalidades en nuestra herramienta *llp*, generando una nueva versión denominada *llpW* (*llp Weighted*). Esta nueva versión de

la herramienta permite asignar un número diferente de procesos virtuales a cada procesador y definir distintos tamaños de paquetes de datos a comunicar, dependiendo del tipo de las máquinas. Además, *llpW* puede ser utilizada en cualquier entorno, puesto que considera la plataforma homogénea como un caso particular del entorno heterogéneo general, donde todas las máquinas son del mismo tipo.

Hemos realizado experimentos con nuestra herramienta para resolver varios problemas mediante programación dinámica: el problema de la mochila unidimensional (subsección 3.6.1), el problema de los caminos mínimos (subsección 3.6.2), el problema de la asignación de recursos (subsección 3.6.3) y el problema de la búsqueda de la subsecuencia común más larga (subsección 3.6.4). Hemos utilizado algoritmos de programación dinámica porque es una técnica fácilmente paralelizable, mediante *pipelines*, sobre un anillo de procesadores. Los resultados computacionales obtenidos al realizar ejecuciones de algoritmos *pipeline* sobre un *cluster* de PCs (sección 1.5.1) demuestran la utilidad de nuestra herramienta frente a la versión homogénea.

También hemos desarrollado un modelo analítico que considera la heterogeneidad debida tanto al cómputo como a las comunicaciones entre los procesadores. Para predecir el tiempo de ejecución sobre una plataforma heterogénea, tomamos en consideración el hecho de que el cómputo de un *pipeline* está fuertemente influido por el procesador más lento que trabaja en él. La capacidad predictiva del modelo la utilizamos para resolver el problema de la distribución (o *mapping*) óptima (encontrar los parámetros que minimizan el tiempo predicho por el modelo analítico), en lugar de requerir una búsqueda *ad hoc* entre varios parámetros. Implementamos un método exacto de fuerza bruta y un método heurístico para resolver esta minimización y verificamos la precisión de nuestras predicciones.

El resto del capítulo lo hemos estructurado de la siguiente manera: En la sección 3.3 definimos el paradigma *pipeline* y presentamos la librería que desarrollamos para resolver algoritmos *pipeline* sobre un anillo de procesadores homogéneos, *llp* (sección 3.4) y su nueva versión, adaptada a entornos heterogéneos, *llpW* (sección 3.5). Validamos nuestra herramienta, comparando los resultados obtenidos con ambas versiones (sección 3.6). En la sección 3.7 presentamos los modelos analíticos que hemos desarrollado para predecir el tiempo de resolución de estos problemas, considerando tres alternativas: cuando el número de etapas del *pipeline* es igual al número de procesadores (subsección 3.7.1), cuando el número de etapas es superior al número de procesadores y se utiliza una distribución cíclica pura (subsección 3.7.2) y cuando el número de etapas es superior y se resuelve mediante una distribución cíclica por bloques (subsección 3.7.3). Validamos nuestro modelo analítico comparando el tiempo estimado por el modelo con el tiempo real obtenido en la ejecución del programa (sección 3.8). Por último, desarrollamos dos métodos para calcular los valores óptimos de los parámetros: un método exacto y un método heurístico (sección 3.9) y realizamos una experiencia computacional que demuestra la precisión de nuestras predicciones (sección 3.10). En la sección 3.11 se encuentran las conclusiones de este capítulo.

3.3. El paradigma *pipeline*

Tradicionalmente, una técnica que ha permitido aumentar la capacidad de ejecución de instrucciones en un procesador consiste en dividir la ejecución completa de la instrucción en varias actividades que se puedan realizar de forma solapada: búsqueda de la siguiente instrucción, búsqueda de los operandos, ejecuciones artiméticas, etc. A esta técnica se le denomina ***pipeline*** y forma parte de la base del diseño de muchos procesadores.

Esta técnica de división del trabajo ha sido adaptada como técnica algorítmica paralela de resolución de problemas, obteniendo el paradigma de programación *pipeline*. Para resolver un problema es necesario particionar el conjunto de tareas en n partes, cada una de las cuales se denominan **etapas** del algoritmo. Una vez establecidas estas etapas, se reparten entre los p procesadores físicos disponibles en el sistema, que se encuentran configurados en forma de *array* unidimensional de procesadores. Cada etapa se alimenta de datos que provienen del procesador situado a su izquierda en el *array* y envía datos al procesador que se encuentra a su derecha. La situación más simple consiste en trabajar con un *pipeline* ideal donde

disponemos de tantos procesadores como etapas. Sin embargo, en la práctica, el número de etapas puede superar al número de procesadores físicos disponibles. En este caso es habitual configurar el conjunto real de procesadores siguiendo una estructura de anillo unidireccional sobre el que se virtualiza el *pipeline* (figura 3.1).

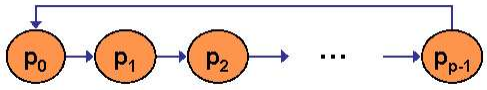

Figura 3.1: Topología del sistema: todos los procesadores están unidos mediante una configuración de anillo unidireccional.

Consideramos que el código ejecutado en cada etapa del pipeline puede representarse mediante el código genérico de la figura 3.2: M iteraciones de un bucle, que consiste en recibir datos de la etapa anterior, operar con esos datos y enviar los valores calculados a la siguiente etapa. Hemos elegido este pseudocódigo porque representa un amplio rango de cómputos *pipelines*, por ejemplo, muchos algoritmos paralelos de programación dinámica [84, 126, 129, 130]. Es necesario indicar que la primera etapa no recibe datos de etapas previas, ni la última realiza ningún envío.

```
1 j = 0;
2 while (j < M) {
3    recibir ();              // Excepto la primera etapa
4    computar ();
5                             // Se envía un elemento de tamaño w
6    enviar (elemento_w);     // Excepto la última etapa
7    j++;
8 }
```

Figura 3.2: Código estándar de un algoritmo *pipeline*: M iteraciones de un bucle que consiste en recibir datos de la etapa anterior, operar con esos datos y enviar los valores calculados a la siguiente etapa.

Para ilustrar el funcionamiento de un *pipeline* utilizamos diagramas de tiempo; es decir, un diagrama donde se especifica, de forma visual, el tiempo de cómputo de cada procesador, así como las comunicaciones realizadas. En la figura 3.3 se muestran diagramas de tiempo típicos obtenidos en la ejecución de un *pipeline* con número de procesadores igual al número de etapas, en entornos homogéneos y heterogéneos. La figura 3.3-Izquierda representa un diagrama de tiempo para un *pipeline* homogéneo con 3 procesadores. Cada procesador es representado por una fila en el diagrama. Para cada uno de ellos, dibujamos con un rectángulo el tiempo t que una máquina invierte en procesar una iteración del algoritmo; y por líneas discontinuas, el tiempo de comunicación entre los procesadores $(\beta + \tau * w)$ donde β y τ son la latencia y ancho de banda respectivamente. En las secciones 1.2 y 1.4 especificamos la forma de modelar el coste de una comunicación mediante estos dos parámetros y en la sección 1.5.1 calculamos los valores de β_{ij} y τ_{ij} para cada par de procesadores disponibles en nuestro *cluster*. En los diagramas puede observarse también las diferencias en el trabajo realizado por la primera etapa (sin recepción de datos), la última etapa (sin enviar información) y las etapas intermedias.

Algunos procesadores deben esperar un período de tiempo antes de poder comenzar a realizar su cómputo, esperando por los datos que necesita. Después de esta fase inicial de arranque el *pipeline* se encuentra trabajando a pleno rendimiento; es decir, ningún procesador del *pipeline* está ocioso. Finalmente, cuando los datos de entrada terminan, el *pipeline* pasa por una fase de finalización, donde los procesadores consecutivos van terminando progresivamente con una iteración de cómputo de diferencia. Este patrón regular aparece en el caso homogéneo y ha sido extensamente estudiado y modelado [11, 12, 85, 133]; en muchos casos, se proporcionan los parámetros adecuados para conseguir ejecuciones óptimas.

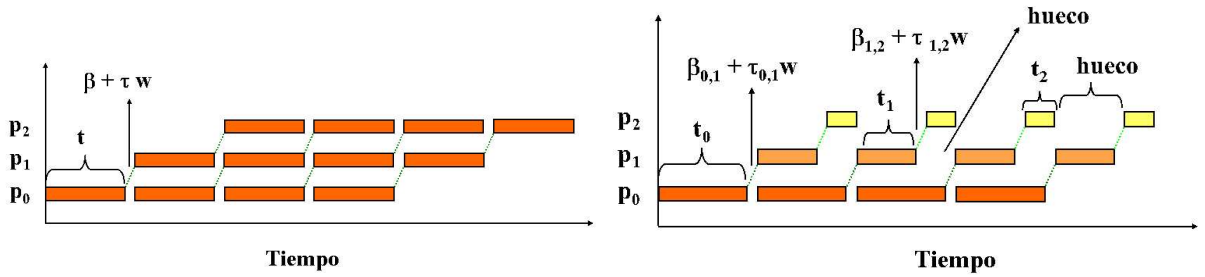

Figura 3.3: Diagramas de tiempo: *Pipeline* homogéneo vs. heterogéneo. El número de procesadores es igual al número de etapas. Izquierda: *Pipeline* homogéneo: El tiempo de cómputo (t) y el tiempo de comunicación ($\beta + \tau * w$) es el mismo para todos los procesadores. Derecha: *Pipeline* heterogéneo: Los procesadores tienen diferentes capacidades e invierten diferentes tiempos para computar su tarea (t_i) y en las comunicaciones ($\beta_{i,i+1} + \tau_{i,i+1} * w$), generando huecos en el diagrama de tiempo.

Aunque en los últimos años se han realizado numerosos trabajos sobre el comportamiento de *pipelines* en sistemas heterogéneos [33, 54, 56, 60, 112, 150, 170, 184], su estudio aún sigue siendo un problema abierto. En la figura 3.3-Derecha mostramos también el diagrama de tiempo para un *pipeline* heterogéneo con 3 procesadores ($p = 3$), cada uno de ellos con una potencia computacional diferente: el procesador p_0 es el procesador más lento y p_2 el procesador más rápido. Debido a la heterogeneidad, no sólo los tiempos de cómputo son distintos, sino también los tiempos invertidos en las comunicaciones entre los procesadores. Los rectángulos representan ahora el tiempo t_i que el procesador p_i necesita para computar una iteración, y las líneas discontinuas el tiempo de comunicación entre los procesadores: $\beta_{i,i+1} + \tau_{i,i+1} * w$ entre los procesadores p_i y p_{i+1}. Debido al hecho de que cada procesador se comunica siempre con el siguiente procesador en el anillo, en el resto de este capítulo vamos a utilizar, por simplicidad, la notación β_i, τ_i para representar los valores $\beta_{i,i+1}$ y $\tau_{i,i+1}$ respectivamente. En el caso homogéneo, como ya ha sido comentado, una vez que el procesador comienza a trabajar es alimentado con datos continuamente. Sin embargo, en el caso heterogéneo, pueden aparecer situaciones asimétricas como consecuencia del efecto producido por el procesador más lento. Los procesadores más rápidos pasan períodos de tiempos (que denominamos **huecos**) sin poder trabajar, esperando recibir la información que necesitan de los procesadores anteriores en el *pipeline*. Por todo lo visto anteriormente se deduce que para obtener una implementación eficiente, los procesadores deben comenzar a trabajar tan pronto como sea posible y ser alimentados con datos cuando sea necesario; el cambio de contexto entre etapas es realizado cada vez que comunican un valor.

Cuando el número de etapas de un *pipeline* es superior al número de procesadores disponibles para ejecutarlas, tenemos que seleccionar el tipo de distribución a realizar: podemos asignar una única etapa por procesador (**distribución cíclica pura**); o bien, obligar a cada procesador a computar varias etapas (**distribución cíclica por bloques**). En este último caso, las etapas realizadas en un mismo procesador se denominan **procesos virtuales**. Si optamos por la última opción, una de las decisiones más importantes a tomar para resolver un problema es la cantidad de trabajo asignada a cada uno de los procesadores; es decir, el número de procesos virtuales que se ejecutan en cada procesador.

El número de procesos virtuales asignado a los procesadores se denomina **grano** (G). En el caso de entornos homogéneos es habitual asignar a todos los procesadores el mismo valor de grano. Sin embargo, en sistemas heterogéneos suele ser interesante asignar a cada tipo de máquina un valor de grano diferente (G_i) en función de su capacidad de cómputo. Denominamos **banda** al conjunto de procesos virtuales que pueden ejecutarse simultáneamente sobre los p procesadores; es decir, una banda está compuesta por $\sum_{i=0}^{p-1} G_i$. Podemos calcular el número de bandas que se producen al realizar el *pipeline* como: $\frac{n}{\sum_{i=0}^{p-1} G_i}$.

Otro factor a tener en cuenta es la forma en la que los datos son comunicados entre los procesadores. En lugar de realizar un envío para cada elemento, el procesador almacena los datos en un ***buffer*** antes

de ser enviados. Cuando el *buffer* se llena, los datos se transmiten como si fueran un único paquete. El uso del *buffer* reduce el número de mensajes necesarios para resolver un problema y, por lo tanto, disminuye también la sobrecarga en las comunicaciones. Sin embargo, al usar el *buffer* se produce un retraso entre los procesadores, ya que un dato calculado por un procesador, no va a poder ser utilizado por el procesador de su derecha, hasta que no haya calculado suficientes datos para llenar el *buffer* y enviarlo. El *buffer* utilizado en todos los procesadores de un sistema homogéneo suele tener el mismo tamaño (B). En sistemas heterogéneos, puede decidirse entre utilizar el mismo tamaño de *buffer* para los distintos procesadores o emplear tamaños diferentes (B_i).

Podemos concluir que existen parámetros fundamentales que debemos calcular para conseguir una ejecución óptima de un *pipeline*: los valores óptimos de G_i y B_i. Obtener estos valores es nuestro objetivo en este capítulo.

Pretendemos buscar un modelo analítico que sea capaz de encontrar la solución óptima para un algoritmo *pipeline* con muchos parámetros. Además, hay que tener en cuenta, que escribir un algoritmo *pipeline* parametrizado requiere un esfuerzo considerable. Por este motivo, pretendemos simplificar este trabajo, desarrollando una herramienta que permita escribir de forma sencilla este tipo de algoritmos.

3.4. *llp*: Una herramienta para algoritmos *pipeline* en entornos homogéneos

La Laguna Pipeline, *llp*, es una herramienta para resolver problemas mediante el paradigma de programación *pipeline*. *llp* está concebida como una librería basada en macros y su portabilidad está garantizada debido a su implementación empleando librerías de paso de mensajes estándar (PVM [63, 78] y MPI [89, 134]).

La herramienta *llp* fue presentada en [85, 129] y permite *mapping* cíclico puro y cíclico por bloques en un algoritmo *pipeline* de acuerdo a las especificaciones del usuario. En [82, 83, 131] se completó el desarrollo de esta herramienta para el caso homogéneo, demostrando la eficiencia de los resultados para diferentes tipos de problemas, añadiéndole la funcionalidad del *buffer* y proporcionándole un mecanismo que permite generar automáticamente el *mapping* óptimo [84, 133, 132].

llp se ha desarrollado siguiendo un diseño estructurado en capas o niveles (figura 3.4); donde cada nivel proporciona una serie de servicios al nivel superior, ocultando los detalles de implementación. El diseño facilita la portabilidad de la herramienta a distintos entornos, así como la inserción de nuevas funcionalidades en la misma. Aunque la versión actual trabaja utilizando las librerías *PVM* y *MPI*, las modificaciones necesarias para incorporar nuevas librerías son mínimas.

El nivel inferior (figura 3.4) lo constituye la **arquitectura** de la máquina sobre la que se están ejecutando los programas; *llp* está preparada para ejecutar sobre cualquier tipo de arquitectura.

El siguiente nivel es el **nivel de comunicaciones entre procesadores** (figura 3.4). Las funciones disponibles para utilizar en niveles superiores incluyen las comunicaciones entre los procesadores físicos y la gestión del *buffer* de comunicaciones. El número de procesadores físicos (p), con los que va a trabajar la herramienta, es definido por el usuario y se almacena en la variable *num_proc*. Se considera que todos los procesadores están unidos formando un anillo unidireccional (figura 3.1). Si se trabaja con *num_proc* procesadores se asigna a cada procesador un nombre que se guarda en la variable *ll_fname*, (desde 0 a *num_proc* - 1). De forma que cada procesador envía siempre los mensajes al procesador que tiene a su derecha en el anillo y los recibe del procesador que tiene a su izquierda. Para un procesador cualquiera p_i, los procesadores con los que se relaciona son p_{right} ($right = (p_i + 1)\ mod\ num\_proc$), el procesador al que envía los mensajes y p_{left} el procesador del que recibe los datos, donde

$$left = \begin{cases} num\_proc - 1 & \text{si } i = 0 \\ i - 1 & \text{en otro caso} \end{cases}$$

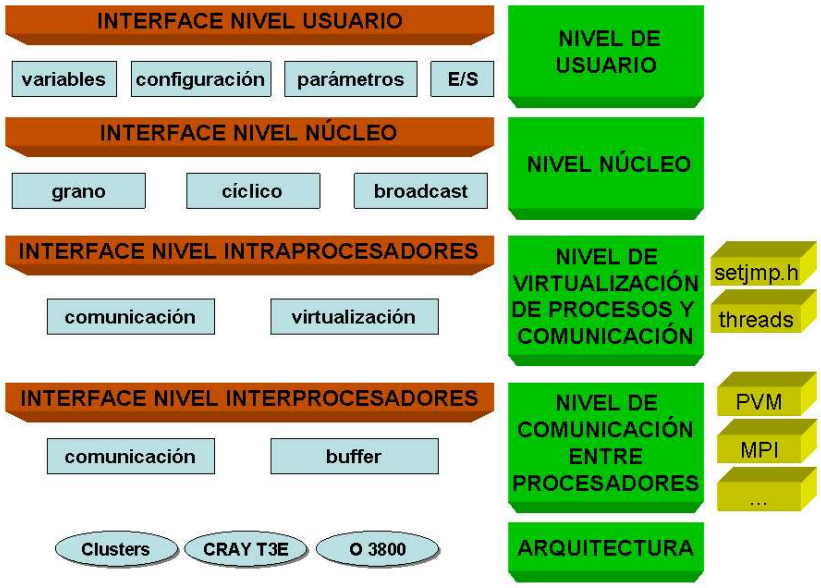

Figura 3.4: Estructura en niveles de la herramienta *llp*. Cada nivel proporciona una serie de servicios al nivel superior, ocultando los detalles de implementación.

El procesador p_i no se comunica con ninguno de los restantes procesadores del anillo. La herramienta permite decidir también el uso o no del *buffer* de datos y, en el caso de usarlo, establecer su tamaño.

Para realizar las comunicaciones se van a utilizar las funciones proporcionadas por dos librerías de paso de mensajes estándar: *MPI* y *PVM*. Unicamente es necesario introducir en el programa una macro indicando cuál de las dos librerías se quiere utilizar (*#define MPI* o *#define PVM*).

El siguiente nivel de *llp* (figura 3.4) se denomina **nivel de virtualización de procesos**. Un problema se divide en tareas independientes y cada una de estas tareas va a ser ejecutada por un procesador diferente. A estos procesos se les llama **procesos virtuales** y se les asigna un identificador (*NAME*), comenzando por el valor 0. Si un problema se divide en *STAGES* etapas virtuales, los valores que se van a asignar a *NAME* están entre 0 y *STAGES* - 1.

Inicialmente, a cada procesador físico se le asigna el trabajo de la primera banda y comienza a ejecutarse el proceso virtual con mayor nombre local (*ll_vname = grano - 1*). Cuando este proceso necesite un dato del proceso anterior (*ll_vname = grano - 2*) le transfiere el control del programa mediante la macro *IN* de la herramienta. Esto lo hacen sucesivamente todos los procesos hasta llegar al primer proceso virtual de la banda. Este proceso, cuando requiera un dato, lo tiene que recibir del procesador físico anterior, es decir, realiza una comunicación entre procesadores (correspondiente al nivel inferior). Cuando el proceso con *ll_vname* = 0 recibe el dato que necesita y calcula el que ha de pasarle al siguiente proceso, le devuelve el control de la ejecución del programa (macro *OUT* de la herramienta). Esto lo realizan todos los procesos hasta volver a llegar al proceso con *ll_vname = grano - 1* que, al calcular su dato, lo envía al procesador físico siguiente. Esto puede producir un retardo ya que un procesador debe esperar a que el anterior realice sus cálculos para poder empezar a trabajar. Las transferencias de control internas al procesador se realizan con funciones de una librería estándar del lenguaje *C* (*setjmp.h*). Esta librería permite realizar saltos condicionales entre etiquetas variables y permite diferenciar entre comunicaciones internas y externas a los procesadores. Debido a que el cambio de contexto está implementado de forma muy eficiente, la sobrecarga introducida por la herramienta es mínima.

El siguiente nivel es el **nivel núcleo** (figura 3.4). Se puede trabajar de dos formas distintas: grano y cíclico. Utilizando la estrategia grano, se trabaja con p procesadores, *STAGES* procesos virtuales y

grano *ll_grain* (figura 3.34). Para trabajar con esta forma de ejecución solamente es necesario declarar la siguiente constante en el programa *#define GRAIN*.

Con la ejecución cíclica no existe el nivel de virtualización de procesos, ya que para este caso sólo existe un proceso virtual en cada instante en un procesador físico (G = 1) (figura 3.31). Para ejecutar utilizando esta técnica es necesario definir la siguiente macro en el programa principal: *#define CICLICO*.

Para estas dos estrategias, la herramienta proporciona todas las macros y funciones necesarias para la ejecución del *pipeline* y las comunicaciones entre los procesos.

Independientemente del método utilizado para resolver el problema (cíclico o con grano), la herramienta proporciona la posibilidad de enviar un mensaje a todos los procesadores del sistema (*broadcast*). Para poder realizarlo, es necesario que todos los procesadores tengan compartidas las variables que van a utilizar.

El nivel más alto de la herramienta *llp* es el **nivel de usuario** (figura 3.4). Para utilizar *llp* no es necesario que el usuario tenga conocimiento de las técnicas de programación paralela, ya que sólo tiene que escribir un algoritmo para un *pipeline* del número de procesadores que él defina y utilizar las macros *IN* y *OUT* para la comunicación entre los procesos.

Este nivel nos proporciona las funciones para inicializar el *pipeline* (*INITPIPE*), finalizar la ejecución del anillo y liberar la memoria asignada (*EXITPIPE*). Estas dos funciones deben ser llamadas por los usuarios en sus programas, pero realizan todo el proceso de forma transparente, sin que el usuario tenga que preocuparse por las comunicaciones. En algunos casos puede interesar invertir el sentido del *pipeline*, la herramienta también proporciona una función (*REVERSE_PIPE*), que permite modificar los nombres físicos de los procesadores e intercambiar los nombres de sus vecinos derecho e izquierdo, para poder seguir estableciendo las comunicaciones.

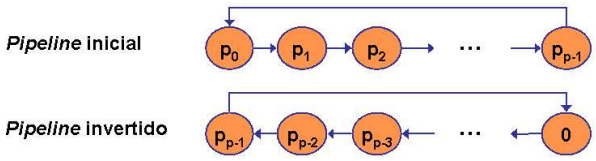

Figura 3.5: Efecto de la macro *REVERSE_PIPE*: se invierte el sentido del *pipeline*.

La herramienta está compuesta por varios ficheros, pero únicamente es necesario incluir en el programa el fichero *ll_pipe.h* e indicar si se va a ejecutar el *mapping* cíclico puro o cíclico por bloques y si se desea emplear la versión *PVM* o la versión *MPI*.

La figuras 3.6 y 3.7 ilustran la facilidad de uso al aplicar un esquema *pipeline* general. En la figura 3.6 se muestra el código general de un programa que utilice esta herramienta. En la parte superior (líneas 1 a 3) se especifican las características del *pipeline*: en este caso, un *pipeline* cíclico por bloques utilizando *MPI* para las comunicaciones; y se incluye la librería. En la función *main*, el programador siempre debe llamar a las macros *INITPIPE* y *EXITPIPE* comentadas anteriormente (líneas 6 y 24 respectivamente). La macro *PIPE* (línea 19) es la encargada de ejecutar el *pipeline*. Recibe como argumento la rutina que se ejecuta en cada proceso (*solver_1()*), el tamaño del *pipeline* (N) y el tamaño del bloque (G) en la política cíclica por bloques. La macro *PIPE* introduce la rutina en un bucle, simulando los procesos virtuales asignados al procesador actual de acuerdo con la política de *mapping*.

La macro *SET_BUFIO* (líneas 18 y 22) nos permite establecer, antes de ejecutar el *pipeline*, el número de elementos que se pueden almacenar en el *buffer*. Como aparece en la figura 3.6, varios *pipelines* pueden ser ejecutados en un mismo programa, con diferentes funciones y distintos valores de grano y *buffer*.

La figura 3.7 muestra el código de la función *solver_1()*, que implementa el *pipeline* virtual que debe ser ejecutado. El usuario debe especificar el código para el primer proceso, para el último proceso y para todos

```
 1 #define GRAIN
 2 #define MPI
 3 #include "ll_pipe.h"
 4         ....
 5 int main (PARAMETERS) {
 6     INITPIPE;
 7     Tipo variable[MAX];
 8
 9     if (ARGC != 7)
10         abort();
11     N = atoi(ARGV(1));                      // Lectura de parámetros
12     M = atoi(ARGV(2));
13     G1 = atoi(ARGV(3));
14     G2 = atoi(ARGV(4));
15     B1 = atoi(ARGV(5));
16     B2 = atoi(ARGV(6));
17                              // Primer pipeline
18     SET_BUFIO(used, B1, sizeof(int));     // Especificar el número de elementos en el buffer
19     PIPE(solver_1(N), N, G1);             // Resolución del pipeline
20
21                              // Segundo pipeline
22     SET_BUFIO(used, B2, sizeof(int));     // Especificar el número de elementos en el buffer
23     PIPE(solver_2(M), M, G2);             // Resolución del pipeline
24     EXITPIPE;
25 }
```

Figura 3.6: Función main(): ejecuta 2 *pipelines*, expandiendo la macro *PIPE* a N y M procesadores virtuales respectivamente.

los procesos intermedios de forma independiente. Debe declarar todas sus variables locales entre las macros *LOCAL_VAR* y *BEGIN*, escribir todo el código y finalizar la función con una macro denominada *END*. La variable *NAME* identifica el proceso virtual en el *pipeline* para ejecutar el código correspondiente. Se usan las macros *IN* y *OUT* para establecer las comunicaciones con los vecinos izquierdo y derecho. *llp* proporciona también rutinas de entrada y salida paralelas, como *GPRINTF* utilizada en la línea 11 de la figura 3.7.

En [85] se demuestra que el rendimiento obtenido con *llp* en un sistema homogéneo es similar al obtenido por un programador experimentado usando una librería estándar de paso de mensajes.

3.5. *llpW*: Una herramienta para algoritmos *pipeline* en entornos heterogéneos

Debido a la portabilidad de las librerías de paso de mensajes, el diseño actual de la herramienta *llp* permite la ejecución de *pipelines* sobre sistemas heterogéneos como cualquier otro programa *PVM* o *MPI*. Sin embargo, al asignar la misma cantidad de trabajo a todas las máquinas sin tener en cuenta sus velocidades, se estaría infrautilizando la capacidad del sistema. Hemos adaptado nuestra herramienta a la heterogeneidad del sistema, de forma que se asigne el trabajo a las máquinas en función de su capacidad computacional. En lugar de realizar un *mapping* cíclico por bloques, con bloques de tamaño G, ahora estudiaremos un *mapping* cíclico por bloques vectorial, con un valor distinto de grano para cada tipo de máquina (G_i). Introducimos dos nuevas macros: *INITPIPE_W* y *PIPE_W* (figura 3.8). Hemos

```
 1 void solver_1 (int N) {
 2     LOCAL_VAR
 3        int result;
 4     BEGIN;
 5
 6     if (NAME == 0)
 7          // Código para la primera etapa
 8     else
 9         if (NAME == (N - 1)) {
10             result = ... // Código para la última etapa
11             GPRINTF("\n%d: Resultado =% d",NAME,result);
12         }
13         else
14               // Código para una etapa intermedia del pipeline
15     END;
16 }
```

Figura 3.7: Función solver_1(): código que ejecutan los procesos virtuales. Se escriben códigos independientes para el primer y el último proceso y para todos los procesos intermedios.

añadido la terminación W por establecer un peso (*Weight*) a cada máquina al realizar la distribución. Estas macros reemplazan las macros análogas en el caso homogéneo. La macro *INITPIPE_W* configura el *pipeline* heterogéneo y la macro *PIPE_W* gestiona la ejecución del *pipeline*. Utilizamos un *array* denominado *arrayG[]* para almacenar la granularidad de los diferentes procesadores que necesitamos en el caso heterogéneo.

```
 1 #define PIPE_W(f, ST, arrayG) {
 2    int aux;
 3    unsigned ll_bands;
 4    STAGES = (ST);
 5           // El array llp_TypeMachines[] contiene los tipos de todas las máquinas
 6           // Se almacena en ll_grain el grano que le corresponde al procesador
 7    ll_grain = arrayG[llp_TypeMachines[ll_fname]];
 8           // El procesador ll_fname calcula el nombre (NAME) de su primer proceso virtual
 9           // Se suman los granos de los procesadores desde 0 hasta ll_fname
10    NAME = sumatorio(0, ll_pname, arrayG[llp_TypeMachines[i]] - 1);
11           // Se calcula el número total de procesos virtuales en una banda como
12           // el sumatorio de los granos asignados a todos los procesadores
13    total_grain = sumatorio(0, numproc -1, arrayG[llp_TypeMachines[i]]);
14           // Se calcula el número de bandas del pipeline heterogéneo
15    ll_bands = num_bands_W(STAGES, arrayG, llp_TypeMachines);
16           // Se realiza toda la ejecución del pipeline
17    for (ll_pipe_i = 0; ll_pipe_i <= ll_bands; ll_pipe_i++) {
18        if (NAME <= (STAGES - 1))  f;
19            NAME =  NAME +  total_grain ;
20    }
21 }
```

Figura 3.8: Macro *PIPE_W*: Macro que realiza la ejecución del *pipeline* heterogéneo, considerando las diferencias entre las máquinas del sistema.

Para realizar la asignación correcta de los valores de granos a las máquinas, el sistema *llpW* necesita información sobre las capacidades de los procesadores del *cluster*. El fichero *llpcluster.conf* almacena el conjunto completo de máquinas que configuran el *cluster*. Cada línea de este fichero contiene una cadena con el nombre de la máquina y un entero que identifica su tipo. La figura 3.9 muestra el fichero de configuración *llpcluster.conf* para nuestro *cluster* (sección 1.5.1), donde el tipo es igual a 0, 1 y 2 para los procesadores rápidos, intermedios y lentos respectivamente.

```
beowulf1.csi.ull.es 0
beowulf2.csi.ull.es 0
beowulf3.csi.ull.es 0
beowulf4.csi.ull.es 0
beowulf5.csi.ull.es 1
beowulf6.csi.ull.es 1
beowulf7.csi.ull.es 1
beowulf8.csi.ull.es 1
beowulf9.csi.ull.es 2
beowulf10.csi.ull.es 2
beowulf11.csi.ull.es 2
beowulf12.csi.ull.es 2
beowulf13.csi.ull.es 2
beowulf14.csi.ull.es 2
```

Figura 3.9: Fichero de configuración *llpcluster.conf* para nuestro *cluster* con 3 tipos de máquinas.

La macro *INITPIPE_W* inicializa el vector *llp_TypeMachines[]* para cada uno de los procesadores involucrados en la ejecución actual, de acuerdo a la especificación del fichero *llpcluster.conf*. El elemento *llp_TypeMachines[i]* almacena el tipo correspondiente al procesador p_i. La macro *PIPE_W(solver( ), N, arrayG)* ejecuta un *pipeline* sobre un entorno heterogéneo. Al proceso virtual i le corresponde una granularidad de $G_i = arrayG[llp\_TypeMachines[i]]$ en cada etapa. Debido al mecanismo de cambio de contexto usado para la virtualización en el caso homogéneo, el rendimiento de la herramienta no se ve afectado por la introducción de estas dos nuevas macros.

3.6. Comparativa *llp* vs *llpW*

La primera prueba que hemos realizado para validar nuestra herramienta y demostrar su eficiencia en entornos heterogéneos, ha consistido en llevar a cabo una comparativa entre los resultados obtenidos para una serie de problemas al utilizar una herramienta diseñada para entornos homogéneos (*llp*), frente a la herramienta adaptada a las plataformas heterogéneos (*llpW*). Hemos empleado la versión *MPI* de la librería en ambos casos.

Para realizar las pruebas, hemos utilizado el *cluster* compuesto únicamente por los dos tipos de máquinas con un nivel de heterogeneidad más alto: las 4 máquinas rápidas y 4 máquinas lentas (sección 1.5.1). En esta sección, no pretendemos encontrar la distribución óptima para minimizar el tiempo total de ejecución paralelo, simplemente deseamos comparar los resultados con las dos herramientas; por lo tanto, vamos a realizar, en cada caso, una ejecución por bloques con diferente número de procesadores (p = 2, 4, 5, 6 y 8) y distintos tamaños de *buffer*. En el caso de la herramienta homogénea se asigna a todos los procesadores la misma cantidad de trabajo (el mismo valor de grano). Para la herramienta heterogénea hemos considerado las máquinas rápidas aproximadamente 3 veces más rápidas que las máquinas lentas, y asignamos un valor de grano $G_R \approx 3 * G_L$. De nuevo queremos indicar, que este valor no es la proporción exacta entre los dos tipos de máquinas, se trata de un valor aproximado elegido para realizar estos experimentos.

En todos los casos, los problemas a resolver que hemos seleccionado son algoritmos de programación dinámica [103]: el problema de la mochila unidimensional, el problema de caminos mínimos, el problema de asignación de recursos y el problema de la búsqueda de la subsecuencia más larga. Como ya comentamos en el capítulo 2, la técnica de programación dinámica es una importante técnica de resolución de problemas que se aplica a problemas de optimización combinatoria. Se emplea cuando la solución a un problema se obtiene como resultado de una secuencia de decisiones: se comprueban todas las secuencias de decisiones y se selecciona la mejor. Con la programación dinámica se consigue evitar calcular la misma solución más de una vez: la solución de un subproblema se almacena en una estructura de datos para usarla posteriormente. Se comienza con los subproblemas más pequeños y se combinan sus soluciones para resolver problemas cada vez mayores, hasta llegar al problema completo. Normalmente, la paralelización de la programación dinámica se realiza mediante algoritmos *pipeline*.

A continuación, describimos los problemas y presentamos los resultados obtenidos para cada uno de ellos con las dos herramientas. Los códigos utilizados para las dos herramientas (homogénea y heterogénea), la única diferencia radica en la modificación de los parámetros de la ejecución.

3.6.1. Problema de la mochila unidimensional

Dado un contenedor o mochila de capacidad M, se trata de introducir en él N objetos, cada uno con un peso w_n y un beneficio obtenido al incluirlo en la mochila p_n. El problema consiste en introducir todos los objetos posibles en la mochila, hasta que no quepa ningún objeto más, maximizando el beneficio obtenido. Matemáticamente se trata de maximizar el beneficio máximo que se puede obtener con esos objetos y para esa capacidad ($f_n^*(C)$):

$f_n(C) = \text{mín}\{f_{n-1} * (C), p_n + f_{n-1} * (C - w_n)\},$ si $C \geq w_n$,

$f_n(C) = f_{n-1} * (C),$ si $C < w_n$,

$f_0(C) = 0,$ para todo C,

Vamos a trabajar con dos instancias del problema (un problema pequeño y otro de mayor tamaño):

1. Disponemos de 5000 objetos ($N = 5000$) y una capacidad de la mochila de $M = 20000$.
2. Disponemos de 12000 objetos ($N = 12000$) y una capacidad de la mochila de $M = 30000$.

En ambos casos, los pesos y los beneficios de los objetos disponibles para ser introducidos en la mochila oscilan entre 1 y 100, no existe ningún elemento con un peso o un beneficio igual a 0.

Algoritmo secuencial

La figura 3.10 presenta el código ejecutado para resolver el problema de forma secuencial. Se inicializa el vector *fanterior* a cero (líneas 8 y 9) y, a continuación, se comienza a modificar el vector de la etapa actual. Para cada iteración se comprueba si el objeto debe incluirse o no en la mochila. Una vez actualizada la fila correspondiente a la etapa actual, se intercambian los valores de los vectores *fanterior* y *factual* para poder usarlos correctamente en la etapa siguiente (líneas 24 a 26).

Algoritmo paralelo

En la paralelización de este algoritmo se necesitan tantos procesos como número de objetos disponibles. Se asigna a cada proceso un objeto (*name*), con peso $w[name]$ y beneficio $p[name]$. A cada uno se le asocia una fila de la matriz de salida. En la figura 3.11 se encuentra la función que resuelve el problema de forma paralela. Este código es el mismo para *llp* y *llpW*.

```
1 void mochila01_sec (void)  {
2    unsigned v1, v2;
3    int c, k;
4    unsigned *factual, *fanterior, *faux;
5
6     // Asignación de memoria a 'factual' y 'fanterior'
7
8    for (c = 0; c <= C; c++)
9        fanterior[c] = 0;
10
11   for (k = 1; k <= N; k++) {
12        for (c = 0; c <= C; c++) {
13             if (c < w[k-1])                // El elemento no cabe en la mochila
14                  factual[c] = fanterior[c];
15              else {                        // El elemento se introduce en la mochila
16                  v1 = fanterior[c - w[k-1]] + p[k-1];
17                  v2 = fanterior[c];
18                  if (v1 > v2)
19                       factual[c] = v1;
20                  else
21                       factual[c] = v2;
22             }
23        }
24        faux = fanterior;                    // Se intercambian los valores de los vectores
25        fanterior = factual;
26        factual = faux;
27   }
28   printf("\tGanancia total: %u", fanterior[C]);
29 }
```

Figura 3.10: Función que resuelve el problema de la mochila unidimensional de forma secuencial.

En la figura 3.11 sólo se muestra el código de las etapas intermedias. La primera y última etapa se implementan de forma similar: en la primera etapa se elimina la recepción de datos (llamada a la macro IN) y en la última se elimina el envío de información (macro OUT). En las etapas intermedias se recibe un elemento del proceso anterior, se actualiza la matriz f y se envía al proceso siguiente. Después se comprueba si el objeto asociado (objeto *name*, donde *name* es el nombre del proceso) cabe en la mochila y, si es así, se modifica f. Vuelve a repetirlo hasta que haya actualizado toda la fila de la matriz.

Resultados computacionales

Hemos resuelto inicialmente el problema de la mochila unidimensional con $N = 50000$ objetos y capacidad de la mochila $M = 20000$. En la tabla 3.1 presentamos los resultados para el subsistema homogéneo de las máquinas rápidas. Los resultados que aparecen en la tabla se han obtenido con *llp*. Estas ejecuciones pueden realizarse también con *llpW*, pero los resultados son tan parecidos que no los hemos incluido. En la tabla aparece el número de procesadores utilizados (columna p), la configuración de los procesadores (columna *Configuración*), el valor del *buffer*, el valor del grano y el tiempo total de ejecución en segundos. En la columna *Configuración* de las tablas de esta sección, representamos por R a un procesador rápido y por L a un procesador lento y escribimos una letra por cada procesador utilizado en la ejecución.

En la tabla 3.1 se observa que el tiempo de ejecución se reduce cuando se ejecuta con 4 procesadores frente al tiempo de ejecución con 2 procesadores. En la tabla 3.2 se introducen los procesadores lentos en el sistema anterior y comparamos los resultados obtenidos con las dos herramientas. La tabla presenta

```
 1 void mochila01_par(void) {
 2    LOCAL_VAR
 3       int h, i;
 4       int result;
 5       unsigned *f;
 6    BEGIN;
 7
 8          . . .
 9
10                  // Procesos intermedios
11
12        for (i = 0; i <= C; i++)              // Inicialización
13             f[i] = 0;
14
15        for (i = 0; i <= C; i++) {
16             IN(&h);                          // Recepción del proceso anterior
17             f[i] = max(f[i], h);
18             if (C >= i + w[name])
19                 f[(i + w[name])] = h + p[name];
20             OUT(&f[i], 1, sizeof(unsigned));  // Envío al siguiente proceso
21         }
22
23          . . .
24
25    END;           // Sincronización entre procesos
26 }
```

Figura 3.11: Función que resuelve el problema de la mochila unidimensional de forma paralela. Los códigos de la primera y la última etapa son muy similares al de las etapas intermedias y no se incluyen en la figura.

Tabla 3.1: Tiempos de ejecución obtenidos al resolver un problema de la mochila unidimensional con $N = 5000$ objetos y una capacidad de la mochila de $M = 20000$ sobre el subsistema homogéneo de las máquinas rápidas. El problema se resuelve utilizando *llp* con una distribución por bloques, diferente número de procesadores y con 3 tamaños de *buffer* distintos.

Problema: N = 5000 - M = 20000				
p	Configuración	*Buffer*	Grano	Tiempo Real
2	R R	10000	2500	79.39
2	R R	5000	2500	70.37
2	R R	2500	2500	65.84
4	R R R R	10000	1250	51.45
4	R R R R	5000	1250	39.89
4	R R R R	2500	1250	32.77

un formato muy similar al de la tabla anterior, añadiendo los datos correspondientes a *llpW* (grano de los procesadores rápidos, grano de los procesadores lentos y tiempo real de ejecución) y el porcentaje de mejora obtenido al ejecutar el problema con *llpW*. Como se especificó en la sección 3.6, los valores de grano se han calculado para una distribución por bloques, utilizando la proporción $G_R \approx 3 * G_L$.

En la tabla 3.2 se observa que existe una gran diferencia entre los resultados de *llp* y *llpW*. En *llp*, al introducir los procesadores lentos, se produce un retraso en la resolución del problema, obteniéndose un mayor tiempo de ejecución. Si se comparan los resultados para $p = 5$, 6 y 8 (las combinaciones con

Tabla 3.2: Tiempos de ejecución obtenidos al resolver un problema de la mochila unidimensional con $N = 5000$ objetos y una capacidad de la mochila de $M = 20000$. El problema se resuelve utilizando una distribución por bloques, empleando diferente número de procesadores y con 3 tamaños de *buffer* distintos.

Problema: N = 5000 - M = 20000								
			Herramienta *llp*		Herramienta *llpW*			
p	Configuración	*Buffer*	Grano	Tiempo Real	Grano_R	Grano_L	Tiempo Real	Porcentaje de mejora
5	R R R R L	10000	1000	77.31	1154	384	50.95	34.10 %
5	R R R R L	5000	1000	64.31	1154	384	38.16	40.66 %
5	R R R R L	2500	1000	58.29	1154	384	31.05	46.73 %
6	R R R R L L	10000	833	83.44	1071	358	55.21	33.83 %
6	R R R R L L	5000	833	64.19	1071	358	37.89	40.97 %
6	R R R R L L	2500	833	55.03	1071	358	29.08	47.15 %
8	R R R R L L L L	10000	625	90.32	938	312	58.01	35.78 %
8	R R R R L L L L	5000	625	61.31	938	312	38.60	37.03 %
8	R R R R L L L L	2500	625	48.39	938	312	26.18	45.91 %

procesadores lentos) vemos que para el tamaño de *buffer* mayor el tiempo empeora al incrementar el número de procesadores, mientras que para los tamaños de *buffer* más pequeños el tiempo se reduce al añadir procesadores. Sin embargo, en ningún caso se consigue mejorar los resultados con el subsistema homogéneo de las máquinas rápidas.

Sin embargo, si ejecutamos el programa utilizando *llpW*, comprobamos que los tiempos de ejecución con las combinaciones que incluyen procesadores lentos, son mejores que los tiempos de ejecución para el subsistema homogéneo de los 4 procesadores rápidos, excepto para el tamaño mayor de *buffer*, donde con 5 procesadores sí se reduce el tiempo de ejecución, pero al introducir más procesadores el tiempo vuelve a aumentar. Por ejemplo, para *buffer* = 2500, el tiempo de ejecución para $p = 8$ reduce en más de un 20 % el tiempo de ejecución para $p = 4$. Observando la columna del *Porcentaje de mejora* observamos que en las configuraciones con procesadores lentos, la mejora obtenida se encuentra entre el 33 % y el 47 %, lo que supone un factor de reducción importante. En la figura 3.12 comparamos de forma gráfica los tiempos conseguidos al resolver este problema empleando las dos herramientas: homogénea (figura 3.12-Izquierda) y heterogénea (figura 3.12-Derecha).

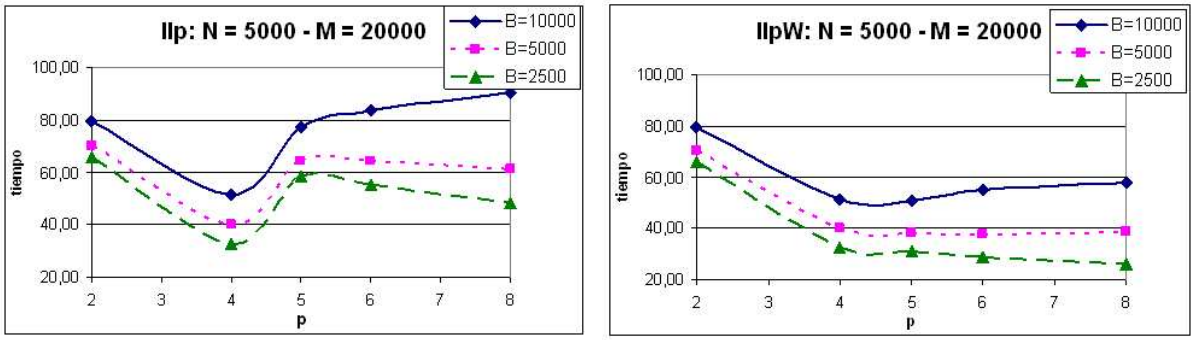

Figura 3.12: Tiempos de ejecución obtenidos al resolver el problema de la mochila unidimensional con $N = 5000$ objetos y capacidad de la mochila $M = 20000$. Izquierda: utilizando la herramienta homogénea (*llp*). Derecha: Utilizando la herramienta para los sistemas heterogéneos (*llpW*).

En la figura 3.12 se observan fácilmente las diferencias entre los dos resultados y la utilidad de la herramienta adaptada a entornos heterogéneos, frente a herramientas diseñadas para sistemas homogéneos:

mientras que en el caso de la ejecución con *llp* se produce un gran incremento en los tiempos obtenidos al añadir los procesadores lentos, al utilizar *llpW* la aceleración prácticamente se mantiene o disminuye dependiendo del tamaño del *buffer*.

A continuación resolvemos el segundo problema de la mochila unidimensional ($N = 12000$ objetos y una capacidad de la mochila de $M = 30000$). En la tabla 3.3 se presentan los resultados para el subsistema homogéneo de los 4 procesadores rápidos y en la tabla 3.4 los resultados para el sistema heterogéneo utilizando las dos herramientas (*llp* y *llpW*) con una distribución por bloques y diferentes tamaños de *buffer*.

Tabla 3.3: Tiempos de ejecución obtenidos al resolver un problema de la mochila unidimensional con $N = 12000$ objetos y una capacidad de la mochila de $M = 30000$ sobre el subsistema homogéneo de las máquinas rápidas. El problema se resuelve utilizando *llp* con distribuciones por bloques, diferente número de procesadores y con 3 tamaños de *buffer* distintos.

Problema: N = 12000 - M = 30000				
p	Configuración	*Buffer*	Grano	Tiempo Real
2	R R	15000	6000	391.54
2	R R	7500	6000	306.19
2	R R	3750	6000	289.70
4	R R R R	15000	3000	336.63
4	R R R R	7500	3000	223.68
4	R R R R	3750	3000	205.82

Tabla 3.4: Tiempos de ejecución obtenidos al resolver un problema de la mochila unidimensional con $N = 12000$ objetos y una capacidad de la mochila de $M = 30000$. El problema se resuelve con distribuciones por bloques, empleando diferentes número de procesadores y con 3 tamaños de *buffer* distintos.

Problema: N = 12000 - M = 30000								
			Herramienta *llp*		Herramienta *llpW*			
p	Configuración	*Buffer*	Grano	Tiempo Real	Grano_R	Grano_L	Tiempo Real	Porcentaje de mejora
5	R R R R L	15000	2400	803.26	2754	984	387.38	51.77 %
5	R R R R L	7500	2400	564.69	2754	984	209.55	62.89 %
5	R R R R L	3750	2400	514.33	2754	984	185.54	63.93 %
6	R R R R L L	15000	2000	572.36	2545	910	426.51	25.48 %
6	R R R R L L	7500	2000	260.04	2545	910	210.25	19.15 %
6	R R R R L L	3750	2000	212.22	2545	910	177.37	16.42 %
8	R R R R L L L L	15000	1500	492.90	2211	789	479.82	2.65 %
8	R R R R L L L L	7500	1500	251.78	2211	789	205.86	18.24 %
8	R R R R L L L L	3750	1500	187.69	2211	789	163.87	12.69 %

Observamos, en las tablas 3.3 y 3.4, que también se produce un incremento en el tiempo de ejecución cuando utilizamos *llp* e introducimos en la plataforma de ejecución los procesadores más lentos (excepto para la ejecución con $p = 8$ y *buffer* $= 3750$). Este empeoramiento de los resultados se observa especialmente cuando ejecutamos el programa con 5 procesadores.

Para la herramienta heterogénea se observa, en las tablas 3.3 y 3.4, que los tiempos de ejecución con las combinaciones que incluyen procesadores lentos reducen los tiempos de ejecución del subsistema homogéneo de los 4 procesadores rápidos, excepto para el tamaño mayor de *buffer*. En este caso, para 8 procesadores y el tamaño más pequeño de *buffer*, el tiempo de ejecución disminuye en un 20 % respecto al tiempo de ejecución para $p = 4$. En este caso, el mayor porcentaje de mejora se obtiene con 5 procesadores,

donde se alcanza el 63 %, mientras que para $p = 6$ y 8 este porcentaje es bastante menor. En la figura 3.13 comparamos los tiempos obtenidos con las dos herramientas: *llp* (figura 3.13-Izquierda) y *llpW* (figura 3.13-Derecha). Para este problema se observa también que se produce un descenso general en los tiempos de ejecución, cuando adaptamos la herramienta a los entornos heterogéneos.

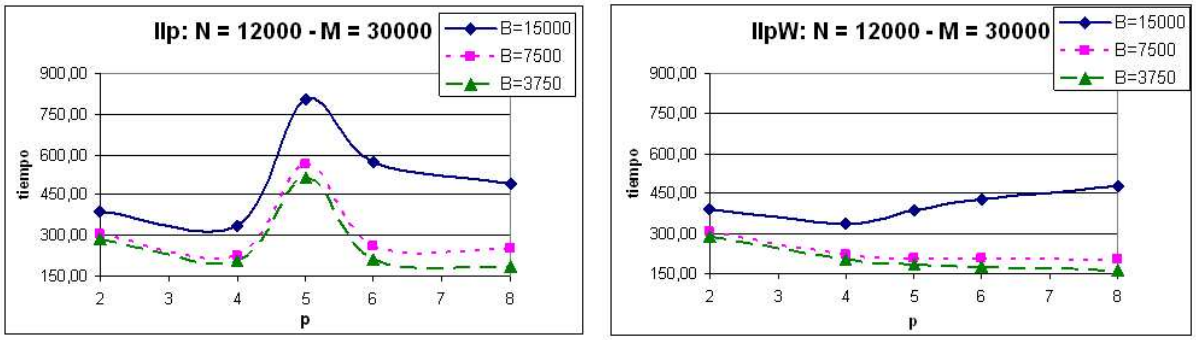

Figura 3.13: Tiempos de ejecución obtenidos al resolver el problema de la mochila unidimensional con $N = 12000$ objetos y capacidad de la mochila $M = 30000$. Izquierda: utilizando la herramienta homogénea (*llp*). Derecha: Utilizando la herramienta para los sistemas heterogéneos (*llpW*).

3.6.2. El problema de caminos mínimos

El problema consiste en encontrar el camino de mínimo coste desde un punto de origen a un conjunto de puntos destino. El problema se denomina de caminos mínimos (o *path planning*). Para resolver el problema utilizamos una matriz de costes, A, de dimensiones $N \times N$, donde el valor $A[i][j]$ con $i = 0, \ldots, N-1$ y $j = 0, \ldots, N-1$, representa el coste asociado al punto $P = (i, j)$. Los valores de los costes mínimos para cada uno de los puntos destino se almacenan en otra matriz, f, también de dimensiones $N \times N$, denominada **matriz de costes mínimos**.

Cada punto $P = (i, j)$ de la matriz puede tener hasta 8 vecinos, denotados por las iniciales de los puntos cardinales (figura 3.14).

NO	N	NE
O	P	E
SO	S	SE

Figura 3.14: Definición de los vecinos de un punto $P(i, j)$ de la matriz de caminos mínimos. Los vecinos se denotan por las iniciales de los puntos cardinales.

El coste para ir desde el punto P hasta cualquiera de sus vecinos Q viene determinado por las ecuaciones:

$$C(P, Q) = (A(P) + A(Q))/2, \text{ si } Q = N, S, E, O,$$

$$C(P, Q) = (A(P) + A(Q)) * \sqrt{2}/2, \text{ si } Q = NO, NE, SO, SE,$$

donde $A(P)$ es el coste asociado al punto P de la matriz. Si $P = (i, j)$ entonces $A(P) = A[i][j]$.

Hemos considerado dos casos particulares de este problema a resolver en este trabajo:

1. Una matriz de tamaño 3000×3000, con un obstáculo central en forma de V invertida.
2. Una matriz de tamaño 4200×4200, con un obstáculo central en forma de U.

Consideramos que el coste de todos los elementos de la matriz es 1, excepto el de aquellos elementos que formen parte de un obstáculo, que representamos mediante puntos de coste infinito (∞) en la matriz. Además, consideramos que el borde de la matriz (primera y última fila y primera y última columna) tiene un coste infinito. En la figura 3.15 se puede observar la forma en la que se almacena la información del problema para un ejemplo pequeño: una matriz de tamaño 8×8 con un obstáculo central en forma de U.

∞	∞	∞	∞	∞	∞	∞	∞
∞	1	1	1	1	1	1	∞
∞	1	∞	1	1	∞	1	∞
∞	1	∞	1	1	∞	1	∞
∞	1	∞	1	1	∞	1	∞
∞	1	∞	∞	∞	∞	1	∞
∞	1	1	1	1	1	1	∞
∞	∞	∞	∞	∞	∞	∞	∞

Figura 3.15: Ejemplo de problema de caminos mínimos, con una matriz de tamaño 8×8 con un obstáculo central en forma de U.

Algoritmo secuencial

Para resolver este problema, se realizan dos tipos de iteraciones, denominadas *red* y *blue*. Estas iteraciones se llevan a cabo de forma alternativa, hasta que no se produzca ningún cambio en la matriz durante una iteración. La figura 3.16 muestra el código de la función principal que resuelve este problema de forma secuencial.

```
 1 void Caminos_Minimos_seq(void) {
 2    cambio = 1;
 3    while (cambio) {
 4       cambio = 0;
 5       RedSweep();               // Iteración 'Red'
 6       if (cambio) {
 7          cambio = 0;
 8          BlueSweep();           // Iteración 'Blue'
 9       }
10    }
11 }
```

Figura 3.16: Función principal que resuelve el problema del camino mínimo de forma secuencial.

La diferencia entre los dos tipos de iteraciones se centra en el orden en el que se recorre la matriz. En la **iteración *red*** se opera sobre la matriz f en orden ascendente para las filas y las columnas, de forma que para cada posición P, se actualiza el valor del mejor coste si existe algún camino mejor que el actual que pase por uno de los vecinos *red* de P, que serían los vecinos O, NO, N, NE. La figura 3.17 muestra el código a ejecutar para este tipo de iteración.

```
1 void RedSweep(void) {
2     unsigned i, j;
3
4     for (i = 1; i < N-1; i++)
5        for (j = 1; j < N-1; j++)
6           Sweep(RED, i, j);
7 }
```

Figura 3.17: Código que implementa la iteración *red*.

En la figura 3.17 se observa que, en la iteración *red*, la matriz se recorre en orden ascendente de filas y columnas. La función *Sweep* se encarga de comprobar los costes debidos a los vecinos correspondientes al tipo de iteración y asignar a la posición (i, j) de la matriz de costo el valor del camino mínimo; almacenando si se produce algún cambio en la matriz. Esta función se utiliza en las dos iteraciones; el primer parámetro indica cuál de las dos iteraciones se ejecuta y decide qué vecinos están involucrados en el cálculo del coste.

El coste mínimo para un punto P se calcula mediante la ecuación:

$$f(P) = \text{mín}\{f(P), f(O) + C(O,P), f(N) + C(N,P), f(NO) + C(NO,P), f(NE) + C(NE,P)\}$$

En la **iteración *blue*** se utiliza un orden descendente en filas y columnas para recorrer la matriz. Para actualizar el valor de la matriz en un punto P se usan los vecinos *blue* de este punto (E, SE, S y SO). La figura 3.18 muestra el código de esta iteración.

```
1 void BlueSweep(void) {
2     int i, j;
3
4      for (i = N-2; i > 0; i--)
5         for (j = N-2; j > 0; j--)
6             Sweep(BLUE, i, j);
7 }
```

Figura 3.18: Código que realiza la iteración *blue*.

En la figura 3.18 se observa que el código es el mismo que para la iteración *red*, modificando únicamente el orden en que se recorre la matriz y el primer parámetro de la función *Sweep*. El coste mínimo para un punto P se calcula mediante la ecuación:

$$f(P) = \text{mín}\{f(P), f(E) + C(E,P), f(S) + C(S,P), f(SO) + C(SO,P), f(SE) + C(SE,P)\}$$

Algoritmo Paralelo

En la paralelización de este algoritmo se mantienen los dos tipos de iteraciones: *red* y *blue*. En la figura 3.19 se encuentra el código correspondiente a la función principal que resuelve el problema de forma paralela.

Para resolver este problema primero es necesario ejecutar un *pipeline* donde la función a ejecutar es una iteración *red* (línea 7 del código) y posteriormente se ejecuta otro *pipeline* donde se realiza la iteración *blue* (línea 14). Es importante indicar que para poder realizar el recorrido de la matriz en orden

```
 1 void Caminos_Minimos_par(void) {
 2     int cambio_aux;
 3
 4     cambio = 1;
 5     while (cambio) {
 6         cambio = 0;
 7         PIPE_W(RedSweep(), N, arrayGrano);
 8
 9               // Sincronización para comprobar el valor de 'cambio' en todos los procesadores
10
11        if (cambio) {
12            cambio = 0;
13            REVERSE_PIPE;
14            PIPE_W(BlueSweep(), N, arrayGrano);
15            REVERSE_PIPE;
16               // Sincronización para comprobar el valor de 'cambio' en todos los procesadores
17        }
18     }
19 }
```

Figura 3.19: Función que resuelve el problema de caminos mínimos de forma paralela.

descendente hay que realizar una inversión en el *pipeline* (líneas 13 y 15). Esto es necesario para que cada procesador pueda trabajar sobre la misma parte de la matriz en ambas iteraciones. Este proceso se realiza mediante la macro *REVERSE_PIPE* (sección 3.4). También realizamos una sincronización entre cada dos iteraciones, para comprobar si se ha producido algún cambio en la parte de la matriz correspondiente a cada procesador. Este paso es necesario puesto que puede ocurrir que sólo se produzcan cambios en las filas de la matriz asignadas a un procesador, y sea necesario una nueva iteración por parte de todos los procesadores.

Resultados computacionales

Hemos resuelto primero el problema de caminos mínimos de tamaño 3000×3000 con un obstáculo central en forma de V invertida. En la tabla 3.5 presentamos los resultados obtenidos al resolver el problema, con una distribución por bloques y diferentes tamaños de *buffer*, en el subsistema homogéneo de los procesadores rápidos y en la tabla 3.6 los resultados para el sistema heterogéneo con las dos herramientas. El esquema de presentación de estas dos tablas es el mismo que el de las tablas con los resultados del problema de la mochila unidimensional.

Tabla 3.5: Tiempos de ejecución obtenidos en el sistema homogéneo de máquinas rápidas al resolver un problema de caminos mínimos de tamaño $N = 3000$, con un obstáculo en forma de V invertida. El problema se resuelve con *llp*, una distribución por bloques, diferente número de procesadores y con 3 tamaños de *buffer* distintos.

Problema: N = 3000				
p	Configuración	*Buffer*	Grano	Tiempo Real
2	R R	1500	1500	47.89
2	R R	750	1500	45.85
2	R R	375	1500	43.89
4	R R R R	1500	750	35.73
4	R R R R	750	750	29.72
4	R R R R	375	750	25.22

Tabla 3.6: Tiempos de ejecución obtenidos al resolver un problema de caminos mínimos de tamaño $N = 3000$, con un obstáculo en forma de V invertida. El problema con una distribución por bloques, empleando diferentes número de procesadores y con 3 tamaños de *buffer* distintos.

Problema: N = 3000								
			Herramienta *llp*		Herramienta *llpW*			
p	Configuración	*Buffer*	Grano	Tiempo Real	Grano_R	Grano_L	Tiempo Real	Porcentaje de mejora
5	R R R R L	1500	600	56.42	692	232	48.77	13,55 %
5	R R R R L	750	600	50.47	692	232	43.29	14,22 %
5	R R R R L	375	600	46.07	692	232	37.49	18,63 %
6	R R R R L L	1500	500	67.43	643	214	55.43	17,80 %
6	R R R R L L	750	500	55.81	643	214	48.34	13,38 %
6	R R R R L L	375	500	48.47	643	214	39.21	19,10 %
8	R R R R L L L L	1500	375	78.70	363	187	53.09	32,54 %
8	R R R R L L L L	750	375	57.80	363	187	36.67	36,56 %
8	R R R R L L L L	375	375	48.69	363	187	32.10	34,08 %

En las tablas 3.5 y 3.6, se observa en los resultados para *llp*, que el tiempo necesario para resolver el problema aumenta al incrementar el número de procesadores lentos en el *cluster*. De forma que el peor resultado se obtiene con la configuración de 8 procesadores. Cuando ejecutamos con *llpW*, comprobamos que, para este problema, tampoco se consigue reducir el tiempo invertido por el subsistema homogéneo de los procesadores rápidos en ninguna ejecución; esto puede deberse a la sincronización de los procesadores entre cada dos iteraciones. A pesar de esta situación, si comparamos los resultados obtenidos con las dos herramientas (figura 3.20), vemos que sí se produce una reducción significativa en el tiempo de ejecución en los resultados de *llpW* sobre los tiempos de *llp*: por ejemplo, para $p = 8$ y *buffer* $= 375$, el tiempo necesario para resolver el problema empleando la librería *llp* es de 48.69 segundos, mientras que con *llpW*, este tiempo se reduce a 32.10 segundos, una reducción en el tiempo de ejecución de casi un 35 %.

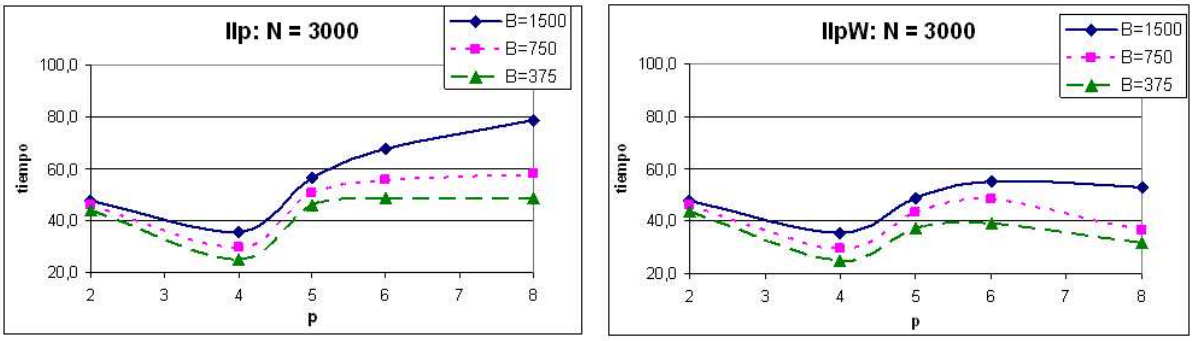

Figura 3.20: Tiempos de ejecución invertidos en resolver el problema de caminos mínimos de tamaño $N = 3000$ con un obstáculo en forma de V invertida. Izquierda: utilizando la herramienta homogénea (*llp*). Derecha: Utilizando la herramienta para los sistemas heterogéneos (*llpW*).

A continuación, resolvemos el segundo problema de caminos mínimos, donde tenemos una matriz de tamaño 4200×4200 con un obstáculo central en forma de U. Como hemos realizado en los problemas anteriores, en las tablas 3.7 y 3.8, mostramos los resultados obtenidos al resolver el problema en el sistema homogéneo y heterogéneo respectivamente.

Los resultados obtenidos para el problema de tamaño 4200×4200 (tablas 3.7 y 3.8) presentan características muy similares a las obtenidas con el tamaño de problema anterior (3000×3000), cuyos resultados

Tabla 3.7: Tiempos de ejecución obtenidos en el sistema homogéneo de los procesadores rápidos, al resolver un problema de caminos mínimos de tamaño $N = 4200$, con un obstáculo en forma de U. El problema se resuelve con *llp*, distribuciones por bloques, diferentes número de procesadores y 3 tamaños de *buffer* distintos.

Problema: N = 4200				
p	Configuración	*Buffer*	Grano	Tiempo Real
2	R R	2100	2100	75.56
2	R R	1050	2100	71.70
2	R R	525	2100	68.40
4	R R R R	2100	1050	54.20
4	R R R R	1050	1050	47.95
4	R R R R	525	1050	41.097

Tabla 3.8: Tiempos de ejecución obtenidos al resolver un problema de caminos mínimos de tamaño $N = 4200$, con un obstáculo en forma de U. El problema se resuelve con distribuciones por bloques, diferentes número de procesadores y con 3 tamaños de *buffer* distintos.

Problema: N = 4200								
			Herramienta *llp*		Herramienta *llpW*			
p	Configuración	*Buffer*	Grano	Tiempo Real	Grano_R	Grano_L	Tiempo Real	Porcentaje de mejora
2	R R	2100	2100	75.56	2100	0	74.24	1.75 %
2	R R	1050	2100	71.70	2100	0	72.48	-1.09 %
2	R R	525	2100	68.40	2100	0	68.96	-0.81 %
4	R R R R	2100	1050	54.20	1050	0	55.55	-2.49 %
4	R R R R	1050	1050	47.95	1050	0	48.76	-1.68 %
4	R R R R	525	1050	41.09	1050	0	41.80	-1.71 %
5	R R R R L	2100	840	83.30	969	324	71.59	14.06 %
5	R R R R L	1050	840	73.03	969	324	60.78	16.77 %
5	R R R R L	525	840	65.43	969	324	51.50	21.30 %
6	R R R R L L	2100	700	94.10	900	300	79.44	15.58 %
6	R R R R L L	1050	700	76.76	900	300	67.26	12.38 %
6	R R R R L L	525	700	66.01	900	300	53.98	18.22 %
8	R R R R L L L L	2100	525	116.97	788	262	89.39	23.58 %
8	R R R R L L L L	1050	525	78.17	788	262	71.45	8.60 %
8	R R R R L L L L	525	525	63.25	788	262	57.69	8.79 %

presentamos en las tablas 3.5 y 3.6. En este problema tampoco se consigue reducir el tiempo de ejecución del subsistema homogéneo de los 4 procesadores rápidos, pero sí se reduce el tiempo empleado por *llp*, alcanzándose más de un 20 % de mejora para la ejecución con 5 procesadores y el tamaño de *buffer* más pequeño. En la figura 3.21 comparamos los tiempos al resolver el problema de caminos mínimos empleando las dos herramientas: homogénea (*llp*) y heterogénea (*llpW*).

3.6.3. El problema de la asignación de recursos

El problema de la asignación de recursos consiste en la asignación de unos recursos limitados a un conjunto de actividades para maximizar su rendimiento o minimizar su costo. Se dispone de M unidades de un recurso indivisible y N actividades o tareas. Por cada tarea N_j existe una función asociada $f_j(x)$, que representa el beneficio (o costo) obtenido cuando se asigna una cantidad x del recurso a la tarea j. Formulamos el problema mediante la siguiente ecuación:

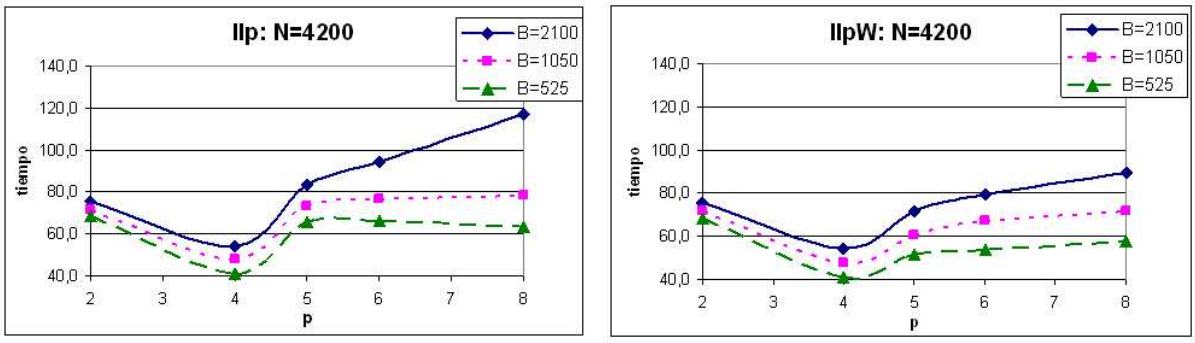

Figura 3.21: Tiempos de ejecución para resolver el problema de caminos mínimos de tamaño $N = 4200$ con un obstáculo en forma de U. Izquierda: utilizando la herramienta homogénea (*llp*). Derecha: Utilizando la herramienta para los sistemas heterogéneos (*llpW*).

$$\text{mín} \sum_{j=1}^{N} f_j(x_j)$$

$$sujeto\ a \sum_{j=1}^{N} x_j = M$$

$$y\ a \leq x_j$$

donde a es el número mínimo de unidades de recurso que se pueden asignar a una tarea.

Este problema ya lo enunciamos en la sección 2.8 cuando lo aplicamos para encontrar los parámetros que minimizan el tiempo de ejecución de un algoritmo maestro-esclavo. En esta sección, hemos resuelto dos casos particulares de problemas de asignación de recursos:

1. Disponemos de $M = 2600$ unidades del recurso y $N = 600$ actividades.
2. Disponemos de $M = 3000$ unidades del recurso y $N = 1000$ actividades.

Algoritmo secuencial

Para resolver el problema de forma secuencial, se emplea el código presentado en la figura 3.22. Se dispone de una matriz Q donde se almacenan los resultados. Esta matriz se inicializa a un valor constante *INFINITO* (líneas 4 a 6), excepto la primera fila que se inicializa a 0 (líneas 7 y 8). A continuación, se aplica la fórmula de programación dinámica para obtener los valores de las siguientes filas de la matriz Q (líneas 10 a 18).

Algoritmo paralelo

La paralelización del problema de asignación de recursos es similar a las implementaciones de los algoritmos anteriores: se asigna al procesador k-ésimo la obtención de los valores óptimos Q_k para todos los recursos m, donde $0 \leq m \leq M$ (figura 3.23). La dependencia en el cálculo de los valores $Q_k[m]$ sólo ocurre en valores adyacentes de k. En la etapa m (con m entre 0 y M), el procesador k-ésimo recibe de su vecino izquierdo el vector óptimo $Q_{k-1}[m]$ y, cuando el cálculo de $Q_k[m]$ esté terminado lo envía a su vecino por la derecha. Por lo tanto, el vector $Q_{k-1}[m]$ se utiliza para actualizar los valores óptimos $Q_k[r]$, con $r \geq m$.

En la figura 3.23 presentamos el código correspondiente a las etapas intermedias. La parte del código para la primera etapa (sin recepción de datos) y la última etapa (sin envío de datos) se implementan de forma independiente, pero no se incluyen en la figura por ser prácticamente idénticas.

```
 1 void rap_seq(void) {
 2    int n, m, i, fnm;
 3
 4    for (n = 1; n < N; n++)          // Inicialización
 5       for (m = 0; m <= M; m++)
 6          Q[n][m] = INFINITO;
 7    for (m = 0; m <= M; m++)
 8       Q[0][m] = 0;
 9
10    for(n = 1; n < N; n++) {        // Resolución del problema
11       for(m = 1; m <= M; m++) {
12           for (i = 1; i <= m - n + 1; i++) {
13              fnm = f(n - 1, i);              // 'f': función de beneficio
14              fnm = max(Q[n-1][m - i], fnm);
15              Q[n][m] = min(Q[n][m],fnm);
16          }
17       }
18    }
19 }
```

Figura 3.22: Función principal que resuelve el problema de asignación de recursos de forma secuencial.

```
 1 void rap_par(void) {
 2    LOCAL_VAR
 3       unsigned m;
 4       int Q[MAX_REC];                  // Beneficios óptimos para cada recurso
 5       int x, j, tmp, results;
 6    BEGIN;
 7
 8                . . .
 9
10                // Procesos intermedios
11        for(m = 0; m <= M; m++)
12            Q[m] = INFINITE;
13        for (m = 0; m <= M; m++) {
14            IN(&x);
15            OUT(&Q[m], 1, sizeof(int));
16            for (j = m + 1; j <= M; j++) {
17               tmp = max(x, f(name - 1, j - m));
18               Q[j] = min(Q[j], tmp);
19            }
20        }
21                . . .
```

Figura 3.23: Función que resuelve el problema de asignación de recursos de forma paralela.

Resultados computacionales

En las tablas 3.9 y 3.10 presentamos los resultados obtenidos al resolver el problema de la asignación de recursos para $N = 600$ actividades y $M = 2600$ unidades de recurso al utilizar las herramientas *llp* y *llpW* con una distribución por bloques y diferentes tamaños de *buffer*.

Tabla 3.9: Tiempos de ejecución obtenidos al resolver un problema de asignación de recursos con $N = 600$ actividades y $M = 2600$ unidades de recurso sobre el sistema homogéneo de las máquinas rápidas. El problema se resuelve con *llp*, una distribución por bloques, empleando diferentes número de procesadores y con 3 tamaños de *buffer* distintos.

Problema: N = 600 - M = 2600				
p	Configuración	*Buffer*	Grano	Tiempo Real
2	R R	1300	300	175.55
2	R R	650	300	138.17
2	R R	325	300	122.16
4	R R R R	1300	150	161.42
4	R R R R	650	150	104.37
4	R R R R	325	150	77.08

Tabla 3.10: Tiempos de ejecución obtenidos al resolver un problema de asignación de recursos con $N = 600$ actividades y $M = 2600$ unidades de recurso. El problema se resuelve con una distribución por bloques, empleando diferentes número de procesadores y con 3 tamaños de *buffer* distintos.

Problema: N = 600 - M = 2600								
			Herramienta *llp*		Herramienta *llpW*			
p	Configuración	*Buffer*	Grano	Tiempo Real	Grano_R	Grano_L	Tiempo Real	Porcentaje de mejora
5	R R R R L	1300	120	194.14	139	44	154.50	20.42 %
5	R R R R L	650	120	150.88	139	44	100.98	33.07 %
5	R R R R L	325	120	121.94	139	44	69.82	42.74 %
6	R R R R L L	1300	100	228.64	129	42	161.85	29.21 %
6	R R R R L L	650	100	167.02	129	42	113.38	32.12 %
6	R R R R L L	325	100	120.04	129	42	71.60	40.35 %
8	R R R R L L L L	1300	75	252.74	113	37	185.02	26.79 %
8	R R R R L L L L	650	75	177.06	113	37	117.66	33.55 %
8	R R R R L L L L	325	75	115.98	113	37	73.90	36.28 %

Como en los problemas anteriores, se observa en los resultados de *llp* un incremento del tiempo de ejecución al introducir en el *cluster* de prueba los procesadores lentos. Cuantos más procesadores lentos se utilicen más tiempo se necesita para resolver el problema, excepto para el tamaño de *buffer* más pequeño ($buffer = 325$).

Si comparamos los resultados obtenidos con *llp* y *llpW*, observamos una gran diferencia en los tiempos de ejecución. En este caso, si introducimos un único procesador lento en el *cluster* de pruebas, observamos una reducción en el tiempo de ejecución frente a los resultados obtenidos con el subsistema homogéneo de los 4 procesadores rápidos. Si aumentamos el número de procesadores lentos, el tiempo se incrementa, pero en ningún caso alcanza los tiempos obtenidos con la librería *llp*. Para 5 procesadores, pasamos de un tiempo de 121.94 segundos con *llp* y un tamaño de $buffer = 325$ a un tiempo de 69.82 segundos con *llpW* y el mismo tamaño de *buffer*. Esto supone una diferencia de más del 40 % en los resultados. En la figura 3.24 comparamos, de forma gráfica, los valores de los tiempos al resolver el problema empleando las dos herramientas.

A continuación, resolvemos el segundo problema de asignación de recursos, con $N = 1000$ actividades y $M = 3000$ unidades de recurso. En las tablas 3.11 y 3.12 presentamos los resultados obtenidos al resolver el problema con una distribución por bloques y diferentes tamaños de *buffer*.

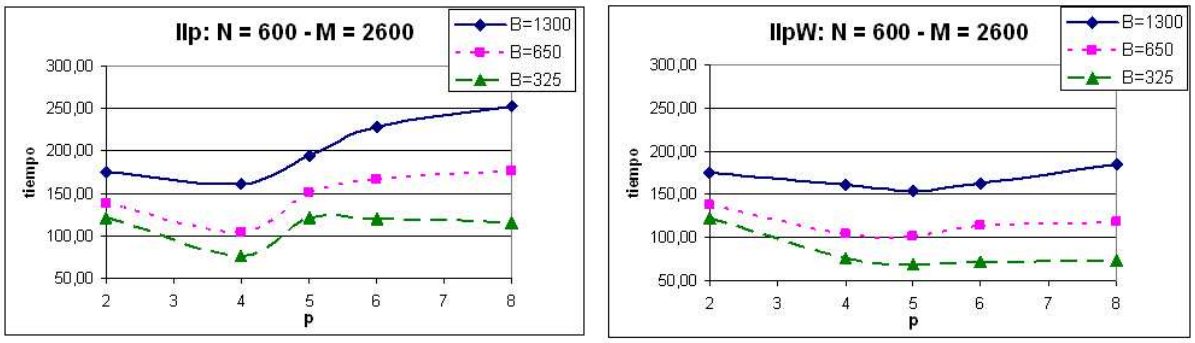

Figura 3.24: Tiempos de ejecución obtenidos al resolver el problema de la asignación de recursos con $N = 600$ tareas y $M = 2600$ unidades de recursos. Izquierda: utilizando la herramienta homogénea (*llp*). Derecha: Utilizando la herramienta para los sistemas heterogéneos (*llpW*).

Tabla 3.11: Tiempos de ejecución obtenidos al resolver un problema de la asignación de recursos con $N = 1000$ tareas y $M = 3000$ unidades de recurso sobre el sistema homogéneo de las máquinas rápidas. El problema se resuelve con *llp*, una distribución por bloques, diferente número de procesadores y 3 tamaños de *buffer* distintos.

Problema: N = 1000 - M = 3000				
p	Configuración	*Buffer*	Grano	Tiempo Real
2	R R	1500	500	343.80
2	R R	750	500	309.46
2	R R	375	500	234.61
4	R R R R	1500	250	312.12
4	R R R R	750	250	220.21
4	R R R R	375	250	163.17

Tabla 3.12: Tiempos de ejecución obtenidos al resolver un problema de la asignación de recursos con $N = 1000$ tareas y $M = 3000$ unidades de recurso. El problema se resuelve con una distribución por bloques, empleando diferentes número de procesadores y con 3 tamaños de *buffer* distintos.

Problema: N = 1000 - M = 3000								
			Herramienta *llp*		Herramienta *llpW*			
p	Configuración	*Buffer*	Grano	Tiempo Real	Grano_R	Grano_L	Tiempo Real	Porcentaje de mejora
5	R R R R L	1500	200	435.78	231	76	342.97	21.30 %
5	R R R R L	750	200	333.78	231	76	226.92	32.02 %
5	R R R R L	375	200	272.00	231	76	155.89	42.69 %
6	R R R R L L	1500	166	520.90	214	72	365.47	29.84 %
6	R R R R L L	750	166	352.10	214	72	247.69	29.65 %
6	R R R R L L	375	166	265.86	214	72	159.99	39.82 %
8	R R R R L L L L	1500	125	550.53	188	62	409.76	25.57 %
8	R R R R L L L L	750	125	382.52	188	62	262.21	31.45 %
8	R R R R L L L L	375	125	257.42	188	62	172.98	32.80 %

Para este tamaño de problema hemos obtenido resultados similares a los observados para el problema anterior. Mientras que en las ejecuciones con la librería homogénea (*llp*), todos los resultados con 5 o más procesadores lentos son mucho peores que los resultados del subsistema homogéneo de procesadores

rápidos; en las ejecuciones con la librería heterogénea, se reduce este tiempo. Cuando incluimos el primer procesador lento, conseguimos reducir el tiempo de ejecución con 4 máquinas, aunque al seguir incrementando el número de procesadores lentos este tiempo empeora. Si comparamos los resultados con 5 procesadores y tamaño del *buffer* = 375, tenemos para *llp* un tiempo de ejecución de 272.00 segundos, mientras que para *llpW* este tiempo es de 155.89 segundos. Esto implica una mejora en el rendimiento del sistema de más del 40 %. En la figura 3.25 mostramos de forma gráfica los resultados obtenidos al resolver el problema.

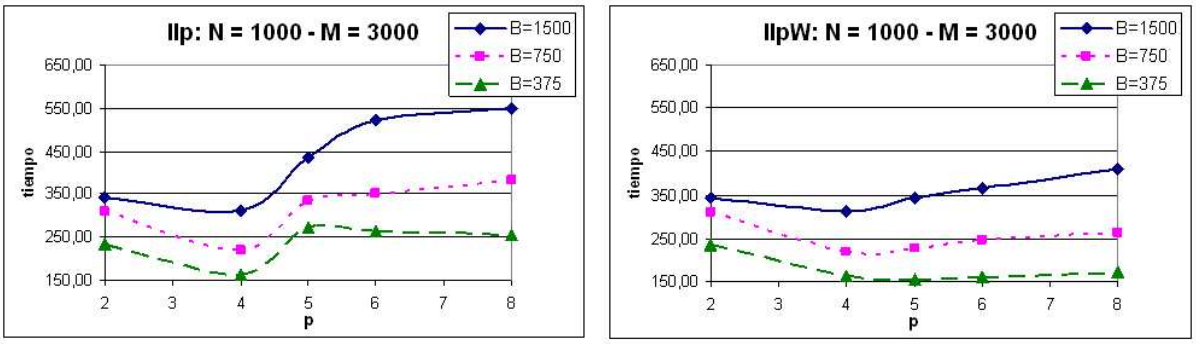

Figura 3.25: Tiempos de ejecución obtenido al resolver el problema de la asignación de recursos con $N = 1000$ tareas y $M = 3000$ unidades de recurso. Izquierda: utilizando la herramienta homogénea (*llp*). Derecha: Utilizando la herramienta para los sistemas heterogéneos (*llpW*).

3.6.4. Problema de la subsecuencia común más larga

El problema de la subsecuencia común más larga consiste en comparar dos cadenas buscando el subconjunto más largo de éstas que sea común a ambas. Se dispone de las dos cadenas de símbolos: la *cadenaA* de longitud N y la *cadenaB* de M símbolos y buscamos en el interior de la *cadenaA* la subsecuencia más larga de la *cadenaB*. Este problema se puede formular mediante las siguientes ecuaciones:

$$F(i,j) = \begin{cases} 0 & \text{si } i = 0 \text{ o } j = 0 \\ \text{máx}\{F(i-1,j), F(i,j-1)\} & \text{si } A_j \neq B_i \\ F(i-1,j-1) & \text{si } A_j = B_i \end{cases}$$

Definamos a continuación las instancias a resolver:

1. La longitud de *cadenaA* es $N = 6000$ y la longitud de *cadenaB* es $M = 10000$.
2. La longitud de *cadenaA* es $N = 12000$ y la longitud de *cadenaB* es $M = 20000$.

Algoritmo secuencial

Este problema se resuelve de forma secuencial utilizando el código presentado en la figura 3.26. Es un algoritmo similar al del problema de la mochila unidimensional (subsección 3.6.1). Se utilizan dos vectores de datos: *fanterior* contiene los datos de la etapa anterior y *factual* con los datos de la etapa actual del algoritmo y un vector auxiliar (*faux*). Primero se inicializa el vector *fanterior* a cero y, a continuación, se aplica la ecuación para calcular los valores de la etapa actual. Una vez finalizado el cálculo de la etapa actual, se intercambian los valores de los vectores *fanterior* y *factual* para poder usarlos correctamente en la etapa siguiente.

```
1 void Subsecuencia_seq(void)  {
2    int i, j, *fanterior, *factual, *faux;
3
4         // Asignación de memoria a 'fanterior' y 'factual' e inicialización a cero
5
6    for (j = 1; j <= N; j++) {
7        for (i = 0; i <= M; i++) {
8            if (i == 0)
9                factual[i] = 0;
10           else
11               if (cadA[j] != cadB[i])
12                   factual[i] = max(factual[i-1], fanterior[i]);
13                else
14                   factual[i] = fanterior[i-1] + 1;
15       }
16       // Intercambio de 'factual' y 'fanterior'
17   }
18   printf("\tLongitud de la subsecuencia :%d \t", fanterior[M]);
19        // Liberar la memoria
20 }
```

Figura 3.26: Función secuencial que resuelve el problema de la búsqueda de la subsecuencia más larga.

Algoritmo paralelo

En la paralelización de este algoritmo se precisan tantos procesos como caracteres en la *cadenaA*. A cada uno se le asocia una fila de la matriz de salida (*cadenaB*). En la figura 3.27 se encuentra la función que resuelve el problema de forma paralela. Se implementa de forma independiente la primera etapa del algoritmo, la última y el resto de las etapas. Como en los problemas anteriores, sólo mostramos en la figura 3.27 el código correspondiente a las etapas intermedias.

```
1 void Subsecuencia_par(void)  {
2    LOCAL_VAR
3       int i, x, *f;
4    BEGIN;
5        . . .
6      for (i = 0; i <= M; i++) {       // Procesos intermedios
7          IN(&x);
8          if (i != 0)
9             if (cadA[name - 1] != cadB[i - 1])
10               f[i] = max(f[i - 1], x);
11         OUT(&f[i], 1, sizeof(int));
12         if (i != M)
13            if (cadA[name-1] == cadB[i])
14               f[i + 1] = x + 1;
15     }
16       . . .
17   END;
18 }
```

Figura 3.27: Función paralela que resuelve el problema de la búsqueda de la subcadena más larga.

Resultados computacionales

En las tablas 3.13 y 3.14 se presentan los resultados obtenidos al resolver el primer problema de la búsqueda de la subsecuencia más larga ($N = 6000$ y $M = 10000$) con la herramienta para un entorno homogéneo (*llp*) y con la herramienta para entornos heterogéneos (*llpW*).

Tabla 3.13: Tiempos de ejecución obtenidos al resolver un problema de la búsqueda de la subsecuencia más larga con $N = 6000$ la longitud de la primera cadena y con $M = 20000$ la longitud de la segunda cadena. El problema se resuelve sobre el sistema homogéneo de las máquinas rápidas utilizando *llp*, una distribución por bloques, diferente número de procesadores y 3 tamaños de *buffer* distintos.

Problema: N = 6000 - M = 10000				
p	Configuración	*Buffer*	Grano	Tiempo Real
2	R R	5000	3000	37.64
2	R R	2500	3000	33.02
2	R R	1250	3000	30.13
4	R R R R	5000	1500	25.22
4	R R R R	2500	1500	18.60
4	R R R R	1250	1500	15.23

Tabla 3.14: Tiempos de ejecución obtenidos al resolver un problema de la búsqueda de la subsecuencia más larga con $N = 6000$ la longitud de la primera cadena y con $M = 20000$ la longitud de la segunda cadena. El problema se resuelve utilizando una distribución por bloques, empleando diferentes número de procesadores y con 3 tamaños de *buffer* distintos.

Problema: N = 6000 - M = 10000								
			Herramienta *llp*		Herramienta *llpW*			
p	Configuración	*Buffer*	Grano	Tiempo Real	Grano_R	Grano_L	Tiempo Real	Porcentaje de mejora
5	R R R R L	5000	1200	33.57	1385	460	24.59	26.75 %
5	R R R R L	2500	1200	29.31	1385	460	17.11	41.63 %
5	R R R R L	1250	1200	23.10	1385	460	13.78	40.36 %
6	R R R R L L	5000	1000	39.56	1286	428	26.63	32.68 %
6	R R R R L L	2500	1000	29.18	1286	428	17.69	39.36 %
6	R R R R L L	1250	1000	23.09	1286	428	12.87	44.27 %
8	R R R R L L L L	5000	750	42.19	1125	375	27.91	33.84 %
8	R R R R L L L L	2500	750	27.81	1125	375	17.71	36.30 %
8	R R R R L L L L	1250	750	21.76	1125	375	12.50	42.56 %

Observando las tablas anteriores (3.13 y 3.14), comprobamos que los resultados obtenidos son similares a los de problemas anteriores: en las ejecuciones con la herramienta para los sistemas homogéneos (*llp*) y configuraciones donde existen procesadores lentos, nos encontramos tiempos de ejecución muy superiores a los tiempos reales del sistema de los 4 procesadores rápidos. Por ejemplo, podemos ver que el tiempo necesario para resolver el problema con la herramienta *llp*, 8 procesadores y tamaño del *buffer* = 375 es más de un 40 % peor que el tiempo requerido por los 4 procesadores rápidos para el mismo tamaño del *buffer*. Si comparamos estos resultados con los de la herramienta *llpW*, podemos comprobar que el tiempo con 8 procesadores y *buffer* = 375 se ha reducido en más de un 42 % (ha pasado de requerir 21.76 segundos con *llp* a necesitar solamente 12.50 segundos para resolver el problema con *llpW*). Además, con *llpW* y 8 procesadores se consigue mejorar el resultado de $p = 4$ en más de un 15 %. En la figura 3.28 comparamos gráficamente los resultados para este problema.

Por último, resolvemos el segundo problema de la búsqueda de la subsecuencia más larga ($N = 12000$ y $M = 20000$). En las tablas 3.15 y 3.16 se encuentran los resultados obtenidos al resolver el problema.

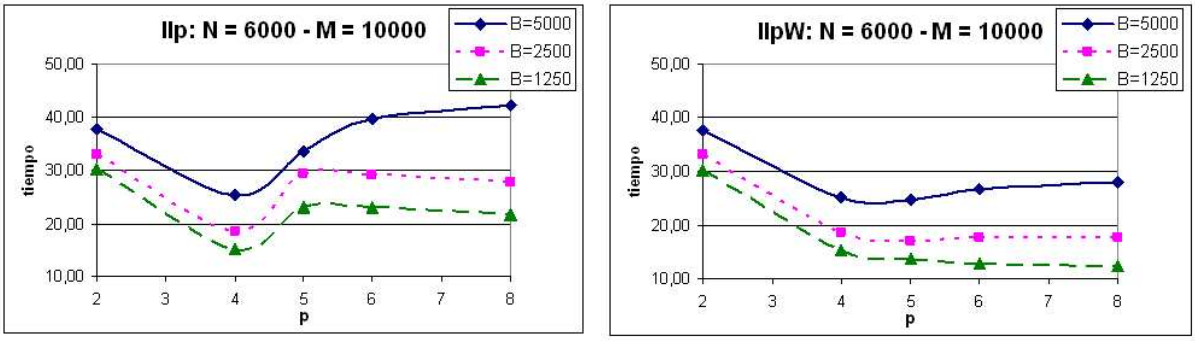

Figura 3.28: Tiempos de ejecución reales al resolver el problema de la búsqueda de la subsecuencia más larga, con $N = 6000$ la longitud de la primera cadena y con $M = 10000$ la longitud de la segunda cadena. Izquierda: utilizando la herramienta homogénea (*llp*). Derecha: Utilizando la herramienta para los sistemas heterogéneos (*llpW*).

Tabla 3.15: Tiempos de ejecución obtenidos al resolver un problema de la búsqueda de la subsecuencia más larga, con $N = 12000$ la longitud de la primera cadena y $M = 20000$ la longitud de la segunda cadena. El problema se resuelve sobre el sistema homogéneo de las máquinas rápidas, utilizando *llp*, una distribución por bloques, diferente número de procesadores y 3 tamaños de *buffer* distintos.

Problema: N = 12000 - M = 20000				
p	Configuración	*Buffer*	Grano	Tiempo Real
2	R R	10000	6000	172.86
2	R R	5000	6000	150.77
2	R R	2500	6000	140.28
4	R R R R	10000	3000	135.64
4	R R R R	5000	3000	106.88
4	R R R R	2500	3000	94.19

Tabla 3.16: Tiempos de ejecución obtenidos al resolver un problema de la búsqueda de la subsecuencia más larga, con $N = 12000$ la longitud de la primera cadena y $M = 20000$ la longitud de la segunda cadena. El problema se resuelve utilizando una distribución por bloques, empleando diferentes número de procesadores y con 3 tamaños de *buffer* distintos.

Problema: N = 12000 - M = 20000								
			Herramienta *llp*		Herramienta *llpW*			
p	Configuración	*Buffer*	Grano	Tiempo Real	Grano_R	Grano_L	Tiempo Real	Porcentaje de mejora
5	R R R R L	10000	2400	178.73	2769	924	138.78	22.35 %
5	R R R R L	5000	2400	139.64	2769	924	104.79	24.96 %
5	R R R R L	2500	2400	120.13	2769	924	87.57	27.10 %
6	R R R R L L	10000	2000	183.73	2571	858	139.38	24.14 %
6	R R R R L L	5000	2000	138.31	2571	858	101.15	26.86 %
6	R R R R L L	2500	2000	113.21	2571	858	82.99	26.69 %
8	R R R R L L L L	10000	1500	192.32	2250	750	148.56	22.75 %
8	R R R R L L L L	5000	1500	126.73	2250	750	97.64	22.95 %
8	R R R R L L L L	2500	1500	94.45	2250	750	73.52	22.15 %

En las tablas 3.15 y 3.16 observamos resultados similares al problema anterior. Cuando asignamos la misma cantidad de grano a todos los procesadores (usamos *llp*), el tiempo aumenta al introducir en la plataforma de prueba los procesadores lentos. Al incrementar el número de procesadores lentos, el tiempo se va reduciendo, pero sin mejorar el tiempo de los 4 procesadores rápidos. Al asignar cantidades distintas de grano a cada tipo de procesador (utilizando *llpW*), se observa que se consigue aumentar el rendimiento del sistema, reduciendo el tiempo de ejecución en más de un 20 % al ejecutar con 8 procesadores frente a la configuración de los 4 procesadores rápidos. En la figura 3.29 pueden observarse de forma gáfica las diferencias entre los resultados obtenidos en las ejecuciones.

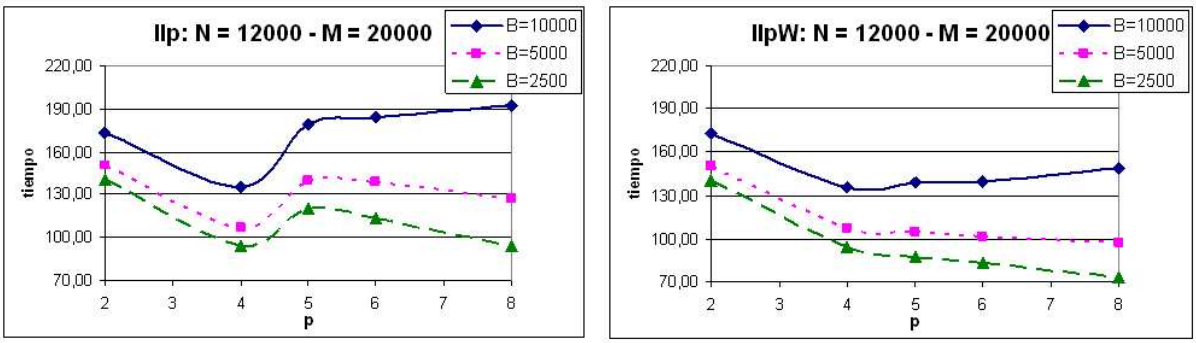

Figura 3.29: Tiempos de ejecución obtenidos al resolver el problema de la búsqueda de la subsecuencia más larga, $N = 12000$ la longitud de la primera cadena y $M = 20000$ la longitud de la segunda cadena. Izquierda: utilizando la herramienta homogénea (*llp*). Derecha: Utilizando la herramienta para los sistemas heterogéneos (*llpW*).

3.7. El modelo analítico

Una vez analizado el funcionamiento de algoritmos *pipeline* en sistemas homogéneos y heterogéneos, pretendemos predecir el tiempo necesario para resolver un problema con unos valores de grano (G) y *buffer* (B) específicos. Para ello, hemos formulado un modelo analítico que determina el tiempo invertido desde el instante de comienzo del *pipeline* heterogéneo, hasta el instante de tiempo en que finaliza el último procesador. Posteriormente, dada una máquina paralela, nuestro objetivo es obtener los valores óptimos de G y B para la instancia de un problema determinado; es decir, los valores de grano y *buffer* que hacen que el tiempo de ejecución del programa sea mínimo. Este problema ha sido formulado previamente en [85, 129].

Para modelar un algoritmo *pipeline* consideramos que puede ser caracterizado por un conjunto de procesadores realizando alguna tarea. Las dependencias entre los datos fuerzan a cada procesador a recibir información del procesador anterior en el anillo y, después de procesar esa información, enviar datos al siguiente procesador en el anillo. Este esquema general puede variar de un problema a otro.

En *pipelines* homogéneos (figura 3.3-Izquierda) se puede derivar una ecuación analítica muy simple para el tiempo paralelo, T_{par}, cuando el número de procesadores (p) es igual al número de etapas del *pipeline* (n): $T_{par} = T_s + T_c$, donde T_s denota el tiempo de arranque correspondiente al procesador p_{p-1}:

$$T_s = (p-2) * (t + \beta + \tau * w)$$

y T_c representa el tiempo que el último procesador (p_{p-1}) invierte en realizar el bucle completo de cómputo:

$$T_c = M * (t + \beta + \tau * w)$$

Al tratarse del caso homogéneo, los valores de t, β y τ son iguales para todos los procesadores.

Hemos presentado este modelo analítico para el caso general (cuando el número de etapas es mayor que el número de procesadores) en sistemas homogéneos en [82, 83, 84, 132, 133], validando nuestro modelo al ejecutar con diferentes problemas y obteniendo los valores de p, G y B para una ejecución óptima.

Para el caso heterogéneo hemos desarrollado nuestro modelo analítico en tres etapas. Estos modelos y su validación han sido presentados en [8, 9]. Primero consideramos la situación donde el número de etapas es igual al número de procesadores que trabajan en el *pipeline* (subsección 3.7.1). A continuación, el caso donde el número de etapas es mayor que el número de procesadores considerando un *mapping* cíclico puro (subsección 3.7.2); y finalizamos con la situación general de un *mapping* cíclico por bloques (subsección 3.7.3). Para cada uno de ellos, hemos validado los resultados teóricos obtenidos con los resultados computacionales en nuestro *cluster* de pruebas (sección 3.8).

3.7.1. El número de etapas es igual al número de procesadores

Consideremos en primer lugar que el número de procesadores físicos, p, es igual al número de etapas del *pipeline*, n ($n = p$). Derivamos una ecuación que calcula el tiempo total de ejecución como el tiempo de arranque del último procesador más el tiempo que invierte el último procesador en realizar el bucle de cómputo completo. Obviamente, el procesador p_i del *pipeline* comienza su cómputo en el instante $\sum_{j=0}^{i-1}(t_j + \beta_j + \tau_j w)$: cada uno de los procesadores anteriores a p_i debe producir y enviar un bloque antes de que p_i pueda comenzar.

Denotamos por T_s, el tiempo de arranque del procesador p_{p-1}:

$$T_s = \sum_{i=0}^{p-2}(t_i + \beta_i + \tau_i w)$$

y por p_{i_0} al procesador más lento del *pipeline*, es decir, el primer procesador del *pipeline*, tal que

$$(t_{i_0} + \beta_{i_0} + \tau_{i_0} w) \geq \max_{i=0}^{p-2}(t_i + \beta_i + \tau_i w)$$

La ecuación siguiente establece el tiempo total de ejecución paralela del *pipeline* como el tiempo de arranque del procesador más lento, más el tiempo que tarda este procesador en realizar su cómputo, añadiéndole el tiempo necesario para que los siguientes procesadores en el *pipeline* computen la última iteración del bucle. Dependiendo de la posición del procesador más lento en el *pipeline* se pueden producir 2 situaciones.

$$T_{par} = \begin{cases} T_s + Mt_{p-1} & si\ t_{p-1} \geq \max_{i=0}^{p-2}(t_i + \beta_i + \tau_i w) \\ \sum_{i=0}^{i_0-1}(t_i + \beta_i + \tau_i w) + M(t_{i_0} + \beta_{i_0} + \tau_{i_0} w) + \sum_{i=i_0+1}^{p-2}(t_i + \beta_i + \tau_i w) + t_{p-1} & en\ otro\ caso \end{cases}$$

donde el primer caso se produce cuando el último procesador es el más lento, puesto que su tiempo de cómputo es mayor o igual que el tiempo invertido por los restantes procesadores en calcular y enviar los datos. Hay que tener en cuenta que el último procesador realiza la última etapa del problema y no lleva a cabo ningún envío de información. En la segunda situación, sin embargo, el último procesador no es el más lento del *pipeline*, por lo que se producen huecos durante su ejecución. En este caso, el tiempo viene determinado por el tiempo del procesador más lento (p_{i_0}).

La figura 3.30 muestra las dos opciones que podemos encontrar. En la figura 3.30-Izquierda se presenta la situación donde el último procesador es el más lento de los que trabajan en el *pipeline*: el tiempo total de ejecución paralela se obtiene sumando el tiempo de arranque más el tiempo de cómputo para este

procesador (p_{p-1}). Mientras que en la figura 3.30-Derecha el procesador más lento se encuentra en una posición intermedia: el tiempo total de ejecución es la suma del tiempo de arranque, más el tiempo de cómputo del procesador más lento, más el tiempo de cómputo de la última iteración para el resto de los procesadores. En ambos casos, la línea en negrita indica el camino que establece el tiempo total de ejecución paralela (T_{par}).

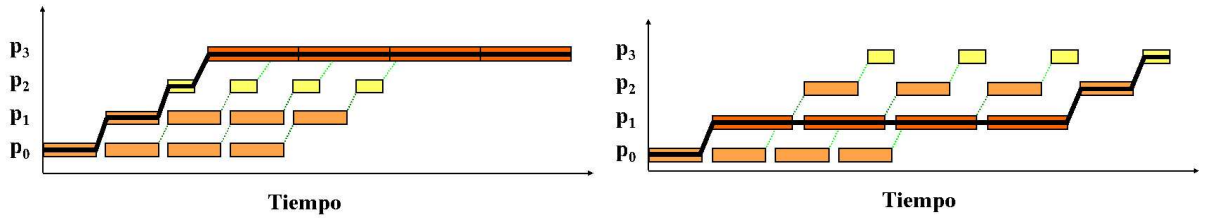

Figura 3.30: Diagramas de tiempo: La línea en negrita indica el camino que establece el tiempo de ejecución. Izquierda: el procesador más lento es el último del *pipeline*, $t_{p-1} \geq \max_{i=0}^{p-2}(t_i + \beta_i + \tau_i w)$. El tiempo total de ejecución es obtenido sumando el tiempo de arranque más el tiempo de cómputo para este procesador. Derecha: El procesador más lento se encuentra en medio del *pipeline*: en las filas correspondientes a los procesadores siguientes aparecen huecos en el cómputo, es decir, existen períodos de tiempo donde los procesadores no están trabajando. El tiempo total de ejecución se obtiene sumando el tiempo de arranque y de cómputo para el procesador más lento, más el tiempo de cómputo de los restantes procesadores durante la última iteración.

Para poder formular el tiempo de ejecución (T_{par}) de forma similar al caso homogéneo, denotamos por T_c al tiempo necesario para realizar completamente todo el cómputo, incluyendo el envío de M paquetes de un elemento ($G_i = 1$ y $B_i = 1$). Para el primer caso, cuando $t_{p-1} \geq \max_{i=0}^{p-2}(t_i + \beta_i + \tau_i w)$, el valor de T_c se obtiene directamente de la fórmula anterior ($T_c = Mt_{p-1}$).

Para el segundo caso, partimos del tiempo de ejecución total:

$$\sum_{i=0}^{i_0-1}(t_i + \beta_i + \tau_i w) + M(t_{i_0} + \beta_{i_0} + \tau_{i_0} w) + \sum_{i=i_0+1}^{p-2}(t_i + \beta_i + \tau_i w) + t_{p-1} =$$

$$\sum_{i=0}^{p-2}(t_i + \beta_i + \tau_i w) + (M-1)(t_{i_0} + \beta_{i_0} + \tau_{i_0} w) + t_{p-1} =$$

$$T_s + (M-1)(t_{i_0} + \beta_{i_0} + \tau_{i_0} w) + t_{p-1}$$

Como T_s es el tiempo de arranque del último procesador (p_{p-1}), ya tenemos su tiempo de cómputo. Resumiendo todo lo anterior, el tiempo que requiere el procesador p_{p-1} para finalizar su ejecución, una vez que ha comienzado a trabajar, viene dado por la siguiente ecuación:

$$T_c = \begin{cases} Mt_{p-1} & si\ t_{p-1} \geq \max_{i=0}^{p-2}(t_i + \beta_i + \tau_i w) \\ Mt_{p-1} + (M-1)(t_{i_0} - t_{p-1} + \beta_{i_0} + \tau_{i_0} w) & en\ otro\ caso. \end{cases}$$

donde $t_{i_0} - t_{p-1} + \beta_{i_0} + \tau_{i_0} w$ es el tiempo invertido por el procesador p_{p-1} esperando por los datos (huecos).

A partir de las consideraciones anteriores, podemos calcular el tiempo total de ejecución paralela de un *pipeline* con $p = n$ como: $T_{par} = T_s + T_c$.

3.7.2. El número de etapas es superior al número de procesadores: *mapping* cíclico puro

Consideremos ahora un número de etapas o procesos virtuales del *pipeline* (n) mayor que el número de procesadores físicos disponibles (p); es decir, $p < n$. Es necesario asignar las etapas del *pipeline* entre los procesadores disponibles. Hemos realizado la primera aproximación considerando sólo el caso más sencillo, donde $G_i = 1$ y $B_i = 1$. Posteriormente desarrollamos el caso general.

En la figura 3.31 se muestra el esquema de la ejecución de un *pipeline* con $p < n$, identificando los procesadores físicos y los procesos virtuales. Puesto que cada procesador contiene un único proceso virtual, una banda está compuesta por p procesos. Por lo tanto, el número de bandas que se producen al ejecutar un *pipeline* utilizando un *mapping* cíclico puro es $NumBandas = \frac{n}{p}$. En la figura 3.31 aparece también indicada la distribución de procesos entre las bandas. De esta forma, el proceso virtual q es ejecutado en la banda número k si y sólo si $\frac{q}{p} = k$; y es ejecutado en el procesador p_i donde $i = q \ mod \ p$.

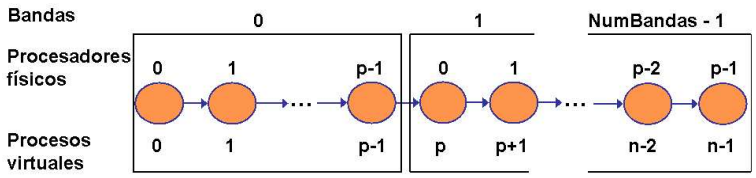

Figura 3.31: Ejecución del *pipeline* mediante una estrategia de ejecución cíclica pura con $p < n$.

Como se observa en la figura 3.31, en este caso el último procesador sí realiza envío de datos cuando está procesando bandas intermedias, pero no en la última (el proceso virtual $n-1$). Y el procesador p_0 recibe datos del procesador anterior (p_{p-1}) en todas las bandas excepto en la primera (proceso virtual 0). A continuación, calculamos el tiempo de ejecución en el *pipeline*. Denotamos por

$$T'_s = \sum_{i=0}^{p-1}(t_i + \beta_i + \tau_i w) = T_s + t_{p-1} + \beta_{p-1} + \tau_{p-1} w$$

al tiempo de arranque entre bandas y por

$$T'_c = M(t_{p-1} + \beta_{p-1} + \tau_{p-1} w + \max_{i=0}^{p-1}(t_i + \beta_i + \tau_i w - (t_{p-1} + \beta_{p-1} + \tau_{p-1} w)))$$

al tiempo de cómputo del procesador p_{p-1} en una banda intermedia. T'_s incluye el tiempo necesario para que el último procesador compute y envíe el primero de sus datos; y T'_c incluye el tiempo de cómputo y envío de los datos más todos los tiempos de espera (huecos). Como se ilustra en la figura 3.32, la relación entre los valores de T'_s y T'_c modifica considerablemente el rendimiento del sistema. La figura 3.32-Izquierda corresponde a la situación donde el conjunto completo de procesadores está continuamente trabajando después de la fase de arranque. La figura 3.32-Derecha representa la situación donde los procesadores se sincronizan entre las bandas, y se producen huecos, durante los cuales, los procesadores tienen que esperar por datos de la banda anterior. Obviamente, la región donde $T'_c < T'_s$ corresponde a un uso ineficiente de los procesadores.

El tiempo total de ejecución para un *mapping* cíclico puro con n etapas y p procesadores, donde $p < n$, sin utilizar *buffer*, viene determinado por:

$$T = \begin{cases} T_1 = T_s + (NumBandas - 1)T'_c + T_c & si \ T'_c > T'_s \\ \\ T_2 = (NumBandas - 1)T'_s + T_s + T_c & si \ T'_s > T'_c \end{cases}$$

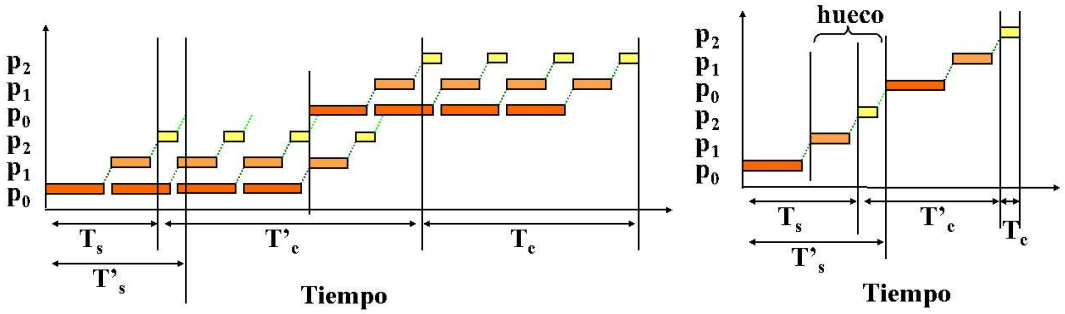

Figura 3.32: Diagramas de tiempo: Izquierda: $T'_c > T'_s$: El conjunto completo de procesadores está continuamente trabajando después de la fase de arranque. Derecha: $T'_c < T'_s$: Los procesadores están sincronizados entre bandas y pueden invertir tiempo esperando por los datos de la banda anterior.

3.7.3. El número de etapas es superior al número de procesadores: *mapping* cíclico por bloques

Consideraremos ahora el caso más general, donde se realiza el *mapping* cíclico por bloques. En una ejecución por bloques se reduce el número de comunicaciones, pero aumenta el tiempo de arranque del *pipeline*. La ejecución cíclica se comporta de forma totalmente opuesta: el tiempo de arranque del pipeline es reducido pero existen muchas comunicaciones. La ejecución cíclica por bloques es una situación intermedia, donde se realizan menos comunicaciones que en la ejecución cíclica y el tiempo de inicio del *pipeline* es menor que en la ejecución por bloques.

Se asignan G_i procesos virtuales al procesador p_i y la comunicación entre los procesadores se realiza en paquetes de tamaño B_i. Por simplicidad no hemos considerado tamaños de *buffer* distintos para los diferentes procesadores. Esto significa que trabajamos con $B_i = B$ para $i = 0, \ldots, p-1$. En este contexto, el procesador p_i computará $G_i * B$ elementos en cada iteración y cada comunicación incluirá B elementos de tamaño w. Siendo más precisos, cada procesador virtual ejecuta un bucle similar al que aparece en la figura 3.33.

```
1 j = 0;
2 while (j < M/B) {
3     recibir ();                  // Excepto el primer proceso virtual
4     computar ();                 // Computa G[i] * B elementos
5                                  // Envía B elementos de tamaño w
6     enviar (B * elemento_w);     // Excepto el último proceso virtual
7     j++;
8 }
```

Figura 3.33: Código estándar para un algoritmo *pipeline* resuelto de forma paralela mediante una distribución cíclica por bloques.

En la figura 3.34 se muestra el funcionamiento del pipeline para este caso general: el grano asignado a cada procesador puede ser distinto de 1 y el número de procesos virtuales que componen cada banda es $\sum_{j=0}^{p-1} G_j$. Por lo tanto, un proceso virtual, q, forma parte de la banda k si y sólo si $\frac{q}{\sum_{j=0}^{p-1} G_j} = k$ y se ejecuta sobre el procesador físico p_i si y sólo si $i = \frac{q}{\sum_{j=0}^{p-1} G_j} \ mod\ p$. El nombre local de los procesos virtuales dentro del procesador físico varía entre 0 y el valor del grano menos 1.

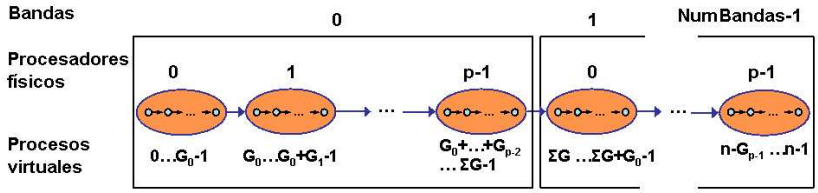

Figura 3.34: Ejecución del *pipeline* mediante una estrategia de ejecución cíclica por bloques con $p < n$.

De acuerdo con la notación utilizada a lo largo de este trabajo, el procesador p_i invierte en cada iteración un tiempo de cómputo de $G_i * B * t_i$ ($G_i * B$ elementos) y un tiempo de comunicación de $\beta_i + \tau_i Bw$ (envía B elementos de tamaño w), en un *pipeline* con $n' = \frac{pn}{\sum_{j=0}^{p-1} G_j}$ etapas.

Por analogía con el caso cíclico, tenemos las siguientes definiciones: El procesador más lento p_{i_0} está caracterizado por ser el primer procesador del *pipeline* tal que

$$(G_{i_0} Bt_{i_0} + \beta_{i_0} + \tau_{i_0} Bw) \geq \max_{i=0}^{p-2}(G_i Bt_i + \beta_i + \tau_i Bw).$$

El tiempo de arranque para el procesador p_{p-1} es

$$T_s = \sum_{i=0}^{p-2}(G_i Bt_i + \beta_i + \tau_i Bw)$$

y el tiempo de arranque entre bandas es

$$T'_s = \sum_{i=0}^{p-1}(G_i Bt_i + \beta_i + \tau_i Bw)$$

El tiempo de cómputo para el último procesador es:

$$T_c = \begin{cases} \frac{M}{B} G_i Bt_{p-1} & si\ G_i Bt_{p-1} \geq \max_{i=0}^{p-2}(G_i Bt_i + \beta_i + \tau_i Bw) \\ \frac{M}{B} G_i Bt_{p-1} + (\frac{M}{B} - 1)(G_{i_0} Bt_{i_0} - G_{p-1} Bt_{p-1} + \beta_{i_0} + \tau_{i_0} Bw) & en\ otro\ caso \end{cases}$$

y el tiempo de cómputo para el último procesador en una banda intermedia es:

$T'_c = M(G_{p-1} Bt_{p-1} + \beta_{p-1} + \tau_{p-1} Bw + G_{i_0} Bt_{i_0} + \beta_{i_0} + \tau_{i_0} Bw - (G_{p-1} t_{p-1} + \beta_{p-1} + \tau_{p-1} Bw))$

El número de bandas en este caso es: $NumBandas = \frac{n'}{p}$.

Considerando que T_s, T_c, T'_s y T'_c están definidas en términos de $G_i Bt_i$: si $p = n'$, trabajamos con una única banda mediante un *mapping* por bloques puro y el tiempo de ejecución es $T_s + T_c$. Si $p < n'$, el tiempo total de ejecución sobre un *mapping* cíclico por bloques con n' etapas y p procesadores, puede obtenerse directamente como:

$$T_{par} = \begin{cases} T_1 = T_s + (NumBandas - 1)T'_c + T_c & si\ T'_c > T'_s \\ T_2 = (NumBandas - 1)T'_s + T_s + T_c & si\ T'_s > T'_c \end{cases}$$

De nuevo el modelo considera las dos situaciones representadas en la figura 3.32.

El modelo presentado hasta ahora, considera la situación donde las combinaciones de los granos generan trabajo para todo el conjunto de procesadores a lo largo de todas las bandas. Sin embargo, en la práctica, es posible que algunos procesadores no trabajen en la última banda; además, el último procesador que trabaja en la última banda (p_{i_1}) puede procesar una cantidad de trabajo diferente al que realiza en el resto de las bandas ($G_{i_1}^{ub}$). Se puede observar un impacto importante en la precisión del modelo si abstraemos y reducimos estos casos particulares al modelo general. Este efecto es también observado en un *pipeline* homogéneo. En sistemas homogéneos se puede derivar una ecuación analítica directamente del caso general. Sin embargo, la casuística del caso heterogéneo es mucho más compleja, puesto que depende de la posición del procesador que más tiempo invierte en el *pipeline* (el procesador más lento) y del conjunto de procesadores incluidos en el procesamiento de la última banda. Denominamos por p_{i_0} al procesador más lento de las bandas intermedias y por p_{i_2} al procesador más lento de la última banda. A continuación mostramos las posibles situaciones que consideramos en el desarrollo del modelo analítico. La figura 3.35 muestra un diagrama de tiempo para un *pipeline* con $p = 4$. Ordenamos los procesadores por su potencia de cómputo: p_3, p_0, p_2 y p_1, donde p_3 es el procesador más rápido y p_1 el procesador más lento, y podemos ver, en la figura, que el procesador más lento es el último procesador de la última banda. Suponemos que p_{i_0} mantiene su grano en la última banda o bien que, a pesar reducir su grano, sigue siendo el procesador más lento en la última banda.

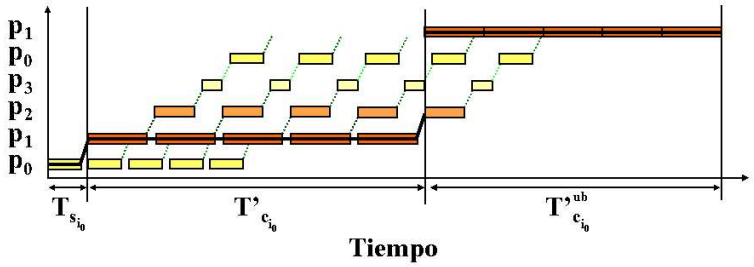

Figura 3.35: Diagrama de tiempo para un *pipeline* con $p = 4$, donde el procesador más lento es el último procesador de la última banda. En este caso, el tiempo total de ejecución es obtenido de acuerdo al tiempo de ejecución del procesador más lento (p_{i_0}): el tiempo de arranque del procesador p_{i_0}, más su tiempo de cómputo de p_{i_0} en las bandas intermedias, más su tiempo de cómputo para la última banda.

Como el procesador más lento en la figura 3.35 es el último procesador de la última banda, el tiempo de ejecución puede ser calculado como:

$$T_{par} = T_{s_{i_0}} + (NumBandas - 1) * T'_{c_{i_0}} + \frac{M}{B} * (G_{i_0}^{ub} * B * t_{i_0})$$

donde p_{i_0} es el procesador más lento; $T_{s_{i_0}}$ es el tiempo de arranque del procesador más lento; $T'_{c_{i_0}}$ es el tiempo invertido por el procesador más lento en computar y enviar sus datos; $NumBandas$ es el número de bandas en la ejecución, $\frac{M}{B}$ es el número de bloques de cómputo que realizan los procesadores y $G_{i_0}^{ub}$ es el grano que le corresponde al procesador más lento en la última banda. El tiempo total de ejecución es el tiempo de arranque del procesador más lento (p_{i_0}), más su tiempo de cómputo en las bandas intermedias, más su tiempo de cómputo en la última banda. Como se observa en la figura 3.35, en la última banda, el procesador p_{i_0} sólo realiza cómputo, pero no lleva a cabo ningún envío.

La segunda situación se produce cuando el procesador p_{i_0} trabaja en la última banda, pero no es el último procesador (figura 3.36). En esta figura tenemos el mismo *pipeline* que el de la figura 3.35, pero en este caso al último procesador de la última banda lo denominamos p_{i_1}, y en este ejemplo $p_{i_1} = p_2$.

Para la situación representada en la figura 3.36, el tiempo total de ejecución paralelo lo calculamos como:

$$T_{par} = T_{s_{i_0}} + NumBandas * T'_{c_{i_0}} + \sum_{i=i_0+1}^{i_1-1} (G_i * B * t_i + \beta_i + \tau_i Bw) + G_{i_1}^{ub} * B * t_{i_1}$$

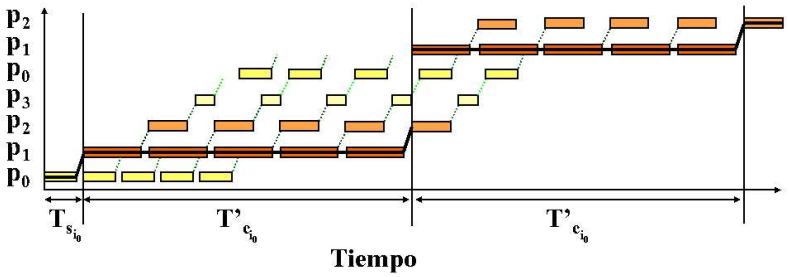

Figura 3.36: Diagrama de tiempo para un *pipeline* con $p = 4$, donde el procesador más lento trabaja en la última banda, pero no es el último procesador. En este caso, el tiempo total de ejecución es obtenido de acuerdo al tiempo de ejecución del procesador más lento (p_{i_0}): el tiempo de arranque del procesador p_{i_0}, más el tiempo de cómputo de p_{i_0} en todas las bandas, más el tiempo de un bloque de cómputo para cada uno de los procesadores posteriores.

Es decir, el tiempo de ejecución paralela viene dado por la suma del tiempo de arranque del procesador más lento ($T_{s_{i_0}}$), más su tiempo de cómputo en todas las bandas ($T'_{c_{i_0}}$), más el tiempo necesario para que cada uno de los procesadores posteriores realice un bloque de cómputo.

La siguiente situación se produce cuando el procesador p_{i_0} no trabaja en la última banda, pero sigue siendo el procesador que determina el tiempo de ejecución (figura 3.37). En la figura se muestra un *pipeline* con 4 procesadores tal que en la última banda solamente trabaja el procesador p_0.

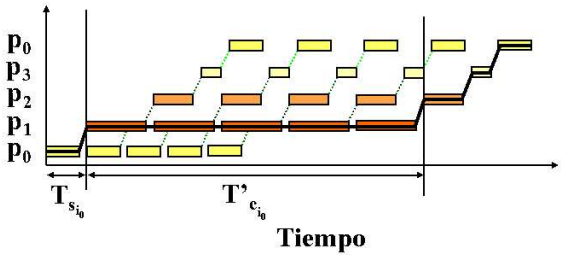

Figura 3.37: Diagrama de tiempo para un *pipeline* con $p = 4$, donde el procesador más lento no trabaja en la última banda, pero es el procesador que determina el tiempo de ejecución. En este caso, el tiempo total se calcula como la suma del tiempo de arranque del procesador p_{i_0}, más el tiempo de cómputo de p_{i_0} en todas las bandas intermedias, más el tiempo de un bloque de cómputo para cada uno de los procesadores posteriores en la penúltima banda, más el tiempo de un bloque de cómputo para todos los procesadores de la última banda.

El tiempo paralelo de ejecución para el *pipeline* representado en la figura 3.37 viene dado por la siguiente ecuación:

$$T_{par} = T_{s_{i_0}} + (NumBandas - 1) * T'_{c_{i_0}} + \sum_{i=i_0+1}^{p-1} (G_i * B * t_i + \beta_i + \tau_i Bw) + \sum_{i=0}^{i_1-1} (G_i * B * t_i + \beta_i + \tau_i Bw) + G_{i_1}^{ub} * B * t_{i_1}$$

Este tiempo se calcula como el tiempo de arranque del procesador más lento (p_{i_0}), más su tiempo de cómputo para todas las bandas intermedias, más el tiempo necesario para que cada uno de los procesadores de la penúltima banda realice un bloque de cómputo, más el tiempo invertido por un bloque de cómputo de cada uno de los procesadores que trabajan en la última banda.

La cuarta situación se produce cuando el procesador más lento de las bandas intermedias (p_{i_0}) es el último procesador de la última banda pero, debido a la reducción de su grano, no es el procesador más

lento si consideramos únicamente la última banda. En el *pipeline* mostrado en la figura 3.38 disponemos de 4 procesadores, donde $p_{i_0} = p_2$ y $p_{i_1} = p_2$. Se observa que el tiempo total de ejecución depende del procesador más lento de la última banda ($p_{i_2} = p_1$).

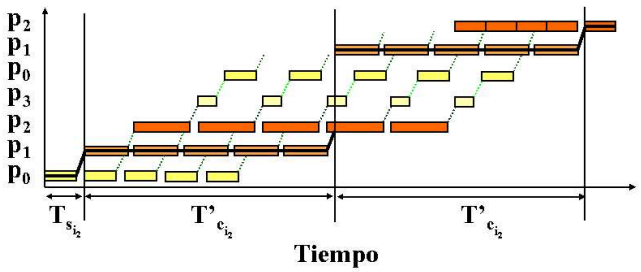

Figura 3.38: Diagrama de tiempo para un *pipeline* con $p = 4$, donde el procesador más lento de las bandas intermedias trabaja en la última banda, pero no es el más lento en ésta. En este caso, el tiempo total de ejecución es obtenido a partir del tiempo de ejecución del procesador más lento de la última banda (p_{i_2}): el tiempo de arranque del procesador p_{i_2}, más el tiempo de cómputo de p_{i_2} en todas las bandas, más el tiempo de un bloque de cómputo para cada uno de los procesadores posteriores.

El tiempo paralelo de ejecución en la figura 3.38 lo calculamos como:

$$T_{par} = T_{s_{i_2}} + T'_{c_{i_2}pb} + (NumBandas - 1) * T'_{c_{i_2}} + \sum_{i=i_2-1}^{i_1-1} (G_i * B * t_i + \beta_i + \tau_i Bw) + G^{ub}_{i_1} * B * t_{i_1}$$

donde $T'_{c_{i_2}pb}$ es el tiempo de cómputo invertido por el procesador p_{i_2} en la primera banda. La diferencia entre la primera y las restantes bandas se debe al efecto del procesador más lento de las bandas intermedias donde produce huecos en la ejecución. En este ejemplo, calculamos el tiempo como el tiempo de arranque del procesador p_{i_2}, más su tiempo de cómputo en todas las bandas, más el tiempo invertido por todos los procesadores posteriores en la última banda para realizar un bloque de cómputo.

Las figuras 3.39 y 3.40 representan las situaciones donde el procesador más lento de las bandas intermedias no trabaja en la última, y el tiempo total de ejecución lo determina el procesador más lento de la última banda (p_{i_2}). En la figura 3.39 se muestra el caso donde p_{i_2} es el último procesador de la última banda.

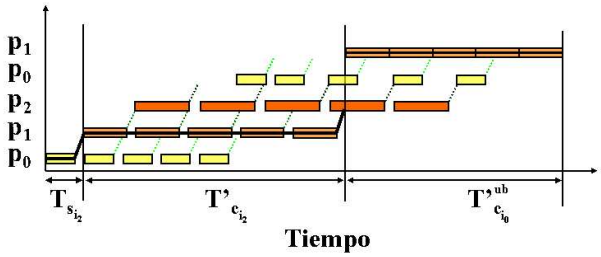

Figura 3.39: Diagrama de tiempo para un *pipeline* con $p = 3$, donde el procesador más lento de las bandas intermedias no trabaja en la última banda, y el procesador más lento de la última banda es el último procesador. El tiempo total de ejecución se obtiene a partir del tiempo de ejecución del procesador más lento de la última banda (p_{i_2}): el tiempo de arranque del procesador p_{i_2}, más el tiempo de cómputo de p_{i_2} en todas las bandas.

El tiempo total de ejecución paralela en la figura 3.39 podemos calculamos como:

$$T_{par} = T_{s_{i_2}} + T'_{c_{i_2}pb} + (NumBandas - 2) * T'_{c_{i_2}} + \frac{M}{B} * (G^{ub}_{i_1} * B * t_{i_1})$$

Esto significa que el tiempo total de ejecución se obtiene como el tiempo de arranque, más el tiempo de cómputo del procesador p_{i_2} para todas las bandas intermedias, más el tiempo de cómputo del procesador para la última banda. Tenemos en cuenta que en esta última banda el procesador p_{i_2} no realiza ningún envío.

La figura 3.40 refleja el caso donde el procesador p_{i_2} no es el último procesador de la última banda.

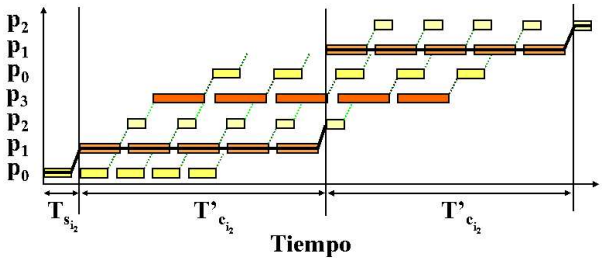

Figura 3.40: Diagrama de tiempo para un *pipeline* con $p = 4$, donde el procesador más lento de las bandas intermedias no trabaja en la última, y el procesador más lento de la última banda no es el último procesador. El tiempo total de ejecución se obtiene a partir del tiempo de ejecución del procesador más lento de la última banda (p_{i_2}): el tiempo de arranque del procesador p_{i_2}, más el tiempo de cómputo de p_{i_2} en todas las bandas más un bloque de cómputo de cada uno de los procesadores posteriores.

El tiempo total de ejecución paralela en la figura 3.40 lo calculamos como:

$$T_{par} = T_{s_{i_2}} + T'_{c_{i_2}pb} + (NumBandas - 1) * T'_{c_{i_2}} + \sum_{i=i_2-1}^{i_1-1} (G_i * B * t_i + \beta_i + \tau_i Bw) + G^{ub}_{i_1} * B * t_{i_1}$$

El tiempo total de ejecución se obtiene como el tiempo de arranque, más el tiempo de cómputo del procesador p_{i_2} para todas las bandas, más un bloque de cómputo para cada uno de los procesadores posteriores en la última banda.

Por último, consideraremos la situación en que el procesador que determina el tiempo total de ejecución es el último procesador de la última banda ($p_{i_1} = p_1$) cuando este procesador no es el más lento de las bandas intermedias ($p_{i_0} = p_2$), ni el más lento de la última banda, $p_{i_2} = p_0$ (figura 3.41).

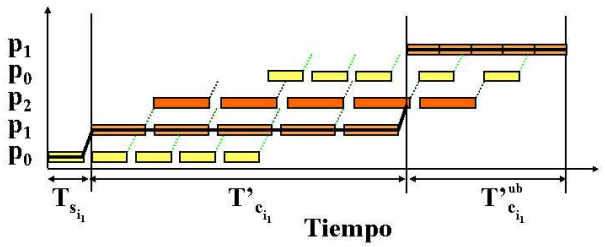

Figura 3.41: Diagrama de tiempo para un *pipeline* con $p = 3$, donde el tiempo total de ejecución se obtiene a partir del tiempo de ejecución del último procesador de la última banda: el tiempo de arranque del procesador p_{i_1}, más el tiempo de cómputo de p_{i_2} en todas las bandas.

El tiempo total de ejecución paralela para el *pipeline* de la figura 3.40 viene dado por la ecuación:

$$T_{par} = T_{s_{i_1}} + T'_{c_{i_1}pb} + (NumBandas - 2) * T'_{c_{i_1}} + \frac{M}{B} * (G^{ub}_{i_1} * B * t_{i_1})$$

Se calcula el tiempo total de ejecución como el tiempo de arranque del procesador p_{i_1}, más el tiempo de cómputo de este procesador en la primera banda (sin los huecos producidos por el procesador más lento), más su tiempo de cómputo en las bandas intermedias, más el tiempo de cómputo en la última banda (sin realizar ningún envío).

Todos los resultados computacionales presentados en este capítulo se han obtenido considerando todas las situaciones especificadas en esta sección.

3.8. Validación del modelo analítico

Para validar nuestro modelo en cada una de las situaciones estudiadas en la sección 3.7 hemos considerado que el cómputo entre las comunicaciones viene dado por un bucle sintético constante. En la figura 3.42 se muestra el código ejecutado por las etapas intermedias del *pipeline*. Todos los resultados que aparecen en esta sección y en las secciones 3.6 y 3.10 se han obtenido ejecutando la versión *MPI* de la librería *llpW* (sección 3.5). En nuestras pruebas, los valores medidos como constantes de cómputo para los procesadores rápidos, intermedios y lentos son $t_R = 0.0072$, $t_I = 0.016$ y $t_L = 0.026$ respectivamente. Estos valores han sido medidos ejecutando el bucle de la línea 12 de la figura 3.42 en cada tipo de máquina. Aunque la herramienta permite asignación de tamaños de *buffer* variables, por motivos de simplicidad, en todos las ejecuciones hemos considerado tamaños de *buffer* iguales para todos los procesadores, independientemente del tipo de la máquina.

```
1  void solver_pipe(void)  {
2     LOCAL_VAR
3        int i;
4        double j;
5        int x;
6     BEGIN;
7
8              . . .
9
10       for (i = 0; i <= M; i++) {
11           IN(&x);
12           for (j = 0; j <= 2000000; j ++);
13           OUT(&i, 1, sizeof(unsigned));
14       }
15            . . .
16
17    END;
18 }
```

Figura 3.42: Función *solver_pipe()* que contiene un bucle sintético constante utilizado para las pruebas de validación y predicción del modelo.

Del mismo modo que hicimos en el capítulo 2, primero vamos a caracterizar nuestra plataforma heterogénea, estableciendo algunas métricas de rendimiento. La tabla 3.17 muestra las relaciones obtenidas entre las constantes de cómputo de las máquinas. Estas constantes son independientes del tamaño del problema, por lo que las relaciones entre estos valores son también constantes. En la tabla, la columna L / R es la relación entre las máquinas lentas y rápidas, la columna I / R es la relación entre máquinas intermedias y rápidas y la columna L / I entre las máquinas lentas e intermedias.

A continuación calculamos la heterogeneidad de nuestro *cluster* (sección 1.4): $H = \frac{\sum_{i=0}^{p-1}(1-CP_i^E)}{p}$. Hay que obtener previamente los valores escalados para las potencias computacionales de los tres tipos

Tabla 3.17: Relación entre las constantes de cómputo para los tres tipos de máquinas que componen el *cluster*: lentas / rápidas (L / R), intermedias / rápidas (I / R) y lentas / intermedias (L / I).

L / R	I / R	L / I
3.61	2.22	1.62

de máquinas, asignándole a los procesadores rápidos el valor más alto ($CP_R^E = 1$). La tabla 3.18 muestra los valores de las potencias computacionales escaladas para las máquinas rápidas (columna CP_R^E), intermedias (columna CP_I^E) y lentas (columna CP_L^E); así como el valor de la heterogeneidad del sistema con los 14 procesadores. En la tabla se observa que la heterogeneidad del sistema alcanza casi el 50 %.

Tabla 3.18: Potencia computacional escalada para los tres tipos de máquinas del cluster (rápidas, intermedias y lentas) y valor de la heterogeneidad del sistema.

CP_R^E	CP_I^E	CP_L^E	Heterogeneidad
1	0.45	0.28	0.47

3.8.1. El número de etapas es igual al número de procesadores

Para resolver este problema sobre p procesadores, asignamos un proceso virtual a cada procesador e invocamos a la macro *PIPE* pasándole como argumentos la función de la figura 3.42, el número de etapas ($n = p$) y el array con los valores de los granos para cada tipo de máquinas ($G_i = 1$).

La tabla 3.19 presenta los tiempos reales de ejecución en segundos (columna etiquetada *Tiempo Real*), el tiempo estimado por el modelo (columna *Modelo*) y los errores relativos obtenidos para las diferentes configuraciones de procesadores rápidos (R), intermedios (I) y lentos (L). La columna *Configuración* muestra el tipo de los procesadores utilizados en la ejecución y su posición en el *pipeline*. La experiencia la hemos repetido para tamaños del bucle externo del problema $M = 50$ y 500, con $p = 4, 6, 8$ y 14 procesadores. Para cada bloque de resultados indicamos el número de procesadores y el valor de M como cabeceras de la tabla.

Observamos, en la tabla 3.19, que el tiempo real de ejecución es prácticamente independiente del orden de los procesadores en la configuración empleada, la diferencia de tiempo entre las distintas configuraciones es mínima. Esta circunstancia también se contempla en los resultados que se obtienen con el modelo analítico. Como podíamos esperar, el error obtenido decrece para valores altos de M. Para $M = 500$ el error no supera en ningún caso el 6 %. Para el problema más pequeño ($M = 50$), la mitad de las ejecuciones presentan un error menor del 8 % y únicamente en 4 de las 48 ejecuciones supera el 10 % de error. Aunque, debido a la escalabilidad de los problemas, es usual observar un incremento de este error con el número de procesadores, las diferencias que se encuentran en la tabla, al modificar el número de procesadores, no son muy importantes. Un segundo hecho notable es la relativa independencia del modelo y de los tiempos de ejecución de una combinación particular de procesadores rápidos, intermedios y lentos. Este comportamiento es debido a que las permutaciones de los procesadores sólo influyen en el valor de Ts.

3.8.2. El número de etapas es superior al número de procesadores: *mapping* cíclico puro

Como segundo paso de la validación, consideramos una distribución cíclica pura ($G_i = 1$) para resolver un problema donde $p < n$; es decir, un *pipeline* donde el número de etapas es superior al número de

Tabla 3.19: Validación del modelo: el número de procesadores es igual al número de etapas del problema.

p = 4 - n = 4						
	M = 50			M = 500		
Configuración	Tiempo Real	Modelo	Error	Tiempo Real	Modelo	Error
I R R L	1.42	1.33	6.43 %	13.62	13.03	4.33 %
L R I R	1.42	1.34	5.63 %	13.63	13.14	3.60 %
L I R R	1.43	1.34	6.29 %	13.64	13.15	3.59 %
I R L R	1.43	1.34	6.29 %	13.78	13.14	4.95 %
L R R I	1.41	1.34	4.96 %	13.54	13.14	2.95 %
I L R R	1.44	1.34	6.94 %	13.81	13.14	4.85 %
p = 6 - n = 6						
	M = 50			M = 500		
Configuración	Tiempo Real	Modelo	Error	Tiempo Real	Modelo	Error
I I R R L L	1.5	1.39	7.33 %	13.75	13.23	3.78 %
I R L I R L	1.52	1.38	9.21 %	13.89	13.19	5.04 %
L L I I R R	1.51	1.39	7.95 %	13.76	13.23	3.85 %
L I I R R L	1.52	1.38	9.21 %	13.69	13.19	3.65 %
I R I R L L	1.49	1.39	6.71 %	13.85	13.23	4.48 %
I R R L I L	1.53	1.38	9.80 %	13.85	13.19	4.77 %
p = 8 - n = 8						
	M = 50			M = 500		
Configuración	Tiempo Real	Modelo	Error	Tiempo Real	Modelo	Error
I I I R R R L L	1.53	1.41	7.84 %	13.76	13.25	3.71 %
L L I I I R R R	1.53	1.41	7.84 %	13.78	13.25	3.85 %
I R L I R L I R	1.55	1.41	9.03 %	13.92	13.22	5.03 %
I I R R L L I R	1.55	1.41	9.03 %	13.95	13.25	5.02 %
I I R R L I R L	1.56	1.41	9.62 %	13.92	13.21	5.10 %
I R R R L L I I	1.52	1.41	7.24 %	13.94	13.25	5.95 %
p = 14 - n = 14						
	M = 50			M = 500		
Configuración	Tiempo Real	Modelo	Error	Tiempo Real	Modelo	Error
R R R R I I I I L L L L L L	1.70	1.54	9.41 %	14.11	13.38	5.17 %
R I L R I L R I L R I L L L	1.76	1.54	12.50 %	14.18	13.38	5.64 %
R R I I L L R R I I L L L L	1.73	1.54	10.98 %	14.12	13.38	5.24 %
I I R R L L I I R R L L L L	1.72	1.54	10.47 %	14.10	13.38	5.11 %
L L L I I I R R R L I R L L	1.76	1.54	12.50 %	14.09	13.38	5.04 %
L L L L L L R R R R I I I I	1.71	1.54	9.94 %	14.07	13.38	4.90 %

procesadores y la ejecución se realiza cíclicamente en bandas. En esta sección presentamos los resultados obtenidos con esta distribución, considerando problemas donde se producen 5 y 10 bandas completas, para $p = 4, 6, 8$ y 14 procesadores, con un tamaño de bucle de $M = 50$ y 500 (tablas 3.20 y 3.21 respectivamente). El número de etapas del *pipeline* se calcula como $p * NumBandas$ en cada caso. Las tablas presentan el mismo esquema que la tabla 3.19: para cada configuración y cada problema incluimos el tiempo real en segundos, el tiempo estimado por el modelo y el error cometido. Separamos los datos en bloques según el número de procesos utilizados para resolver cada problema. Hemos realizado las ejecuciones para los dos tamaños de problemas resueltos en la sección anterior: $M = 50$ en la tabla 3.20 y $M = 500$ en la tabla 3.21.

Como se comprueba en los resultados presentados en las tablas 3.20 y 3.21, los tiempos calculados por nuestro modelo analítico son muy precisos. En la tabla 3.20 observamos que en el problema de tamaño $M = 50$, sólo 5 ejecuciones presentan un error superior al 7 %; mientras que para el problema de tamaño $M = 500$, únicamente en 4 de las 48 ejecuciones realizadas se obtiene un error superior al 5 %. Esta cantidad supone solamente un 8.33 % del total de las ejecuciones. Además, se observa que existe un comportamiento similar en todos los casos, presentando un porcentaje constante de error, independientemente del número de procesadores y del número de bandas. El factor que sigue influyendo en los resultados es el tamaño del problema: se comprueba que con el problema de mayor tamaño los porcentajes de error obtenidos son inferiores.

Por último, comparamos estos resultados con los de la tabla 3.19: la introducción de bandas en el *pipeline* no incrementa los porcentajes de error; al contrario, los errores presentados en las tablas 3.20 y 3.21 se reducen, obteniéndose un error medio de las ejecuciones para $M = 500$ inferior al 5 %.

3.8.3. El número de etapas es superior al número de procesadores: *mapping* cíclico por bloques

Para validar nuestro modelo en el caso general, realizamos una experiencia computacional similar a la desarrollada en las subsecciones previas y hemos obtenido una precisión comparable en las predicciones. Desarrollamos nuevos experimentos, variando ahora el tamaño del *buffer* y los valores de grano para todas las máquinas La tabla 3.22 muestra los tiempos de ejecución del algoritmo paralelo y el tiempo estimado por el modelo para el problema sintético con $M = 100$, $n = 20$ y $p = 4$ cuando se ejecuta el problema sobre el *cluster* compuesto únicamente por las máquinas intermedias y lentas. La configuración utilizada en este caso ha sido $I\ L\ I\ L$, donde I representa una máquina intermedia y L una máquina lenta. Hemos ejecutado el programa que utiliza la librería $llpW$ variando el tamaño del *buffer* y el tamaño del grano para los procesadores intermedios (G_I) y lentos (G_L) a lo largo del rango de valores posibles: $B = 1 \ldots 12$, $G_I = 1 \ldots 9$ y $G_L = 1 \ldots 9$. La única restricción que hemos considerado ha sido exigir que el grano asignado a los procesadores lentos no pueda ser superior al asignado a los procesadores intermedios. La tabla presenta solamente un subconjunto de todas las ejecuciones realizadas.

En la tabla 3.22 también aparece como información una columna con el número de bandas que se generan utilizando la combinación correspondiente de granos. El número de bandas no tiene que ser un valor exacto: si lo es, se trata de ejecuciones donde las bandas están completas, si no es así, estamos trabajando con *pipelines* donde algún procesador no trabaja en la última banda. Todas estas situaciones se tuvieron en cuenta en la descripción del modelo (sección 3.7.3). Se comprueba que el error relativo que ha cometido nuestro modelo es muy reducido. En 26 pruebas, de un total de 45 ejecuciones que aparecen en la tabla, el error está por debajo del 3 %. Esta cantidad supone un 57.78 % de las ejecuciones; y en el 88.89 % del total de las ejecuciones, el error no supera el 5 %. Además, en ningún caso obtenemos un error superior al 6 %.

Esta experiencia no permite únicamente validar nuestro modelo analítico, sino también comprobar experimentalmente qué combinación de parámetros es la mejor solución. El mínimo tiempo de ejecución se alcanza con la combinación $B = 1$, $G_I = 7$ y $G_L = 3$, con 56.50 segundos de ejecución. Esta combinación es también la que obtiene el mínimo de los tiempos calculados por el modelo analítico (línea en negrita

Tabla 3.20: Validación del modelo: *mapping* cíclico puro con 5 y 10 bandas para el tamaño del problema $M = 50$.

p = 4						
	n = 20 - M = 50			n = 40 - M = 50		
Configuración	Tiempo Real	Modelo	Error	Tiempo Real	Modelo	Error
I R R L	6.90	6.59	4.49 %	13.75	13.14	4.44 %
L R I R	6.92	6.59	4.77 %	13.82	13.14	4.92 %
L I R R	6.92	6.59	4.77 %	13.78	13.15	4.57 %
I R L R	6.93	6.59	4.91 %	13.78	13.14	4.64 %
L R R I	6.87	6.59	4.08 %	13.85	13.14	5.13 %
I L R R	6.93	6.59	4.91 %	13.80	13.14	4.78 %
p = 6						
	n = 30 - M = 50			n = 60 - M = 50		
Configuración	Tiempo Real	Modelo	Error	Tiempo Real	Modelo	Error
I I R R L L	6.95	6.65	4.32 %	13.75	13.23	3.78 %
I R L I R L	7.01	6.63	5.42 %	13.89	13.19	5.04 %
L L I I R R	7.00	6.65	5.00 %	13.83	13.23	4.34 %
L I I R R L	7.15	6.63	7.27 %	14.54	13.21	9.15 %
I R I R L L	7.00	6.65	5.00 %	13.87	13.23	4.61 %
I R R L I L	7.01	6.63	5.42 %	14.38	13.19	8.28 %
p = 8						
	n = 40 - M = 50			n = 80 - M = 50		
Configuración	Tiempo Real	Modelo	Error	Tiempo Real	Modelo	Error
I I I R R R L L	6.95	6.67	4.03 %	13.81	13.25	4.06 %
L L I I I R R R	7.03	6.67	5.12 %	13.89	13.25	4.61 %
I R L I R L I R	7.04	6.66	5.40 %	13.96	13.22	5.30 %
I I R R L L I R	7.32	6.67	8.88 %	13.94	13.25	4.95 %
I I R R L I R L	7.04	6.66	5.40 %	13.96	13.21	5.37 %
I R R R L L I I	7.27	6.67	8.25 %	13.91	13.25	4.74 %
p = 14						
	n = 70 - M = 50			n = 140 - M = 50		
Configuración	Tiempo Real	Modelo	Error	Tiempo Real	Modelo	Error
R R R R I I I I L L L L L L	7.24	6.80	6.08 %	14.07	13.38	4.90 %
R I L R I L R I L R I L L L	7.29	6.80	6.72 %	14.16	13.38	5.51 %
R R I I L L R R I I L L L L	7.26	6.80	6.34 %	14.11	13.38	5.17 %
I I R R L L I I R R L L L L	7.26	6.80	6.34 %	14.09	13.38	5.04 %
L L L I I I R R R L I R L L	7.30	6.80	6.85 %	14.80	13.38	9.59 %
L L L L L L R R R R I I I I	7.24	6.80	6.08 %	14.13	13.38	5.31 %

Tabla 3.21: Validación del modelo: *mapping* cíclico puro con 5 y 10 bandas para el tamaño del problema $M = 500$.

p = 4						
	n = 20 - M = 500			n = 40 - M = 500		
Configuración	Tiempo Real	Modelo	Error	Tiempo Real	Modelo	Error
I R R L	68.30	65.50	4.10 %	137.75	131.10	4.83 %
L R I R	68.46	65.58	4.21 %	137.80	131.13	4.84 %
L I R R	68.51	65.62	4.22 %	137.19	131.22	4.35 %
I R L R	68.67	65.58	4.50 %	137.16	131.13	4.40 %
L R R I	68.89	65.58	4.80 %	138.85	131.13	6.24 %
I L R R	68.72	65.58	4.57 %	137.35	131.13	4.53 %
p = 6						
	n = 30 - M = 500			n = 60 - M = 500		
Configuración	Tiempo Real	Modelo	Error	Tiempo Real	Modelo	Error
I I R R L L	68.20	65.84	3.46 %	136.49	131.61	3.58 %
I R L I R L	69.05	65.67	4.90 %	137.33	131.26	4.42 %
L L I I R R	68.63	65.84	4.07 %	137.24	131.61	4.10 %
L I I R R L	72.48	65.69	9.37 %	137.25	131.45	4.23 %
I R I R L L	68.69	65.84	4.15 %	137.21	131.61	4.08 %
I R R L I L	68.81	65.67	4.56 %	137.91	131.26	4.82 %
p = 8						
	n = 40 - M = 500			n = 80 - M = 500		
Configuración	Tiempo Real	Modelo	Error	Tiempo Real	Modelo	Error
I I I R R R L L	68.20	65.86	3.43 %	136.30	131.63	3.48 %
L L I I I R R R	68.72	65.86	4.16 %	137.43	131.63	4.22 %
I R L I R L I R	68.85	65.69	4.59 %	137.55	131.28	4.56 %
I I R R L L I R	68.80	65.86	4.27 %	137.39	131.63	4.19 %
I I R R L I R L	68.80	65.69	4.52 %	137.62	131.28	4.61 %
I R R R L L I I	68.89	65.86	4.40 %	137.57	131.63	4.32 %
p = 14						
	n = 70 - M = 500			n = 140 - M = 500		
Configuración	Tiempo Real	Modelo	Error	Tiempo Real	Modelo	Error
R R R R I I I I L L L L L L	69.09	65.99	4.45 %	138.03	131.76	4.54 %
R I L R I L R I L R I L L L	69.16	65.99	4.58 %	137.94	131.76	4.48 %
R R I I L L R R I I L L L L	69.16	65.99	4.58 %	137.69	131.76	4.31 %
I I R R L L I I R R L L L L	69.21	65.99	4.65 %	137.94	131.76	4.48 %
L L L I I I R R R L I R L L	72.86	65.99	9.43 %	146.90	131.76	10.31 %
L L L L L L R R R R I I I I	69.05	65.99	4.43 %	138.19	131.76	4.65 %

Tabla 3.22: Validación del modelo: *mapping* cíclico por bloques.

p = 4 - M = 100 - n = 20 - Configuration: I L I L						
Buffer	G_I	G_S	Núm. Bandas	Tiempo Real	Modelo	Error
1	1	1	5	77.66	76.39	1.64 %
1	2	1	3.33	60.92	60.44	0.80 %
1	3	3	1.67	92.85	91.58	1.37 %
1	4	2	1.67	62.88	61.25	2.59 %
1	5	5	1	78.68	77.53	1.46 %
1	6	4	1	63,19	62.33	1.36 %
1	**7**	**3**	**1**	**56.50**	**53.95**	**4.51 %**
1	8	2	1	64.87	61.23	5.62 %
1	9	1	1	72.84	68.50	5.96 %
2	1	1	5	77.88	76.65	1.57 %
2	2	1	3.33	63.09	60.42	4.24 %
2	3	3	1.67	93.06	91.95	1.19 %
2	4	2	1.67	62.55	61.70	1.35 %
2	5	5	1	80.22	79.03	1.48 %
2	6	4	1	64.69	63.83	1.34 %
2	7	3	1	58.85	55.38	5.88 %
2	8	2	1	65.76	62.43	5.07 %
2	9	1	1	73.03	69.47	4.88 %
5	1	1	5	78.57	77.53	1.33 %
5	2	1	3.33	61.73	60.41	2.15 %
5	3	3	1,67	94.77	93.08	1.79 %
5	4	2	1.67	65.63	63.05	3.93 %
5	5	5	1	84.80	83.55	1.47 %
5	6	4	1	69.43	68.34	1.56 %
5	7	3	1	63.19	59.69	5.55 %
5	8	2	1	69.39	66.04	4.82 %
5	9	1	1	75.68	72.40	4.33 %
7	1	1	5	79.23	78.13	1.39 %
7	2	1	3.33	62.35	59.81	4.08 %
7	3	3	1.67	94.43	93.83	0.64 %
7	4	2	1.67	64.33	63.95	0.60 %
7	5	5	1	87.69	86.57	1.28 %
7	6	4	1	72.43	71.36	1.48 %
7	7	3	1	65.50	62.56	4.49 %
7	8	2	1	71.17	68.46	3.81 %
7	9	1	1	77.52	74.36	4.08 %
11	1	1	5	80.30	79.33	1.21 %
11	2	1	3.33	62.46	60.10	3.77 %
11	3	3	1.67	96.57	95.32	1.29 %
11	4	2	1.67	67.55	65.75	2.66 %
11	5	5	1	94.26	92.61	1.75 %
11	6	4	1	79.04	77.39	2.09 %
11	7	3	1	70.73	68.31	3.42 %
11	8	2	1	76.46	73.29	4.15 %
11	9	1	1	81.38	78.27	3.82 %

en la tabla 3.22). Se puede observar en la tabla que la combinación $B = 1$, $G_I = 3$ y $G_L = 3$, requiere un tiempo de ejecución de 92.85 segundos. Esto significa un rendimiento relativo peor en un 65 % y da una idea de la importancia de realizar una buena selección de parámetros.

Con estos resultados podemos considerar validado nuestro modelo para todas las situaciones posibles en sistemas heterogéneos.

3.9. Métodos de predicción de los parámetros óptimos

La asignación óptima de trabajo a procesadores es un factor crucial que determina el rendimiento del algoritmo paralelo, como hemos ilustrado en los resultados de la sección 3.8. En esta sección analizaremos el problema de encontrar el *mapping* óptimo de un algoritmo *pipeline* sobre un anillo de procesadores. Consideramos dos variables: el número de procesos virtuales sobre cada procesador físico y el tamaño de los paquetes que se envían. El problema del *mapping* se define como el problema de encontrar la combinación óptima de los parámetros que minimiza el tiempo total de ejecución.

Para el caso homogéneo, la herramienta *llp* permite predecir los valores óptimos de los parámetros de forma automática, seleccionando también el número óptimo de procesadores físicos [132, 133] (sección 3.4).

Para los sistemas heterogéneos aún no es posible obtener esta distribución utilizando *llpW* de forma automática, pero las capacidades predictivas del modelo analítico presentado en la sección 3.7, pueden ser utilizadas para predecir los valores de los parámetros en una ejecución óptima; es decir, establecer los valores G_i y B para que la función de complejidad analítica, T_{par}, alcance su valor mínimo. El problema de encontrar el *mapping* óptimo puede ser resuelto por varios métodos. Una primera aproximación, considerada en la literatura, es la minimización analítica de T_{par} [11]. La minimización analítica impone un importante desarrollo teórico matemático, que en el caso de nuestro modelo analítico parece una tarea inabordable debido al número de variables involucradas. La segunda aproximación a ser considerada es la minimización numérica de la función de complejidad [84]. Esta sección describe dos estrategias numéricas que hemos considerado para abordar el problema de predecir los valores óptimos de los parámetros. La primera estrategia consiste en un método exacto que enumera cada solución válida del espacio de soluciones. La segunda es utilizar la heurística de recocido simulado (*simulated annealing*) que ha demostrado ser muy eficiente.

3.9.1. Método exacto enumerativo

En [84] hemos presentado un método exacto para resolver el problema del *mapping* óptimo en un sistema homogéneo. Es un método tipo *Golden Section* [147], que aprovecha las propiedades de convexidad de las funciones analíticas. La figura 3.43 muestra el comportamiento de la función cuando fijamos uno de los parámetros y modificamos el valor del otro. La idea básica, para encontrar el mínimo, es ir moviéndonos a lo largo de la dirección descendente mientras la función analítica decrece. El algoritmo finaliza con la solución óptima después de unas pocas evaluaciones de la función analítica. Esta estrategia ha probado ser muy eficiente y precisa.

El comportamiento de la función analítica en sistemas heterogéneos es diferente. La figura 3.44 muestra la forma de esta función en las ejecuciones experimentales. En la figura 3.44-Izquierda presentamos el comportamiento de la función T_{par} cuando fijamos el número de procesadores ($p = 4$) y el valor del grano para los procesadores lentos ($G_L = 1$), para distintos valores de *buffer* y variamos el grano asignado a los procesadores intermedios. En la figura 3.44-Derecha se muestra su comportamiento cuando fijamos el número de procesadores y la combinación de los granos para todas las máquinas, y variamos el tamaño del *buffer*. Ahora no observamos la misma forma convexa del caso homogéneo. Este hecho nos fuerza a una enumeración completa del espacio de soluciones válidas como un primer método exacto. Si la máquina heterogénea comprende sólo 2 tipos de procesadores, este método exacto presenta una

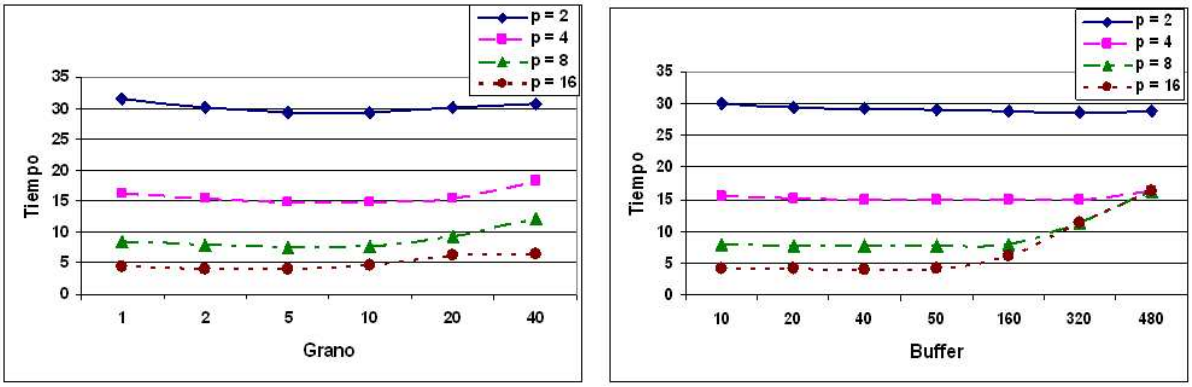

Figura 3.43: Comportamiento de los sistemas homogéneos. Izquierda: Fijamos el valor del *buffer* y variamos el valor del grano para diferente número de procesadores. Derecha: Fijamos el valor del grano y movemos el *buffer* para distinto número de procesadores.

complejidad de tiempo de orden $O((M/p) * (n^2/p_R^2))$ donde n es el número de etapas del *pipeline*, M el número de estados de cada etapa, p es el número de procesadores y p_R es el número de procesadores rápidos. Desafortunadamente, este orden de complejidad crece cuando se incrementa el número de tipos de procesadores.

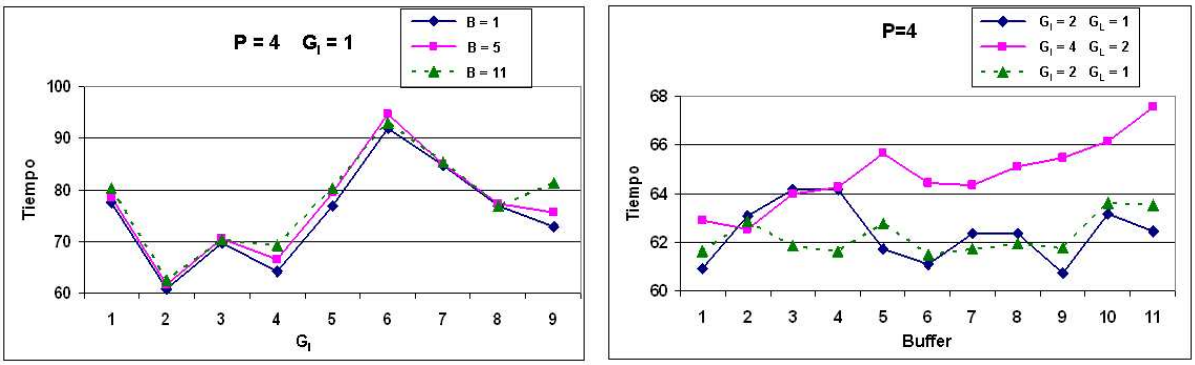

Figura 3.44: Comportamiento de los sistemas heterogéneos. Izquierda: Fijamos el valor del grano de los procesadores lentos (G_L) y movemos el grano de los procesadores intermedios (G_I) para diferentes valores del *buffer*. Derecha: Fijamos una combinación de G_I y G_L y movemos el valor del *buffer*.

3.9.2. Un método heurístico

Consideremos ahora la alternativa de una aproximación heurística para encontrar el *mapping* óptimo. Es un procedimiento rápido que proporciona los parámetros para alcanzar una solución próxima a la óptima. El algoritmo básico que proponemos aparece en la figura 3.45.

El algoritmo consiste en una ejecución multiarranque de una heurística de recocido simulado, ejecutando desde puntos de inicio aleatorios, para encontrar la distribución óptima (*buffer*, G_R, G_I, G_L). Dada una distribución, el procedimiento encuentra una nueva solución en su entorno. La nueva distribución es generada siguiendo un método aleatorio: seleccionamos al azar un parámetro a modificar (*buffer*, G_R, G_I o G_L), y decidimos si el valor de este parámetro vamos a incrementarlo o reducirlo. El nuevo *mapping* es

```
 1 BestDistribution = InitializeDistribution ();
 2 BestTime = T(BestDistribution);
 3 repeat five times {
 4     Temperature = InitialTemperature;
 5     while (Temperature > FinalTemperature) {
 6         for (i = 1; i <= NumberOfIterations; i++)  {
 7             NewDistribution = GenerateDistribution (NewDistribution);
 8             NewTime = T (NewDistribution);
 9             if (NewTime < BestTime) {
10                 BestTime = NewTime;
11                 BestDistribution = NewDistribution;
12              }
13              else {
14                  // Aceptar con probabilidad exp(-(NewTime - BestTime)/Temperature)
15              }
16         }
17         ReduceTemperature;

18     }
19     NewDistribution = GenerateRandomDistribution();
20     NewTime = T (NewDistribution);
21     if (NewTime < BestTime) {
22         BestTime = NewTime;
23         BestDistribution = NewDistribution;
24     }
25 }
```

Figura 3.45: Código del método heurístico para calcular la distribución óptima de B, G_R, G_I y G_S.

aceptado si se mejora el resultado hasta el momento. Debido a la presencia de muchos mínimos locales en la función objetivo, algunas distribuciones que no mejoran el resultado actual son aceptadas con una cierta probabilidad.

3.10. Predicción de los parámetros óptimos

Esta sección presenta los resultados computacionales que nos ayudan a encontrar el *mapping* óptimo para problemas específicos. Las distribuciones proporcionadas por el método exacto son comparadas con las soluciones dadas por la técnica heurística. La tabla 3.23 muestra los resultados para diferentes tamaños de problemas con $p = 4, 6, 8, 14$. El número de etapas (n) y el tamaño del problema (M), junto con la configuración de procesadores utilizada, se muestra como cabecera por encima de los resultados obtenidos. Las columnas etiquetadas como *Buffer*, G_R, G_I y G_L muestran la distribución de trabajos. La columna *Tiempo Real* contiene el tiempo real de ejecución obtenido al ejecutar el programa paralelo con la distribución correspondiente. La columna *Modelo* presenta el tiempo que predice el modelo analítico. La columna *Error* es el error relativo cometido en la predicción realizada por el modelo al compararla con el tiempo real de ejecución. La columna *Dif.* es la diferencia relativa entre el tiempo real de ejecución con la distribución obtenida al ejecutar el método exacto y el tiempo real de la distribución proporcionada por el método heurístico. La última columna (*Tiempo Ejecución*) muestra el tiempo invertido en la ejecución para el método exacto y la heurística para obtener las distribuciones óptimas.

Para todos los casos considerados, el método heurístico obtiene la distribución óptima o soluciones muy próximas a la óptima. Mencionar que en un caso ($M = 1000$ y $n = 200$ con $p = 14$) el método

Tabla 3.23: Predicción de los parámetros óptimos: comparación de los resultados del método exacto frente a los resultados obtenidos con la heurística.

p = 4									
M = 500 - n = 50 - Configuración: L I R R									
Método	*Buffer*	G_F	G_I	G_S	Tiempo Real	Modelo	Error	Dif.	Tiempo Ejecución
Exacto	1	19	8	4	68.61	68.77	-0.23 %		0.58
Heurística	5	18	9	4	73.80	73.96	-0.22 %	7.56 %	0.34
M = 1000 - n = 200 - Configuración: I L R R									
Exacto	1	37	16	10	539.89	533.71	1.14 %		68.40
Heurística	33	15	6	4	574.67	550.18	4.26 %	6.44 %	0.52
M = 2000 - n = 100 - Configuración: I R R L									
Exacto	1	37	16	10	543.55	534.01	1.76 %		17.45
Heurística	12	37	16	10	567.84	542.23	4.46 %	4.47 %	0.51
p = 6									
M = 500 - n = 50 - Configuración: L I I R R L									
Método	*Buffer*	G_F	G_I	G_S	Tiempo Real	Modelo	Error	Dif.	Tiempo Ejecución
Exacto	1	15	6	4	54.45	54.62	-0.31 %		0.42
Heurística	3	15	6	4	56.87	55.56	2.30 %	4.44 %	0.39
M = 1000 - n = 200 - Configuración: I L R I L R									
Exacto	1	29	13	8	435.14	418.90	3.73 %		47.98
Heurística	30	20	7	5	453.30	455.00	-0.38 %	4.17 %	0.48
M = 2000 - n = 100 - Configuration: I R I R L L									
Exacto	1	29	13	8	432.16	419.08	3.03 %		12.48
Heurística	4	10	4	4	434.30	433.15	0.26 %	0.50 %	0.51
p = 8									
M = 500 - n = 50 - Configuración: L L I I I R R R									
Método	*Buffer*	G_F	G_I	G_S	Tiempo Real	Modelo	Error	Dif.	Tiempo Ejecución
Exacto	1	11	4	1	40.00	40.18	-0.45 %		0.29
Heurística	4	11	4	1	44.52	41.51	7.76 %	11.3 %	0.40
M = 1000 - n = 200 - Configuración: I I L L R R R I									
Exacto	1	21	9	5	304.51	303.66	0.28 %		34.72
Heurística	2	21	8	6	358.08	306.49	14.40 %	17.59 %	0.53
M = 2000 - n = 100 - Configuración: I I R R L I R L									
Exacto	1	21	9	5	302.21	303.83	-0.54 %		8.98
Heurística	5	7	3	2	325.69	313.76	3.66 %	7.79 %	0.53
p = 14									
M = 500 - n = 50 - Configuración: I I R R L L I I R R L L L L									
Método	*Buffer*	G_F	G_I	G_S	Tiempo Real	Modelo	Error	Dif.	Tiempo Ejecución
Exacto	1	4	1	1	29.36	29.28	0.27 %		0.25
Heurística	5	4	1	1	30.16	30.12	0.13 %	2.72 %	0.56
M = 1000 - n = 200 - Configuración: I R L I R L I R L I R L L L									
Exacto	2	28	11	7	228.21	208.07	8.83 %		30.35
Heurística	3	7	3	2	217.27	209.77	3.45 %	-4.79 %	0.61
M = 2000 - n = 100 - Configuración: I I I I R R R R L L L L L L									
Exacto	2	7	3	2	217.72	209.72	3.67 %		7.77
Heurística	1	7	3	2	218.02	209.78	3.78 %	0.14 %	0.61

heurístico ofrece un mejor resultado que el método exacto, debido a que el error cometido por el modelo con la distribución del método exacto alcanza casi el 9 % de error, mientras que con la heurística el error no llega al 4 %. Podemos ver también que el tiempo de ejecución para el algoritmo exacto depende del tamaño del problema. El problema más grande es el que consume mayor cantidad de tiempo para obtener el resultado. Este tiempo de ejecución se incrementa también con el número de tipos de procesadores incluidos. Sin embargo, como esperábamos, el tiempo de ejecución del método heurístico permanece constante. Aunque para problemas pequeños, el método exacto invierte muy poco tiempo, para problemas grandes es recomendable usar la técnica heurística.

3.11. Conclusiones

Proponemos la herramienta *llpW* y un modelo analítico que cubre una amplia clase de algoritmos *pipeline* y considera la heterogeneidad debida tanto al cómputo como a las comunicaciones. La herramienta permite una implementación sencilla y eficiente de *pipelines* sobre una plataforma heterogénea. Hemos comprobado, ejecutando algoritmos de programación dinámica, que con herramientas adaptadas a entornos heterogéneos se consigue incrementar el rendimiento del sistema en la resolución de un problema. También validamos el modelo analítico desarrollado sobre un *cluster* heterogéneo, demostrándose la precisión de nuestras predicciones.

Hemos desarrollado un modelo analítico que predice el efecto del grano y del *buffer* en los algoritmos *pipeline*. El modelo analítico permite una estimación sencilla de estos parámetros a partir de una simple aproximación numérica.

El modelo presentado constituye la base para resolver el problema del *mapping* óptimo en los algoritmos *pipelines* en sistemas heterogéneos. El *mapping* óptimo es calculado usando una aproximación algorítmica, que considera dos estrategias diferentes: un método heurístico que permite obtener, de forma muy eficiente, los parámetros para una ejecución óptima y un algoritmo exacto que enumera todas las soluciones posibles.

Como trabajo futuro, nos proponemos extender el modelo para un esquema *pipeline* más general y seguiremos una aproximación similar a la presentada en [84], sobre sistemas homogéneos para proporcionar un *mapping* óptimo automático para *pipelines* en sistemas heterogéneos. Este modelo ya se ha incluido en la herramienta para el caso homogéneo (*llp*): cuando se ejecuta la primera banda, la herramienta estima los parámetros definiendo la función $T(G, B)$ y calcula los valores óptimos de G y B. La sobrecarga introducida es despreciable, debido a que sólo se requieren unas pocas evaluaciones de la función objetivo. Después de esta primera banda, la ejecución del algoritmo paralelo continúa con las siguientes bandas haciendo uso de los parámetros óptimos de grano y *buffer* calculados ([82, 83]). Al añadir el número de procesadores como parámetro en esta función ($T(G, B, p)$), una minimización analítica de esta nueva función es una tarea impracticable. Sin embargo, una aproximación numérica propociona el mínimo con sólo algunas evaluaciones de la función objetivo.

Del mismo modo que en sistemas homogéneos, el bajo error observado y la simplicidad de nuestra propuesta, permite hacer uso de esta metodología en herramientas paralelas que calculan los parámetros, automáticamente, en tiempo de ejecución; incluyendo también el número de procesadores en el cálculo del *mapping* y utilizar una estrategia dinámica de distribución de tareas basada en la carga de las máquinas del sistema. Mediante esta estrategia, el programa calcula la carga de las máquinas al comenzar la ejecución y predice la distribución óptima para las características actuales del sistema, resolviendo el problema con los parámetros calculados.

Capítulo 4

El Problema de la Predicción de la Estructura Secundaria del RNA

4.1. Resumen

En este capítulo resolvemos un problema importante en el ámbito de la biología molecular: el **problema de la predicción de la estructura secundaria del RNA**. Utilizamos algoritmos de programación dinámica [103] que, como ya hemos comentado en capítulos anteriores, es una técnica de optimización combinatoria ampliamente utilizada en múltiples campos: teoría de control, investigación operativa, computación y, más recientemente, en el ámbito de biología. Muchos autores han desarrollado algoritmos de programación dinámica para familias de problemas multietapa. Sin embargo, la clase más general de problemas, donde las dependencias aparecen entre etapas no consecutivas, no ha sido estudiada en profundidad. El problema del emparejamiento de bases del RNA es uno de los problemas incluidos en esta última clase.

Uno de los paquetes de software más importantes que resuelven este problema es el paquete de Viena [72, 98]. Hemos paralelizado la función principal de este paquete, derivando instancias diferentes que se corresponden con un recorrido horizontal y con un recorrido vertical del dominio del problema. Desarrollamos una primera aproximación paralela donde el número de tareas asignadas a cada procesador se mantiene constante durante toda la resolución del problema. Aún cuando la cantidad asignada sea la misma para todos los procesadores y todas las iteraciones del algoritmo, nos encontramos en un sistema heterogéneo según la definición dada en la sección 1.2: un programa donde el trabajo realizado varía entre las distintas iteraciones, ejecutado sobre una plataforma homogénea. Esto es debido a que la cantidad de trabajo que requiere cada iteración es diferente y depende del elemento a calcular en cada momento.

Además, formulamos y resolvemos analíticamente un problema de optimización para determinar el tamaño del bloque de código que minimiza el tiempo total de ejecución del programa paralelo. La validación de nuestro modelo analítico para el algoritmo paralelo la presentamos en [4] y demostramos que los resultados experimentales obtenidos presentan un comportamiento acorde a las predicciones de nuestro modelo teórico. En [5] ampliamos el estudio desarrollando un modelo estadístico. Este modelo estadístico es válido en cualquier instancia, incluso cuando las hipótesis del modelo analítico no son satisfechas.

Posteriormente, consideramos el hecho de que en cada iteración se realiza una cantidad distinta de trabajo, para plantearnos una modificación de nuestro programa. En esta nueva versión permitimos resolver el problema asignando un número diferente de tareas en cada iteración del algoritmo. De esta forma, podemos adaptar el tamaño del bloque de código óptimo a la cantidad de trabajo a realizar en cada momento y reducir el tiempo de ejecución del programa. Comparamos los resultados obtenidos con las dos versiones para verificar el rendimiento del sistema al ejecutar con la segunda versión. Empleamos

los modelos anteriores para predecir la cantidad óptima de tareas a asignar a los procesadores en cada iteración.

4.2. Introducción

La investigación en biología molecular está básicamente orientada a la comprensión de la estructura y funcionamiento de los ácidos nucleicos y de las proteínas. Este campo ha sufrido un avance espectacular desde el descubrimiento de la estructura del DNA (ácido desoxirribonucleico) en 1953. Con el incremento de la capacidad para manipular secuencias de biomoléculas, se ha generado una gran cantidad de información. La necesidad de procesar esta información ha creado nuevos problemas de naturaleza interdisciplinar. Los biólogos son los creadores y usuarios finales de estos datos; sin embargo, debido al tamaño y complejidad de esta información, se necesita la ayuda de otras disciplinas, en particular de las matemáticas y de la informática. Esto ha derivado en la creación de un nuevo campo muy amplio, denominado **biología molecular computacional**. El término **bioinformática** ha sido utilizado por diferentes disciplinas para referirse a varias cuestiones. En general, se puede aplicar este término para referirse a las tecnologías de la información aplicadas a la gestión y análisis de los datos biológicos. Esto engloba desde la inteligencia artificial y la robótica hasta la biología molecular. En el apéndice A introducimos el contexto en el que se encuentra el problema que resolvemos en este capítulo y la terminología adecuada para asimilar los conceptos.

En biología molecular existen problemas que requieren algoritmos eficientes, porque deben resolverse miles de veces al día; un ejemplo clásico de estos problemas es la comparación de secuencias: dadas dos secuencias, que representan biomoléculas, pretendemos saber cuál es la similitud entre las dos. Constantemente se diseñan nuevos y mejores algoritmos y aparecen nuevos subcampos de investigación. La programación dinámica es uno de los métodos más utilizados en estos algoritmos [194].

En este capítulo hemos resuelto el **problema de la predicción paralela de la estructura secundaria del RNA** Este problema también se conoce como el **problema del plegamiento de la molécula de RNA** o el **problema del emparejamiento de bases**. Para determinar su solución implementamos el algoritmo de programación dinámica presentado en [162]. Como ya ha sido comentado anteriormente, la programación dinámica es una importante técnica de optimización combinatoria, aplicada a una amplia variedad de campos (teoría de control, investigación operativa, biología computacional, etc.). En la sección 2.8 hemos presentado el método de funcionamiento de esta técnica. Muchos autores describen algoritmos paralelos de programación dinámica para problemas específicos, como el problema de la mochila entera 0/1, la asignación de recursos, la búsqueda de la subsecuencia más larga [85]. Esta clase de problemas ha sido extensamente estudiada y paralelizada de forma eficiente sobre las arquitecturas actuales durante la última década. Esta primera familia de problemas sigue un patrón similar: son problemas multietapa, donde las ecuaciones de recurrencia de la programación dinámica producen sólo dos tipos de dependencias: dentro de una misma etapa y entre etapas consecutivas. Podemos caracterizar esta clase de problemas mediante ecuaciones de recurrencia uniformes [106]. Esto se refiere tanto a ecuaciones de recurrencia uniforme clásicas (por ejemplo, el problema de comparación de cadenas [162]), como a recurrencias no uniformes solamente en una de las coordenadas. En este último caso, el problema es tratado como si fuera uniforme simplemente cambiando el eje de coordenadas no uniforme; es decir, la dependencia no uniforme es proyectada y ocultada en la memoria local. De esta manera se han paralelizado el problema de la mochila y el problema de asignación de recursos [12, 85].

Una familia más general de recurrencias aparece cuando aplicamos la programación dinámica al problema de obtener el camino más corto, la búsqueda óptima sobre un árbol binario, la parentización óptima del producto encadenado de matrices o el problema de la predicción de la estructura secundaria del RNA [77, 79, 155]. En esta segunda clase se observa que aparecen nuevas dependencias entre etapas no consecutivas. Estas nuevas dependencias pueden ser consideradas como una nueva fuente de comunicaciones; obviamente, esto hace que la paralelización sea más compleja. Se han realizado diversos estudios sobre este tipo de recurrencias, buscando modelos que minimicen el coste debido a las comunicaciones. En

[76, 90, 120] se encuentran trabajos donde se utilizan matrices triangulares para resolver las recurrencias asociadas. En [23, 67, 77, 79, 158] se describen varios algoritmos que resuelven los mismos problemas usando el modelo *CREW-PRAM* para sistemas de memoria compartida.

En este capítulo estudiamos esta segunda clase de ecuaciones de recurrencia, con las peculiaridades del problema de la predicción de la estructura secundaria del RNA y realizamos una paralelización eficiente usando una topología simple de *pipeline* sobre un anillo. Debido a que es un problema bioquímico de considerable importancia, se han propuesto diversos algoritmos en las últimas décadas, algunos de ellos pueden encontrarse en [183]. Posteriormente, en [123], se presenta un algoritmo diseñado para computar todas las posibilidades del emparejamiento de bases y en [72] se presenta una implementación de este algoritmo para una arquitectura de paso de mensajes. Todos ellos requieren muchos recursos computacionales. Como consecuencia de esto se han propuesto varios algoritmos paralelos. Por ejemplo, en [72, 98] se presenta uno de los paquetes de *software* más utilizados para resolver el problema del emparejamiento de bases del RNA (el paquete de Viena), realizando una adaptación de los algoritmos a máquinas paralelas con memoria distribuida y arquitectura de hipercubo. Estos trabajos consideran una distribución equilibrada de tareas, pero sin tener en cuenta el coste debido a las comunicaciones. En este capítulo presentamos una paralelización de la función principal de este paquete, que predice el plegamiento de una secuencia RNA, considerando la heterogeneidad producida por el programa (sección 1.2). En [126] los autores emplean una técnica conocida como **particionado del espacio de direcciones** (o ***tiling***) [105, 188] para resolver esta clase de ecuaciones de recurrencia y demuestran su utilidad en los resultados obtenidos. Otro trabajo importante se encuentra en [34], donde se demuestra experimentalmente que, para las secuencias de longitud 9212, las comunicaciones requieren aproximadamente el 50 % del tiempo de ejecución. Esto demuestra claramente las ventajas de agrupar los cómputos en bloques para reducir el número de comunicaciones; sin embargo, los autores trabajan con un sistema considerado hoy obsoleto. En [99] también consideran secuencias de RNA muy largas (concretamente la del HIV) y resuelven el problema de la predicción de su estructura secundaria y la comparación entre estructuras secundarias sobre máquinas masivamente paralelas, demostrando su utilidad. En [163] se diseña un algoritmo genético paralelo masivo y se prueba sobre máquinas *SGI Origin 2000* y *Cray T3E*. Otros problemas biológicos relacionados con el análisis de secuencias y la síntesis de proteínas también están siendo abordadas mediante programación dinámica [173].

Considerando los trabajos anteriores, concluimos que encontrar un buen equilibrio entre los costes debidos al cómputo y a las comunicaciones, es la principal dificultad para obtener una implementación paralela eficiente. La técnica *tiling* o particionado del espacio de direcciones, es una técnica especial desarrollada para este propósito [105, 188]. Esta aproximación es utilizada de forma automática por los compiladores para controlar la granularidad de los programas e incrementar la relación entre el cómputo y la comunicación [45, 109, 149, 161, 187] o bien, empleado para sintonizar manualmente códigos secuenciales y paralelos. Un **bloque de cómputo** (o ***tile***) en el espacio de direcciones es un grupo de iteraciones con forma de paralelepípedo que va a ser ejecutado como una unidad; todos los datos no locales, requeridos para cada uno de los bloques, deben ser obtenidos previamente (por ejemplo, con llamadas a funciones de librerías como *MPI* [89, 134]). El cuerpo del *tile* es ejecutado a continuación. Este código no contiene llamadas a ninguna rutina de comunicación. Por último, se transmiten los datos necesarios para los siguientes cómputos. Este proceso se repite de forma iterativa. El **problema del *tiling*** puede ser definido como el problema de elegir los parámetros del *tile* (forma y tamaño) para una ejecución óptima. Generalmente, esto requiere resolver un problema de optimización discreta no-lineal. La mayoría de los trabajos sobre *tiling* óptimo, consideran jerarquías de bucles perfectos con un espacio de iteraciones (o dominio) con forma de paralelepípedo y dependencias uniformes. El *tiling* se denomina **ortogonal** [12, 13], cuando los límites del *tile* son paralelos a los límites del dominio. Cuando se relaja esta restricción aparece un *tiling* obliquo [61, 100, 101].

En este capítulo resolvemos el problema del *tiling* óptimo cuando disponemos de un espacio de iteraciones triangular $\frac{n \times n}{2}$ con dependencias no uniformes, utilizando una topología de anillo con p procesadores. Nuestra aproximación se compone de 2 fases: en la primera fase (el **preprocesamiento**), los p procesadores trabajan de forma independiente sobre una parte del espacio de iteraciones triangular de tamaño

$\frac{n}{p} \times \frac{n}{p}$. Esta fase finaliza con un intercambio de datos entre todos los procesadores. En la segunda fase, los procesadores trabajan en un *pipeline*. Aplicamos la aproximación *tiling* a esta segunda fase, donde disponemos de un dominio de iteración que permite usar bloques de cómputo rectangulares. Adaptamos las técnicas propuestas en [12, 13] al caso considerado y desarrollamos una extensión de la aproximación *tiling* ortogonal para las recurrencias no uniformes. Determinamos una ecuación analítica para obtener el tiempo de ejecución de nuestros programas y formulamos un problema de optimización discreta no-lineal para encontrar el tamaño del *tile*. Nuestro modelo está basado en el modelo *BSP* [124, 178], un modelo portable ampliamente aceptado para la predicción del rendimiento sobre máquinas paralelas. Nuestra solución puede ser fácilmente incorporada a un compilador sin producir sobrecarga, debido a que es un método exacto y analítico. Además también desarrollamos una aproximación estadística que nos permite encontrar un tamaño del *tile* próximo al óptimo en las situaciones donde no se cumplen las hipótesis del modelo analítico.

Nuestras contribuciones específicas para resolver este problema pueden resumirse en la siguiente lista [4, 5]:

- Proponemos un nuevo esquema paralelo para una amplia clase de recurrencias, con un espacio de iteraciones triangular y dependencias no uniformes. El problema del emparejamiento de bases del RNA es un caso particular de esta clase de recurrencias.
- Desarrollamos y extendemos una aproximación *tiling* para este tipo de recurrencias. Proponemos y analizamos diferentes recorridos del dominio de iteraciones: horizontal y vertical.
- Usando el modelo *BSP* [124, 178] formulamos un problema de optimización para determinar el tamaño del bloque de cómputo, que minimice el tiempo total de ejecución del programa, sobre una máquina paralela con p procesadores configurados como un anillo.
- En nuestro esquema paralelo elegimos una distribución donde fijamos el ancho y alto del bloque de cómputo, realizando una distribución por columnas entre los procesadores.
- Desarrollamos un modelo estadístico válido para todos los casos, incluso cuando las hipótesis del modelo analítico no son satisfechas.
- Realizamos numerosos experimentos computacionales para validar nuestro modelo teórico, incluyendo la aplicación de estos algoritmos a secuencias reales de RNA.
- Modificamos nuestro programa paralelo para procesar cada iteración del algoritmo empleando un ancho del *tile* diferente y reducir el tiempo total de ejecución del programa

Nuestra aproximación es similar a la realizada en el primer trabajo donde se demuestra el éxito de la aproximación *tiling* para esta clase de recurrencias [126]: pretendemos reducir el coste de las comunicaciones realizando una distribución por columnas entre los procesadores. Sin embargo, existen algunas diferencias:

1. Debido a la fase inicial de preprocesamiento, nuestro esquema paralelo equilibra mejor la carga.
2. En nuestras ecuaciones, tenemos en cuenta el valor de la latencia (β), mientras que en [126] los autores consideran que este valor es igual a 0.
3. Realizamos un análisis completo de la función de tiempo en todas las situaciones: cuando los procesadores del *pipeline* están permanentemente ocupados y cuando los procesadores permanecen inactivos durante un tiempo esperando por los datos (como ha sido comentado en el capítulo 3). En [126] solamente consideran la primera de estas dos situaciones.
4. Consideramos bloques de cómputo rectangulares, mientras que los autores de [126] emplean únicamente bloques cuadrados. Además, permitimos bloques de tamaño variable en cada iteración del algoritmo.

El resto del capítulo lo hemos estructurado de la siguiente manera: En la sección 4.3 introducimos el problema que vamos a resolver, en la sección 4.4 describimos uno de los paquetes de *software* más importantes para este problema (el paquete de Viena) y definimos las estrategias de paralelización de este problema (sección 4.5). A continuación, desarrollamos la paralelización del paquete de Viena, utilizando un tamaño constante de *tile* (sección 4.6) y presentamos los resultados computacionales obtenidos (sección 4.7). Diseñamos un modelo analítico para calcular el tamaño óptimo del *tile* (sección 4.8) y comprobamos la validez de este modelo (sección 4.9). También hemos desarrollado un modelo estadístico alternativo al modelo analítico anterior (sección 4.10) y comparamos los resultados obtenidos con ambos métodos (sección 4.11). En la sección 4.12 modificamos el programa para poder utilizar tamaños variables de *tile* en cada uno de los rectángulos, presentamos los resultados computacionales para esta nueva versión (sección 4.13) y utilizamos los modelos para obtener los valores óptimos de los parámetros (sección 4.14). Por último, en la sección 4.15 presentamos las conclusiones de este trabajo.

4.3. El problema de la predicción de la estructura secundaria del RNA

Una molécula de RNA es una cadena monohélice compuesta por 4 **nucleótidos: Acetina (A), Citosina (C), Guanina (G), Uracilo (U)** (sección A.6). Un nucleótido de alguna parte de la molécula puede unirse, formando una **base**, con un nucleótido **complementario** de otra parte de la molécula. Esto da lugar a **pliegues** y forma la estructura secundaria del RNA. La secuencia de nucleótidos determina de forma única la manera en que la molécula se pliega; por tanto, el problema podría enunciarse de la siguiente manera: **Dada una secuencia de nucleótidos correspondientes a una molécula de RNA, predecir la estructura secundaria de dicha molécula, determinando cuáles son las bases que se aparean entre sí**. Es un problema que puede resolverse en tiempo polinomial si se restringe a estructuras que no posean **nudos** (estructura secundaria). Si aceptamos que pueda poseer nudos (estructura terciaria), el problema se convierte en NP-completo.

En esta molécula, las bases complementarias C-G y A-U forman parejas de bases estables; además también existe la unión débil G-U. Consideramos la molécula como una cadena de n caracteres (su estructura primaria):

$$R = r_1 r_2 \dots r_n \qquad r_i \in \{A, C, G, U\}$$

Normalmente, n es del orden de cientos o incluso miles; por ejemplo, el genoma RNA del bacteriófago $Q\beta$ contiene $n \approx 4220$ nucleótidos; el virus de la polio, $n \approx 7500$; y el virus HIV, $n \approx 10000$.

La estructura secundaria de la molécula consiste un conjunto S de pares de bases (r_i, r_j), tal que $1 \leq i < j \leq n$. Si $(r_i, r_j) \in S$ se dice que r_i es una **base complementaria** de r_j y $j - i > t$, donde t es la distancia mínima entre bases para que puedan unirse ($j - i > 4$). Si (r_i, r_j) y (r_k, r_l) son dos parejas de bases existen tres alternativas:

1. Es el mismo par de bases: $i = k$ y $j = l$. Esta condición indica que cada nucleótido puede tomar parte, como mucho, en un único par de bases.
2. (r_i, r_j) precede a (r_k, r_l): $i < j < k < l$.
3. (r_i, r_j) incluye a (r_k, r_l): $i < k < l < j$. En este caso, se dice que el par de bases (r_k, r_l) es interior a (r_i, r_j) y es inmediatamente interior si no existe ningún par de bases (r_p, r_q) tal que $i < p < k < l < q < j$.

Hay una configuración natural que no se ha considerado: los llamados **nudos** o ***knots***. Un nudo se produce cuando $(r_i, r_j) \in S$, donde $i < k < j < l$. Descartando los nudos, S forma un grafo plano. Por

lo tanto, aunque los nudos forman parte de la estructura de RNA, su predicción puede llevarse a cabo en la predicción de la estructura tridimensional de la molécula. Eliminar los nudos de la predición de la estructura secundaria es un requisito necesario para utilizar algoritmos de programación dinámica.

Un método sencillo para determinar la estructura del RNA es mediante la definición de reglas de energía. Suele emplearse una función e, tal que $e(r_i, r_j)$ es la energía liberada por la pareja de bases $(r_i, r_j) \in S$.

Por convención, se asume que

$$e(r_i, r_j) < 0 \quad \text{si } i \neq j$$
$$e(r_i, r_j) = 0 \quad \text{si } i = j$$

Algunos valores de e utilizados a $37°C$ son:

CG $\Leftrightarrow$ -3 AU $\Leftrightarrow$ -2 GU $\Leftrightarrow$ -1

La energía total, E, liberada por la estructura S, viene determinada por $E(S) = \sum_{(r_i,rj)\in S} e(r_i, r_j)$

Se denota por $S_{i,j}$ a la estructura secundaria de la cadena $R = r_i r_{i+1} ... r_j$ y por $E(S_{i,j})$ a la energía mínima liberada por la estructura. El objetivo es encontrar el valor de $E(S_{1,n})$ (la energía liberada por el plegamiento de toda la molécula). Se aplican las siguientes ecuaciones [34, 162, 183, 194]:

$$E(S_{i,j}) = \text{mín} \begin{cases} E(S_{i+1,j}) & r_i \text{ queda libre} \\ E(S_{i,j-1}) & r_j \text{ queda libre} \\ e(i,j) + E(S_{i+1,j-1}) & (r_i, r_j) \in S \\ \text{mín}\{E(S_{i,k-1}) + E(S_{k,j})\} & (r_i, r_k) \in S \text{ y } (r_l, r_j) \in S, \text{ pero } (r_i, r_j) \notin S \end{cases}$$

Hay que tener en cuenta la existencia de estructuras con bucles. Supongamos que (r_i, r_j) es una pareja de bases de S y consideramos las posiciones k, u y l, tales que $i < k < u < l < j$. Se dice que la base r_k es **accesible** desde (r_i, r_j) si (r_k, r_l) no forma una pareja de bases en S (para cualquier k y l). También se dice que el par de bases (r_k, r_l) es accesible si tanto r_k como r_l son accesibles. Se denomina **bucle** al conjunto de bases accesibles desde una pareja (r_i, r_j). La suposición de que no hay nudos en la estructura implica que cualquier estructura S descompone la cadena R en bucles disjuntos.

El tamaño de un bucle puede determinarse de dos formas distintas:

- El número de bases **libres** o **aisladas** que contiene el bucle: es decir, aquellas bases que no forman pareja. Existen bucles de tamaño 0 (parejas de bases de forma consecutiva).
- Se considera el bucle como el conjunto de los $(k-1)$ pares de bases y las k' bases accesibles desde (r_i, r_j) y se denomina ***k-bucle***.

Existen diferentes tipos de bucles (figura 4.1) que podemos clasificar según el valor k del *k-bucle*.

1. Si $k = 1$: bucle **hairpin** (figura 4.1-a). No existe ningún par de bases accesible desde (r_i, r_j).
2. Si $k = 2$: Existe un par de bases (r_k, r_l) que es accesible desde (r_i, r_j).
   a) **Par apilado** o ***stack pair*** (figura 4.1-b): si $k - i = 1$ y $j - l = 1$. Son parejas de bases que se suceden de forma consecutiva (bucle de tamaño 0 si contamos el número de bases libres). También se denomina **región helicoidal**.
   b) Bucle ***bulge*** (figura 4.1-c): si $k - i > 1$ o $j - l > 1$, pero no los dos.

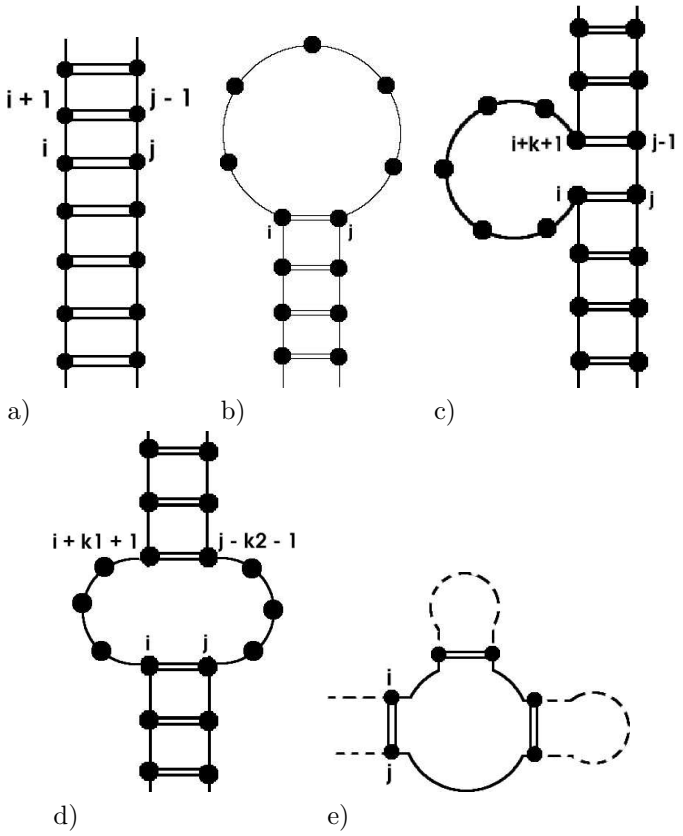

Figura 4.1: Diferentes tipos de bucles en la estructura secundaria de una molécula de RNA: a) Región helicoidal, par apilado o *stack*. b) Bucle hairpin. c) Bucle *bulge*. d) Bucle interior. e) Bucle *multi-loop* o multibucle.

c) Bucle **interior** (figura 4.1-d): si $k - i > 1$ y $j - l > 1$.

3. Si $k \geq 3$: **Multibucle** o ***multi-loop*** (figura 4.1-e).

En la expresión anterior de la energía total ($E(S_{i,j})$) no se tenía en cuenta la existencia de bucles. En la expresión siguiente sí se consideran estas estructuras. La energía de la estructura secundaria se calcula como la suma de las contribuciones de las energías de cada bucle. Como consecuencia, la mínima energía liberada se puede calcular recursivamente mediante programación dinámica [34, 162, 183, 194].

$$E(S_{i,j}) = \text{mín} \begin{cases} E(S_{i+1,j}) & r_i \text{ queda libre} \\ E(S_{i,j-1}) & r_j \text{ queda libre} \\ E(L_{i,j}) & r_i \text{ y } r_j \text{ se unen entre sí} \\ \text{mín}\{E(S_{i,k-1}) + E(S_{k,j})\} & i < k \leq j \end{cases}$$

La expresión $L_{i,j}$ refleja la situación donde $(r_i, r_j) \in S$ y refleja lo que sucede entre r_i y r_j. Para poder calcular el valor de $E(L_{i,j})$ se necesitan varias funciones:

$$\begin{aligned} eh(k) &= \text{energía liberada por un bucle hairpin de tamaño } k \\ \eta &\equiv \text{energía liberada por parejas de bases adyacentes} \\ eb(k) &\equiv \text{energía liberada por un } \textit{bulge} \text{ de tamaño } k \\ ei(k) &\equiv \text{energía liberada por un bucle interior de tamaño } k \end{aligned}$$

Utilizando estas funciones, puede calcularse $E(L_{i,j})$ mediante el siguiente conjunto de expresiones:

$e(r_i, r_j) + eh(j - i - 1)$	si $L_{i,j}$ es un hairpin
$e(r_i, r_j) + \eta + E(S_{i+1,j-1}$	si $L_{i,j}$ es una región helicoidal
$\text{mín}_{k \geq 1}\{e(r_i, r_j) + eb(k) + E(S_{i+k+1,j-1}\}$	si $L_{i,j}$ es un *bulge* sobre i
$\text{mín}_{k \geq 1}\{e(r_i, r_j) + eb(k) + E(S_{i+1,j-k-1}\}$	si $L_{i,j}$ es un *bulge* sobre j
$\text{mín}_{k_1,k_2 \geq 1}\{e(r_i, r_j) + ei(k_1 + k_2) + E(S_{i+1+k_1,j-1-k_2}\}$	si $L_{i,j}$ es un bucle interior

Mediante las ecuaciones anteriores se dispone de toda la información necesaria para calcular la matriz programación dinámica. En la figura 4.2 se muestra cuál es la región de la matriz cuyos valores debemos conocer antes de calcular el valor $E(S_{i,j})$. Se observa que la cantidad de elementos que es necesario conocer previamente depende de los valores de i y j.

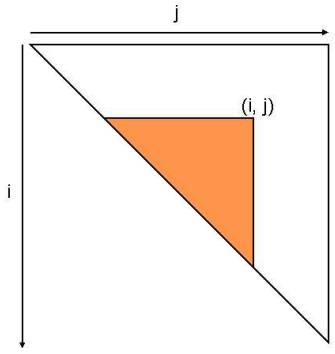

Figura 4.2: Espacio de direcciones triangular. El espacio sombreado representa el área de dependencia de un punto (i, j). La solución se encuentra en la parte superior derecha, en la posición $(1, n)$.

Los puntos del interior de la región sombreada de la figura 4.2, lejos de los bordes (filas i, $i + 1$ y columnas j, $j-1$), solamente se consideran en el caso del bucle interior. Corresponde a la situación cuando $L_{i,j}$ está en el interior de un bucle y es el caso que más tiempo consume. En la figura 4.3 se muestran las regiones que previamente deben haber sido calculadas para poder computar $E(L_{i,j})$ y se especifica a qué tipo de bucle corresponde cada elemento (i, j).

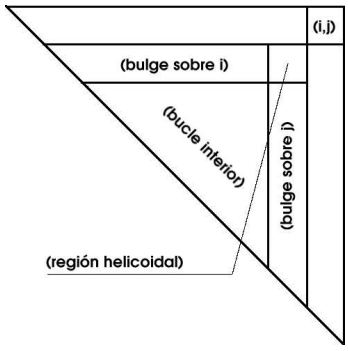

Figura 4.3: Dependencias de datos para la evaluación de $E(L_{i,j})$.

Cada punto (i, j) de la matriz requiere $O(n^2)$ operaciones, dando un tiempo total del algoritmo de $O(n^4)$. Como se explicó en [34], es razonable asumir que el tamaño en el interior del bucle es menor que

algún número fijo. Bajo está hipótesis, el tiempo para calcular $E(L_{i,j})$ para un punto fijo (i, j) requiere un orden $O(1)$. Sin embargo, las ecuaciones anteriores suponen una simplificación del problema, porque no se está considerando el caso del *multi-loop* [34, 193]. Para integrarlo en esta aproximación solamente es necesario tener en cuenta que:

1. El dominio de dependencia del *multi-loop* es el mismo, pero requiere considerar más casos para calcular $E(S_{i,j})$.
2. Considerar el *multi-loop* necesita un $O(n)$ para resolverse, el tiempo total del algoritmo se convierte en $O(n^3)$.

Este caso sí ha sido considerado por los autores del paquete de Viena, para la predicción de la estructura secundaria del RNA, en el desarrollo de su trabajo.

4.4. El paquete de Viena

En esta sección describimos el paquete desarrollado por la Universidad de Viena para la predicción y comparación de las estructuras secundarias del RNA [72, 98]. Este paquete está basado en los algoritmos de programación dinámica presentados en la sección 4.6 para predecir las estructuras con la mínima energía liberada. Introducen una heurística eficiente para el problema del plegamiento inverso y presentan programas para la comparación de estructuras secundarias del RNA. El núcleo del paquete consiste en códigos compactos para obtener la estructura de mínima energía o funciones de partición de moléculas de RNA, incluyendo también un paquete estadístico que incluye rutinas para el análisis *cluster*, la geometría estadística y la descomposición. Además, también permite a los usuarios imponer restricciones al plegado de las moléculas.

Podemos clasificar todas las funciones que nos ofrece el paquete en las siguientes actividades:

- Predecir la estructura secundaria de mínima energía. Es la función más utilizada del paquete y proporciona 3 clases de algoritmos de programación dinámica para realizar la predicción.
- Calcular la función de partición para el ensamblaje de estructuras.
- Calcular las estructuras subóptimas dado un rango de energía.
- Diseñar secuencias dada una estructura predefinida (**plegamiento inverso**).
- Comparación de estructuras secundarias.

El paquete de Viena (versión *1.4*) está constituido por 10 módulos:

- **RNAfold:** Predicción de la estructura secundaria de una molécula de RNA. Es el módulo más importante y en el que hemos centrado nuestro trabajo.
- **RNAsubopt:** Cálculo de estructuras subóptimas dentro de un rango de energía dado.
- **RNAeval:** Evaluación de la energía para una cadena de nucleótidos y una estructura dada.
- **RNAheat:** Cálculo de las curvas de derretimiento (*melting curves*).
- **RNAdistance:** Comparación de estructuras secundarias.
- **RNApdist:** Comparación de ensamblaje (*ensembles*) de estructuras secundarias.
- **RNAinverse:** Búsqueda de secuencias de pliegues que coincidan con estructuras dadas.

- **AnalyseSeqs:** Análisis de secuencias de datos.
- **AnalyseDist:** Análisis de matrices de distancias.
- **RNAplot:** Dibuja una estructura secundaria de RNA.

Todos los módulos utilizan la entrada y salida estándar (*stdin* y *stdout*) y utilizan opciones desde la línea de comandos, en lugar de menús, para poder anidarlos de una forma fácil mediante *pipes*.

En la figura 4.4 se observa la salida del módulo RNAplot. Se muestra la estructura secundaria de una molécula de RNA: en ella se observan claramente los nucleótidos que forman la secuencia y los pares de bases y bucles que se establecen durante el plegamiento.

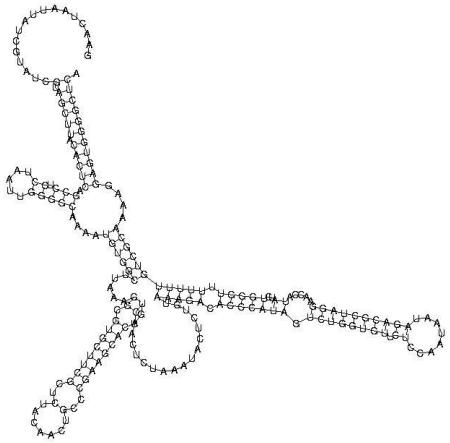

Figura 4.4: Estructura secundaria de una molécula de RNA obtenida con el paquete de Viena.

Para realizar las funciones anteriores, los autores definen 2 matrices triangulares que almacenan la energía liberada asociada a cada subproblema. Así, $c[i, j]$ almacena la energía liberada para el subproblema $R = r_i, r_{i+1}, ..., r_j$. En una de las matrices no se tiene en cuenta la energía asociada a los *multi-loops*, mientras que en la segunda sí se considera. En la figura 4.5 se muestra el aspecto de una matriz triangular y numeramos sus elementos.

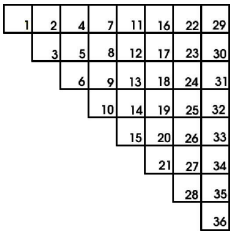

Figura 4.5: Aspecto de una matriz triangular de tamaño 8×8 y numeración de sus elementos.

Para simular una matriz triangular emplean un vector de datos y un vector de índices. El vector utilizado para representar a esta matriz y evitar el uso ineficiente de la memoria aparece en la figura 4.6. Esta estructura de datos se denomina ***Tvector*** y almacena los datos de la tabla por columnas.

Para acceder a los *Tvectores* utilizan un vector de índices (*index*), que tiene tantos elementos como filas (o columnas) tengan las matrices triangulares. El elemento $index[i]$ almacena el elemento del *Tvector*

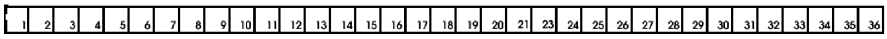

Figura 4.6: Aspecto del Tvector que simula la matriz de la figura 4.5. La tabla se almacena en este vector por columnas.

donde comienza la columna i. En la figura 4.7 se encuentra el vector de índices para el *Tvector* de la figura 4.6.

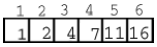

Figura 4.7: Vector de índice. El elemento $index[i]$ contiene la celda del *Tvector* donde comienza la columna i.

Para acceder al elemento (i, j) de la matriz triangular solamente es necesario aplicar la siguiente expresión:

$$m[i, j] = Tm[index[j] + i]$$

donde *index* es el vector de índice y Tm es el *Tvector* equivalente a la matriz m.

Empleando la versión secuencial de este paquete para resolver diferentes instancias del problema, puede comprobarse que el tiempo de ejecución se incrementa, de forma considerable, al aumentar el tamaño de la molécula de RNA (figura 4.8). Para un problema de tamaño 8192, la ejecución secuencial invierte más de 1500 segundos y, en la práctica, podemos encontrar moléculas de tamaño superior. Observando estos resultados, nos planteamos realizar la paralelización de la función principal del paquete (la función que se encarga de obtener el plegamiento de la molécula de RNA) para intentar reducir los tiempos de ejecución en los problemas de mayor tamaño.

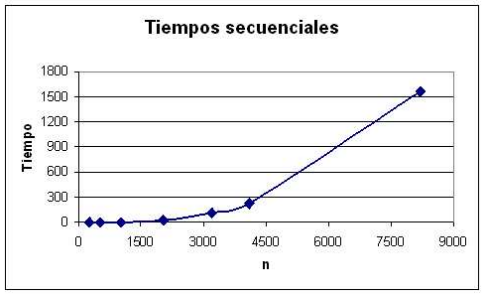

Figura 4.8: Tiempos de ejecución obtenidos al resolver el problema de la predicción de la estructura secundaria del RNA utilizando el paquete de Viena. Los resultados han sido obtenidos en una *SGI Origin 3800* (sección 4.7).

4.5. Estrategias de paralelización

Hemos empleado dos estrategias de paralelización para resolver el problema. La diferencia entre los dos esquemas radica en la forma de recorrer el dominio de iteraciones del problema. Comenzamos con una fase de preprocesamiento (idéntica en ambos métodos), donde procesamos espacios de direcciones triangulares, T_i con $i = 1, \ldots, p$, de tamaño $(\frac{n}{p} \times \frac{n}{p})/2$ puntos (figura 4.9-a). Los triángulos son calculados simultáneamente sobre p procesadores, cada procesador calcula uno de esos triángulos de forma secuencial.

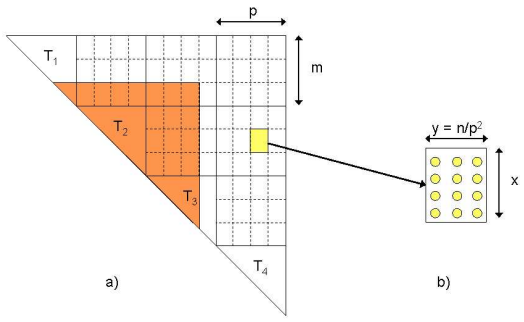

Figura 4.9: a) Espacio de direcciones transformado para procesarlo utilizando $p = 4$ procesadores. Se identifican 4 triángulos que se calculan durante la fase de preprocesamiento. Los restantes puntos del espacio de direcciones se agrupan en bloques cuadrados que, a su vez, se dividen en *tiles*. El espacio sombreado representa el área de dependencia de un punto (i, j). b) Imagen ampliada de un *tile*.

En este capítulo hemos modificado la numeración aplicada a los procesadores por motivos de simplicidad, ahora comienzan a numerarse desde p_1 en lugar de hacerlo desde p_0.

Una vez finalizada la fase de preprocesamiento, el resto del dominio de iteraciones podemos visualizarlo como una escalera invertida de $\frac{p(p-1)}{2}$ cuadrados: R_s^t, donde s indica la fila ($s = 1, \ldots, p-1$) y t indica la columna, $t = 2, \ldots, p$ (figura 4.10-a).

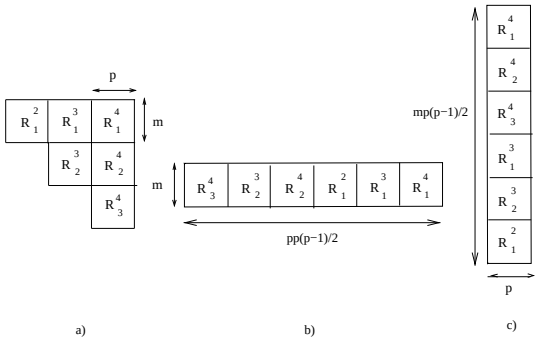

Figura 4.10: a) Espacio de direcciones transformado visto como un conjunto de cuadradros. b) Recorrido horizontal del dominio. c) Recorrido vertical del dominio.

A continuación, aplicamos la aproximación *tiling* al dominio de la figura 4.10-a, considerando *tiles* rectangulares. Cada *tile* contiene x filas con y puntos en cada fila (figura 4.9-b). Denominamos a los enteros x e y altura y anchura del *tile* respectivamente. Cada *tile* es considerado como un único punto en un nuevo dominio transformado.

Suponiendo que $p^2 \leq n$, decidimos fijar el ancho del *tile* al valor $y = \frac{n}{p^2}$. Por motivos de simplicidad, establecemos $m = \frac{n}{px}$ y $\bar{p} = p - 1$. Considerando estos parámetros, cada cuadrado R_s^t contiene m filas de *tiles*, con p *tiles* por fila. La figura 4.9 ilustra un caso particular de un dominio transformado, cuando $n = 48$, $p = 4$, $m = 3$, $x = 4$ y $\frac{n}{p^2} = 3$. Las dependencias de este dominio transformado son las mismas que las del dominio original del problema.

Hay tres recorridos obvios del dominio de iteraciones que permiten satisfacer esas dependencias: diagonal (figura 4.10-a), horizontal (figura 4.10-b) y vertical (figura 4.10-c). En el **recorrido diagonal** los cuadrados son computados en diagonales; es decir, primero se calculan en *pipeline* los cuadrados R_i^s tal que $t - s = 1$, después se calculan los cuadrados R_i^s tal que $t - s = 2$, etc. (figura 4.10-a). En el **recorrido**

horizontal los cuadrados son procesados por filas: computamos primero el cuadrado inferior y después vamos procesando las filas hacia arriba, para cada fila calculamos los cuadrados de izquierda a derecha. El número total de cuadrados es $\frac{p\bar{p}}{2}$ y el dominio de iteraciones puede ser visto como un gran rectángulo de tamaño $m \times \frac{p^2\bar{p}}{2}$ tiles. Para el ejemplo considerado en la figura 4.10-b, el orden en que se procesan los cuadrados es R_3^4, R_2^3, R_2^4, R_1^2, R_1^3, R_1^4. En el **recorrido vertical** los cuadrados son procesados por columnas: primero el cuadrado superior izquierdo y, a continuación, vamos procesando el resto de las columnas de izquierda a derecha; para cada una de ellas los cuadrados se procesan de abajo a arriba. En el ejemplo presentado en la figura 4.10-c, el orden en que se computan los cuadrados es R_1^2, R_2^3, R_1^3, R_3^4, R_2^4, R_1^4. El número total de cuadrados es $\frac{p\bar{p}}{2}$ y el dominio de iteraciones puede ser visto en este caso como un gran rectángulo de tamaño $\frac{pm\bar{p}}{2} \times p$ *tiles*. La cantidad de puntos calculados por cada uno de los procesadores está distribuida de forma uniforme entre todos ellos. En este trabajo empleamos solamente los recorridos horizontal y vertical.

4.6. La paralelización del paquete de Viena: tamaño constante del *tile*

En esta sección describimos el paquete paralelo que hemos desarrollado a partir de la versión secuencial de la función que predice la estructura secundaria del RNA del paquete de Viena [72, 98]. Nuestro objetivo es computar la matriz triangular que permite obtener la estructura secundaria de una molécula de RNA, teniendo en cuenta las dependencias existentes entre los datos (figura 4.11). Dada una secuencia de nucleótidos $R = r_1r_2r_3...r_n$, el elemento (i, j) de la matriz representa la energía liberada por el pliegue de la subsecuencia $R' = r_i...r_j$, con $j > i$. Esto significa que la solución del problema se encuentra en la posición $(1, n)$, donde se encuentra la energía liberada para toda la molécula (figura 4.11).

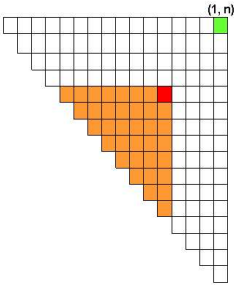

Figura 4.11: Dependencias entre los datos: el elemento (i, j) depende de todos los valores que forman el triangulo inferior izquierdo, que aparece coloreado en la figura. El elemento $(1, n)$ contiene la energía total liberada por el pliegue de la molécula (la solución al problema).

Como se observa en la figura 4.11, para obtener el elemento (i, j) necesitamos tener calculados previamente todos los elementos $\{(a, b)/a \geq i \wedge b \leq j\}$; es decir, su triángulo inferior izquierdo, que aparece coloreado en la figura.

En el programa se recogen, como parámetros, el fichero que contiene la molécula de RNA y el volumen del *tile* con el que vamos a ejecutar (ancho y alto). El procesador p_0 lee la molécula de RNA del fichero y la envía al resto de los procesadores.

La función principal del paquete se denomina *fold()* y calcula el plegamiento de la molécula para obtener su estructura secundaria. Esta función recibe como parámetro una molécula de RNA y devuelve su estructura secundaria y la energía liberada por el pliegue asociado a la molécula. En la figura 4.12 mostramos un pseudocódigo del algoritmo que realiza esta función: primero se procesan los triángulos, cada procesador calcula de forma secuencial la parte de la matriz asociada a su triángulo (fase de pre-

procesamiento). Una vez calculados, cada procesador envía al resto de las máquinas los datos obtenidos (línea 7) y, a continuación, los cuadrados se calculan de forma paralela mediante un *pipeline* (línea 8). El último paso del algoritmo consiste en recuperar el pliegue de la estructura secundaria (línea 11). Esta última parte del problema no la hemos paralelizado porque sólo requiere un tiempo de cómputo lineal.

```
 1 float fold(char *string, char *structure) {
 2    Inicialización
 3
 4          // Código paralelo
 5    Inicio de medición del tiempo
 6    Se calculan los triángulos
 7    Intercambio de triángulos entre todos los procesadores
 8    Se procesan los cuadrados
 9    Fin de la medición del tiempo
10
11    Recuperación del pliegue de la estructura secundaria (bactracking)
12 }
```

Figura 4.12: Código de la rutina *fold()* que obtiene la estructura secundaria de una molécula de RNA a partir de la secuencia de la molécula.

En las siguientes subsecciones vamos a comentar los diferentes tipos de paralelizaciones que hemos realizado: la fase de preprocesamiento, la paralelización de un cuadrado cuando trabajamos únicamente con dos procesadores, y las tres estrategias utilizadas para calcular de forma paralela los cuadrados cuando resolvemos el problema con p procesadores.

4.6.1. Paralelización de los triángulos

El primer paso de la paralelización ha consistido en la división del trabajo en triángulos. Dividimos la matriz generando tantos triángulos como procesadores tengamos disponibles para resolver el problema. En la figura 4.13 podemos ver el caso más simple, cuando solamente trabajan 2 procesadores.

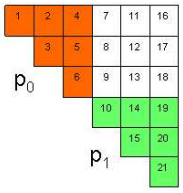

Figura 4.13: División de la matriz triangular en 2 triángulos. Se asigna a cada procesador uno de los triángulos para que sean calculados de forma paralela.

Como ha sido comentado en la sección 4.4, la matriz triangular se almacena en un vector denominado ***Tvector***. En la figura 4.14 mostramos el contenido del *Tvector* correspondiente a la matriz triangular anterior y representamos por los mismos colores los elementos que forman parte del triángulo que calcula cada procesador.

Cuando los procesadores finalizan el cálculo de su triángulo se intercambian la información obtenida. De esta manera, al terminar la fase de preprocesamiento, todos los procesadores disponen de todos los datos calculados. En las figuras 4.13, 4.14 y 4.15 mostramos el proceso de cálculo y empaquetado de los triángulos.

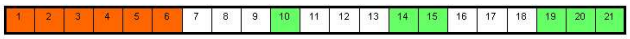

Figura 4.14: Representación de la matriz triangular en un vector auxiliar (*Tvector*). Los elementos que forman parte de los triángulos aparecen coloreados.

Figura 4.15: Empaquetar un triángulo para su transmisión: Izquierda: *Tpaquete* asociado al primer triángulo (procesador p_0). Derecha: *Tpaquete* asociado al segundo triángulo (procesador p_1).

Inicialmente disponemos de la matriz donde hemos calculado los triángulos (figura 4.13). Se puede ver en la figura 4.14 que, mientras que todos los elementos del triángulo asociado al procesador p_0 están almacenados de forma contigua, el triángulo correspondiente a p_1 tiene sus elementos dispersos por el *Tvector*. Para poder enviar la información de los triángulos a los restantes procesadores se empaquetan los datos de forma contigua, utilizando una estructura de datos denominada ***Tpaquete***. En la figura 4.15-Izquierda se muestra el *Tpaquete* asociado al triángulo del procesador p_0, en este caso el *Tpaquete* es una parte del *Tvector*, con sus elementos almacenados en el mismo orden. En la figura 4.15-Derecha se observa el *Tpaquete* correspondiente al triángulo de p_1. Ahora sí se observa que los datos del triángulo se encuentran almacenados de forma consecutiva.

Por último, mostramos en la figura 4.16 la generación de triángulos en una ejecución con mayor número de procesadores. En la figura se observa la subdivisión del trabajo para una molécula de tamaño 16 entre 4 procesadores.

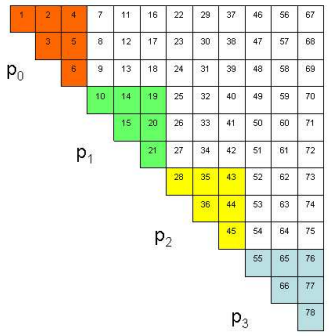

Figura 4.16: Fase de preproceso para resolver el problema para una molécula de tamaño 16 entre 4 procesadores: se generan 4 triángulos.

4.6.2. Paralelización de los cuadrados utilizando 2 procesadores

Una vez finalizada la fase de preprocesamiento, debemos paralelizar el dominio de iteraciones restante. La primera aproximación que hemos desarrollado considera que trabajamos únicamente con 2 procesadores. Esto significa que existe solamente un cuadrado a calcular. En esta sección comentamos las diferentes maneras de distribuir el trabajo entre los 2 procesadores y de realizar las comunicaciones para reducir el tiempo de ejecución.

La aproximación más simple podría consistir en dividir el cuadrado en 2 partes iguales y asignarle una a cada procesador (figura 4.17).

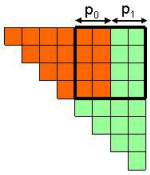

Figura 4.17: Paralelización de 1 cuadrado empleando 2 procesadores. Utilizamos la aproximación más simple, dividimos el cuadrado en 2 partes iguales y asignamos uno a cada procesador.

En el cálculo de los elementos del cuadrado de la figura 4.17, sí se producen dependencias entre los datos repartidos entre los dos procesadores. El procesador p_0 tiene disponibles todos los datos que necesita antes de comenzar el cálculo de sus elementos, pero el procesador p_1 debe esperar por los datos calculados por p_0.

En esta primera aproximación consideramos que cada vez que p_0 calcula un valor lo transmite a p_1 (todos los elementos etiquetados con *OUT* en la figura 4.18). El procesador p_1 únicamente recibe datos antes de calcular la primera columna asignada (los elementos etiquetados con *IN* en la columna 13 de la figura 4.18). Cuando p_1 recibe el elemento inmediatamente anterior al que está calculando (el elemento de la misma fila y columna 12 en la figura 4.18) puede asegurarse que, previamente, también ha recibido todos los valores anteriores y puede comenzar a procesar todos los elementos de esa fila.

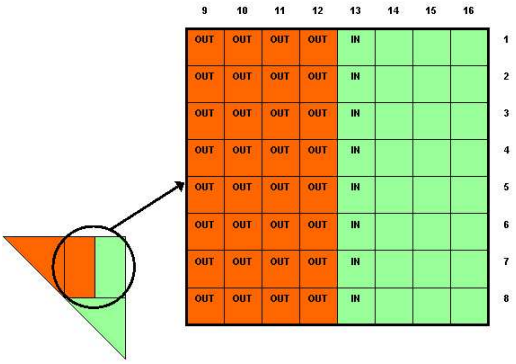

Figura 4.18: Comunicaciones entre los dos procesadores: cada vez que el procesador p_0 calcula un valor lo transmite a p_1. El procesador p_1 solamente recibe datos antes de calcular la primera columna asignada.

Un método para mejorar esta aproximación consiste en utilizar un *buffer*, que almacene los valores a transmitir, hasta que se haya calculado un número adecuado de ellos. De esta forma reducimos el número de comunicaciones establecidas para resolver el problema. La figura 4.19 muestra los envíos realizados por el procesador p_0 para transmitir los datos calculados al procesador p_1, cuando utilizamos un *buffer* para las comunicaciones de tamaño 7. Hemos asignado un mismo color a las celdas que son transmitidas en el mismo paquete.

En la figura 4.19 se ha marcado con *OUT* el momento en el que se produce una transmisión física: cuando se calculan 7 elementos, se produce un envío físico. Cuando el procesador p_0 termina el cálculo de su bloque vacía el contenido del *buffer*, aunque el número de elementos en el *buffer* no haya llegado al máximo. Por ejemplo, en la figura 4.19, se observa que el último paquete se envía al procesador p_1 únicamente con 4 elementos en el *buffer*, en lugar de los 7 elementos que contienen los paquetes

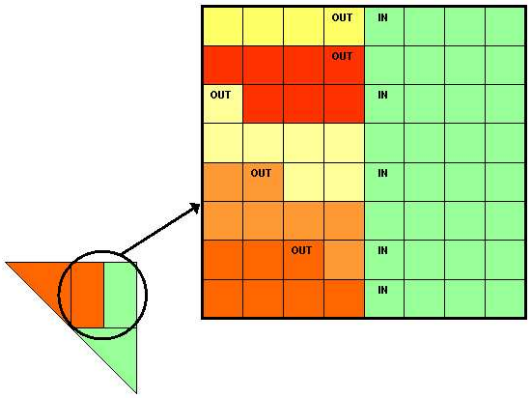

Figura 4.19: Comunicaciones entre los dos procesadores con un *buffer* de transmisión de tamaño 7. Asignamos un mismo color a todas los elementos que son transmitidos en el mismo paquete.

anteriores. En la figura se observa el reducido número de comunicaciones que se realizan en este caso, si lo comparamos con los envíos realizados en la figura 4.18.

Hasta ahora hemos considerado la distribución por bloques del trabajo entre los dos procesadores. Pero esta asignación de las columnas a los procesadores no tiene que ser óptima: podemos utilizar una distribución cíclica pura (figura 4.20-a), una distribución cíclica por bloques (figura 4.20-b) o una distribución por bloques (figura 4.20-c).

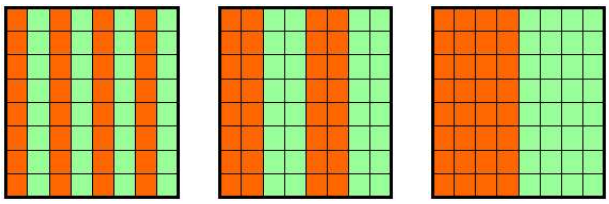

Figura 4.20: Diferentes distribuciones de trabajo para una misma matriz. a) Distribución cíclica pura. b) Distribución cíclica por bloques. c) Distribución por bloques.

4.6.3. Paralelización de los cuadrados utilizando p procesadores

En esta sección consideramos la situación general donde trabajamos con p procesadores y existe más de un rectángulo. Definimos 3 estrategias para su paralelización.

- Estrategia horizontal no sincronizada.
- Estrategia vertical no sincronizada.
- Estrategia horizontal sincronizada.

En la figura 4.21 mostramos un ejemplo de la matriz triangular para resolver el problema de predecir la estructura secundaria de una molécula de 16 bases, utilizando 4 procesadores. En la figura se muestran los elementos calculados por cada procesador con un color diferente. Se observan los triángulos calculados en la fase de preprocesamiento y los rectángulos, que son procesados con una distribución cíclica pura.

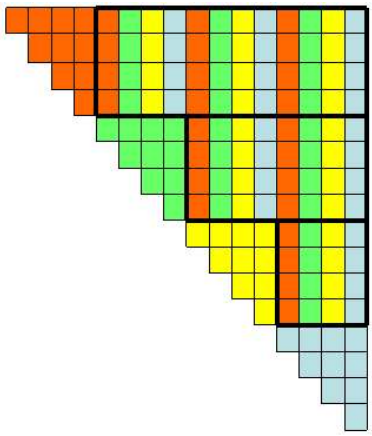

Figura 4.21: Visión global de la paralelización de los cuadrados con 4 procesadores. Los elementos calculados por cada procesador presentan un color diferente.

En la figura 4.21 se puede observar que hay un conjunto de elementos (la última columna en el ejemplo), que sólo las posee el último procesador. Bajo determinadas circunstancias este comportamiento es correcto. Sin embargo, hay combinaciones de parámetros para los que es necesario que todos los procesadores dispongan de todos los datos calculados, una vez que haya finalizado el procesamiento del rectángulo. Para conseguirlo se utiliza un **proceso de sincronización** que consiste en realizar una iteración más del algoritmo (realizando sólo los envíos y recepciones), para garantizar que todos los procesadores posean todos los datos. En el ejemplo de la figura 4.21, este proceso de sincronización no es necesario puesto que un procesador siempre va a tener los datos inferiores que necesita. Esto sucede porque el mismo procesador tiene asignada la misma columna para todos los rectángulos.

Planificación horizontal frente a planificación vertical

Esta clasificación especifica el orden en que se evalúan los cuadrados en la matriz y la forma en la que se agrupan estos cuadrados para dar lugar a los rectángulos. En la figura 4.22 se muestran estas 2 estrategias. En la planificación horizontal se agrupan los cuadrados por filas para generar los rectángulos e, internamente, se procesan de izquierda a derecha. En la planificación vertical, se agrupan por columnas y, dentro de los rectángulos, se calculan los cuadrados de abajo a arriba.

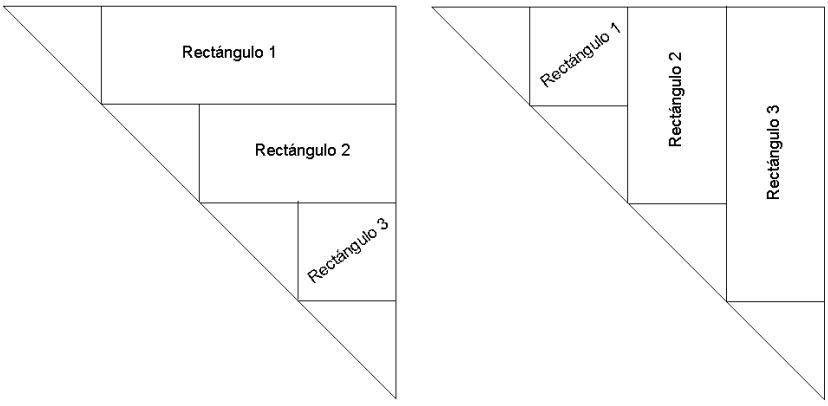

Figura 4.22: Planificación de los cuadrados. Izquierda: Planificación horizontal: los cuadrados se agrupan por filas. Derecha: Planificación vertical: los cuadrados se agrupan por columnas.

Estrategia horizontal no sincronizada

Es la estrategia más sencilla, aunque solamente puede aplicarse bajo unas condiciones de número de procesadores y de ancho del *tile* determinadas. En la figura 4.23-Izquierda presentamos un ejemplo de la forma de trabajar en esta estrategia. La sincronización no es necesaria debido a las características de ancho del *tile*, número de procesadores y longitud de la molécula. Esto permite resolver el problema sin que todos los procesadores requieran disponer de todos los datos.

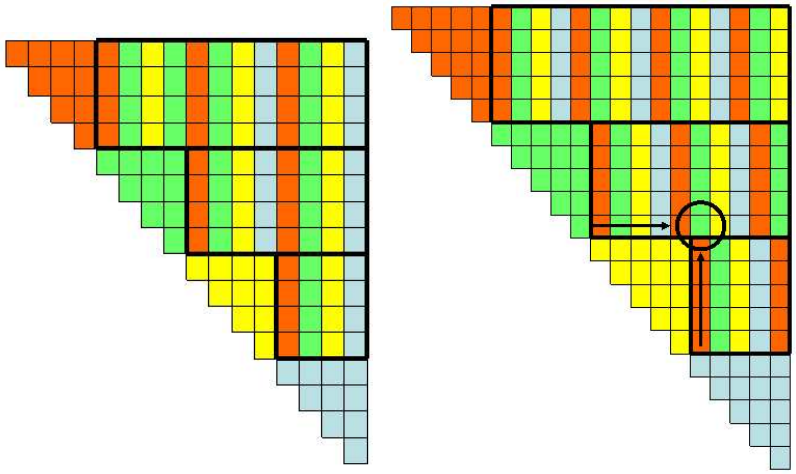

Figura 4.23: Condiciones válidas y no válidas para la estrategia. Izquierda: La relación entre el ancho del *tile*, número de procesadores y la longitud de la molécula cumplen las condiciones de validación. Derecha: No es posible resolver el problema porque existen dependencias no resueltas.

En la figura 4.23-Derecha mantenemos el mismo número de procesadores y ancho del *tile* pero modificamos el tamaño del problema y utilizamos una molécula de tamaño 20. Ahora sí es necesaria la sincronización, puesto que no disponer de todos los datos imposibilita el cálculo de los cuadrados. Como se observa en la figura, el procesador p_1 no puede calcular el elemento en el interior del círculo, al no tener todos los datos que necesita (su triángulo inferior izquierdo).

Las condiciones necesarias para poder aplicar esta estrategia son las siguientes:

- El problema debe ser "perfecto", **sin residuos**, o lo que es lo mismo, se tiene que cumplir que $n \bmod p = 0$.
- No pueden existir bandas residuales; es decir, bandas donde no trabajen todos los procesadores. La longitud de una banda es $p * y$, por lo que se tiene que cumplir que $(n/p) \bmod (p * y) = 0$.

Comprobamos estas dos condiciones para los dos ejemplos de la figura 4.23. En el diagrama de la izquierda tenemos una molécula de 16 bases ($n = 16$) y trabajamos con 4 procesadores ($p = 4$) y ancho del *tile* 1 ($y = 1$). En estas condiciones se cumple la primera condición ($16 \bmod 4 = 0$) y también la segunda ($4 \bmod 4 = 0$). Sin embargo, para la figura 4.23-Derecha, mantenemos el número de procesadores ($p = 4$) y el ancho del *tile* ($y = 1$) pero aumentamos el tamaño de la molécula ($n = 20$). En este caso sí se cumple la primera condición ($20 \bmod 4 = 0$) pero no se cumple la segunda ($5 \bmod 4 \neq 0$).

Estrategia vertical no sincronizada

En esta estrategia realizamos una planificación vertical (figura 4.22-Derecha) y no realizamos proceso de sincronización al finalizar el cómputo de cada cuadrado. Por tanto, igual que en la estrategia horizontal no sincronizada, hay restricciones al ancho del *tile* y al número de procesadores (se tienen que cumplir las mismas condiciones que en el caso anterior).

Estrategia horizontal sincronizada

Esta estrategia es igual que la versión horizontal no sincronizada, con la diferencia de que al finalizar cada rectángulo se produce la sincronización entre los procesadores. Esto produce una parada del *pipeline*, pues hay que vaciarlo para que todos los procesadores reciban todos los datos. Mediante esta estrategia no tenemos limitaciones a la hora de elegir el ancho del *tile* ni el número de procesadores. Sin embargo, sí se introduce una cierta sobrecarga como consecuencia de la sincronización. Con esta estrategia es posible resolver problemas como el de la figura 4.23-Derecha.

4.7. Resultados computacionales: tamaño constante del *tile*

En esta sección se muestra los resultados computacionales obtenidos con las tres versiones presentadas en la sección 4.6. Antes vamos a comentar las características de la plataforma paralela sobre la que hemos realizado las pruebas. Como ya hemos comentado, en este capítulo trabajamos con un sistema heterogéneo según la definición dada en la sección 1.2: un programa heterógeneo ejecutado sobre una plataforma homogénea.

Plataforma homogénea

En este capítulo hemos utilizado una plataforma de pruebas diferente a la empleada en los capítulos 2 y 3: la máquina *SGI Origin 3800* del Centro de Investigación de Energías, Medio Ambiente y Tecnología (CIEMAT) del Ministerio de Ciencia y Tecnología español, denominada *jen50*, que dispone de 128 procesadores *MIPS R14000* a 600 MHz, con 128 Gbytes de memoria y una arquitectura *ccNuma*. Es una plataforma homogénea puesto que todos los procesadores disponen de las mismas características. Los 128 procesadores se encuentran distribuidos en 32 nodos de 4 procesadores cada uno, los nodos están organizados en una topología de hipercubo (figura 4.24). En cada uno de los vértices de este hipercubo existe un *router* que conecta 4 nodos con el resto del sistema.

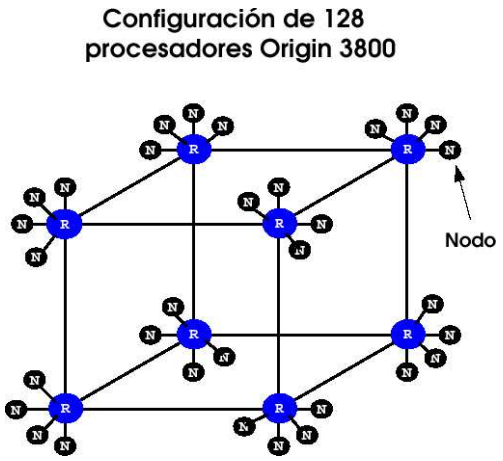

Figura 4.24: Arquitectura de la *SGI-Origin 3800*: los procesadores se encuentran distribuidos en 32 nodos de 4 procesadores, conectados mediante una topología de hipercubo.

En la figura 4.25 se observa uno de los vértices del hipercubo, detallando principalmente el contenido de un nodo. Se observa que cada nodo contiene 4 procesadores y cada procesador dispone de una memoria caché integrada. Además existe otra memoria caché externa a los procesadores pero interna al nodo. Los 4 nodos se encuentran conectados mediante un *router* que, a su vez, se conecta con los *routers* de los restantes vértices del hipercubo. Es importante resaltar la diferencia de velocidad en los *buses* de

comunicaciones existentes. Como se observa en la figura, la comunicación interna al nodo será más rápida que con procesadores situados en otros nodos.

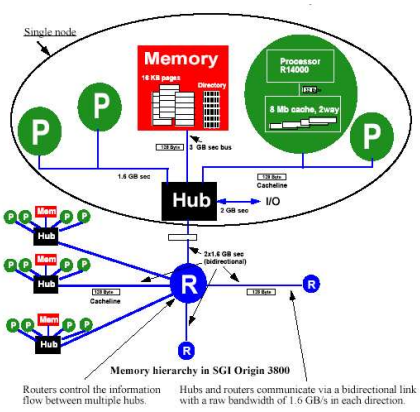

Figura 4.25: Arquitectura de la *SGI-Origin 3800*: Detalle interno de uno de los nodos en uno de los vértices del hipercubo.

La máquina está gestionada por el sistema operativo *IRIX* de 64 bits, que permite a los usuarios trabajar con *jen50* como si fuera un único recurso computacional. Además, es un sistema operativo multiproceso y *multithread*. Como ya ha sido comentado la memoria compartida es de tipo *cc-NUMA* y sus características son las siguientes:

- Baja latencia tanto en el acceso a la memoria local como a la remota. Esto permite la **escalabilidad del sistema**.
- **Gestión eficiente de los recursos:** La memoria se muestra como un recurso único, no hay que programar un modelo de datos con restricciones de capacidad para problemas de gran consumo de memoria.
- **Flexibilidad de crecimiento:** Esta arquitectura es compatible con una gran variedad de entornos, procesadores y componentes. Soporta procesadores tanto *MIPS* como *Intel*, esquemas de interconexión *PCI* o *XIO* y con sistema operativo *IRIX* o *Linux*.

Resultados obtenidos en la ejecución

Inicialmente ejecutamos los problemas de forma secuencial para comparar posteriormente estos resultados con los obtenidos al ejecutar el problema en paralelo. En la tabla 4.1 mostramos los resultados para 7 tamaños distintos de problemas: hemos generado de forma aleatoria cadenas de *RNA* para los tamaños 256, 512, 1024, 2048, 3000, 4096 y 8192.

Todas las ejecuciones realizadas en este capítulo, tanto las secuenciales como las paralelas, han sido repetidas 5 veces para evitar perturbaciones producidas como consecuencia de la ejecución en modo no exclusivo. Los resultados que se muestran son los tiempos mínimos de las cinco ejecuciones.

A continuación hemos realizado la experiencia computacional con el programa paralelo para las tres versiones del código: versión horizontal no sincronizada, vertical no sincronizada y horizontal sincronizada. En todos los casos, el alto y ancho del *tile* se le pasan al programa como parámetros. Hemos realizado una ejecución intensiva para los problemas de tamaño 256, 512, 1024, 2048 y 4096, variando los valores de ancho y alto del *tile*, de forma que $y \in [1, \frac{n}{p^2}]$ y los valores de x dependen del problema considerado. Mediante esta experiencia computacional, pretendemos encontrar el tamaño del *tile* que minimiza el tiempo de resolución del problema para, posteriormente, utilizar estos valores en la validación de nuestros modelos (secciones 4.9 y 4.10).

Tabla 4.1: Tiempos secuenciales obtenidos en la ejecución del problema de la predicción de la estructura secundaria del RNA para diferentes tamaños de moléculas.

Tamaño de problema	Tiempo de ejecución
256	0.26
512	1.22
1024	5.97
2048	30.49
3200	107.36
4096	219.39
8192	1565.63

Versión horizontal no sincronizada.

En el problema de tamaño más pequeño ($n = 256$) variamos el valor de x entre 1 y 100. En la tabla 4.2 mostramos los resultados solamente para algunas de las 1500 combinaciones de número de procesadores, y y x utilizadas en las ejecuciones. Los resultados presentados son únicamente una muestra, que ha sido seleccionada de forma que se encuentran los valores máximo y mínimo que puede tomar un parámetro y valores intermedios equidistantes, para que cubra todo el rango de valores. En ésta y las siguientes tablas no se busca localizar el valor mínimo, sino mostrar la variación de resultados que se produce con distintas combinaciones de los parámetros. También mostraremos, de forma más visual, estos resultados mediante gráficas, para algunos de los problemas. Las combinaciones que necesitan el menor tiempo de ejecución se mostrarán en una tabla posterior (sección 4.9).

El número de procesadores que hemos empleado para resolver problemas ha sido siempre potencia de 2, igual que los tamaños de los problemas. Esto se debe a que el valor $\frac{n}{p}$ debe ser un valor exacto para poder realizar la fase de preprocesamiento y dividir el espacio de iteraciones en triángulos. Por tanto, el número de procesadores empleados han sido $p = 2, 4$ y 8. Los valores utilizados para el ancho del *tile* son también potencia de 2 a fin de que se cumplan las condiciones necesarias para aplicar esta versión del algoritmo. A pesar de que con $p = 16$ se cumple también que $\frac{n}{p} = \frac{256}{16} = 16$ es un valor exacto, si aplicamos la segunda condición, se tiene que cumplir que $\frac{n/p}{py} = \frac{1}{y}$ también debe ser un valor exacto. Por lo tanto, sólo podríamos usar un valor $y = 1$ y hemos descartado utilizar este número de procesadores.

En la tabla 4.2 la columna y representa el ancho del *tile* y la columna x es el alto del *tile*. También mostramos el tiempo real de la ejecución en segundos y la aceleración obtenida. En la cabecera de cada bloque se muestra el número de procesadores utilizados. Además, en la parte superior de la tabla se puede ver el tamaño del problema y el tiempo secuencial requerido para resolverlo. Para cada valor de p, marcamos en negrita la combinación de parámetros que requiere el menor tiempo de ejecución (de entre la selección mostrada en la tabla).

En la tabla 4.2 se observa cómo varían los tiempos de ejecución dependiendo de los parámetros utilizados. Esto indica la importancia de una selección adecuada de parámetros al ejecutar un programa. Por ejemplo, para 2 procesadores la aceleración conseguida en los resultados presentados se encuentra entre 0.73 y 1.60. Es decir, existen varias combinaciones cuyo tiempo de ejecución es peor que el tiempo secuencial; sin embargo, también existen otras combinaciones donde sí se consigue una aceleración superior a 1. Es necesario recordar que en estas tablas no se encuentran necesariamente las combinaciones óptimas de parámetros, simplemente es una muestra de los resultados obtenidos. Además, hay que resaltar el hecho de que pueden encontrarse aceleraciones idénticas para valores de tiempo de ejecución diferentes; esto es debido al redondeo que hemos realizado de las cantidades a únicamente dos cifras decimales.

Al aumentar el número de procesadores se observa que la aceleración máxima alcanzada también aumenta. Sin embargo, es un problema muy pequeño y los tiempos de ejecución son muy reducidos, esto explica el alto número de resultados peores que el tiempo del algoritmo secuencial: por ejemplo, para 8 procesadores sólo 3 de las 18 aceleraciones mostradas son superiores a 1.

Tabla 4.2: Tiempos de ejecución del programa paralelo de la predicción de la estructura secundaria del RNA para una molécula de tamaño 256, utilizando la versión horizontal no sincronizada.

Tamaño de problema: 256 - Tiempo secuencial: 0.260							
Número de procesadores: $p = 2$							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	0.330	0.79	1	20	0.272	0.96
1	40	0.270	0.96	1	60	0.277	0.94
1	80	0.295	0.88	1	100	0.355	0.73
2	1	0.273	0.95	2	20	0.212	1.23
2	40	0.216	1.20	2	60	0.220	1.18
2	80	0.237	1.10	2	100	0.283	0.92
4	1	0.218	1.19	4	20	0.193	1.35
4	40	0.191	1.36	4	60	0.196	1.32
4	80	0.218	1.19	4	100	0.248	1.05
16	1	0.176	1.48	16	20	0.169	1.54
16	40	0.180	1.44	16	60	0.177	1.47
16	80	0.196	1.33	16	100	0.219	1.19
32	1	0.168	1.55	32	20	0.168	1.55
32	40	0.178	1.46	32	60	0.180	1.44
32	80	0.196	1.32	32	100	0.219	1.19
64	**1**	**0.163**	**1.60**	64	20	0.173	1.50
64	40	0.184	1.42	64	60	0.198	1.31
64	80	0.210	1.24	64	100	0.230	1.13
Número de procesadores: $p = 4$							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	0.299	0.87	1	20	0.212	1.22
1	40	0.360	0.72	1	60	0.543	0.48
1	80	0.589	0.44	1	100	0.604	0.43
2	1	0.187	1.39	2	20	0.163	1.60
2	40	0.281	0.93	2	60	0.408	0.64
2	80	0.444	0.59	2	100	0.445	0.58
4	1	0.149	1.75	4	20	0.138	1.88
4	40	0.239	1.09	4	60	0.329	0.79
4	80	0.371	0.70	4	100	0.369	0.70
8	1	0.119	2.18	8	20	0.128	2.03
8	40	0.200	1.30	8	60	0.293	0.89
8	80	0.318	0.82	8	100	0.324	0.80
16	**1**	**0.104**	**2.49**	16	20	0.124	2.09
16	40	0.182	1.43	16	60	0.249	1.05
16	80	0.261	1.00	16	100	0.276	0.94
Número de procesadores: 8							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	0.183	1.42	1	20	0.438	0.59
1	40	0.931	0.28	1	60	0.815	0.32
1	80	0.812	0.32	1	100	0.855	0.30
2	1	0.149	1.74	2	20	0.336	0.77
2	40	0.542	0.48	2	60	0.588	0.44
2	80	0.611	0.43	2	100	0.666	0.39
4	**1**	**0.100**	**2.60**	4	20	0.262	0.99
4	40	0.435	0.60	4	60	0.461	0.56
4	80	0.494	0.53	4	100	0.511	0.51

También se observa que los mejores resultados se obtienen con valores altos de y y con valores bajos de x; los peores resultados se obtienen para $y = 1$ (distribución cíclica pura) y con tamaños altos de x. En la figura 4.26 se observa de forma gráfica las variaciones de tiempos de ejecución si fijamos 2 parámetros y variamos el tercero.

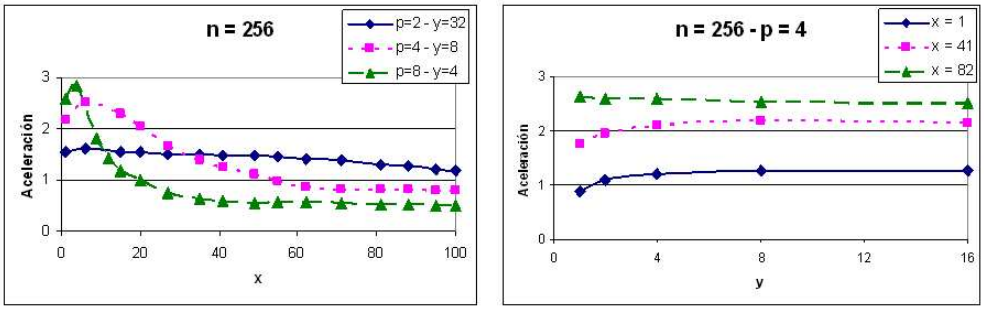

Figura 4.26: Aceleraciones obtenidas con el problema de tamaño 256 y la versión horizontal no sincronizada fijando 2 parámetros y variando el tercero. Izquierda: Fijamos número de procesadores y el alto del *tile* y modificamos el ancho del *tile*. Derecha: Fijamos número de procesadores y el ancho del *tile* y variamos su alto.

En la tabla 4.3 mostramos los resultados para el siguiente tamaño de problema ($n = 512$). En este caso hemos variado el valor de x entre 1 y 50. En este problema el número total de combinaciones utilizadas ha sido de 900, debido a la reducción de los valores de x a la mitad. Tampoco hemos ejecutado el problema con 16 procesadores puesto que únicamente podríamos utilizar dos valores distintos de ancho del *tile* (1 y 2). El formato de esta tabla es el mismo que el utilizado en la tabla 4.2.

En la tabla 4.3 se observa también que los tiempos de ejecución se reducen al incrementar el número de procesadores, la aceleración llega a 3.48 para 8 procesadores. Sin embargo, también existen varias combinaciones de parámetros que hacen que el tiempo de ejecución sea peor que el tiempo secuencial. De nuevo es necesario indicar la amplia variación de la aceleración dependiendo de los valores usados para el ancho y alto del *tile*, lo que convierte la elección de los parámetros en un aspecto fundamental de la ejecución en paralelo. Por último, si comparamos los resultados obtenidos para el tamaño de problema $n = 256$ con los resultados para el problema de tamaño $n = 512$, vemos que los peores resultados se siguen obteniendo con $y = 1$ aunque, en este caso, el valor de x varía: para 8 procesadores el peor tiempo se obtiene con un valor de x alto, mientras que para $y = 1$ y x pequeño se mejora el tiempo secuencial; para 4 procesadores con $y = 1$ la aceleración es prácticamente idéntica para $x = 1$ y para $x = 50$, para los valores intermedios se obtiene un tiempo equivalente al tiempo secuencial; para 2 procesadores, sin embargo, todos los valores de x producen una aceleración similar para $y = 1$. Si comprobamos las aceleraciones máximas, podemos observar que para 4 y 8 procesadores el efecto es el mismo que en el problema de tamaño 256: se obtienen para los valores más altos de y, con un valor de x pequeño. En cambio, para 2 procesadores esta situación es diferente, hemos ejecutado con anchos del *tile* hasta 128 pero los resultados obtenidos para este valor son mucho peores que los tiempos con $y = 32$.

En el problema de tamaño $n = 1024$ utilizamos valores del alto del *tile* entre 1 y 30. Ahora ejecutamos también con $p = 16$, puesto que ya podemos utilizar tres valores de ancho del *tile* ($y = 1, 2$ y 4). Sin embargo, aunque $\frac{1024}{32} = 32$ cumple con la primera condición, sólo podríamos utilizarlo con $y = 1$ y lo descartamos. El número de ejecuciones realizadas con combinaciones distintas han sido 720. El formato utilizado en la tabla 4.4 es el mismo que el de las tablas anteriores.

De nuevo comprobamos, en la tabla 4.4, la importancia en la elección de los parámetros. Por ejemplo, para 16 procesadores se observa que el mejor resultado se obtiene para $y = 4$ y $x = 1$, con una aceleración de 3.79, para $y = 2$ y el mismo valor de x se llega a una aceleración de 2.36. Para el resto de los 10 resultados presentados, la aceleración alcanzada es inferior a 2 y en la mayoría de ellos incluso se encuentra por debajo del tiempo secuencial.

Tabla 4.3: Tiempos de ejecución del programa paralelo de la predicción de la estructura secundaria del RNA para una molécula de tamaño 512, utilizando la versión horizontal no sincronizada.

Tamaño de problema: 512 - Tiempo secuencial: 1.22							
Número de procesadores: 2							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	2.05	0.60	1	10	1.76	0.70
1	20	1.72	0.71	1	30	1.71	0.72
1	40	1.70	0.72	1	50	1.72	0.71
4	1	1.12	1.09	4	10	0.99	1.24
4	20	0.98	1.25	4	30	0.99	1.24
4	40	0.97	1.26	4	50	0.97	1.26
8	1	0.95	1.29	8	10	0.85	1.44
8	20	0.85	1.44	8	30	0.84	1.45
8	40	0.84	1.45	8	50	0.86	1.43
32	1	0.80	1.53	32	10	0.77	1.59
32	**20**	**0.75**	**1.63**	32	30	0.78	1.57
32	40	0.76	1.61	32	50	0.77	1.59
64	1	0.86	1.43	64	10	0.82	1.48
64	20	0.82	1.49	64	30	0.83	1.48
64	40	0.83	1.48	64	50	0.84	1.46
128	1	1.05	1.16	128	10	1.04	1.18
128	20	1.03	1.19	128	30	1.00	1.22
128	40	1.05	1.17	128	50	1.04	1.18
Número de procesadores: 4							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	1.61	0.76	1	10	1.24	0.99
1	20	1.14	1.08	1	30	1.16	1.05
1	40	1.43	0.86	2	50	1.60	0.77
4	1	0.72	1.70	4	10	0.60	2.06
4	20	0.59	2.07	4	30	0.61	2.01
4	40	0.73	1.67	4	50	0.85	1.45
8	1	0.56	2.20	8	10	0.50	2.43
8	20	0.52	2.35	8	30	0.55	2.22
8	40	0.64	1.92	8	50	0.74	1.61
16	1	0.52	2.37	16	10	0.51	2.41
16	20	0.48	2.57	16	30	0.49	2.50
16	40	0.56	2.17	16	50	0.64	1.91
32	**1**	**0.45**	**2.73**	32	10	0.46	2.65
32	20	0.47	2.63	32	30	0.50	2.44
32	40	0.56	2.20	32	50	0.63	1.94
Número de procesadores: 8							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	1.02	1.21	1	10	0.83	1.48
1	20	1.55	0.79	1	30	2.43	0.50
1	40	2.80	0.44	2	50	3.53	0.35
4	1	0.48	2.57	4	10	0.47	2.60
4	20	0.80	1.53	4	30	1.13	1.08
4	40	1.53	0.80	4	50	1.87	0.66
8	**1**	**0.35**	**3.48**	8	10	0.39	3.12
8	20	0.82	1.50	8	30	0.91	1.34
8	40	1.11	1.10	8	50	1.42	0.86

Tabla 4.4: Tiempos de ejecución del programa paralelo de la predicción de la estructura secundaria del RNA para una molécula de tamaño 1024, utilizando la versión horizontal no sincronizada.

Tamaño de problema: 1024 - Tiempo secuencial: 5.97							
Número de procesadores: 2							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	16.23	0.37	1	10	14.79	0.40
1	20	14.82	0.40	1	30	14.55	0.41
8	1	5.24	1.14	8	10	4.92	1.21
8	20	4.89	1.22	8	30	4.94	1.21
32	1	4.15	1.44	32	10	4.02	1.48
32	**20**	**3.98**	**1.50**	32	30	3.98	1.50
64	1	4.11	1.45	64	10	4.069	1.47
64	20	4.00	1.49	64	30	4.07	1.47
128	1	4.53	1.32	128	10	4.44	1.34
128	20	4.47	1.33	128	30	4.42	1.35
256	1	5.33	1.12	256	10	5.37	1.11
256	20	5.28	1.13	256	30	5.35	1.12
Número de procesadores: 4							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	12.14	0.49	1	10	9.41	0.63
1	20	9.60	0.62	1	30	9.38	0.64
8	1	3.19	1.87	8	10	2.88	2.07
8	20	2.87	2.08	8	30	2.90	2.06
16	1	2.70	2.21	16	10	2.42	2.47
16	20	2.41	2.48	16	30	2.46	2.43
32	1	2.38	2.51	32	10	2.19	2.72
32	**20**	**2.22**	**2.69**	32	30	2.22	2.69
64	1	2.63	2.27	64	10	2.49	2.39
64	20	2.57	2.32	64	30	2.66	2.24
Número de procesadores: 8							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	6.47	0.92	1	10	5.69	1.05
1	20	6.62	0.90	1	30	9.92	0.60
4	1	2.81	2.12	4	10	2.28	2.62
4	20	2.70	2.21	4	30	3.85	1.55
8	1	2.07	2.89	8	10	1.78	3.37
8	20	2.39	2.50	8	30	2.88	2.07
16	**1**	**1.61**	**3.71**	16	10	1.72	3.48
16	20	1.78	3.34	16	30	2.51	2.38
Número de procesadores: 16							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	4.99	1.20	1	10	7.45	0.80
1	20	13.50	0.44	1	30	19.84	0.30
2	1	2.53	2.36	2	10	4.41	1.35
2	20	8.43	0.71	2	30	12.81	0.47
4	**1**	**1.57**	**3.79**	4	10	3.42	1.75
4	20	6.04	0.99	4	30	8.22	0.73

Utilizando únicamente los resultados presentados en la tabla, la mejor opción para resolver el problema parece ser con 16 procesadores, aunque la diferencia con los tiempos usando 8 procesadores es mínima para haber duplicado el número de procesadores. Para conocer realmente el número de procesadores óptimo tendremos que tener en cuenta todos los resultados obtenidos en las ejecuciones (sección 4.9).

Para este tamaño de problema y para un número de procesadores de 2 ó 4 se observa que el valor de y para la mejor ejecución ya no es el valor más alto, mientras que para 8 y 16 procesadores el valor máximo de y sigue siendo la mejor elección. Los peores resultados se siguen obteniendo para las distribuciones cíclicas puras ($y = 1$) con cualquier número de procesadores.

En la figura 4.27 mostramos los resultados obtenidos cuando fijamos el número de procesadores y el valor de y y variamos x y cuando fijamos el número de procesadores y x y modificamos el valor de y.

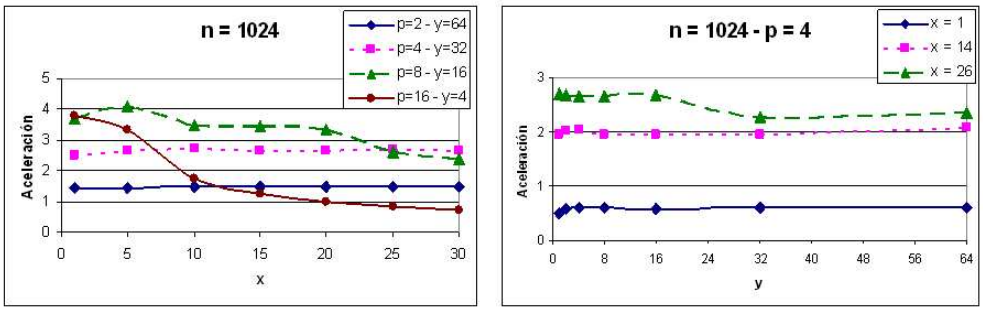

Figura 4.27: Aceleraciones obtenidas para el problema de tamaño 1024 con la versión horizontal no sincronizada fijando 2 parámetros y variando el tercero. Izquierda: Fijamos número de procesadores y x. Derecha: Fijamos número de procesadores y el valor de y.

En la tabla 4.5 mostramos una selección de los 420 resultados calculados para el problema de tamaño $n = 2048$. Los valores de x utilizados varían entre 1 y 15 y el número de procesadores entre 2 y 16 (únicamente para potencias de 2).

Las ejecuciones para el problema de tamaño 2048 se encuentran en un rango de valores de x más reducido que en los problemas anteriores, solamente entre 1 y 15. Ahora observamos que los tiempos más reducidos se alcanzan para algunos de los valores más altos de x utilizados, entre 10 y 15 elementos. De nuevo, los peores resultados se consiguen con la distribución cíclica pura, en ninguno de estos casos se consigue reducir el tiempo empleado por el algoritmo secuencial.

En la sección 4.9 comprobaremos la combinación donde se alcanza el tiempo mínimo para la ejecución de este problema, sin embargo, para los resultados presentados, resulta mejor resolver el problema con 8 procesadores en lugar de utilizar 16.

Por último, en la tabla 4.6 se encuentran los resultados para el problema de mayor tamaño ($n = 4096$). En este caso hemos realizado únicamente 350 ejecuciones con combinaciones diferentes de y y x al reducir x a valores entre 1 y 10. Aunque para este último tamaño de problema ya es posible emplear 32 procesadores con 3 valores de y ($y = 1$, 2 y 4), no mostramos los resultados obtenidos en la tabla por ser mucho peores que los conseguidos con menos procesadores.

Si comparamos los resultados presentados en la tabla 4.6 con los resultados de las tablas anteriores podemos observar que, a medida que incrementamos el tamaño del problema, también aumenta el valor del ancho del *tile* con el que debemos ejecutar el programa para obtener los mejores resultados. Como consecuencia, los tiempos de ejecución para los valores de y más reducidos son cada vez peores. Por ejemplo, para 2 procesadores con $y = 1$ la aceleración obtenida se mantiene constante en 0.12. Esto supone un tiempo de ejecución casi 9 veces peor que el tiempo secuencial. Además, existen valores de y mayores que 1 que siguen siendo peores que los resultados secuenciales (por ejemplo, para 2 procesadores con $y = 8$).

Tabla 4.5: Tiempos de ejecución del programa paralelo de la predicción de la estructura secundaria del RNA para una molécula de tamaño 2048, utilizando la versión horizontal no sincronizada.

Tamaño de problema: 2048 - Tiempo secuencial: 30.49							
Número de procesadores: 2							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	164.92	0.18	1	5	158.95	0.19
1	10	158.61	0.19	1	15	158.48	0.19
8	1	39.23	0.78	8	5	35.01	0.87
8	10	34.82	0.88	8	15	34.96	0.87
32	1	24.61	1.24	32	5	21.85	1.40
32	10	21.91	1.39	32	15	22.11	1.38
64	1	21.26	1.43	64	5	20.40	1.49
64	10	20.41	1.49	64	15	20.34	1.50
128	1	20.73	1.47	128	5	20.43	1.49
128	**10**	**20.19**	**1.51**	128	15	20.21	1.51
512	1	26.94	1.13	512	5	26.89	1.13
512	10	26.75	1.14	512	15	26.68	1.14
Número de procesadores: 4							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	108.29	0.28	1	5	101.37	0.30
1	10	99.66	0.31	1	15	99.28	0.31
8	1	22.07	1.38	8	5	21.26	1.43
8	10	20.68	1.47	8	15	20.84	1.46
16	1	16.21	1.88	16	5	15.34	1.99
16	10	15.49	1.97	16	15	15.19	2.01
32	1	13.40	2.28	32	5	13.20	2.31
32	10	12.92	2.36	32	15	12.95	2.35
64	**1**	**12.42**	**2.46**	64	5	12.70	2.40
64	10	12.54	2.43	64	15	12.10	2.52
Número de procesadores: 8							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	59.66	0.51	1	5	56.55	0.54
1	10	57.60	0.53	1	15	56.22	0.54
8	1	14.03	2.17	8	5	12.79	2.38
8	10	12.19	2.50	8	15	12.50	2.44
16	1	9.72	3.14	16	5	9.026	3.38
16	10	9.04	3.37	16	15	9.05	3.37
32	1	8.05	3.79	32	5	7.74	3.94
32	10	7.72	3.95	**32**	**15**	**7.58**	**4.02**
Número de procesadores: 16							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	38.39	0.79	1	5	32.06	0.95
1	10	35.32	0.86	1	15	53.07	0.57
4	1	11.93	2.56	4	5	11.27	2.70
4	10	13.00	2.35	4	15	18.10	1.68
8	1	8.94	3.41	**8**	**5**	**8.41**	**3.63**
8	10	9.19	3.32	8	15	12.21	2.50

Tabla 4.6: Tiempos de ejecución del programa paralelo de la predicción de la estructura secundaria del RNA para una molécula de tamaño 4096, utilizando la versión horizontal no sincronizada.

Tamaño de problema: 4096 - Tiempo secuencial: 219.39							
Número de procesadores: 2							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	1906.15	0.12	1	4	1876.75	0.12
1	7	1864.63	0.12	1	10	1854.37	0.12
8	1	345.26	0.64	8	4	322.46	0.68
8	7	318.53	0.69	8	10	317.68	0.69
64	1	138.15	1.59	64	4	127.62	1.72
64	7	127.51	1.72	64	10	130.39	1.68
128	1	130.17	1.69	128	4	116.58	1.88
128	7	119.97	1.83	128	10	117.02	1.87
256	1	117.34	1.87	256	4	115.48	1.90
256	**7**	**114.98**	**1.91**	256	10	116.11	1.89
1024	1	184.74	1.19	1024	4	183.86	1.19
1024	7	182.82	1.20	1024	10	184.56	1.19
Número de procesadores: 4							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	1226.62	0.18	1	4	1188.15	0.18
1	7	1178.27	0.19	1	10	1176.15	0.19
8	1	204.31	1.07	8	4	193.83	1.13
8	7	193.07	1.14	8	10	203.89	1.08
64	1	78.41	2.80	64	4	71.38	3.07
64	7	71.50	3.07	64	10	72.10	3.04
128	1	70.08	3.13	128	4	67.49	3.25
128	**7**	**66.52**	**3.30**	128	10	67.58	3.25
256	1	70.73	3.10	256	4	69.07	3.18
256	7	68.86	3.19	256	10	68.19	3.22
Número de procesadores: 8							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	665.67	0.33	1	4	643.57	0.34
1	7	704.71	0.31	1	10	644.85	0.34
8	1	124.61	1.76	8	4	112.91	1.94
8	7	114.84	1.91	8	10	112.94	1.94
16	1	75.62	2.90	16	4	76.98	2.85
16	7	75.07	2.92	16	10	73.86	2.97
64	1	44.12	4.97	64	4	44.48	4.93
64	**7**	**42.53**	**5.16**	64	10	42.59	5.15
Número de procesadores: 16							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	394.61	0.56	1	4	358.97	0.61
1	7	379.03	0.58	1	10	332.30	0.66
4	1	118.85	1.85	4	4	110.21	1.99
4	7	111.92	1.96	4	10	110.86	1.98
16	1	45.99	4.77	**16**	**4**	**41.23**	**5.32**
16	7	44.72	4.90	16	10	44.61	4.92

Con este problema de mayor tamaño conseguimos aceleraciones más altas: con 16 procesadores se alcanza una aceleración de 5.32; incluso con 2 procesadores estamos muy cerca de la aceleración máxima: para $y = 256$ y $x = 7$ la aceleración es de 1.91 frente a la aceleración de 2 que se debería obtener en una situación ideal. En la figura 4.28 se muestra de forma gráfica los resultados para el problema de mayor tamaño cuando fijamos 2 de los parámetros.

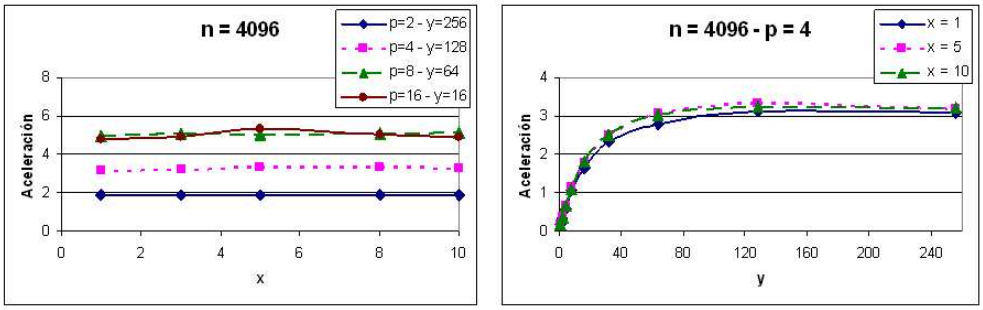

Figura 4.28: Aceleraciones obtenidas con el problema de tamaño 4096 y la versión horizontal no sincronizada fijando 2 parámetros y variando el tercero. Izquierda: Fijamos número de procesadores y x. Derecha: Fijamos número de procesadores y el valor de y.

Como se observa en las tablas anteriores, para los problemas pequeños con cualquier número de procesadores, los tiempos mínimos se obtienen en el tamaño más alto de y con el que es posible la ejecución. A medida que el tamaño de problema aumenta, esta situación deja de producirse cuando utilizamos un número reducido de procesadores, pero se mantiene cuando trabajamos con 8 ó 16. En la figura 4.29 se observan las aceleraciones máximas alcanzadas para cada combinación de tamaño de problema y número de procesadores. En la mayoría de los casos al aumentar cualquiera de los dos parámetros se incrementa el valor de la aceleración. Existen algunas excepciones a esta situación: por ejemplo, para $n = 2048$ la aceleración obtenida es menor que la aceleración para $n = 1024$ cuando ejecutamos con $p = 4$ y 16. Además, la aceleración para $n = 2048$ se reduce cuando ejecutamos con 16 procesadores frente a los resultados obtenidos cuando $p = 8$.

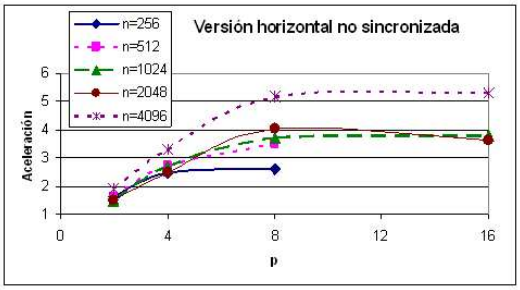

Figura 4.29: Aceleraciones máximas alcanzadas en las ejecuciones con la versión horizontal no sincronizada para cada tamaño de problema y número de procesadores.

Versión vertical no sincronizada

En esta subsección mostramos los resultados obtenidos con la versión vertical no sincronizada para las mismas configuraciones de n, p, y y x que para la versión horizontal no sincronizada. Las restricciones impuestas en cada problema a los valores posibles de número de procesadores y ancho del *tile*, los valores de x y el número de ejecuciones realizadas son también las mismas, así como el formato de las tablas empleado. En la tabla 4.7 se encuentran los resultados para el problema $n = 256$, en la tabla 4.8 para el

problema $n = 512$, en la tabla 4.9 los resultados para $n = 1024$, en la tabla 4.10 para $n = 2048$ y, por último, en la tabla 4.11 se muestran los resultados seleccionados para el tamaño más grande de problema ($n = 4096$).

Comparando los resultados del problema de tamaño 256 de la versión vertical no sincronizada (tabla 4.7) con los de la versión horizontal no sincronizada (tabla 4.2) observamos unos resultados similares: los mejores resultados se consiguen con los valores de y altos y de x pequeños, mientras que los peores resultados se obtienen con valores pequeños de y. Los tiempos de ejecución disminuyen al incrementar el número de procesadores, aunque existen combinaciones de valores de los parámetros para los cuales los resultados son peores que el tiempo de la ejecución secuencial, especialmente para un reducido número de procesadores.

Observamos también que los resultados obtenidos con la versión vertical no sincronizada, son similares a los presentados para la versión horizontal no sincronizada para el problema de tamaño 512. Las combinaciones de y y x donde se encuentran los mejores y peores resultados son prácticamente idénticos. Incluso los tiempos de ejecución son parecidos, las mayores diferencias se encuentran en 8 procesadores, donde la aceleración varía entre 0.35 y 3.48 para la versión horizontal no sincronizada (tabla 4.3), mientras que para la versión vertical no sincronizada (tabla 4.8) estos valores están entre 1.11 y 3.61.

En la tabla 4.9 se observa que los resultados para el problema $n = 1024$ son semejantes a los de la versión horizontal no sincronizada (tabla 4.4). Comparando estos datos, vemos que los resultados para 2 procesadores son prácticamente idénticos; mientras que, para el resto, los mejores y peores resultados presentan una aceleración un poco más alta para la versión vertical, excepto para el mejor resultado con 16 procesadores. Por ejemplo, para 8 procesadores se reduce el tiempo de ejecución de 1.61 segundos con la versión horizontal a 1.46 con la vertical. Al aumentar la aceleración conseguida para este número de procesadores, se mejora el tiempo conseguido con 16 procesadores.

En la tabla 4.10 se observan, de nuevo, resultados análogos a los de la versión horizontal no sincronizada. En este caso, el mejor resultado se consigue cuando ejecutamos con el mayor número posible de procesadores ($p = 16$). Esto se debe a que la aceleración se ha reducido para 8 procesadores y se ha incrementado para 16 procesadores.

Para el problema de mayor tamaño también se observan, en la tabla 4.10 de la versión vertical no sincronizada, resultados similares a los de la tabla 4.6 (versión horizontal no sincronizada). Sin embargo, para este problema, los mejores resultados para 8 y 16 procesadores son peores que los que se obtuvieron para la versión horizontal.

Como se ha comentado en las tablas anteriores, se observa que las versiones no sincronizadas presentan un comportamiento equivalente para cualquier tamaño de problema. Por este motivo presentamos, de forma gráfica, únicamente un resumen con las aceleraciones para las distintas combinaciones de número de procesadores y tamaños de problemas (figura 4.30).

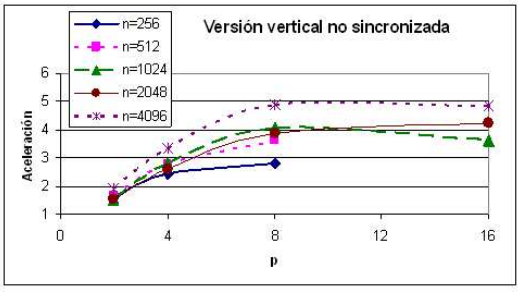

Figura 4.30: Aceleraciones máximas alcanzadas en las ejecuciones con la versión vertical no sincronizada para cada tamaño de problema y número de procesadores.

Tabla 4.7: Tiempos de ejecución del programa paralelo de la predicción de la estructura secundaria del RNA para una molécula de tamaño 256, utilizando la versión vertical no sincronizada.

Tamaño de problema: 256 - Tiempo secuencial: 0.260							
Número de procesadores: 2							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	0.315	0.83	1	20	0.277	0.94
1	40	0.271	0.96	1	60	0.273	0.95
1	80	0.289	0.90	1	100	0.357	0.73
2	1	0.262	0.99	2	20	0.212	1.23
2	40	0.213	1.22	2	60	0.219	1.19
2	80	0.231	1.13	2	100	0.276	0.94
4	1	0.217	1.20	4	20	0.190	1.37
4	40	0.189	1.38	4	60	0.194	1.34
4	80	0.205	1.27	4	100	0.241	1.08
16	1	0.169	1.54	16	20	0.162	1.61
16	40	0.170	1.53	16	60	0.174	1.49
16	80	0.189	1.37	16	100	0.216	1.20
32	1	0.163	1.60	32	20	0.165	1.58
32	40	0.172	1.51	32	60	0.181	1.44
32	80	0.194	1.34	32	100	0.220	1.24
64	**1**	**0.159**	**1.63**	64	20	0.169	1.54
64	40	0.179	1.45	64	60	0.194	1.34
64	80	0.207	1.26	64	100	0.223	1.17
Número de procesadores: 4							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	0.290	0.90	1	20	0.188	1.38
1	40	0.202	1.29	1	60	0.282	0.92
1	80	0.376	0.69	1	100	0.342	0.76
2	1	0.189	1.37	2	20	0.141	1.84
2	40	0.158	1.65	2	60	0.259	1.00
2	80	0.234	1.11	2	100	0.240	1.08
4	1	0.139	1.87	4	20	0.121	2.15
4	40	0.138	1.89	4	60	0.183	1.42
4	80	0.187	1.39	4	100	0.193	1.35
8	1	0.113	2.29	8	20	0.109	2.38
8	40	0.127	2.04	8	60	0.159	1.63
8	80	0.171	1.52	8	100	0.177	1.47
16	**1**	**0.106**	**2.45**	16	20	0.108	2.40
16	40	0.143	1.81	16	60	0.151	1.72
16	80	0.165	1.57	16	100	0.175	1.48
Número de procesadores: 8							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	0.194	1.34	1	20	0.175	1.48
1	40	0.210	1.24	1	60	0.249	1.04
1	80	0.298	0.87	1	100	0.232	1.12
2	1	0.145	1.79	2	20	0.149	1.75
2	40	0.147	1.77	2	60	0.161	1.62
2	80	0.177	1.47	2	100	0.228	1.14
4	**1**	**0.093**	**2.80**	4	20	0.099	2.64
4	40	0.144	1.81	4	60	0.139	1.88
4	80	0.174	1.50	4	100	0.158	1.64

Tabla 4.8: Tiempos de ejecución del programa paralelo de la predicción de la estructura secundaria del RNA para una molécula de tamaño 512, utilizando la versión vertical no sincronizada.

Tamaño de problema: 512 - Tiempo secuencial: 1.22							
Número de procesadores: 2							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	2.027	0.60	1	10	1.744	0.70
1	20	1.683	0.73	1	30	1.672	0.73
1	40	1.674	0.73	1	50	1.701	0.72
4	1	1.111	1.10	4	10	0.976	1.26
4	20	0.957	1.28	4	30	0.945	1.30
4	40	1.009	1.21	4	50	0.951	1.29
8	1	0.900	1.36	8	10	0.844	1.45
8	20	0.824	1.49	8	30	0.833	1.47
8	40	0.835	1.47	8	50	0.801	1.53
32	1	0.791	1.55	**32**	**10**	**0.747**	**1.64**
32	20	0.753	1.63	32	30	0.755	1.62
32	40	0.760	1.61	32	50	0.750	1.63
64	1	0.829	1.48	64	10	0.806	1.52
64	20	0.796	1.54	64	30	0.796	1.54
64	40	0.801	1.53	64	50	0.817	1.50
128	1	1.005	1.22	128	10	0.984	1.25
128	20	0.991	1.24	128	30	1.018	1.20
128	40	1.004	1.22	128	50	1.002	1.22
Número de procesadores: 4							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	1.597	0.77	1	10	1.177	1.04
1	20	1.212	1.01	1	30	1.183	1.04
1	40	1.212	1.01	2	50	1.210	1.01
4	1	0.747	1.64	4	10	0.601	2.04
4	20	0.594	2.06	4	30	0.589	2.08
4	40	0.634	1.93	4	50	0.661	1.85
8	1	0.580	2.11	8	10	0.501	2.45
8	20	0.507	2.42	8	30	0.516	2.37
8	40	0.523	2.34	8	50	0.551	2.22
16	1	0.541	2.26	16	10	0.452	2.71
16	20	0.482	2.54	16	30	0.464	2.64
16	40	0.479	2.56	16	50	0.490	2.50
32	1	0.481	2.54	**32**	**10**	**0.443**	**2.76**
32	20	0.451	2.72	32	30	0.451	2.72
32	40	0.490	2.50	32	50	0.492	2.49
Número de procesadores: 8							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	1.019	1.20	1	10	0.971	1.26
1	20	0.731	1.68	1	30	0.817	1.50
1	40	0.958	1.28	2	50	1.104	1.11
4	1	0.534	2.29	4	10	0.447	2.74
4	20	0.480	2.55	4	30	0.435	2.81
4	40	0.455	2.69	4	50	0.550	2.23
8	1	0.361	3.39	**8**	**10**	**0.340**	**3.61**
8	20	0.457	2.68	8	30	0.420	2.92
8	40	0.499	3.07	8	50	0.536	2.29

Tabla 4.9: Tiempos de ejecución del programa paralelo de la predicción de la estructura secundaria del RNA para una molécula de tamaño 1024, utilizando la versión vertical no sincronizada.

Tamaño de problema: 1024 - Tiempo secuencial: 5.97							
Número de procesadores: 2							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	16.16	0.37	1	10	14.75	0.40
1	20	14.54	0.41	1	30	14.53	0.41
8	1	5.23	1.14	8	10	4.91	1.22
8	20	4.77	1.25	8	30	4.99	1.20
32	1	4.14	1.44	32	10	3.97	1.50
32	**20**	**3.85**	**1.55**	32	30	4.03	1.48
64	1	4.12	1.45	64	10	4.09	1.46
64	20	3.98	1.50	64	30	4.11	1.45
128	1	4.38	1.36	128	10	4.28	1.39
128	20	4.33	1.38	128	30	4.39	1.36
256	1	5.21	1.17	256	10	5.18	1.15
256	20	5.19	1.15	256	30	5.29	1.13
Número de procesadores: 4							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	11.28	0.53	1	10	10.55	0.57
1	20	9.13	0.65	1	30	9.20	0.65
8	1	3.31	1.80	8	10	3.06	1.95
8	20	2.87	2.08	8	30	2.97	2.01
16	1	2.75	2.17	16	10	2.39	2.49
16	20	2.40	2.49	16	30	2.44	2.44
32	1	2.42	2.47	**32**	**10**	**2.12**	**2.81**
32	20	2.39	2.50	32	30	2.16	2.77
64	1	2.45	2.43	64	10	2.51	2.38
64	20	2.45	2.44	64	30	2.46	2.43
Número de procesadores: 8							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	7.23	0.82	1	10	5.65	1.06
1	20	5.61	1.06	1	30	5.52	1.08
4	1	3.05	1.96	4	10	2.61	2.28
4	20	2.40	2.49	4	30	2.84	2.10
8	1	2.21	2.70	8	10	1.95	3.06
8	20	1.78	3.36	8	30	2.01	2.97
16	1	1.87	3.19	**16**	**10**	**1.46**	**4.08**
16	20	1.65	3.63	16	30	1.69	3.53
Número de procesadores: 16							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	5.28	1.13	1	10	3.46	1.76
1	20	3.61	1.65	1	30	3.81	1.57
2	1	2.75	2.17	2	10	2.01	2.98
2	20	2.45	2.43	2	30	2.57	2.32
4	1	2.06	2.89	**4**	**10**	**1.64**	**3.64**
4	20	1.76	3.39	4	30	1.72	3.46

Tabla 4.10: Tiempos de ejecución del programa paralelo de la predicción de la estructura secundaria del RNA para una molécula de tamaño 2048, utilizando la versión vertical no sincronizada.

Tamaño de problema: 2048 - Tiempo secuencial: 30.49							
Número de procesadores: 2							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	166.52	0.18	1	5	161.00	0.19
1	10	159.67	0.19	1	15	158.62	0.19
8	1	36.32	0.84	8	5	34.99	0.87
8	10	34.69	0.88	8	15	34.68	0.88
32	1	24.16	1.26	32	5	21.73	1.40
32	10	21.58	1.41	32	15	21.71	1.40
64	1	21.22	1.44	64	5	19.98	1.53
64	10	20.33	1.50	64	15	19.88	1.53
128	1	20.56	1.48	128	5	20.03	1.52
128	**10**	**19.85**	**1.54**	128	15	20.00	1.52
512	1	26.42	1.15	512	5	26.17	1.17
512	10	26.19	1.16	512	15	26.11	1.17
Número de procesadores: 4							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	111.74	0.27	1	5	103.19	0.30
1	10	106.02	0.29	1	15	101.55	0.30
8	1	24.56	1.24	8	5	21.22	1.44
8	10	21.20	1.44	8	15	20.68	1.47
16	1	18.51	1.65	16	5	15.85	1.92
16	10	15.05	2.03	16	15	15.10	2.02
32	1	14.32	2.13	32	5	13.05	2.34
32	10	12.74	2.39	32	15	12.34	2.47
64	1	12.63	2.41	64	5	12.02	2.54
64	10	11.93	2.56	**64**	**15**	**11.82**	**2.58**
Número de procesadores: 8							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	67.12	0.45	1	5	57.05	0.53
1	10	58.90	0.52	1	15	57.62	0.53
8	1	14.50	2.10	8	5	12.65	2.41
8	10	12.27	2.48	8	15	14.07	2.17
16	1	10.88	2.80	16	5	9.60	3.18
16	10	9.39	3.25	16	15	9.49	3.21
32	1	9.12	3.34	**32**	**5**	**7.87**	**3.87**
32	10	8.16	3.74	32	15	8.05	3.78
Número de procesadores: 16							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	42.62	0.72	1	5	36.64	0.83
1	10	34.40	0.89	1	15	34.86	0.87
4	1	13.68	2.23	4	5	11.14	2.74
4	10	10.68	2.85	4	15	11.05	2.76
8	1	8.45	3.61	**8**	**5**	**7.22**	**4.22**
8	10	7.51	4.06	8	15	7.96	3.83

Tabla 4.11: Tiempos de ejecución del programa paralelo de la predicción de la estructura secundaria del RNA para una molécula de tamaño 4096, utilizando la versión vertical no sincronizada.

Tamaño de problema: 4096 - Tiempo secuencial: 219.39							
Número de procesadores: 2							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	1913.22	0.11	1	4	1889.72	0.12
1	7	1872.12	0.12	1	10	1879.96	0.12
8	1	346.65	0.63	8	4	322.07	0.68
8	7	319.16	0.69	8	10	318.15	0.69
64	1	139.08	1.58	64	4	127.06	1.73
64	7	126.65	1.73	64	10	126.76	1.73
128	1	120.82	1.82	128	4	116.02	1.89
128	7	115.17	1.90	128	10	115.69	1.90
256	1	117.29	1.87	256	4	114.93	1.91
256	7	114.93	1.91	**256**	**10**	**114.70**	**1.91**
1024	1	182.82	1.20	1024	4	182.88	1.20
1024	7	182.06	1.21	1024	10	182.07	1.20
Número de procesadores: 4							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	1251.21	0.18	1	4	1299.34	0.17
1	7	1233.24	0.18	1	10	1182.51	0.19
8	1	233.89	0.94	8	4	198.44	1.11
8	7	207.00	1.06	8	10	192.11	1.14
64	1	78.97	2.78	64	4	72.37	3.03
64	7	71.89	3.05	64	10	71.85	3.05
128	1	71.14	3.08	128	4	66.10	3.32
128	7	65.77	3.34	**128**	**10**	**65.64**	**3.34**
256	1	71.31	3.08	256	4	70.22	3.12
256	7	69.49	3.16	256	10	69.45	3.16
Número de procesadores: 8							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	749.26	0.29	1	4	722.38	0.30
1	7	674.42	0.33	1	10	655.42	0.33
8	1	128.43	1.71	8	4	125.20	1.75
8	7	121.10	1.81	8	10	113.83	1.93
16	1	84.22	2.60	16	4	71.39	3.07
16	7	69.81	3.14	16	10	71.29	3.08
64	1	51.061	4.30	64	4	45.14	4.86
64	**7**	**44.72**	**4.91**	64	10	47.78	4.59
Número de procesadores: 16							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	440.16	0.50	1	4	396.85	0.55
1	7	387.16	0.57	1	10	378.41	0.58
4	1	122.26	1.79	4	4	113.69	1.93
4	7	113.34	1.94	4	10	108.31	2.03
16	1	49.49	4.43	16	4	45.24	4.85
16	**7**	**45.11**	**4.86**	16	10	45.64	4.81

Para esta versión, también podemos comprobar que las aceleraciones máximas aumentan al incrementarse el tamaño del problema o el número de procesadores. Aunque siguen produciéndose algunas excepciones: la aceleración para $n = 2048$ cuando ejecutamos con 4 y 8 procesadores es menor que la conseguida para el problema de tamaño $n = 1024$. Para este último problema, el rendimiento obtenido empeora al emplear 16 procesadores, frente a utilizar solamente 8.

Versión horizontal sincronizada

Para la versión horizontal sincronizada no es necesario establecer restricciones en la asignación del ancho del *tile* (sección 4.6.3). Por este motivo, hemos realizado un número de ejecuciones mucho mayor que para las versiones anteriores: hemos utilizado los mismos valores de y que en las versiones no sincronizadas y hemos añadido valores intermedios para cubrir todo el rango de valores: $y \in [1, \frac{n}{p^2}]$. Para cada problema se especificarán el número de ejecuciones realizadas y los valores de y utilizados. En todas las tablas se presentarán algunos de los valores de y que son potencia de 2 para comparar con las versiones no sincronizadas y algunos de los valores intermedios. Respecto al alto del *tile*, hemos ejecutado con el mismo rango de valores que en las versiones no sincronizadas y, en la mayoría de los casos, mostramos las mismas cantidades en las tablas.

En la tabla 4.12 presentamos los resultados del problema del tamaño 256 para la versión horizontal sincronizada. Para este problema el valor máximo de y que podemos utilizar es $\frac{256}{p^2}$, si consideramos únicamente 2 procesadores ($\frac{256}{2^2} = \frac{256}{4} = 64$). Por lo tanto, utilizamos como ancho del *tile* para resolver el problema todas las potencias de 2 hasta 64, empleando como valores intemedios las siguientes cantidades: 12, 20, 25, 30, 40, 50 y 60. Para el resto de número de procesadores, calculamos el valor máximo de y mediante la fórmula anterior e incluimos los valores intermedios inferiores al máximo. Esto significa que, considerando todos los valores de p, y y x, hemos realizado, para resolver este problema, la cifra de 2300 combinaciones distintas en las ejecuciones del programa paralelo, de las cuales solamente mostramos en la tabla 84 resultados.

En la tabla 4.12 se observa que los resultados, para un problema tan pequeño, no difieren demasiado de los resultados para la versión horizontal no sincronizada. Aunque si observamos los mejores tiempos, podemos ver que se obtienen para la misma combinación de ancho y alto del *tile*, pero se ha incrementado el tiempo de ejecución: para 4 procesadores, el tiempo de ejecución para la versión sincronizada es de 0.115 segundos, mientras que en la no sincronizada era de 0.104 segundos. Lo mismo ocurre con 8 procesadores, donde el tiempo aumenta de 0.100 segundos en la versión no sincronizada a 0.124 segundos utilizando la sincronización. Al ser un problema tan pequeño, las diferencias en tiempo son también muy reducidas. La principal diferencia se encuentra en el número óptimo de procesadores, que se reduce de 8 a 4.

De nuevo se comprueba la gran diferencia que puede haber en los resultados modificando únicamente uno de los parámetros. Por ejemplo, si consideramos la ejecución utilizando 8 procesadores con $y = 4$ vemos que se consigue el mejor tiempo de ejecución con el tamaño más pequeño de x (la aceleración es mayor de 2). Sin embargo, cuando incrementamos el tamaño de x observamos que el tiempo de ejecución aumenta, no consiguiendo mejorar el tiempo del algoritmo secuencial en ninguna de las restantes ejecuciones. En la figura 4.31 mostramos de forma gráfica los resultados del problema más pequeño cuando fijamos dos parámetros y variamos uno (y o x).

En la tabla 4.13 se encuentran los resultados correspondientes al problema de tamaño 512. Para este problema el valor máximo de y con el que podemos ejecutar para 2 procesadores es $\frac{n}{p^2} = \frac{512}{2^2} = 128$ y, además de las potencias de 2, hemos realizado ejecuciones con los siguientes valores intermedios: 12, 20, 25, 30, 40, 50, 60, 75, 85, 100, 110 y 120. Como en la tabla anterior, para 4 y 8 procesadores, se utilizan los valores menores que el máximo y correspondiente ($\frac{n}{p^2}$). Teniendo en cuenta el número de procesadores y los valores de y y x utilizados, hemos ejecutado el programa para un total de 1700 combinaciones distintas.

La tabla 4.13 es el único caso donde las combinaciones de y y x que se muestran, no coinciden con las de las tablas de las versiones no sincronizadas para el mismo tamaño de problema. Esto se debe al incremento en el número de valores de y utilizados en la ejecución del programa: para 2 procesadores

Tabla 4.12: Tiempos de ejecución del programa paralelo de la predicción de la estructura secundaria del RNA para una molécula de tamaño 256, utilizando la versión horizontal sincronizada.

Tamaño de problema: 256 - Tiempo secuencial: 0.260							
Número de procesadores: 2							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	0.330	0.79	1	20	0.275	0.94
1	40	0.268	0.97	1	60	0.272	0.96
1	80	0.292	0.89	1	100	0.349	0.75
4	1	0.293	0.89	4	20	0.186	1.40
4	40	0.192	1.36	4	60	0.193	1.35
4	80	0.206	1.26	4	100	0.237	1.10
20	1	0.171	1.52	20	20	0.170	1.53
20	40	0.165	1.58	20	60	0.223	1.16
20	80	0.191	1.36	20	100	0.212	1.23
32	1	0.167	1.56	32	20	0.165	1.57
32	40	0.171	1.52	32	60	0.183	1.42
32	80	0.203	1.28	32	100	0.220	1.18
50	1	0.177	1.47	50	20	0.177	1.47
50	40	0.178	1.46	50	60	0.190	1.37
50	80	0.191	1.36	50	100	0.238	1.09
64	**1**	**0.162**	**1.60**	64	20	0.168	1.55
64	40	0.196	1.33	64	60	0.214	1.22
64	80	0.213	1.22	64	100	0.231	1.13
Número de procesadores: 4							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	0.279	0.93	1	20	0.220	1.18
1	40	0.402	0.65	1	60	0.557	0.47
1	80	0.618	0.42	1	100	0.611	0.43
4	1	0.173	1.50	4	20	0.172	1.51
4	40	0.246	1.06	4	60	0.348	0.75
4	80	0.378	0.69	4	100	0.398	0.65
8	1	0.124	2.09	8	20	0.137	1.90
8	40	0.221	1.18	8	60	0.307	0.85
8	80	0.336	0.77	8	100	0.387	0.67
12	1	0.130	2.00	12	20	0.140	1.85
12	40	0.224	1.16	12	60	0.309	0.84
12	80	0.339	0.77	12	100	0.348	0.75
16	**1**	**0.115**	**2.25**	16	20	0.162	1.61
16	40	0.226	1.15	16	60	0.308	0.85
16	80	0.326	0.80	16	100	0.334	0.78
Número de procesadores: 8							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	0.267	0.98	1	20	0.478	0.54
1	40	0.747	0.35	1	60	0.806	0.32
1	80	0.941	0.28	1	100	0.949	0.27
2	1	0.133	1.95	2	20	0.460	0.56
2	40	0.626	0.42	2	60	0.685	0.38
2	80	0.720	0.36	2	100	0.811	0.32
4	**1**	**0.124**	**2.09**	4	20	0.392	0.66
4	40	0.519	0.50	4	60	0.589	0.44
4	80	0.584	0.45	4	100	0.617	0.42

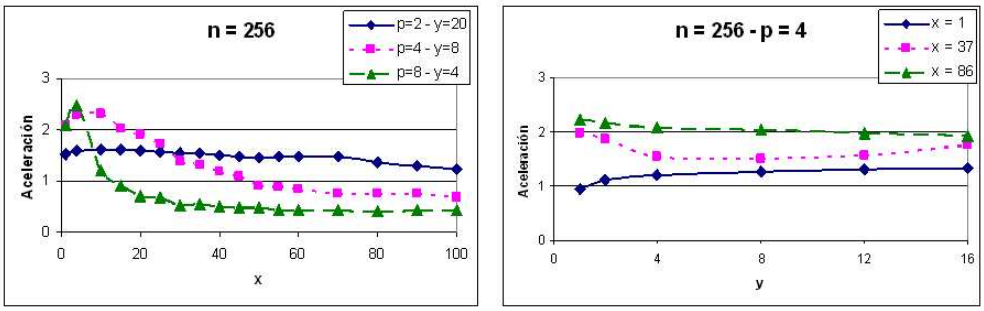

Figura 4.31: Aceleraciones obtenidas para el problema de tamaño 256 con la versión horizontal sincronizada fijando 2 parámetros y variando el tercero. Izquierda: Fijamos número de procesadores y valor de x y modificamos y. Derecha: Fijamos número de procesadores y valor de y y variamos x.

empleamos 8 valores distintos de y en las versiones no sincronizadas, mientras que para esta nueva versión hemos utilizado 20 diferentes. Para poder mostrar información sobre un mayor número valores de y, hemos reducido el número de tamaños de x mostrados en la tabla: hemos pasado de incluir 6 valores por cada y a mostrar sólo 4.

En el problema de tamaño 512 (tabla 4.13) también podemos ver que los resultados no son muy diferentes a los de la versión horizontal no sincronizada; sin embargo, se observa, en general, un pequeño incremento en los tiempos de ejecución. Por ejemplo, para 8 procesadores, con $y = 8$ y $x = 1$, el tiempo sin la sincronización entre los procesadores es 0.35 segundos (una aceleración de 3.48), mientras que la misma combinación de parámetros estableciendo la sincronización requiere un tiempo de 0.44 segundos (la aceleración es de 2.74). Al ser un problema pequeño, donde el tiempo total de ejecución es menor de un segundo, no se nota la diferencia en tiempo (0.09 segundos), pero sí se observa la reducción del rendimiento en la resolución del problema. A pesar de estas diferencias, las combinaciones donde se obtiene el mínimo tiempo de ejecución son similares.

Para el tamaño de problema 1024 (tabla 4.14) y 2 procesadores podemos ejecutar con un valor de máximo de $y = \frac{1024}{2^2} = 256$, por lo que el número de valores de y que no son potencia de 2 también se incrementan. Hasta 128 hemos utilizado los mismos valores intermedios que en la ejecución para el problema de tamaño 512, entre 128 y 256 los valores utilizados han sido: 150, 175, 200 y 250. Al reducir el rango de valores de x, el número de ejecuciones se reduce respecto a los problemas anteriores, realizando 1500 ejecuciones.

En los resultados de la tabla 4.14 se observa que, a diferencia de los problema anteriores, ahora existen combinaciones de y y x para los que se reduce el tiempo de ejecución frente a la versión no sincronizada. Por ejemplo, para 2 procesadores, $y = 256$ y $x = 10$ obtenemos ahora un tiempo de ejecución de 5.26 (aceleración 1.14), mientras en la versión horizontal no sincronizada el tiempo fue de 5.37 (una aceleración de 1.11). Evidentemente, al ser un problema que requiere poco tiempo de ejecución (menos de 6 segundos en la mayoría de las ejecuciones, las diferencias en los tiempo no pueden ser excesivamente grandes).

Observando los resultados de la tabla 4.14 comprobamos que existe una diferencia importante respecto a los problemas anteriores: se ha modificado el tamaño del *tile* que requiere el mínimo tiempo de ejecución. Para resolver este problema con 2 ó 4 procesadores, la mejor solución consiste en utilizar un ancho del *tile* de $y = 40$ en lugar de emplear $y = 32$, que era el valor óptimo en las versiones no sincronizadas. La figura 4.32 muestra de forma gráfica los resultados del problema de tamaño $n = 1024$ cuando variamos y o x.

En la tabla 4.15 presentamos los resultados para el problema de tamaño 2048. Para este problema y 2 procesadores el valor máximo de y se incrementa hasta $\frac{2048}{2^2} = 512$ y añadimos los siguientes valores que no son potencia de 2: 300, 400 y 500. Empleando todas estas cantidades, llegamos a ejecutar con 29

Tabla 4.13: Tiempos de ejecución del programa paralelo de la predicción de la estructura secundaria del RNA para una molécula de tamaño 512, utilizando la versión horizontal sincronizada.

Tamaño de problema: 512 - Tiempo secuencial: 1.22							
Número de procesadores: 2							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	2.01	0.61	1	15	1.73	0.71
1	35	1.70	0.72	1	50	1.68	0.73
16	1	0.83	1.47	16	15	0.78	1.58
16	35	0.79	1.55	16	50	0.79	1.55
25	1	0.80	1.53	25	15	0.76	1.60
25	35	0.76	1.60	25	50	0.76	1.62
32	1	0.79	1.55	**32**	**15**	**0.73**	**1.67**
32	35	0.78	1.57	32	50	0.75	1.63
60	1	0.81	1.50	60	15	0.76	1.61
60	35	0.76	1.60	60	50	0.78	1.57
85	1	0.84	1.46	85	15	0.82	1.49
85	35	0.83	1.48	85	50	0.85	1.44
110	1	0.90	1.37	110	15	0.87	1.41
110	35	0.93	1.32	110	50	0.95	1.28
128	1	1.02	1.20	128	15	0.99	1.24
128	35	0.99	1.23	128	50	1.01	1.22
Número de procesadores: 4							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	1.68	0.73	1	15	1.15	1.07
1	35	1.23	0.99	1	50	1.64	0.75
8	1	0.61	1.99	8	15	0.54	2.28
8	35	0.58	2.11	8	50	0.73	1.68
12	1	0.54	2.28	12	15	0.52	2.37
12	35	0.68	1.80	12	50	0.72	1.69
16	1	0.52	2.36	16	15	0.51	2.38
16	35	0.55	2.21	16	50	0.72	1.70
20	1	0.54	2.25	**20**	**15**	**0.49**	**2.49**
20	35	0.55	2.22	20	50	0.73	1.69
25	1	0.53	2.30	25	15	0.50	2.47
25	35	0.58	2.12	25	50	0.72	1.69
30	1	0.54	2.25	30	15	0.53	2.32
30	35	0.58	2.10	30	50	0.74	1.67
32	1	0.56	2.18	32	15	0.50	2.46
32	35	0.58	2.12	32	50	0.73	1.68
Número de procesadores: 8							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	1.04	1.18	1	15	1.20	1.03
1	35	2.45	0.50	1	50	3.49	0.35
2	1	0.74	1.66	2	15	0.82	1.49
2	35	1.71	0.72	2	50	2.40	0.51
4	1	0.48	2.55	4	15	0.73	1.69
4	35	1.30	0.94	4	50	1.94	0.63
8	**1**	**0.45**	**2.74**	8	15	0.59	2.07
8	35	1.23	1.00	8	50	1.61	0.76

Tabla 4.14: Tiempos de ejecución del programa paralelo de la predicción de la estructura secundaria del RNA para una molécula de tamaño 1024, utilizando la versión horizontal sincronizada.

Tamaño de problema: 1024 - Tiempo secuencial: 5.97							
Número de procesadores: 2							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	16.23	0.37	1	10	14.79	0.40
1	20	14.52	0.41	1	30	14.56	0.41
32	1	4.15	1.44	32	10	3.96	1.51
32	20	3.95	1.51	32	30	3.94	1.52
40	1	4.04	1.48	40	10	3.80	1.57
40	**20**	**3.71**	**1.61**	40	30	3.86	1.55
85	1	4.19	1.43	85	10	4.13	1.45
85	20	4.16	1.43	85	30	4.14	1.44
120	1	4.28	1.39	120	10	4.20	1.42
120	20	4.17	1.43	120	30	4.25	1.40
256	1	5.26	1.13	256	10	5.26	1.14
256	20	5.26	1.13	256	30	5.25	1.14
Número de procesadores: 4							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	11.26	0.53	1	10	9.41	0.63
1	20	9.31	0.64	1	30	9.38	0.64
12	1	2.97	2.01	12	10	2.61	2.29
12	20	2.56	2.33	12	30	2.62	2.28
32	1	2.38	2.51	32	10	2.28	2.61
32	20	2.38	2.50	32	30	2.40	2.48
40	1	2.38	2.51	**40**	**10**	**2.24**	**2.66**
40	20	2.38	2.51	40	30	2.39	2.50
64	1	2.72	2.19	64	10	2.46	2.42
64	20	2.47	2.42	64	30	2.62	2.27
Número de procesadores: 8							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	7.41	0.81	1	10	5.66	1.05
1	20	6.87	0.87	1	30	10.05	0.59
4	1	2.70	2.21	4	10	2.32	2.57
4	20	2.79	2.14	4	30	4.15	1.44
12	1	1.88	3.18	12	10	1.82	3.28
12	20	2.18	2.74	12	30	2.85	2.10
16	1	1.70	3.50	**16**	**10**	**1.60**	**3.73**
16	20	1.93	3.09	16	30	2.63	2.27
Número de procesadores: 16							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	4.84	1.23	1	10	7.40	0.81
1	20	13.44	0.44	1	30	19.97	0.30
2	1	3.07	1.95	2	10	4.33	1.38
2	20	8.65	0.69	2	30	12.47	0.48
4	**1**	**2.17**	**2.75**	4	10	3.62	1.65
4	20	6.28	0.95	4	30	8.57	0.70

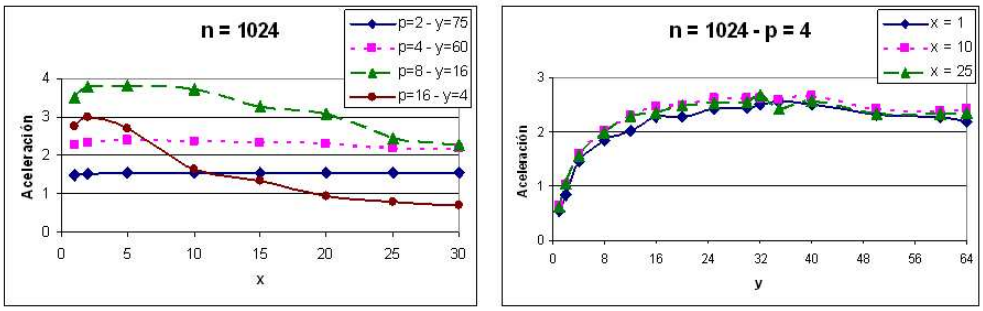

Figura 4.32: Aceleraciones alcanzadas para el problema de tamaño 1024 con la versión horizontal sincronizada fijando 2 parámetros y variando el tercero. Izquierda: Fijamos número de procesadores y x. Derecha: Fijamos número de procesadores y el valor de y.

valores de distintos de y para 2 procesadores. La cifra total de combinaciones diferentes de número de procesadores, y y x es de 1005.

Si comparamos los resultados de la tabla 4.15 con los correspondientes a la versión horizontal no sincronizada (tabla 4.6), observamos que existen combinaciones para las que el tiempo de ejecución se reduce y otras para las que aumenta, aunque en todos los casos son tiempos similares. Sin embargo, sí se produce un cambio en los valores óptimos de y para 2 y 4 procesadores. Para 2 procesadores se consigue incrementar la aceleración de 1.51 (con $y = 64$) a 1.60 (con $y = 60$) al introducir la sincronización; mientras que con 4 procesadores la aceleración máxima es prácticamente idéntica, pero se alcanza con distinto valor de y, con 150 en lugar de 128. Se observa en estos dos últimos problemas que el ancho óptimo del *tile* ya no es una potencia de 2.

Por último, en la tabla 4.16 presentamos los resultados para el problema de mayor tamaño, $n = 4096$. Ahora el valor máximo de y con 2 procesadores es 1024 y añadimos además otros 5 valores: 600, 700, 800, 900, 1000. Al reducir sólo a 10 los posibles tamaños de x, el número total de combinaciones posibles que realizamos es únicamente de 880.

En los resultados del problema de tamaño 4096 (tabla 4.16) observamos que se produce un incremento significativo de los tiempos mínimos, excepto para 2 procesadores. Por ejemplo, para 8 procesadores la aceleración máxima para la versión no sincronizada era de 5.16, mientras que usando la sincronización sólo es de 4.67. La figura 4.33 muestra la variación de los resultados obtenidos para este problema cuando fijamos 2 de los parámetros.

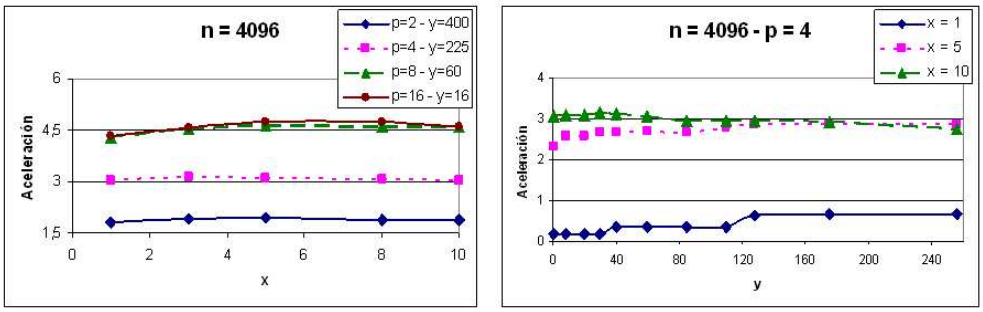

Figura 4.33: Aceleraciones alcanzadas con el problema de tamaño 4096 y la versión horizontal sincronizada fijando 2 parámetros y variando el tercero. Izquierda: Fijamos número de procesadores y x. Derecha: Fijamos número de procesadores y el valor de y.

Tabla 4.15: Tiempos de ejecución del programa paralelo de la predicción de la estructura secundaria del RNA para una molécula de tamaño 2048, utilizando la versión horizontal sincronizada.

Tamaño de problema: 2048 - Tiempo secuencial: 30.49							
Número de procesadores: 2							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	166.57	0.18	1	5	161.10	0.19
1	10	159.23	0.19	1	15	158.71	0.19
32	1	24.46	1.25	32	5	21.95	1.39
32	10	21.75	1.40	32	15	21.90	1.39
64	1	21.15	1.44	64	5	20.14	1.51
64	10	20.00	1.52	64	15	19.99	1.52
128	1	20.52	1.49	128	5	20.07	1.52
128	10	20.13	1.51	128	15	20.16	1.51
150	1	19.78	1.54	**150**	**5**	**19.01**	**1.60**
150	10	19.05	1.60	150	15	19.17	1.59
300	1	21.03	1.45	300	5	20.81	1.47
300	10	20.85	1.46	300	15	21.18	1.44
512	1	26.61	1.15	512	5	26.54	1.15
512	10	26.50	1.15	512	15	26.53	1.15
Número de procesadores: 4							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	108.82	0.28	1	5	102.06	0.30
1	10	100.16	0.30	1	15	99.61	0.31
12	1	18.15	1.68	12	5	17.05	1.79
12	10	17.05	1.79	12	15	17.05	1.79
32	1	13.61	2.24	32	5	12.84	2.37
32	10	12.69	2.40	32	15	12.63	2.41
60	1	12.75	2.39	**60**	**5**	**12.14**	**2.51**
60	10	12.14	2.51	60	15	12.21	2.50
64	1	12.88	2.37	64	5	12.30	2.48
64	10	12.42	2.45	64	15	12.95	2.35
128	1	13.31	2.29	128	5	13.11	2.33
128	10	13.38	2.28	128	15	13.83	2.20
Número de procesadores: 8							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	60.42	0.50	1	5	55.07	0.55
1	10	54.59	0.56	1	15	54.40	0.56
8	1	12.95	2.35	8	5	12.23	2.49
8	10	12.10	2.52	8	15	12.08	2.52
20	1	9.08	3.36	20	5	8.73	3.49
20	10	8.77	3.48	20	15	8.78	3.47
32	1	8.19	3.72	**32**	**5**	**7.65**	**3.98**
32	10	7.74	3.94	32	15	7.88	3.87
Número de procesadores: 16							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	36.60	0.83	1	5	30.66	0.99
1	10	34.17	0.89	1	15	51.15	0.60
4	1	11.71	2.60	4	5	10.77	2.83
4	10	12.01	2.54	4	15	17.39	1.75
8	1	8.29	3.68	**8**	**5**	**7.88**	**3.87**
8	10	8.70	3.51	8	15	11.95	2.55

Tabla 4.16: Tiempos de ejecución del programa paralelo de la predicción de la estructura secundaria del RNA para una molécula de tamaño 4096, utilizando la versión horizontal sincronizada.

Tamaño de problema: 4096 - Tiempo secuencial: 219.39							
Número de procesadores: 2							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	1918.08	0.11	1	4	1908.47	0.11
1	7	1880.84	0.12	1	10	1871.63	0.12
120	1	126.69	1.73	120	4	114.96	1.91
120	7	114.18	1.92	120	10	114.87	1.91
256	1	119.15	1.84	256	4	115.45	1.90
256	7	115.18	1.90	256	10	115.96	1.89
300	1	114.26	1.92	300	4	110.25	1.99
300	**7**	**109.06**	**2.01**	300	10	109.31	2.01
600	1	126.61	1.73	600	4	123.64	1.77
600	7	125.95	1.74	600	10	127.05	1.73
1024	1	185.27	1.18	1024	4	192.57	1.14
1024	7	212.55	1.03	1024	10	183.78	1.19
Número de procesadores: 4							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	1236.21	0.18	1	4	1194.96	0.18
1	7	1191.99	0.18	1	10	1202.30	0.18
64	1	77.17	2.84	64	4	73.61	2.98
64	7	73.55	2.98	64	10	73.34	2.99
120	1	70.85	3.10	120	4	68.51	3.20
120	**7**	**68.05**	**3.22**	120	10	68.95	3.18
128	1	71.29	3.08	128	4	68.90	3.18
128	7	70.00	3.13	128	10	69.48	3.16
256	1	75.18	2.92	256	4	74.31	2.95
256	7	75.03	2.92	256	10	79.92	2.75
Número de procesadores: 8							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	712.93	0.31	1	4	673.24	0.33
1	7	676.68	0.32	1	10	688.88	0.32
8	1	113.70	1.93	8	4	112.62	1.95
8	7	110.03	1.99	8	10	113.74	1.93
20	1	74.50	2.94	20	4	66.14	3.32
20	7	68.23	3.22	20	10	64.67	3.39
64	1	48.62	4.51	**64**	**4**	**46.88**	**4.68**
64	7	47.57	4.61	64	10	49.65	4.42
Número de procesadores: 16							
y	x	Tiempo Real	Aceleración	y	x	Tiempo Real	Aceleración
1	1	404.99	0.54	1	4	369.81	0.59
1	7	364.13	0.60	1	10	357.64	0.61
8	1	71.33	3.08	8	4	68.64	3.20
8	7	68.52	3.20	8	10	70.31	3.12
16	1	50.72	4.33	16	4	48.58	4.52
16	7	47.80	4.59	**16**	**10**	**47.60**	**4.61**

Como se observa en las tablas anteriores, para los problemas pequeños con cualquier número de procesadores, el ancho del *tile* que minimiza el tiempo de ejecución sigue siendo potencia de 2 (como en las versiones no sincronizadas). Sin embargo, al aumentar el tamaño de los problemas, para 2 y 4 procesadores se obtienen los tiempos mínimos en valores de y intermedios, mientras que para 8 y 16 sigue siendo mejor ejecutar con valores que son potencia de 2. En la figura 4.34 se observan las aceleraciones máximas alcanzadas para cada combinación de tamaño de problema y número de procesadores. Como en las versiones anteriores, en la mayor parte de las ejecuciones, aumenta la aceleración al incrementarse uno cualquiera de los dos parámetros. Las excepciones encontradas a esta situación se encuentran en el problema $n = 2048$ con 4 procesadores, cUYA Aceleración es inferior a la alcanzada para $n = 1024$ y el mismo valor de p. Además, excepto para el problema de mayor tamaño, se produce un descenso de la aceleración al ejecutar con 16 procesadores, frente a los resultados para $p = 8$.

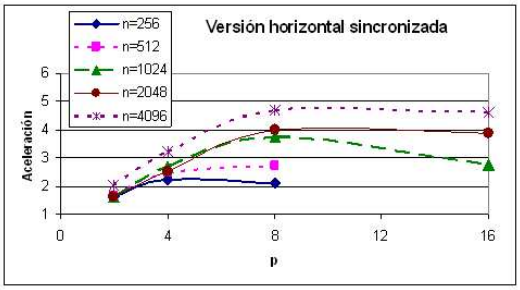

Figura 4.34: Aceleraciones máximas alcanzadas en las ejecuciones con la versión horizontal sincronizada para cada tamaño de problema y número de procesadores.

Como se puede comprobar en todos los resultados mostrados en esta sección, no es posible establecer un patrón sencillo para calcular los valores del ancho y alto del *tile*, para obtener el tiempo mínimo de ejecución. Por este motivo, desarrollamos un modelo analítico y un modelo estadístico que nos resuelvan este problema y nos permitan obtener la combinación óptima de parámetros, evitándonos la necesidad de realizar un alto número de ejecuciones.

En la figura 4.35 se muestra una comparativa entre las tres versiones: *vHns* representa la versión horizontal no sincronizada, *vVns* representa la versión vertical no sincronizada y *vHs* la versión horizontal sincronizada.

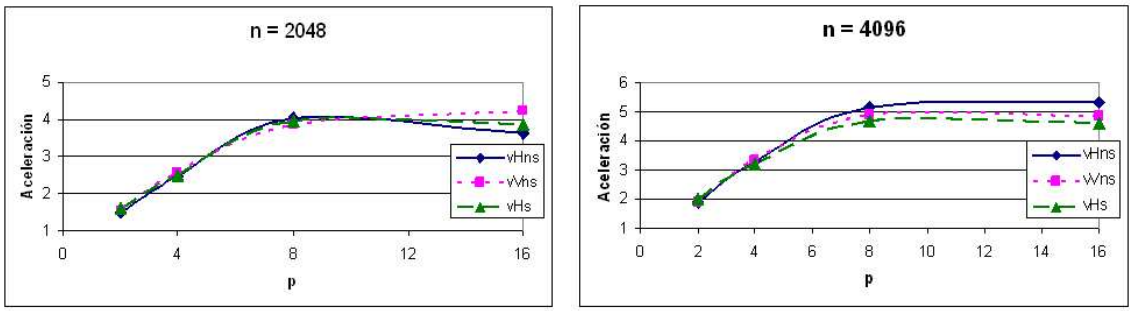

Figura 4.35: Aceleraciones alcanzadas en las ejecuciones con las tres versiones del programa. Izquierda: para el problema de tamaño $n = 2048$. Derecha: para el problema de tamaño $n = 4096$.

En la figura 4.35-Izquierda se puede comprobar la similitud existente entre los resultados para las tres versiones con el problema de tamaño $n = 2048$, únicamente se diferencian cuando $p = 16$. Para este número de procesadores se obtiene un mayor rendimiento con la versión vertical frente a la horizontal

y con la versión sincronizada frente a la que no lo es. Sin embargo, cuando consideramos el problema de tamaño $n = 4096$ (figura 4.35-Derecha), el rendimiento del sistema ya es distinto con 8 procesadores. En este caso, la versión horizontal no sincronizada resulta ser la mejor de las tres. Esto indica que no podemos asegurar con cuál de las versiones deberíamos resolver un problema para obtener el mínimo tiempo de ejecución.

4.8. Modelo analítico para el problema de la estructura secundaria del RNA

En esta sección presentamos el modelo analítico que hemos desarrollado para calcular el tamaño óptimo del *tile* (ancho y alto) con el que resolver el problema. Inicialmente, consideramos una máquina ideal, donde se puede despreciar el tiempo debido a las comunicaciones y, posteriormente, modelamos el tiempo de ejecución en una máquina real y optimizamos el volumen del *tile*.

4.8.1. Desarrollo del modelo sobre una máquina ideal

Primero vamos a modelar el tiempo de ejecución sobre una máquina ideal, donde el coste debido a las comunicaciones es despreciable y se requiere una unidad de tiempo para ejecutar cada *tile*. Calculamos el tiempo paralelo ideal para resolver el problema con p procesadores, usando una distribución cíclica por bloques de *tiles* a los procesadores.

Dividimos los cuadrados R_k^l por columnas y asignamos las columnas a los procesadores, cualquier cuadrado de la columna j es asignado al procesador p_j. A continuación, procesamos las columnas en un *pipeline*. Con las características anteriores, es fácil derivar el tiempo de ejecución necesario para resolver un cuadrado dado, aplicando una clase de *tiling* ortogonal.

Como cualquier cuadrado R_k^l contiene m líneas y p columnas, podemos calcular ahora el tiempo total paralelo, teniendo en cuenta que las dependencias entre los cuadrados, son las mismas que entre los bloques de cómputo y que un cuadrado comienza a ejecutarse por el *tile* inferior izquierdo y finaliza su ejecución en el *tile* superior derecho. Por este motivo, un cuadrado no puede comenzar a ser calculado (procesar su primer bloque) antes de que el cuadrado previo haya completado la ejecución de su último bloque.

Consideramos primero el recorrido horizontal. Si el cuadrado R_k^l sigue inmediatamente al cuadrado R_k^{l-1} en el orden del recorrido horizontal de la figura 4.10-b, hay dos condiciones que determinan cuando puede comenzar el cálculo del cuadrado R_k^l:

1. Todos los m bloques de cómputo de la primera columna (la situada más a la izquierda) en el cuadrado R_{k+1}^l deben haber sido calculados.

2. Todos los p bloques de cómputo de la primera fila (la fila inferior) en el cuadrado R_k^{l-1} deben haber sido calculados también.

Mediante la técnica del *tiling* ortogonal, los cuadrados sucesivos son computados de izquierda a derecha mediante etapas de un *pipeline*; tenemos $\frac{p\bar{p}}{2}$ cuadrados ($\frac{p\bar{p}}{2}$ etapas en el *pipeline*) y el intervalo de tiempo entre el inicio de dos cuadrados sucesivos es $\max(m, p)$. Esto significa que el último cuadrado comienza en el instante de tiempo $(\frac{p\bar{p}}{2} - 1)\max(m, p)$ y necesita $p+m$ pasos adicionales para completarse. Podemos unir y simplificar estas dos expresiones para obtener la ecuación del tiempo paralelo, Σ:

$$\Sigma \;=\; \left(\tfrac{p\bar{p}}{2} - 1\right)\max(m,p) + m + p \;=\; \begin{cases} \Sigma_1 = \frac{p\bar{p}}{2}m + p & \text{si } m \geq p \\ \Sigma_2 = \frac{p\bar{p}}{2}p + m & \text{en otro caso} \end{cases}$$

En el recorrido vertical se deriva el tiempo total de forma ligeramente diferente. Debido al esquema del recorrido, los dos primeros cuadrados (R_1^2 y R_2^3) pueden ser calculados sin sincronización entre los procesadores (figura 4.10-c). Sin embargo, al comenzar el cálculo del tercer cuadrado (R_1^3), es necesaria una sicronización. Como la columna $p+1$ en el dominio transformado es asignada al primer procesador, esta sincronización es equivalente a considerar máx$(2m, p)$ como el tiempo de comienzo del cálculo de este cuadrado. Debido a la inclinación del espacio de direcciones el cálculo de la columna k, donde $k = (p-1) \bmod p$ (calculada por el último procesador) envía los datos a la columna $k+1$ (calculada por el primer procesador) antes del comienzo de cada nueva etapa. Podemos deducir la siguiente expresión para el tiempo paralelo ideal:

$$\Sigma \quad = \quad \text{máx}(2m, p) + \left(\tfrac{p\bar{p}}{2} - 2\right) m + p \quad = \quad \begin{cases} \Sigma_3 = \frac{p\bar{p}}{2} m + p & \text{si } 2m \geq p \\ \Sigma_4 = \left(\frac{p\bar{p}}{2} - 2\right) m + 2p & \text{en otro caso} \end{cases}$$

4.8.2. Desarrollo del modelo sobre una máquina real

Basado en el tiempo ideal obtenido en la sección anterior, desarrollamos ahora el tiempo de ejecución sobre una máquina paralela real configurada como un anillo de p procesadores. Modelamos los costes de las comunicaciones entre los procesadores usando el modelo *BSP* [178]. Finalmente, usando el tiempo de ejecución estimado, formulamos un problema de optimización para determinar los parámetros del *tile* que minimizan el tiempo de ejecución.

El valor del tiempo paralelo ideal (Σ), calculado en la sección anterior, refleja el tiempo en número de etapas paralelas, donde cada etapa corresponde a la ejecución de un bloque de cómputo y la comunicación de su resultado. Para convertirlo en tiempo de ejecución de una máquina real, necesitamos un modelo del coste de las comunicaciones; vamos a utilizar para este propósito el modelo *BSP* [178], donde el cómputo se realiza en una secuencia de macroetapas. Cada macroetapa consiste en:

1. Algunos cómputos locales.
2. Algunos eventos de comunicación durante el macroetapa.
3. Una barrera global de sincronización, durante la cual efectúan realmente las comunicaciones y los resultados pasan a estar disponibles para la macroetapa siguiente.

En nuestra paralelización, cada macroetapa es idéntica y corresponde a lo que hemos denominado una etapa al calcular el valor de Σ. Por tanto, el tiempo total de ejecución es simplemente $\Sigma\mathcal{P}$, donde $\mathcal{P}$ es el tiempo necesario para realizar una macroetapa. A continuación, determinamos una ecuación analítica precisa del tiempo de ejecución de nuestro programa y formulamos nuestro problema de optimización discreta no lineal.

Dos bloques de cómputo se dice que son **sucesivos** si uno de ellos depende directamente del otro. En nuestra implementación, la estructura de datos es replicada en cada procesador; es decir, cada procesador trabaja con el espacio de iteraciones completo. Por otra parte, los bloques de cómputo son asignados a los procesadores por columnas: la columna j es calculada por el procesador $j \bmod p$. Esta distribución de *tiles* garantiza que los sucesivos bloques en vertical son asignados al mismo procesador y no es necesario realizar comunicaciones de los datos. Para calcular un *tile* cualquiera, el procesador p_k necesita recibir del procesador p_{k-1} sólo los $\bar{p}$ bloques situados a la izquierda, en la misma fila del *tile* considerado. Debido a la condición de ortogonalidad, los restantes *tiles* han sido transmitidos previamente a la memoria de p_k. El código ejecutado para cada uno de los *tiles* aparece en la figura 4.36.

Denotamos por α al tiempo de ejecución de una única instancia del bucle. Podemos asumir que este tiempo es constante, al menos en algún entorno relativamente pequeño; esto es únicamente una aproximación, pero nos permite obtener una fórmula analítica para calcular el tamaño óptimo del bloque de cómputo.

```
1 repeat
2    recibir (r_1, r_2,..., r_{p-1});
3    computar (tile, r_p);
4    enviar (r_2, r_3, ..., r_p);
5 end
```

Figura 4.36: Código ejecutado para cada uno de los *tiles* del espacio de direcciones.

Como el volumen del bloque de cómputo es $v = x\frac{n}{p^2}$ (alto por ancho del *tile*), podemos calcular el tiempo de la macroetapa mediante la siguiente ecuación:

$$\mathcal{P}(x) = 2(\beta + \bar{p}\tau x\tfrac{n}{p^2}) + \alpha x\tfrac{n}{p^2}$$

donde β es la latencia y $\frac{1}{\tau}$ es el ancho de banda de la red.

Para encontrar el tamaño óptimo del *tile*, necesitamos minimizar el tiempo de ejecución $\Sigma\mathcal{P}(x)$. Para el recorrido horizontal minimizamos la siguiente función:

$$H(x) = \begin{cases} H_1(x) = \Sigma_1\mathcal{P}(x) = \left(\frac{n\bar{p}}{2x} + p\right)\mathcal{P}(x) & \text{si } x \leq \frac{n}{p^2} \\ H_2(x) = \Sigma_2\mathcal{P}(x) = \left(\frac{p^2\bar{p}}{2} + \frac{n}{px}\right)\mathcal{P}(x) & \text{en otro caso} \end{cases}$$

Mientras que para el recorrido vertical, debemos minimizar la función $V(x)$ definida mediante la expresión:

$$V(x) = \begin{cases} V_1(x) = \Sigma_3\mathcal{P}(x) = \left(\frac{n\bar{p}}{2x} + p\right)\mathcal{P}(x) & \text{si } x \leq \frac{2n}{p^2} \\ V_2(x) = \Sigma_4\mathcal{P}(x) = \left((\frac{p\bar{p}}{2} - 2)\frac{n}{px} + 2p\right)\mathcal{P}(x) & \text{en otro caso} \end{cases}$$

4.8.3. Optimizando el tamaño del *tile*

En esta subsección damos soluciones analíticas para el tamaño óptimo del *tile* (alto y ancho) que minimiza el tiempo de ejecución en los dos recorridos. Hay que tener en cuenta que se cumplen las siguientes propiedades:

$$\begin{aligned} H_1(x) &\geq H_2(x) \quad \text{para } x \leq \tfrac{n}{p^2}; \\ H_1(x) &= H_2(x) \quad \text{para } x = \tfrac{n}{p^2}; \\ H_1(x) &\leq H_2(x) \quad \text{en otro caso} \end{aligned}$$

y

$$\begin{aligned} V_1(x) &\geq V_2(x) \quad \text{para } x \leq \tfrac{2n}{p^2}; \\ V_1(x) &= V_2(x) \quad \text{para } x = \tfrac{2n}{p^2}; \\ V_1(x) &\leq V_2(x) \quad \text{en otro caso} \end{aligned}$$

Por lo tanto, tenemos que resolver los siguientes problemas:

$$\text{Problema } \mathcal{P}_1\text{: Minimizar } H(x) = \text{mín máx}(H_1(x), H_2(x))$$
$$\text{Problema } \mathcal{P}_2\text{: Minimizar } V(x) = \text{mín máx}(V_1(x), V_2(x))$$

Resolvemos estos problemas basándonos en el hecho de que las funciones $H_1(x)$, $H_2(x)$, $V_1(x)$ y $V_2(x)$ son funciones convexas de la forma $T(x) = \frac{A}{x} + Bx + C$, donde la solución óptima se obtiene para $x^* = \sqrt{\frac{A}{B}}$. Desarrollando las ecuaciones anteriores obtenemos:

$H_1 = V_1 = \frac{n\beta\bar{p}}{x} + \frac{nx}{p}(2\tau\bar{p} + \alpha) +$ const

y el mínimo global de H_1 y V_1 se encuentra en: $x_1^* = \sqrt{\frac{\beta\bar{p}p}{2\tau\bar{p}+\alpha}}$.

$H_2 = \frac{2n\beta}{px} + \frac{nx\bar{p}}{2}(2\tau\bar{p} + \alpha) +$ const

y el mínimo global de H_2 se encuentra en: $x_2^* = \sqrt{\frac{4\beta}{\bar{p}p(2\tau\bar{p}+\alpha)}}$.

$V_2 = \frac{n\beta(\bar{p}p-4)}{px} + \frac{2nx}{p}(2\tau\bar{p} + \alpha) +$ const

y el mínimo global de V_2 está en: $x_3^* = \sqrt{\frac{\beta(\bar{p}p-4)}{2(2\tau\bar{p}+\alpha)}}$.

Obviamente, tenemos que $x_1^* > x_2^*$ y $x_1^* > x_3^*$. Vamos a denominar x_h^{intsc} al punto donde $H_1(x) = H_2(x)$ ($x_h^{intsc} = \frac{n}{p^2}$) y x_v^{intsc} al punto donde $V_1(x) = V_2(x)$ ($x_v^{intsc} = \frac{2n}{p^2}$). Observamos que en el intervalo $[1, x_h^{intsc}]$ el mínimo de la función $H(x)$ es el mínimo de $H_1(x)$, mientras que en el intervalo $[x_h^{intsc}, \frac{n}{p}]$ viene dado por el mínimo de $H_2(x)$. Respectivamente, en el intervalo $[1, x_v^{intsc}]$ el mínimo de $V(x)$ es el mínimo de la función $V_1(x)$, mientras que en $[x_v^{intsc}, \frac{n}{p}]$ es el mínimo de $V_2(x)$. Considerando los resultados anteriores, obtenemos las siguientes ecuaciones como soluciones a los problemas $\mathcal{P}_1$ y $\mathcal{P}_2$.

$$x_h^* = \begin{cases} x_1^* = \sqrt{\frac{\beta\bar{p}p}{2\tau\bar{p}+\alpha}} & \text{si } x_1^* \leq x_h^{intsc} \\ x_h^{intsc} = \frac{n}{p^2} & \text{si } x_2^* \leq x_h^{intsc} \leq x_1^* \\ x_2^* = \sqrt{\frac{4\beta}{\bar{p}p(2\tau\bar{p}+\alpha)}} & \text{si } x_h^{intsc} \leq x_2^* \end{cases}$$

$$x_v^* = \begin{cases} x_1^* = \sqrt{\frac{\beta\bar{p}p}{2\tau\bar{p}+\alpha}} & \text{si } x_1^* \leq x_v^{intsc} \\ x_v^{intsc} = \frac{2n}{p^2} & \text{si } x_3^* \leq x_v^{intsc} \leq x_1^* \\ x_3^* = \sqrt{\frac{\beta(\bar{p}p-4)}{2(2\tau\bar{p}+\alpha)}} & \text{si } x_v^{intsc} \leq x_3^* \end{cases}$$

Las tres posibles situaciones para calcular el mínimo global en el caso del recorrido vertical es ilustrado en las figuras 4.37.

Hay que tener en cuenta que en las situaciones donde el tiempo de ejecución del algoritmo está dado por las funciones H_2 o V_2, existe un período de tiempo para el cual los procesadores se encuentran ociosos entre las diferentes etapas. Sin embargo, cuando el tiempo de ejecución lo establecen las funciones H_1 o V_1, los procesadores están ocupados de forma permanente. Cuando los puntos de interesección x_h^{intsc} o x_v^{intsc} son menores que 1, los algoritmos se ejecutan únicamente en la situación donde el tiempo depende

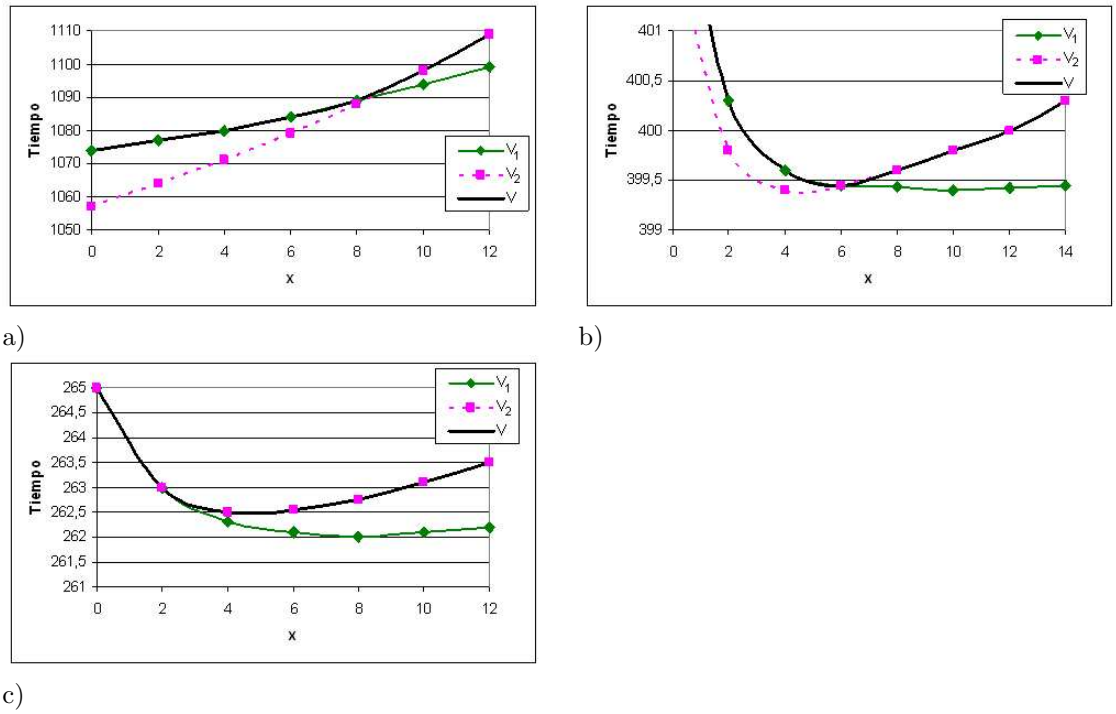

Figura 4.37: Encontrando el mínimo global para el *tile* de altura x_v^*. a) El mínimo global coincide con el mínimo de la función $V_1(x)$. b) El mínimo global se encuentra en la intersección de $V_1(x)$ y $V_2(x)$. c) El mínimo global coincide con el mínimo de la función $V_2(x)$.

de H_2 y V_2; esto significa que se están empleando demasiados procesadores para el problema de tamaño n dado.

Otro método para calcular el alto del *tile* es buscar el valor óptimo de su volumen $v^* = y^* x^*$, donde $y^* = \frac{n}{p^2}$ y disponemos de un espacio de iteraciones rectágular de tamaño $l \times h$ (l es la longitud del dominio y h es su altura).

Si realizamos una distribución de los *tiles* a los p procesadores por columnas, de forma cíclica, obtenemos $\frac{l}{py}$ bandas de tamaño $py \times h$. Bajo la hipótesis de que los procesadores se encuentran permanentemente ocupados (tiempo de ejecución establecido por las funciones $H_1(x)$ y $V_1(x)$), una banda necesita $\frac{h}{x}$ macroetapas, por lo tanto, la última banda comienza después de

$$\left(\frac{l}{py} - 1\right) \frac{h}{x}$$

y requiere $\frac{h}{x} + p$ macroetapas adicionales. Para obtener el tiempo real de ejecución, multiplicamos el número de macroetapas realizadas por el tiempo real de cada una de ellas ($\mathcal{P}$):

$$(\tfrac{lh}{pv} + p)P(v) = (\tfrac{lh}{pv} + p)(2(\beta + \tau\bar{p}v) + \alpha v)$$

Simplificando esta ecuación obtenemos:

$$\tfrac{2hb\beta}{pv} + vp(2\tau\bar{p} + \alpha) + \text{const}$$

y el volumen del *tile* que minimiza esta función es:

$$v^* = \sqrt{\frac{2lh\beta}{p^2(2\tau\bar{p}+\alpha)}}$$

Debido a que el tamaño del dominio para el recorrido vertical (respectivamente horizontal) es $\frac{n}{p} \times \frac{n\bar{p}}{2}$ (respectivamente $\frac{n\bar{p}}{2} \times \frac{n}{p}$), obtenemos, en ambos casos, que el volumen óptimo del *tile* es el mismo:

$$v^* = \frac{n}{p^2}\sqrt{\frac{p\bar{p}\beta}{2\tau\bar{p}+\alpha}}.$$

Esto significa que cuando: $p\bar{p}\beta \geq 2\tau\bar{p} + \alpha$, el tamaño del *tile*

$$(y^*, x^*) = (\frac{n}{p^2}, \sqrt{\frac{p\bar{p}\beta}{2\tau\bar{p}+\alpha}})$$

es una de las infinitas soluciones posibles del volumen óptimo.

4.9. Validación del modelo analítico

En esta sección vamos a validar nuestro modelo, comparando los resultados obtenidos de forma empírica frente al comportamiento estimado en la aplicación de nuestro modelo analítico. Vamos a utilizar los resultados de la ejecución intensiva, presentados en la sección 4.7, para comparar los valores mínimos de todas las ejecuciones, con los resultados correspondientes a las combinaciones de parámetros que obtenemos al aplicar nuestro modelo. El primer paso que debemos realizar antes de la validación es calcular cuáles son los mejores tiempos obtenidos en la ejecución intensiva realizada (sección 4.7) y cuáles son las combinaciones de ancho y alto del *tile* con las que se alcanzan esos resultados. En la tabla 4.17 mostramos la información de los mejores tiempos para las dos versiones no sincronizadas.

Hay que resaltar el hecho de que, en los resultados experimentales, el ancho del *tile* es el mismo en ambos algoritmos (horizontal y vertical). Además, si comprobamos el rendimiento de los algoritmos observamos que no existen diferencias significativas, las aceleraciones alcanzadas en ambos casos son similares (figura 4.38). Esta característica ya la habíamos comentado también en la sección 4.7.

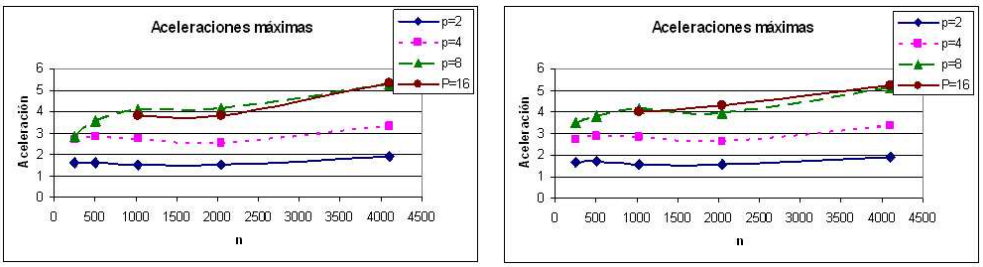

Figura 4.38: Aceleraciones máximas alcanzadas en la resolución de los diferentes problemas con distinto número de procesadores. Izquierda: versión horizontal no sincronizada. Derecha: versión vertical no sincronizada.

En la tabla 4.17 también podemos ver que los valores óptimos de y, obtenidos de forma experimental, se comportan de acuerdo a la ecuación de la sección 4.8: $y = \frac{n}{p^2}$, para $p \in [8, 16]$. También se cumple cuando $p = 4$ para los problemas pequeños ($n = 256, 512$). A continuación vamos a comprobar que podemos predecir estos valores mediante nuestra ecuación analítica. En las secciones posteriores, utilizaremos también una aproximación estadística para predecir las combinaciones óptimas de parámetros y compararemos los resultados obtenidos con ambos métodos.

Tabla 4.17: Tiempos de ejecución mínimos y aceleraciones del algoritmo *tiling* para los recorridos horizontal y vertical no sincronizados obtenidos de forma experimental y las combinaciones óptimas de ancho y alto del *tile*.

		Versión horizontal no sincronizada				Versión vertical no sincronizada			
p	n	y	x	Tiempo	Aceleración	y	x	Tiempo	Aceleración
2	256	32	6	0.16	1.61	32	8	0.16	1.64
2	512	32	14	0.75	1.63	32	12	0.72	1.69
2	1024	32	22	3.94	1.52	32	20	3.85	1.55
2	2048	128	14	20.18	1.51	128	13	19.80	1.54
2	4096	256	7	114.98	1.91	256	3	114.52	1.92
4	256	16	5	0.10	2.68	16	6	0.09	2.75
4	512	32	8	0.44	2.81	32	2	0.42	2.89
4	1024	32	14	2.18	2.74	32	11	2.10	2.84
4	2048	64	9	12.02	2.54	64	12	11.52	2.65
4	4096	128	5	66.11	3.32	128	9	65.54	3.35
8	256	4	4	0.09	2.84	4	11	0.07	3.51
8	512	8	5	0.34	3.57	8	6	0.32	3.80
8	1024	16	3	1.46	4.09	16	4	1.45	4.12
8	2048	32	13	7.31	4.17	32	4	7.74	3.94
8	4096	64	9	41.70	5.26	64	8	42.69	5.14
16	1024	4	1	1.57	3.79	4	6	1.49	4.01
16	2048	8	9	7.99	3.82	8	11	7.08	4.31
16	4096	16	4	41.23	5.32	16	6	42.00	5.22

En la sección 4.8 comentamos que debe cumplirse la desigualdad $p\bar{p}\beta \geq 2\tau\bar{p} + \alpha$ para poder aplicar el modelo. Si comprobamos los resultados de la tabla 4.17, observamos que esta desigualdad se mantiene para cada instancia con un número de procesadores mayor de 2 ($p = 4, 8$ y 16). A pesar de esto, vamos a comparar los valores estimados frente a los resultados reales obtenidos para todas las instancias (incluido $p = 2$).

Para aplicar el modelo analítico es necesario calcular el valor de varios parámetros (sección 4.8). Los valores de β y τ han sido calculados mediante regresión lineal a partir de la ejecución de un *ping-pong* para diferentes tamaños. El proceso utilizado en la máquina *SGI Origin 3800* para obtener β y τ ha sido el mismo que el empleado en nuestra plataforma heterogénea en la sección 1.5.1.

La evaluación del parámetro α requiere una consideración especial: el valor de α depende de la pareja (i, j) y requiere un proceso costoso. Por motivos de simplicidad estimamos α para un tamaño de problema dado, n, y lo aproximamos como la media del tiempo del algoritmo secuencial sobre el conjunto de puntos: $\alpha(n) = \frac{T(n)}{n^2/2}$, donde $T(n)$ denota el tiempo empleado en el cómputo de la matriz de dependencias triangular; es decir, el tiempo de ejecución del algoritmo secuencial. El tiempo $T(n)$ puede ser aproximado mediante una regresión lineal de las ejecuciones secuenciales y obtenemos la siguiente ecuación:

$$T(n) = \begin{cases} 1{,}479e-09 * n^3 + 4{,}199e-06 * n^2 - 7{,}472e-06 * n - 1{,}516e-02 & \text{si } n \leq 1024 \\ 2{,}201e-09 * n^3 + 7{,}552e-06 * n^2 - 2{,}038e-02 * n + 1{,}992e+01 & \text{en otro caso} \end{cases}$$

La tabla 4.18 muestra el *tile* óptimo obtenido por los modelos y los errores relativos cometidos al comparar estos resultados con los valores óptimos de la tabla 4.17. En la tabla se encuentra información sobre los dos recorridos: horizontal no sincronizada y vertical no sincronizada.

Si observamos las cifras de los errores en la tabla 4.18, comprobamos que las predicciones teóricas se ajustan a los resultados reales para el rango de validación de las ecuaciones ($p = 4, 8$ y 16), con la

Tabla 4.18: Predicción de los parámetros para una ejecución óptima usando la predicción analítica, para las versiones horizontal y vertical no sincronizada.

		Versión horizontal no sincronizada				Versión vertical no sincronizada			
p	n	y	x	Tiempo Real	Error	y	x	Tiempo Real	Error
2	256	64	1	0.16	1.04 %	64	1	0.16	0.48 %
2	512	128	1	1.05	40.38 %	128	1	1.01	39.06 %
2	1024	256	1	5.33	35.37 %	256	1	5.21	35.35 %
2	2048	512	1	26.94	33.49 %	512	1	26.42	33.44 %
2	4096	1024	1	184.74	60.66 %	1024	1	182.82	59.64 %
4	256	16	2	0.10	2.19 %	16	2	0.10	3.41 %
4	512	32	2	0.45	3.25 %	32	2	0.42	0.00 %
4	1024	64	2	2.53	16.29 %	64	2	2.41	14.75 %
4	2048	128	2	12.95	7.73 %	128	2	13.09	13.65 %
4	4096	256	1	70.73	6.99 %	256	1	71.31	8.80 %
8	256	4	5	0.10	13.39 %	4	5	0.07	1.08 %
8	512	8	5	0.34	0.00 %	8	5	0.33	3.02 %
8	1024	16	4	1.81	24.31 %	16	4	1.45	0.00 %
8	2048	32	4	7.79	6.52 %	32	4	7.74	0.00 %
8	4096	64	3	43.10	3.35 %	64	3	46.64	9.24 %
16	1024	4	9	2.99	90.37 %	4	9	1.76	18.39 %
16	2048	8	8	8.23	3.08 %	8	8	7.26	2.58 %
16	4096	16	6	45.79	11.05 %	16	6	42.00	0.00 %

excepción del alto porcentaje de error obtenido para la instancia de tamaño $n = 1024$, especialmente para la versión horizontal no sincronizada. Se observa que los errores para la versión vertical no sincronizada son siempre algo más reducidos. Sin embargo, para 2 procesadores la hipótesis del modelo no se satisfacen y los errores llegan al 60 %.

4.10. Modelo estadístico para el problema de la estructura secundaria del RNA

En esta sección vamos a desarrollar un modelo estadístico con capacidad de predicción del *tile* óptimo para resolver el problema de la estructura secundaria del RNA. Vamos a utilizar la información que nos ofrece la tabla 4.17. En particular, la regresión lineal puede ser desarrollada sobre y^* y x^*, de modo que podamos obtener para cualquier procesador ecuaciones de la forma $y^* = a * n + b$. De forma análoga se pueden obtener ecuaciones para x^*. La tabla 4.19 muestra las ecuaciones de nuestro modelo estadístico.

A continuación comprobamos nuestro modelo estadístico mediante el mismo proceso de validación realizado con el modelo analítico: se calculan los valores de ancho y alto del *tile* para los problemas resueltos mediante la ejecución masiva en la sección 4.7 y comparamos los tiempos de ejecución obtenidos con esos parámetros, con los resultados mínimos presentados en la tabla 4.17. En la tabla 4.20 presentamos los parámetros (y^*, x^*) que obtenemos con las ecuaciones de la tabla 4.19 y el error cometido en cada caso.

Comparamos los resultados de la tabla 4.20 con los resultados correspondientes al modelo analítico (tabla 4.18) y observamos que los errores se han reducido considerablemente en la mayoría de los casos al ejecutar con el modelo estadístico, frente a los datos del modelo analítico, incluso para el problema $n = 1024$. En la tabla 4.20 comprobamos que el modelo estadístico predice correctamente el rendimiento de nuestra aplicación paralela para cualquier número de procesadores.

Tabla 4.19: Ecuaciones para la predicción estadística de los valores (y^*, x^*) en los recorridos horizontal y vertical no sincronizados.

	Horizontal				Vertical	
	y		x		x	
p	b	a	b	a	b	a
2	-2.66667	0.06216	14.458333	-0.001171	14.666667	-0.002184
4	9.33333	0.02839	9.2500000	-0.0006615	6.083333	0.001208
8	-5.507e-15	1.563e-02	4.166667	0.001659	6.833333	-0.000147
16	-2.072e-15	3.906e-03	3.5000000	0.0004883	8.5000000	-0.0003488

Tabla 4.20: Predicción de los parámetros para una ejecución óptima usando la predicción estadística, para las versiones horizontal y vertical no sincronizada.

		Versión horizontal no sincronizada				Versión vertical no sincronizada			
p	n	y	x	Tiempo Real	Error	y	x	Tiempo Real	Error
2	256	16	14	0.17	4.46 %	16	14	0.17	5.39 %
2	512	32	14	0.75	0.00 %	32	14	0.74	1.95 %
2	1024	64	13	4.03	2.34 %	64	14	4.04	5.04 %
2	2048	128	12	20.27	0.43 %	128	10	19.85	0.25 %
2	4096	256	10	116.11	0.98 %	256	6	114.59	0.06 %
4	256	16	9	0.10	5.60 %	16	6	0.09	0.00 %
4	512	32	9	0.45	2.10 %	32	7	0.49	16.64 %
4	1024	32	9	2.23	2.31 %	32	7	2.15	2.57 %
4	2048	64	8	12.03	0.10 %	64	9	11.92	3.44 %
4	4096	128	7	66.52	0.62 %	128	10	65.64	0.15 %
8	256	4	5	0.10	13.39 %	4	7	0.07	0.01 %
8	512	8	5	0.34	0.00 %	8	7	0.36	12.77 %
8	1024	16	6	1.52	4.14 %	16	7	1.63	12.93 %
8	2048	32	8	7.72	5.55 %	32	7	8.09	4.52 %
8	4096	64	10	42.59	2.12 %	64	6	44.08	3.26 %
16	1024	4	4	1.92	22.04 %	4	8	1.71	14.71 %
16	2048	8	4	8.39	5.03 %	8	8	7.26	2.58 %
16	4096	16	5	43.15	4.63 %	16	7	45.11	7.39 %

4.11. Comparativa modelo analítico vs modelo estadístico

Como experimento de validación adicional, decidimos observar el comportamiento del algoritmo para valores de n no incluidos en el estudio computacional intensivo. La tabla 4.21 presenta los resultados para dos casos: $n = 3200$ y $n = 8192$. Los valores de y y x fueron obtenidos a partir de las ecuaciones de la sección 4.8 (modelo analítico) y la sección 4.10 (modelo estadístico). Las aceleraciones obtenidas se comportan de acuerdo a la progresión de las aceleraciones óptimas observadas en las tablas 4.18 y 4.20.

Tabla 4.21: Predicción de los parámetros óptimos para dos tamaños de problemas ($n = 3000$ y $n = 8192$) con los modelos analítico y estadístico.

Predicción Analítica						
			Horizontal		Vertical	
p	n	y	x	Aceleración	x	Aceleración
4	3200	200	2	2.95	2	2.90
4	8192	512	1	2.47	2	2.55
8	3200	50	3	4.41	3	4.71
8	8192	128	1	5.05	1	4.71
Predicción Estadística						
			Horizontal		Vertical	
p	n	y	x	Aceleración	x	Aceleración
2	3200	200	11	1.79	8	1.80
2	8192	512	5	1.63	1	1.62
4	3200	100	7	2.81	10	3.08
4	8192	256	4	2.79	16	2.68
8	3200	50	9	4.37	6	4.64
8	8192	128	18	4.12	6	5.37

En la tabla 4.21 podemos comprobar que no existe un tipo de predicción (analítica o estadística), ni un esquema de resolución (horizontal o vertical) que sea siempre mejor que los otros. Aunque, para la mayoría de los problemas, las aceleraciones máximas alcanzadas se consiguen con la versión vertical no sincronizada, con los parámetros estimados mediante el modelo estadístico: por ejemplo, con 8 procesadores y el problema de tamaño $n = 8192$ se consigue una aceleración de 5.37 con la versión vertical y la predicción estadística, frente a una aceleración de 4.12 con la versión horizontal y el mismo tipo de predicción.

4.11.1. Casos reales

Aplicamos ahora nuestra metodología formal a moléculas de RNA no sintéticas. Descargamos varias moléculas desde la base de datos *European Ribosomal RNA* [70]: el *Bacillus Anthracis* (el bacilo del ántrax) con $n = 1506$, el *Bacillus Cereus* (una bacteria que provoca un envenenamiento alimenticio) con $n = 2782$, la *Simkania Negevensis* (una bacteria que produce infecciones respiratorias) con $n = 2950$ y la *Acidianus Brierleyi* (molécula con capacidad para oxidar la pirita) con $n = 3046$. La tabla 4.22 muestra los tiempos secuenciales obtenidos para todos ellos. Obtenemos el *tiling* óptimo para ejecutarlos utilizando las predicciones analítica y estadística. La tabla 4.22 resume los resultados obtenidos en las ejecuciones paralelas. El algoritmo utilizado fue la versión horizontal, donde los procesadores se sincronizan al final del cómputo del rectángulo y no es necesario que se cumpla que y^* sea un divisor de $\frac{n}{p^2}$. La precisión del modelo estadístico es sensiblemente mejor para los casos considerados. Debido a que la evaluación de los valores de a y b para el modelo estadístico implica realizar un conjunto adicional de experimentos, el modelo analítico puede ser la opción para las situaciones donde se cumplan las hipótesis para su aplicación.

En la figura 4.39 presentamos, de forma gráfica, las aceleraciones obtenidas al resolver el problema de la predicción de la estructura secundaria para las moléculas reales anteriores utilizando las combinaciones de parámetros calculadas por el modelo analítico y el modelo estadístico.

Tabla 4.22: Predicción de los parámetros óptimos mediante la predicción analítica y estadística para moléculas no sintéticas RNA.

Bacillus Anthracis - n = 1506 - Tiempo secuencial: 14.10 seg.								
	Analítico				Estadístico			
p	y	x	Tiempo	Aceleración	y	x	Tiempo	Aceleración
2	376	1	12.75	1.11	105	14	9.22	1.53
4	94	2	6.36	2.22	50	5	6.50	2.17
8	24	4	4.15	3.40	24	5	4.12	3.42
16	6	9	6.52	2.16	6	4	5.48	2.57
Bacillus Cereus - n = 2782 - Tiempo secuencial: 70.60 seg.								
	Analítico				Estadístico			
p	y	x	Tiempo	Aceleración	y	x	Tiempo	Aceleración
2	695	1	58.20	1.21	199	12	39.63	1.78
4	174	2	28.72	2.46	83	7	25.81	2.73
8	43	4	18.06	3.91	43	5	17.81	3.96
16	11	7	19.33	3.65	11	5	17.89	3.95
Simkania Negevensis - n = 2950 - Tiempo secuencial: 83.94 seg.								
	Analítica				Estadística			
p	y	x	Tiempo	Aceleración	y	x	Tiempo	Aceleración
2	737	1	68.87	1.22	212	11	46.10	1.82
4	184	2	32.66	2.57	88	7	33.14	2.53
8	46	3	19.62	4.28	46	6	21.74	3.86
16	12	7	19.55	4.29	12	5	21.03	3.99
Acidianus Brierleyi - n = 3046 - Tiempo secuencial: 92.39 seg.								
	Analítica				Estadística			
p	y	x	Tiempo	Aceleración	y	x	Tiempo	Aceleración
2	761	1	73.58	1.26	219	11	48.51	1.90
4	190	2	34.42	2.68	90	7	32.88	2.81
8	46	3	21.32	4.33	48	6	23.84	3.88
16	12	7	21.78	4.24	12	5	20.23	4.57

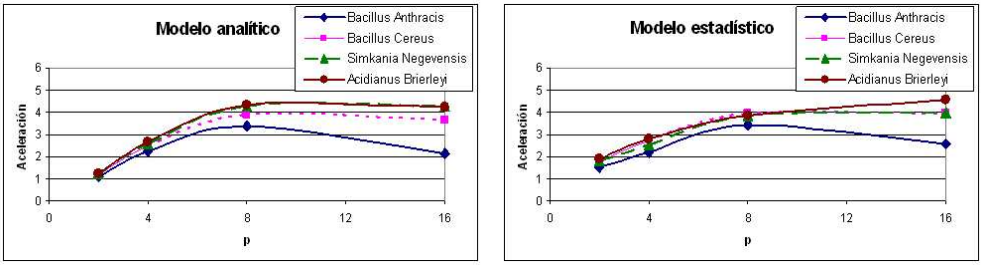

Figura 4.39: Aceleraciones obtenidas en las ejecuciones con la versión horizontal sincronizada para las moléculas reales. Izquierda: Utilizando las combinaciones de parámetros calculadas por el modelo analítico. Derecha: Utilizando las combinaciones de parámetros calculadas por el modelo estadístico.

Se observa en las gráficas de la figura 4.39 que las aceleraciones se incrementan al aumentar el tamaño de las moléculas. Utilizando las combinaciones del modelo analítico, las dos moléculas de mayor tamaño (*Simkenia Negevensis* y *Acidianus Brierleyi*) presentan un rendimiento similar; mientras que para el modelo estadístico, 16 procesadores y el problema de mayor tamaño se obtiene aceleración más alta.

4.12. La paralelización del paquete de Viena: tamaño variable del *tile*

En esta sección describimos una modificación realizada en el programa paralelo explicado en la sección 4.6. El programa mantiene el proceso de resolución explicado en la secciones 4.5 y 4.6: primero se realiza la fase de preprocesamiento (el cálculo de los triángulos) y, a continuación, se opera sobre los rectángulos. La única diferencia con el proceso explicado en la sección 4.6 consiste en la resolución con p procesadores: el cálculo de los triángulos y la ejecución del programa con 2 procesadores no cambia.

Cuando trabajamos con p procesadores nos interesa trabajar con un ancho del *tile* distinto para cada rectángulo, manteniendo constante el alto del *tile*. De esta forma podemos comparar los resultados obtenidos en la sección 4.7 con los conseguidos con la nueva versión del programa y comprobar si la simplificación realizada en el modelo analítico afecta a la optimalidad.

En la figura 4.40 mostramos un ejemplo de resolución de un problema para una molécula de tamaño $n = 16$ con 4 procesadores.

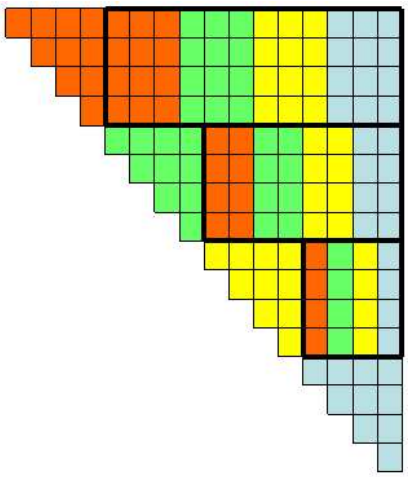

Figura 4.40: Resolución del problema utilizando un volumen de *tile* diferente para cada rectángulo.

En la figura 4.40 se observa que el ancho del *tile* utilizado en cada rectángulo es diferente y esto implica que es necesario realizar una sincronización al final del cálculo de cada uno; sin ella, no sería posible resolver el problema, puesto que los procesadores no dispondrían de todos los datos necesarios. Por este motivo, hemos aplicado estas modificaciones únicamente a la versión horizontal sincronizada. Como ya comentamos en la sección 4.6, en la versión sincronizada no se producen limitaciones a los valores de y. Además, al permitir modificar el valor de y en cada uno de los rectángulos, el número de combinaciones posibles se ve incrementado de forma considerable. La única restricción que hemos impuesto, en la resolución del problema, consiste en considerar que el ancho del *tile* asignado a un rectángulo nunca será menor que el ancho del *tile* en los rectángulos inferiores (los rectángulos más pequeños).

4.13. Resultados computacionales: tamaño variable del *tile*

En esta sección presentamos algunos de los resultados computacionales obtenidos en una ejecución intensiva de la nueva versión del programa (sección 4.12). Hay que tener en cuenta que, en esta versión, la cantidad de combinaciones de número de procesadores, ancho y alto del *tile* para una validación experimental es considerable. Para 2 procesadores, el número de ejecuciones realizadas es el mismo que en la versión horizontal sincronizada con un tamaño constante de *tile*, puesto que sólo existe un cuadrado. Sin embargo, para 4 y 8 procesadores, es necesario probar todas las combinaciones posibles con los valores del ancho del *tile* para la versión horizontal sincronizada (sección 4.7), en cada uno de los rectángulos. Debido al número de rectángulos a calcular en cada caso y al conjunto de valores que pueden tomar y y x, el número de ejecuciones realizadas en la ejecución se ha visto incrementado considerablemente. El problema de tamaño $n = 2048$ no ha resuelto con 8 procesadores, mediante la ejecución intensiva, porque supondría realizar más de 170000 ejecuciones. En la sección 4.14 resolvemos este problema con las combinaciones de parámetros calculadas por los modelos desarrollados (modelo analítico y estadístico). Debido al alto número de ejecuciones realizadas, no resulta sencillo mostrar todos los resultados en tablas; por este motivo hemos decidido presentar solamente dos comparativas entre los tiempos de ejecución obtenidos con la primera versión del programa (esquema horizontal sincronizado) y los tiempos para la nueva versión.

En la tabla 4.23 se muestra una comparativa de los tiempos de ejecución obtenidos con las dos versiones del programa utilizando las mismas combinaciones de parámetros en los dos casos. Queremos comprobar la sobrecarga introducida al permitir la asignación de diferentes valores de ancho del *tile* en cada uno de los rectángulos. Las combinaciones de parámetros utilizadas en esta prueba, han sido las combinaciones óptimas de la versión horizontal sincronizada con un tamaño constante del *tile*. En la nueva versión hemos empleado el mismo ancho del *tile* a todos los rectángulos para imitar el funcionamiento de la primera versión. La columna p de la tabla 4.23 contiene el número de procesadores con el que se ha resuelto el problema; en n se encuentra el tamaño del problema; las columnas y y x muestran el volumen del *tile* (ancho y alto respectivamente); *Tiempo - tile constante* y *Tiempo - tile variable* son los tiempos de ejecución de las dos versiones; la última columna muestra el Porcentaje de mejora alcanzado con la versión que varía el tamaño del *tile*, frente a la versión que lo mantiene fijo. Este porcentaje es mayor que cero cuando modificar el ancho del *tile* es mejor y negativo cuando el tiempo de ejecución de esta nueva versión es peor. Debido a que los resultados se encuentran redondeados a 2 decimales, aunque los tiempos de ejecución parezcan iguales, el porcentaje que aparece en la tabla es distinto de cero.

Tabla 4.23: Comparación de los tiempos de ejecución obtenidos con las dos versiones del programa, para las combinaciones de parámetros óptimas para un tamaño constante de *tile* con el esquema horizontal sincronizado. En la nueva versión se asigna a todos los rectángulos el mismo ancho de *tile*.

p	n	y	x	Tiempo - *tile* constante	Tiempo - *tile* variable	Porcentaje de mejora
2	256	32	6	0.16	0.16	-0.29 %
2	512	32	12	0.73	0.74	-1.04 %
2	1024	40	29	3.71	3.77	-1.53 %
2	2048	150	13	18.96	18.99	-0.18 %
4	256	16	3	0.10	0.12	-15.28 %
4	512	32	4	0.47	0.47	-0.15 %
4	1024	32	4	2.21	2.21	-0.09 %
4	2048	60	6	11.96	11.97	-0.09 %
8	256	4	3	0.11	0.11	-8.72 %
8	512	8	2	0.37	0.37	-0.37 %
8	1024	16	7	1.52	1.68	-10.48 %

En la tabla 4.23 se observa que los tiempos son prácticamente idénticos aunque ligeramente peores para la versión con el *tile* variable. Este efecto se produce por la sobrecarga añadida al permitir la modificación del *tile*. Sin embargo, existen únicamente 3 casos donde el porcentaje es mayor del 5 %. Esto significa que la sobrecarga introducida es mínima.

En la tabla 4.24 presentamos los resultados mínimos obtenidos al resolver los problemas de tamaño $n = 256$, 512, 1024 y 2048, con el programa con tamaños variables de *tile*, comparándolos con los datos ejecutando con un tamaño de *tile* constante. La columna y contiene los diferentes valores de ancho de *tile* empleados en toda la ejecución. Estos valores se muestran en orden descendente de rectángulos; por ejemplo: para $n = 2048$ y $p = 4$, el rectángulo superior se resuelve con un ancho de *tile* $y = 75$, el rectángulo intermedio con $y = 60$ y el rectángulo inferior se resuelve con un *tile* más reducido ($y = 40$).

Tabla 4.24: Tiempos de ejecución mínimos y aceleraciones al ejecutar con tamaños variables de *tile* y el algoritmo horizontal sincronizado. También presentamos las combinaciones de ancho y alto del *tile* donde se obtienen los mejores resultados y el porcentaje de mejora de la nueva versión

p	n	Tiempo - *tile* constante	y	x	Tiempo - *tile* variable	Aceleración	Porcentaje de mejora
2	256	0.16	32	3	0.15	1.69	2.23 %
2	512	0.73	32	17	0.71	1.71	1.90 %
2	1024	3.71	75	16	3.65	1.64	1.59 %
2	2048	18.96	150	11	18.77	1.62	1,00 %
4	256	0.10	16 - 16 - 8	3	0.10	2.54	1.13 %
4	512	0.47	32 - 32 - 16	5	0.44	2.79	6.43 %
4	1024	2.21	40 - 40 - 32	11	2.14	2.78	2.95 %
4	2048	11.96	75 - 60 - 40	11	11.67	2.61	2.47 %
8	256	0.11	4 - 4 - 4 - 4 - 4 - 4 - 2	3	0.09	2.75	9.91 %
8	512	0.37	8 - 8 - 8 - 8 - 8 - 8 - 2	4	0.34	3.56	7.61 %
8	1024	1.52	16 - 16 - 16 - 16 - 16 - 8 - 4	3	1.46	4.08	3.64 %

Se observa en la tabla de resultados que, a pesar de la pequeña sobrecarga introducida en la nueva versión, se consigue reducir el tiempo mínimo de ejecución, llegando a alcanzar un porcentaje de 9.91 % con 8 procesadores y el problema más pequeño. Sin embargo, se observa que, en la mayoría de los casos, el porcentaje de mejora crece al aumentar el número de procesadores. En la figura 4.41 comparamos los resultados obtenidos con las dos versiones del programa.

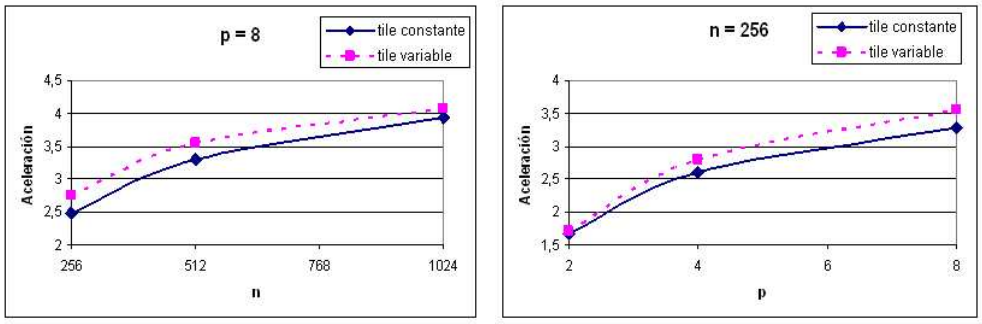

Figura 4.41: Aceleraciones obtenidas al resolver los problemas utilizando las dos versiones del programa: con *tiles* de tamaño constante o variable. Izquierda: Fijando el número de procesadores. Derecha: Fijando el tamaño del problema.

4.14. Validación de los modelos: tamaño variable de *tile*

Desarrollar modelos específicos para la versión del programa donde modificamos el tamaño del *tile* en cada rectángulo, implica una gran dificultad. Por este motivo, aplicamos los modelos desarrollados en las secciones 4.8 y 4.10 a este nuevo esquema de ejecución, utilizando el siguiente método: consideramos cada uno de los rectángulos de forma independiente. Suponemos que cada uno de ellos es el rectángulo superior (rectángulo de mayor tamaño) de un problema alternativo y aplicamos los modelos diseñados anteriormente sobre ese nuevo problema. El resultado obtenido al aplicar el modelo es el ancho del *tile* que utilizamos sobre el rectángulo correspondiente en el problema original.

Vamos a validar nuestros modelos para el caso con tamaño variable de *tiles* mediante el mismo proceso seguido en la sección 4.9, 4.10 y 4.11: primero aplicamos los modelos a los problemas resueltos en la sección 4.13 (problemas ejecutados mediante una ejecución masiva) y comparamos los resultados. A continuación, los aplicamos a problemas no resueltos comprobando las aceleraciones alcanzadas con los valores de los problemas anteriores. Por último, utilizamos las combinaciones calculadas por los modelos para resolver los casos reales presentados en la sección 4.11. En la tabla 4.25 mostramos los resultados reales obtenidos con las combinaciones de parámetros calculadas por el modelo analítico y el modelo estadístico.

Tabla 4.25: Predicción de los parámetros para una ejecución óptima utilizando tamaños variables de *tile*, empleando el modelo analítico y el modelo estadístico.

Modelo analítico					
p	n	y	x	Tiempo Real	Error
2	256	64	1	0.16	2.97 %
2	512	128	1	1.09	52.36 %
2	1024	256	1	5.23	39.04 %
2	2048	512	1	26.35	40.37 %
4	256	16 - 11 - 5	2	0.18	79.07 %
4	512	32 - 21 - 11	2	0.47	7.10 %
4	1024	64 - 43 - 21	2	2.44	13.62 %
4	2048	128 - 85 - 43	2	12.26	5.13 %
8	256	4 - 3 - 3 - 2 - 2 - 1 - 1	5	0.12	22.99 %
8	512	8 - 7 - 6 - 5 - 3 - 2 - 1	5	0.38	8.76 %
8	1024	16 - 14 - 11 - 9 - 7 - 5 - 2	4	1.64	6.52 %
Modelo estadístico					
p	n	y	x	Tiempo Real	Error
2	256	12	15	0.16	3.40 %
2	512	31	15	0.74	3.11 %
2	1024	69	14	3.87	3.05 %
2	2048	145	13	19.05	1.48 %
4	256	20 - 17 - 14	3	0.15	43.12 %
4	512	30 - 23 -17	4	0.47	6.57 %
4	1024	50 - 36 - 23	6	2.37	10.62 %
4	2048	89 - 63 - 36	8	12.50	7.11 %
8	256	13 - 12 -10 - 8 - 6 - 4 - 2	1	0.12	26.55 %
8	512	24 - 21 - 18 - 15 - 12 - 8 - 4	1	0.37	6.18 %
8	1024	41 - 37 - 32 - 27 - 21 - 15 - 8	1	1.58	2.87 %

Comparando los resultados correspondientes a los dos modelos (tabla 4.25) con los presentados en las tablas 4.18 y 4.20 (ejecuciones con un tamaño constante de *tile*), observamos que con 2 procesadores se sigue sin cumplir las hipótesis del modelo analítico y los errores cometidos son altos; sin embargo, con 2 procesadores y las combinaciones aportadas por el modelo estadístico los errores son inferiores al

3.5 %. Con 4 y 8 procesadores observamos que los errores más altos se encuentran en el problema de tamaño $n = 256$. Esto se debe a que el tiempo necesario para resolver el problema es muy reducido (0.09 segundos para 8 procesadores) y aumentar ligeramente el tiempo de ejecución hasta 0.12 segundos supone un error superior al 25 %. Sin embargo, el error obtenido al resolver el problema de tamaño $n = 1024$, se ha reducido frente a los resultados de las tablas 4.18 y 4.20. En general se observa que el error se reduce cuando se incrementa el tamaño del problema.

A continuación, hemos realizado la validación del modelo utilizando problemas que no hemos ejecutado de forma intensiva y comparamos las aceleraciones alcanzadas con las aceleraciones óptimas de los problemas anteriores. En la tabla 4.26 mostramos los resultados para los problemas de tamaño $n = 2048$ y 8192 con 8 procesadores y para el problema de tamaño $n = 4096$ con 16 procesadores.

Tabla 4.26: Predicción de los parámetros óptimos para los problemas $n = 4096$ $n = 8192$ usando predicción analítica y estadística.

Predicción Analítica					
p	n	y	x	Tiempo Real	Aceleración
8	2048	32 - 27 - 23 - 18 - 14 - 9 - 5	4	10.70	2.85
8	8192	128 - 110 - 91 - 73 - 55 - 37 - 18 - 128	1	364.24	4.30
16	4096	16 - 15 - 14 - 13 - 12 - 11 - 10 - 9 - 7 - 6 - 5 - 4 - 3 - 2 - 1	6	56.32	3.90
Predicción Estadística					
p	n	y	x	Tiempo Real	Aceleración
8	2048	60 - 56 - 51 - 45 - 37 - 27 - 15	2	9.77	3.12
8	8192	174 - 105 - 72 - 63 - 63 - 62 - 45	1	361.07	4.34
16	4096	47 - 47 - 47 - 47 - 47 - 46 - 45 - 43 - 40 - 37 - 33 - 28 - 23 - 16 - 9	1	43.47	5.05

Se observa que las aceleraciones obtenidas en todos los casos son altas, especialmente con las combinaciones correspondientes al modelo estadístico, donde se alcanza una aceleración de 5.05 con 16 procesadores.

4.14.1. Casos reales

Por último, hemos resuelto los problemas de moléculas no sintéticas. Utilizamos las mismas moléculas que en la sección 4.11, pero ahora resolvermos estos problemas utilizando tamaños variables de *tiles* (tablas 4.27 y 4.28).

En las tablas 4.27 y 4.28 observamos que seguimos obteniendo mejores resultados cuando aplicamos el modelo estadístico, frente a los resultados con los parámetros calculados mediante el modelo analítico. Las aceleraciones se incrementan al aumentar el tamaño del problema; este incremento se observa especialmente con 16 procesadores y el modelo analítico. Además, si comparamos estos resultados con los de la tabla 4.22 (ejecutando con un tamaño constante de *tile*), observamos que, en la mayoría de las ejecuciones, se reduce el tiempo de ejecución al utilizar distintos tamaños de *tile* (figura 4.42).

4.15. Conclusiones

En este capítulo proponemos un nuevo esquema de paralelización para una amplia clase de recurrencias, con un espacio de iteraciones triangular y dependencias no uniformes. Estamos particularmente

Tabla 4.27: Predicción de los parámetros óptimos mediante la predicción analítica para la ejecución con moléculas no sintéticas de RNA.

Bacillus Anthracis - n = 1506 - Tiempo secuencial: 14.10 seg.				
p	y	x	Tiempo	Aceleración
2	376	1	12.56	1.12
4	94 - 63 - 31	2	6.52	2.16
8	24 - 20 - 17 - 13 - 10 - 7 - 3	4	4.37	3.23
16	6 - 5 - 5 - 5 - 4 - 4 - 4 - 3 - 3 - 2 - 2 - 2 - 1 - 1 - 1	9	9.52	1.48
Bacillus Cereus - n = 2782 - Tiempo secuencial: 70.60 seg.				
p	y	x	Tiempo	Aceleración
2	695	1	57.59	1.23
4	174 - 116 - 58	2	26.73	2.64
8	43 - 37 - 31 - 25 - 19 - 12 - 6	4	19.18	3.68
16	11 - 10 - 9 - 9 - 8 - 7 - 7 - 6 - 5 - 4 - 4 - 3 - 2 - 1 - 1	7	25.87	2.73
Simkania Negevensis - n = 2950 - Tiempo secuencial: 83.94 seg.				
p	y	x	Tiempo	Aceleración
2	737	1	68.59	1.22
4	184 - 123 - 61	2	30.44	2.76
8	46 - 40 - 33 - 26 - 20 - 13 - 7	3	21.86	3.84
16	12 - 11 - 10 - 9 - 8 - 7 - 6 - 5 - 5 - 4 - 3 - 2 - 2 - 1	7	23.63	3.55
Acidianus Brierleyi - n = 3046 - Tiempo secuencial: 92.39 seg.				
p	y	x	Tiempo	Aceleración
2	761	1	72.57	1.27
4	190 - 127 - 63	2	31.88	2.90
8	48 - 41 - 34 - 27 - 20 - 14 - 7	3	21.88	4.22
16	12 - 10 - 10 - 9 - 8 - 7 - 6 - 6 - 5 - 4 - 3 - 2 - 2 - 1	7	26.21	3.52

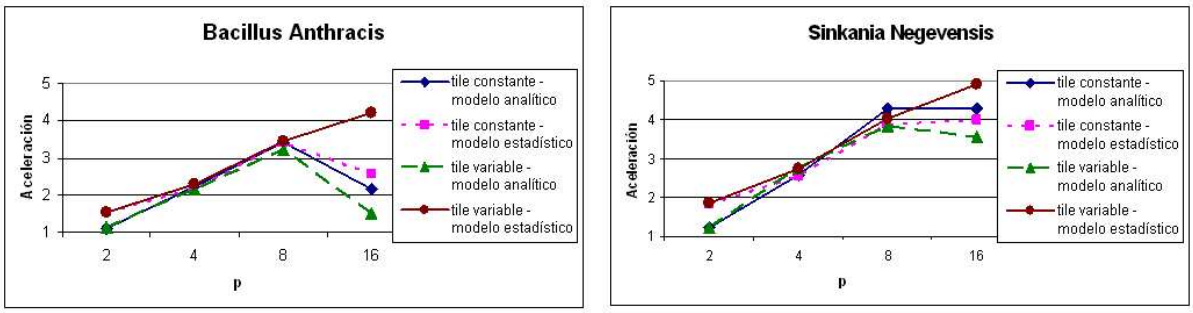

Figura 4.42: Aceleraciones máximas alcanzadas en la resolución de diferentes problemas con moléculas reales utilizando los parámetro calculados con los modelos analítico y estadístico para las dos versiones del programa (tamaño constante y variable de *tile*). Izquierda: *Bacillus Anthracis* ($n = 1506$). Derecha: *Simkania Negevensis* ($n = 2950$).

interesados en las recurrencias de la programación dinámica para resolver el problema de la predicción de la estructura secundaria del RNA y paralelizamos una aplicación real que resuelve el problema (el paquete de Viena).

Hemos estudiado y comparado tres versiones de nuestra estrategia paralela: horizontal no sincronizada, vertical no sincronizada y horizontal sincronizada. Extendemos la aproximación *tiling* para la clase

Tabla 4.28: Predicción de los parámetros óptimos mediante la predicción estadística para la ejecución de moléculas no sintéticas de RNA.

Bacillus Anthracis - n = 1506 - Tiempo secuencial: 14.10 seg.				
p	y	x	Tiempo	Aceleración
2	105	14	9.13	1.54
4	68 - 49 - 29	7	6.17	2.29
8	52 - 48 - 43 - 37 - 29 - 21 - 11	2	4.10	3.44
16	35 - 33 - 32 - 30 - 28 - 26 - 24 - 22 - 20 - 18 - 15 - 13 - 10 - 7 - 5	1	3.36	4.19
Bacillus Cereus - n = 2782 - Tiempo secuencial: 70.60 seg.				
p	y	x	Tiempo	Aceleración
2	199	12	39.15	1.80
4	116 - 82 - 46	9	24.98	2.83
8	63 - 62 - 59 - 53 - 46 - 35 - 20	2	16.78	4.21
16	46 - 46 - 45 - 43 - 41 - 39 - 37 - 35 - 32 - 28 - 25 - 21 - 17 - 12 - 7	1	14.49	4.87
Simkania Negevensis - n = 2950 - Tiempo secuencial: 83.94 seg.				
p	y	x	Tiempo	Aceleración
2	212	11	45.52	1.84
4	121 - 86 - 48	10	30.57	2.75
8	64 - 63 - 60 - 55 - 47 - 36 - 21	2	20.85	4.02
16	47 - 46 - 45 - 44 - 43 - 41 - 38 - 36 - 33 - 30 - 26 - 22 - 17 -13 - 7	1	17.09	4.91
Acidianus Brierleyi - n = 3046 - Tiempo secuencial: 92.39 seg.				
p	y	x	Tiempo	Aceleración
2	219	11	48.37	1.91
4	124 - 89 - 49	10	31.39	2.94
8	64 - 63 - 61 - 56 - 49 - 37 - 21	2	22.54	4.10
16	47 - 47 - 46 - 45 - 43 - 41 - 39 - 37 - 34 - 30 - 27 - 22 - 18 - 13 - 8	1	20.21	4.57

considerada de recurrencias y derivamos un modelo teórico para calcular el tamaño óptimo del *tile*. Esto nos permite predecir el comportamiento óptimo del algoritmo para una plataforma computacional dada.

Validamos el modelo analítico desarrollado para los algoritmos paralelos, realizando un conjunto amplio de experimentos. Hemos presentado, también, un modelo estadístico, demostrando que es una alternativa válida para resolver el problema, con la ventaja de que podemos utilizarlo en todos los casos, incluso en aquellos donde las hipótesis del modelo analítico no se satisfacen. Los resultados computacionales prueban la eficiencia del esquema y la precisión de nuestras predicciones.

Las aceleraciones, aunque satisfactorias, no escalan para algunas instancias ($p \geq 8$). Este hecho es probablemente debido al uso de tamaño del *tile* estático. Por este motivo introducimos una nueva versión donde incluimos la posibilidad de utilizar un tamaño del *tile* variable, modificando su ancho para cada rectángulo. Comprobamos la utilidad de esta versión y validamos también las predicciones realizadas para obtener los parámetros óptimos.

Hay varias propuestas y cuestiones abiertas. Uno de nuestros objetivos es establecer un método que determine analíticamente el número óptimo de procesadores. Otro de los objetivos planteados, consiste en establecer el modelo estadístico reduciendo considerablemente el tamaño de la muestra realizada. Además, muchos de los razonamientos desarrollados en este trabajo, también pueden ser aplicados a arquitecturas de memoria compartida utilizando la librería *OpenMP*.

El esquema propuesto para los algoritmos RNA puede ser implementado de manera que se adapte, dinámicamente, a la plataforma sobre la que se ejecuta. Después de una primera etapa para evaluar los parámetros, el algoritmo puede analizar la arquitectura, determinar el tamaño óptimo del *tile* y resolver

el problema con esa combinación de parámetros. Nuestra solución al problema del tamaño óptimo del *tile* puede ser fácilmente incorporada en una herramienta que permita resolver el problema sin sobrecarga de cómputo adicional.

También nos planteamos adaptar nuestro programa a plataformas heterogéneas, donde consideraremos la heterogeneidad debida a los procesadores y a la red.

Bibliografía

[1] S. G. Akl. *Parallel Computation Models and Methods*. Prentice Hall, 1997.

[2] E. Alba, A. J. Nebro, and J. M. Troya. Heterogeneous Computing and Parallel Genetic Algorithms. *Journal of Parallel and Distributed Computing*, 62(9):1362–1385, 2002.

[3] M. Aldinucci, S. Ciarpaglini, M. Coppola, M. Danelutto, L. Folchi, S. Orlando, S. Pelagatti, and A. Zavanella. The P3L Project. Página Web. `http://www.di.unipi.it/~susanna/p3l.html/`.

[4] F. Almeida, R. Andonov, D. González, L. M. Moreno, V. Poirriez, and C. Rodríguez. Optimal Tiling for the RNA Base Pairing Problem. In 14^{th} *Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 173–182, 2002.

[5] F. Almeida, R. Andonov, L. M. Moreno, V. Poirriez, M. Pérez, and C. Rodríguez. On the Parallel Prediction of the RNA Secondary Structure. In *Parallel Computing: Sotfware Technology, Algorithms, Architectures and Applications (PARCO)*, pages 525–532, 2003.

[6] F. Almeida, D. González, and L. M. Moreno. The Master-Slave Paradigm on Heterogeneous Systems: A Dynamic Programming Approach for the Optimal Mapping. In 12^{th} *Euromicro Conference on Parallel, Distributed and Network-Based Processing (Euro-PDP'04)*, pages 266–272, February 2004.

[7] F. Almeida, D. González, and L. M. Moreno. The Master-Slave Paradigm on Heterogeneous Systems: A Dynamic Programming Approach for the Optimal Mapping. *Journal of Systems Architecture*, 2005 (To appear).

[8] F. Almeida, D. González, L. M. Moreno, and C. Rodríguez. An Analytical Model for Pipeline Algorithms on Heterogeneous Clusters. In *Recent Advances in Parallel Virtual Machine and Mesaage Passing Interface,* 9^{th} *European PVM/MPI Users' Group Meeting (Lecture Notes in Computer Science (LNCS), vol. 2474)*, pages 441–449. Springer-Verlag, 2002.

[9] F. Almeida, D. González, L. M. Moreno, and C. Rodríguez. Pipelines on Heterogeneous Systems: Models and Tools. *Concurrency and Computation: Practice and Experience*, 2005 (to appear).

[10] F. Almeida, D. González, L. M. Moreno, C. Rodríguez, and J. Toledo. On the Prediction of Master-Slave Algorithms over Heterogeneous Clusters. In 11^{th} *Euromicro Conference on Parallel, Distributed and Network-Based Processing (Euro-PDP'03)*, pages 433–437, February 2003.

[11] R. Andonov, S. Balev, S. Rajopadhye, and N. Yanev. Optimal Semi-Oblique Tiling and its Application to Sequence Comparison. In 13^{th} *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 153–162, July 4-6 2001.

[12] R. Andonov and S. Rajopadhye. Optimal Orthogonal Tiling of 2-D Iterations. *Journal of Parallel and Distributed Computing*, 45:159–165, 1997.

[13] R. Andonov, S. Rajopadhye, and N. Yanev. Optimal Orthogonal Tiling. In *Euro-Par 1998 Parallel Processing, 4^{th} International Euro-Par Conference (Lecture Notes in Computer Science (LNCS), vol. 1470)*, pages 480–490. Springer-Verlag, 1998.

[14] M. Banikazemi, J. Sampathkumar, S. Prabhu, D. K. Panda, and P. Sadayappan. Communication Modeling of Heterogeneous Networks of Workstations for Performance Characterization of Collective Operations. In *International Workshop on Heterogeneous Computing (HCW'99), in conjunction with IPPS'99*, pages 159–165, April 1999.

[15] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 15(4):319–330, 2004.

[16] O. Beaumont, A. Legrand, and Y. Robert. The Master-Slave Paradigm with Heterogeneous Processors. Technical Report RR-4156, Rapport de recherche de l'INRIA- Rhone-Alpes, April 2001.

[17] O. Beaumont, A. Legrand, and Y. Robert. Scheduling Divisible Workloads on Heterogeneous Platforms. *Parallel Computing*, 19:1121–1152, 2003.

[18] O. Beaumont, A. Legrand, and Y. Robert. Scheduling Strategies for Mixed Data and Task Parallelism on Heterogeneous Clusters and Grids. In 11^{th} *Euromicro Conference on Parallel, Distributed and Network-Based Processing (Euro-PDP'03)*, pages 209–216, February 2003.

[19] H. M. Berman, W. K. Olson, D. L. Beveridge, J. Westbrook, A. Gelbin, T. Demeny, S. H. Hsieh, A. R. Srinivasan, and B. Schneider. The Nucleic Acid Database: A Comprehensive Relational Database of Three-Dimensional Structures of Nucleic Acids. *Biophysical Journal*, 63:751–759, 1992.

[20] V. Blanco, F. F. Rivera, D. B. Heras, M. Amor, O. Plata, and E. L. Zapata. Modeling Superlinear Speedup on Distributed Memory Multiprocessors. In *Parallel Computing Conference (ParCo'97)*, pages 689–692, September 1997.

[21] S. Booth. Performance and Code Writing Bottlenecks on the Cray T3D. Technical Report EPCC-TR97-01, EPCC (The University of Edinburgh), 2001. `http://www.epcc.ed.ac.uk/computing/services/cray_service/documents//`.

[22] P. Bourke. 2 Dimensional FFT. Página Web, 1998. `http://astronomy.swin.edu.au/`.

[23] P. G. Bradford. Efficient Parallel Dynamic Programming. In 30^{th} *Annual Allerton Conference on Communication, Control and Computing*, pages 185–194, 1992.

[24] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[25] H. Brunst, H.-C. Hoppe, W. E. Nagel, and M. Winkler. Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach. In *International Conference on Computational Science*, pages 751–760, 2001.

[26] G. Casino. Dossier: Historia de la Biología. *MUY Especial: El Misterio de la Vida*, 49:41–61, 2000.

[27] P. Castro. Hacia una Sociedad Saludable. *MUY Especial: La Nueva Medicina*, 24:18–22, 1996.

[28] E. César, J. G. Mesa, J. Sorribes, and E. Luque. POETRIES: Performance Oriented Environment for Transparent Resource-Management, Implementing End-user Parallel/Distributed Applications. In *Euro-Par 2003 Parallel Processing, 9^{th} International Euro-Par Conference (Lecture Notes in Computer Science (LNCS), vol. 2790)*, pages 141–146. Springer-Verlag, 2003.

[29] E. César, J. G. Mesa, J. Sorribes, and E. Luque. Modeling Master-Worker Applications in POETRIES. In *Proceedings of the* 9^{th} *International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pages 22–30. IEEE Computer Society, April 2004.

[30] E. Cesar, A. Morajko, T. Margalef, J. Sorribes, A. Espinosa, and E. Luque. Dynamic Performance Tuning Supported by Program Specification. *Scientific Programming*, 10:35–44, 2002.

[31] D. Chaiken, C. Fields, and K. K. A. Agarwal. Directory-based Cache Coherence in Large-scale Multiprocessors. *IEEE Computer*, 23(6):49–58, June 1990.

[32] R. D. Chamberlain, D. Chace, and A. Patil. How are we doing?. An Efficiency Measure for Shared, Heterogeneous Systems. In *ISCA* 11^{th} *Int'l Conference on Parallel and Distributed Computing Systems*, pages 15–21, September 1998.

[33] S. Chatterjee and J. Strosnider. Distributed Pipeline Scheduling: End-to-End Analysis of Heterogeneous, Multi-Resource Real-Time Systems. In 15^{th} *International Conference on Distributed Computing Systems (ICDCS'95)*, pages 204–211, 1995.

[34] J. H. Chen, S. Y. Le, B. A. Shapiro, and J. V. Maizel. Optimization of an RNA Folding Algorithm for Parallel Architectures. *Parallel Computing*, 24(11):1617–1634, 1998.

[35] R. Chisleanschi. Para qué Servirá el Mapa del Ser Humano. *MUY Especial: Biotecnología*, 12:42–47, 1993.

[36] A. Choudary, B. Narahary, D. M. Nicol, and R. Simha. Optimal Processor Assignment for a Class of Pipelined Computations . *IEEE Transactions on Parallel and Distributed Systems*, 5(4):439–445, April 1994.

[37] M. Cierniak, M. Zaki, and W. Li. Compile-Time Scheduling Algorithms for Heterogeneous Network of Workstations. *The Computer Journal*, 40(60):236–239, 1997.

[38] A. Clematis and A. Corana. Modelling Performance of Heterogeneous Parallel Computing Systems. *Parallel Computing*, 25:1131–1145, 1999.

[39] A. Clematis, G. Dodero, and V. Gianuzzi. Efficient Use of Parallel Libraries on Heterogeneous Networks of Workstations. *Journal of Systems Architecture*, 46:641–653, 2000.

[40] A. Clematis, G. Dodero, and V. Gianuzzi. A Practical Approach to Efficient Use of Heterogeneous PC Network for Parallel Mathematical Computation. In *European High Performance Computing and Networking (HPCN 2001)*, pages 464–473, June 2001.

[41] M. J. Clement and M. J. Quinn. Multivariate Statistical Techniques for Parallel Performance Prediction. In 28^{th} *Annual Hawaii International Conference on System Scieces (HICSS-28)*, pages 446–455. IEEE Computer Society, January 1995.

[42] Sun Global Glossary. Sun Product Documentation. Página Web. `http://docs.sun.com/app/docs/doc/805-4368/`.

[43] Resource Kit Glossary. Windows 2000 Resource Kits. Microsoft. Página Web. `http://www.microsoft.com/`.

[44] Nebula Glossary. University of Washington. Página Web. `http://www.washington.edu/nebula/glossary.html/`.

[45] S. Coleman and K. S. McKinley. Tile Size Selection using Cache Organization and Data Layout. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 279–290, June 1995.

[46] J. M. Colmenar, O. Garnica, S. López, J. I. Hidalgo, J. Lanchares, and R. Hermida. Empirical Characterization of the Latency of Long Asynchronous Pipelines with Data-Dependent Module Delays. In 12^{th} *Euromicro Conference on Parallel, Distributed and Network-Based Processing (Euro-PDP'04)*, pages 112–119, February 2004.

[47] L. Colombet and L. Desbat. Speedup and Efficiency of Large Size Applications on Heterogenous Networks. *Theoritical Computer Sciencie*, 196(1-2):31–41, 1992.

[48] E. M. Coperías. Dossier: Los Cinco Prodigios. *MUY Especial: Biotecnología*, 12:49–68, 1993.

[49] E. M. Coperías. Dossier: Todas las Terapias de Vanguardia. *MUY Especial: La Nueva Medicina*, 24:37–56, 1996.

[50] E. M. Coperías. Esto es Vivir. *MUY Especial: El Milagro de la Evolución*, 27:22–27, 1996.

[51] E. M. Coperías. Vivir al Límite. *MUY Especial: El Misterio de la Vida*, 49:34–39, 2000.

[52] E. M. Coperías. Ya está Descifrado el Genoma Humano: ¿Y Ahora Qué? *MUY Especial: El Misterio de la Vida*, 49:78–83, 2000.

[53] E. M. Coperías. Las Claves de la Conducta. *MUY Especial: Etología Humana*, 63:49–65, 2003.

[54] A. Corana. Computing Long- and Short-Range Interactions with Ring-based Algorithms on Homogeneous and Heterogeneous Systems. In 11^{th} *Euromicro Conference on Parallel, Distributed and Network-Based Processing (Euro-PDP'03)*, pages 357–364, February 2003.

[55] P. F. Crain, J. Rozenski, and J. A. McCloskey. The Modification Database. Página Web. `http://medstat.med.utah.edu/RNAmods/`.

[56] J. Cuenca, D. Giménez, and J. P. Martínez. Heuristics for Work Distribution of a Homogeneous Parallel Dynamic Programming Scheme on Heterogeneous Systems. In *3rd International Symposium on Parallel and Distributed Computing (3rd International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (ISPDC / HeteroPar'04))*, pages 354–361, 2004.

[57] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A Practical Model of Parallel Computation. *Communications of the ACM (CAMC)*, 39(11):78–85, 1996.

[58] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In 4^{th} *ACM SIGPLAN, Sum. Principles and Practice of Parallel Programming*, pages 1–12, May 1993.

[59] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for Data Parallelism in P^3L. In *Euro-Par 1997 Parallel Processing,* 3^{th} *International Euro-Par Conference (Lecture Notes in Computer Science (LNCS), vol. 1300)*, pages 619–628. Springer-Verlag, 1997.

[60] M. Danelutto and M. Stigliani. SKElib: Parallel Programming with Skeletons in C. In *Euro-Par 2001 Parallel Processing,* 8^{th} *International Euro-Par Conference (Lecture Notes in Computer Science (LNCS), vol 2150)*, pages 1175–1185. Springer-Verlag, 2001.

[61] F. Desprez, J. Dongarra, F. Rastello, and Y. Robert. Determining the Idle Time of a Tiling: New Results. *Journal of Information Science and Engineering*, 14(1):167–190, 1998.

[62] V. Donaldson, F. Berman, and R. Paturi. Program Speedup in a Heterogenous Computing Network. *Journal of Parallel and Distributed Computing*, 21(3):316–322, 1994.

[63] J. Dongarra, A. Geist, W. Jiang, J. Kohl, R. Manchek, P. Papadopoulos, V. Sunderam, and M. Fischer. PVM: Parallel Virtual Machine. Página Web. `http://www.csm.ornl.gov/pvm/pvm_home.html`.

[64] A. J. Dorta, J. A. González, C. Rodríguez, and F. de Sande. llc: A Parallel Skeletal Language. *Parallel Processing Letters*, 13(3):427–448, September 2003.

[65] N. Drakos. High Performance Fortram Language Specification, versión 2.0. Página Web. `http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/hpf-v20/`.

[66] M. Drozdowski and P. Wolnievicz. Experiments with Scheduling Divisible Tasks in Clusters of Workstations. In *Euro-Par 2000 Parallel Processing, 6^{th} International Euro-Par Conference (Lecture Notes in Computer Science (LNCS), vol. 1900)*, pages 311–319. Springer-Verlag, 2000.

[67] P. Edmonds, E. Chu, and A. George. Dynamic Programming on a Shared-Memory Multi-Processor. *Parallel Computing*, 19(1):9–22, 1993.

[68] A. Espinosa, T. Margalef, and E. Luque. Automatic Performance Evaluation of Parallel Programs. In *6^{th} Euromicro Workshop on Parallel and Distributed Processing (Euro-PDP'98)*, pages 43–49. IEEE, 1998.

[69] A. Espinosa, T. Margalef, and E. Luque. Automatic Performance Analysis of Master/Worker PVM Applications with Kpi. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7^{th} European PVM/MPI Users' Group Meeting (Lecture Notes in Computer Science (LNCS), vol. 1908)*, pages 47–55. Springer-Verlag, 2000.

[70] The European Ribosomal RNA Database. Página Web. `http://www.psb.ugen.be/rRNA/`.

[71] T. Fahringer and H. Zima. Static Parameter Based Performance Prediction Tool for Parallel Programs. In *Proceedings of ACM International Conference of Supercomputing*, pages 207–219, 1993.

[72] M. Fekete, I. L. Hofacker, and P. F. Stadler. Prediction of RNA Base Pairing Probabilities using Massively Parallel Computers. *Journal of Computational Biology*, 7(1-2):171–182, 2000.

[73] S. Fortune and J. Wyllie. Parallelism in Random Access Memories. In *10^{th} ACM Symposium on the Theory of Computing*, pages 114–118, May 1978.

[74] M. Frigo and S. G. Johnson. Fast Fourier Transform in the West (FFTW). Página Web. `http://www.fftw.org//`.

[75] M. Frigo and S. G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the IEEE Intl. Conf. Acoustics Speech and Signal Processing (ICASSP'98), vol. 3*, pages 1381–1384. IEEE, 1998.

[76] P. Gachet, B. Joinnault, and P. Quinton. Synthetizing Systolic Arrays using DIASTOL. In *1^{st} International Workshop on Systolic Arrays*, pages 25–36, July 1986.

[77] Z. Galil and K. Park. Parallel Algorithms for Dynamic Programming Recurrences with more than O(1) Dependency. *Journal of Parallel and Distributed Computing*, 21:213–222, 1994.

[78] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide And Tutorial for Network Parallel Computing*. MIT Press, Cambridge, Massachussets, 1994.

[79] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[80] S. Girona, J. Labarta, and R. M. Badía. Validation of Dimemas Comunication Model for MPI Collective Operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7^{th} European PVM/MPI Users' Group Meeting (Lecture Notes in Computer Science (LNCS), vol. 1908)*, pages 39–46. Springer-Verlag, 2000.

[81] M. Goldsmith and G. Jones. *Programming in Occam 2*. Prentice Hall, 1988.

[82] D. González, F. Almeida, L. M. Moreno, and C. Rodríguez. Optimal mapping of pipeline algorithms. In *Euro-Par 2000 Parallel Processing, 6^{th} International Euro-Par Conference (Lecture Notes in Computer Science (LNCS), vol. 1900)*, pages 321–324. Springer-Verlag, 2000.

[83] D. González, F. Almeida, L. M. Moreno, and C. Rodríguez. Pipeline Algorithms on MPI: Optimal Mapping of the Path Planning Problem. In *Recent Advances in Parallel Virtual Machine and Mesaage Passing Interface, 7^{th} European PVM/MPI Users' Group Meeting (Lecture Notes in Computer Science (LNCS), vol. 1908)*, pages 104–112. Springer-Verlag, 2000.

[84] D. González, F. Almeida, L. M. Moreno, and C. Rodríguez. Towards the Automatic Optimal Mapping of Pipelines Algorithms. *Parallel Computing*, 29:241–254, 2003.

[85] D. González, F. Almeida, J. L. Roda, and C. Rodríguez. From the Theory to the Tools: Parallel Dynamic Programming. *Concurrency: Practice and Experience*, 12:21–34, 2000.

[86] J. A. González, M. Printista, G. Rodríguez, C. Rodríguez, and F. de Sande. CALL. Página Web. `http://nereida.deioc.ull.es/~call/`.

[87] J. A. González, C. Rodríguez, G. Rodríguez, F. de Sande, and M. Printista. A Tool for Performance Modeling of Parallel Programs. *Scientific Computing*, 11(3):191–198, 2003.

[88] J. R. González, C. León, and C. Rodríguez. An Asynchronous Branch and Bound Skeleton for Heterogeneous Clusters. In *Recent Advances in Parallel Virtual Machine and Mesaage Passing Interface, 11^{th} European PVM/MPI Users' Group Meeting (Lecture Notes in Computer Science (LNCS), vol. 3241)*, pages 191–198. Springer-Verlag, 2004.

[89] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B.Ñitzberg, W. Saphir, and M. Snir. *MPI - The Complete Reference Volume 2, The MPI Extensions*. MIT Press, Cambridge, Massachussetts, 1998.

[90] L. Guibas, H. Kung, and C. Thomson. Direct VLSI Implementation of Combinatorial Algorithms. *Caltech Conference on VLSI*, pages 509–525, 1979.

[91] J. Gustafson. Fixed Time, Tiered Memory and Superlinear Speedup. In 5^{th} *Conference on Distributed Memory Computers*, pages 1255–1260, 1990.

[92] P. Hansen. *Studies in Computational Science*. Prentice Hall, 1995.

[93] K. Hawick. High Performance Computing and Communications Glossary (version 2.1). Página Web. `http://wotug.ukc.ac.uk/parallel/acronyms/hpccgloss/all.html/`.

[94] M. T. Heath. Performance Visualization with ParaGraph. In *2nd Workshop on Environments and Tools for Parallel Scientific Computing*, pages 221–230, 1994.

[95] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8(5):29–39, 1991.

[96] D. Helmbold and C. McDowell. Speedup (n) Greater than n. *IEEE Transaction on Parallel and Distributed Systems*, 1(2):250–256, 1990.

[97] High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[98] I. L. Hofacker, W. Fontana, P. F. Stadler, L. S. Bonhoeffer, M. Tacker, and P. Schuster. Fast Folding and Comparison of RNA Secondary Structures. *Monatshefte Fr Chemie*, 125(2):167–188, 1994.

[99] I. L. Hofacker, M. A. Huynen, P. F. Stadler, and P. E. Stolorz. Knowledge Discovery in RNA Sequence Families of HIV Using Scalable Computers. In *Proceedings of the 2^{nd} International Conference on Knowledge Discovery and Data Mining*, pages 20–25, 1996.

[100] K. Högstedt, L. Carter, and J. Ferrante. Determining the Idle Time of a Tiling. In *The* 24^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 160–173, January 1997.

[101] K. Högstedt, L. Carter, and J. Ferrante. Selecting Tile Shape for Minimal Execution Time. In 11^{th} *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 201–211, June 1999.

[102] R. M. Hord. *Understanding Parallel Supercomputing.* IEEE Press Understanding Science and Technology Series, 1999.

[103] T. Ibaraki. *Enumerative Approaches to Combinatorial Optimization, Part II.* Annals of Operations Research. Volume 11, 1-4, 1988.

[104] T. Ibaraki and N. Katoh. *Resource Allocation Problems. Algorithmic Approaches.* The MIT Press, 1988.

[105] F. Irigoin and R. Triolet. Supernode Partitioning. In 15^{th} *ACM Symposium on Principles of Programming Languages*, pages 319–328, January 1988.

[106] R. M. Karp, R. E. Miller, and S. Winograd. The Organization of Computations for Uniform Recurrence Equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[107] D. Kearney and N. W. Bermann. Performance Evaluation of Asynchronous Logic Pipelines with Data Depedent Processing Delays. In 2^{nd} *Working Conference on Asynchronous Design Methodologies*, pages 4–13. IEEE Computer Society Press, May 1995.

[108] K. Kempf. Electronic Computers within the Ordenance Corps (chapter III). Historial Monograph from 1961. U. S. Army. Página Web, 1961. `http://ftp.arl.mil/~mike/comphist/61ordnance/index.html/`.

[109] C. T. King, W. H. Chou, and L. M. Ni. Pipelined Data-Parallel Algorithms: Part II–Design. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):486–499, October 1990.

[110] H. Kuchen. A Skeleton Library. Página Web. `http://danae.uni-muenster.de/lehre/kuchen/Skeletons/`.

[111] H. Kuchen. A Skeleton Library. Technical Report 6/02-I, University of Münster, 2002.

[112] H. Kuchen. A Skeleton Library. In *Euro-Par 2002 Parallel Processing,* 8^{th} *International Euro-Par Conference (Lecture Notes in Computer Science (LNCS), vol. 2400)*, pages 620–629. Springer-Verlag, 2002.

[113] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing Design and Analysis of Algorithms.* The benjamin / Cummings Publishing Company, Inc., 1994.

[114] J. Labarta, S. Girona, and T. Cortés. Analyzing Scheduling Policies using Dimemas. *Parallel Computing*, 23(1-2):23–34, 1997.

[115] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. Dip: A Parallel Program Development Environment. In *Euro-Par 1996 Parallel Processing vol. II,* 2^{th} *International Euro-Par Conference (Lecture Notes in Computer Science (LNCS), vol. 1124)*, pages 665–674. Springer-Verlag, 1996.

[116] C. A. Lee, C. DeMatteis, J. Stepanek, and J. Wang. Cluster Performance and the Implications for Distributed, Heterogeneous Grid Performance. In *Heterogeneous Computing Workshop*, pages 253–261, 2000.

[117] M. Lee, W. Liu, and V. K. Prasanna. A Mapping Methodology for Designing Software Task Pipelines for Embedded Signal Processing. In *IPPS/SPDP Workshops*, pages 937–944, 1998.

[118] A. Legrand, H. Renard, Y. Robert, and F. Vivien. Load-balancing Iterative Computations in Heterogenous Clusters with Shared Communication Links. Technical report, Laboratorie de l'Informatique du Parallelisme (cole Normale Suprieure de Lyon), 2003.

[119] P. A. Limbach, P. F. Crain, and J. A. McCloskey. Summary: The Modified Nucleosides of RNA. *Nucleic Acids Research*, 22(12):2183–2196, 1994.

[120] B. Louka and M. Tchuente. Dynamic Programming on Two-Dimensional Systolic Arrays. *Information Processing Letters*, 29(2):97–104, 1988.

[121] S. Lurueña. EDVAC (Electronic Discrete Variable Automatic Calculator). Historia de la Informática. Universidad Politcnica de Madrid. Página Web. `http://www.dma.eui.upm.es/historia_informatica/Flash/principal.htm/`.

[122] J. A. Martínez and I. García. Cut-Through versus Store and Forward en la Implementación de la FFT bidimensional. In *VIII Jornadas de Paralelismo*, pages 291–297, September 1997.

[123] J. S. McCaskill. The Equilibrium Partition Function and Base Pair Binding Probabilities for RNA Secondary Structure. *Biopolymers*, 29(6-7):1105–1119, 1990.

[124] W. F. McColl. Scalable Computing. In *Computer Science Today: Recent Trends and Developments (Lecture Notes in Computer Science (LNCS), vol. 1000)*, pages 46–61. Springer Verlag, 1995.

[125] H. Meuer, E. Strohmaier, J. Dongarra, and H. D. Simon. Lista Top500. Página Web. `http://clusters.top500.org/`.

[126] S. Miguet and Y. Robert. Dynamic Programming on a Ring of Processors. In *Hypercube and Distributed Computers*, pages 19–33, 1989.

[127] Answer.com: The New Standard in Reference. Página Web. `http://www.answers.com/topic/statistical-model/`.

[128] A. Morajko, O. Morajko, J. Jorba, T. Margalef, and E. Luque. Automatic Performance Analysis and Dynamic Tuning of Distributed Applications. *Parallel Processing Letters*, 13(2):169–187, 2003.

[129] D. Morales, F. Almeida, F. García, J. Roda, and C. Rodríguez. A Skeleton for Parallel Dynamic Programming. In *Euro-Par 1999 Parallel Processing, 5^{th} International Euro-Par Conference (Lecture Notes in Computer Science (LNCS), vol 1685)*, pages 877–887. Springer-Verlag, 1999.

[130] D. Morales, J. Roda, F. Almeida, C. Rodríguez, and F. García. Integral Knapsack Problem Parallel Algorithms and their Implementations on Distributed Systems. In *Proceedings of the 1995 International Conference on Supercomputing ACM Press*, pages 218–226, 1995.

[131] L. M. Moreno. *LLP2000: Un Estudio de la Granularidad en Aplicaciones Pipeline*. Proyecto final de carrera, Centro Superior de Informática (Universidad de La Laguna), March 2000.

[132] L. M. Moreno, F. Almeida, D. González, and C. Rodríguez. Adaptive Execution of Pipelines. In *Recent Advances in Parallel Virtual Machine and Mesaage Passing Interface, 8^{th} European PVM/MPI Users' Group Meeting (Lecture Notes in Computer Science (LNCS), vol. 2131)*, pages 217–224. Springer-Verlag, 2001.

[133] L. M. Moreno, F. Almeida, D. González, and C. Rodríguez. The Tuning Problem of Pipelines. In *Euro-Par 2001 Parallel Processing, 8^{th} International Euro-Par Conference (Lecture Notes in Computer Science (LNCS), vol 2150)*, pages 117–121. Springer-Verlag, 2001.

[134] MPI: Message Passing Interface. Página Web. `http://www-unix.mcs.anl.gov/mpi/`.

[135] V. Murthy. The RNA Structure Database. Página Web. `http://www.rnabase.org/`.

[136] Dossier: La Máquina Humana. *MUY Especial: Nuestro Cuerpo*, 1:37–64, 1990.

[137] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer 63*, 12(1):69–80, 1996.

[138] NCBI. GenBank (National Center for Biotechnology Information). Página Web. `http://http://www.ncbi.nlm.nih.gov/Genbank/index.html/`.

[139] The Nucleic Acid Database (NDB). Página Web. `http://ndbserver.rutgers.edu/`.

[140] R. Nuñez. ¿Qué es la Vida? *MUY Especial: El Misterio de la Vida*, 49:16–20, 2000.

[141] M. Oberhuber. Institute for Computer Graphics and Vision. Glossary. Página Web. `http://www.icg.tu-graz.ac.at/mober/pub/thesis/html/node49.htm/`.

[142] OpenMP. Página Web. `http://www.openmp.org/`.

[143] OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface*. OpenMP Forum, November 2000. `http://www.openmp.org/specs/mp-documents/fspec20.pdf`.

[144] OpenMP Architecture Review Board. *OpenMP C/C++ Application Program Interface*. OpenMP Forum, March 2002. `http://www.openmp.org/specs/mp-documents/cspec20.pdf`.

[145] L. Pastor and J. L. Bosque. An Efficiency and Scalability Model for Heterogeneous Clusters. In *2001 IEEE International Conference on Cluster Computing (CLUSTER 2001)*, pages 427–434, October 2001.

[146] Protein Data Bank (PDB). Página Web. `http://www.rcsb.org/pdb/`.

[147] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2 edition, 1993.

[148] H. Quinn, L. A. Smith, M. Leeser, and W. Meleis. Runtime Assignment of Reconfigurable Hardware Components for Image Processing Pipelines. In 11^{th} *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 173–182, April 2003.

[149] J. Ramanujam and P. Sadayappan. Tiling Multidimensional Iteration Spaces for Non Shared-Memory Machines. In *Supercomputing 91*, pages 111–120, 1991.

[150] S. Ranaweera and D. P. Agrawal. Scheduling of Periodic Time Critical Applications for Pipelined Execution on Heterogeneous Systems. In *International Conference on Parallel Processing (ICPP'01)*, pages 131–140, September 2001.

[151] H. Riley. The Von Neumann Architecture of Computer Systems. Página Web. `http://www.csupomona.edu/~hnriley/www/VonN.html/`.

[152] Database of Non-Canonical Base Pairs. Página Web. `http://prion.bchs.uh.edu/bp_type/`.

[153] J. L. Roda, C. Rodríguez, F. Almeida, and D. González. Prediction of Parallel Algorithms Performance on Bus Based Network Using PVM. In 6^{th} *Euromicro Workshop on Parallel and Distributed Processing (Euro-PDP'98)*, pages 57–63, April 1998.

[154] J. L. Roda, C. Rodríguez, D. G. Morales, and F. Almeida. Predicting the Execution Time of Message Passing Models. *Concurrency: Practice and Experience*, 11(9):461–477, 1999.

[155] C. Rodríguez, D. González, F. Almeida, J. Roda, and F. García. Parallel Algorithms for Polyadic Problems. In *Proceedings of the* 5^{th} *Euromicro Workshop on Parallel and Distributed Processing*, pages 394–400. IEEE Computer Society Press, January 1997.

[156] G. Rodríguez. *CALL: A Complexity Analysis Tool*. Proyecto final de carrera, Escuela Técnica Superior de Ingeniería Informática (Universidad de La Laguna), 2000.

[157] P. Rodríguez, M. Pattichis, and R. Jordan. Parallel Single Instruction Multiple Data (SIMD) FFT: Algorithm and Implementation. Technical Report HPCERC2003-002, University of New Mexico, 2003.

[158] W. Rytter. On Efficient Parallel Computations for some Dynamic Programming Problems. *Theoretical Computer Science*, 59:297–307, 1988.

[159] G. Sánchez-Crepo and V. Manzano. Sobre la definición de Estadística. *Hipótesis Alternativa*, 3(2):6–11, 2002.

[160] J. M. Schopf. Structural Prediction Models for High-Performance Distributed Applications. In *Cluster Computing Conference (CCC'97)*, March 1997. `http://www-unix.mcs.anl.gov/~schopf/Pubs/CCC97/index.html`.

[161] R. Schreiber and J. Dongarra. Automatic Blocking of Nested Loops. Technical Report UT-CS-90-108, University of Tennesse, August 1990.

[162] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology.* PWS Publishing Company, 1997.

[163] B. A. Shapiro, J. C. Wu, D. Bengali, and M. J. Potts. The Massively Parallel Genetic Algorithm for RNA Folding: MIMD Implementation and Population Variation. *Bioinformatics*, 17(2):137–148, 2001.

[164] P. Strenström. A Survey of Cache Coherence Schemes for Multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[165] L. Stryer. *Bioquímica. Cuarta edición, tomo 1.* Editorial Reverté, 1995.

[166] J. Subhlok and G. Vondran. Optimal Mapping of Sequences of Data Parallel Tasks. In 5^{th} *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 134–143, 1995.

[167] J. Subhlok and G. Vondran. Optimal Latency-Throughput Tradeoffs for Data Parallel Pipelines. In 8^{th} *Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 62–71, 1996.

[168] J. Subhlok and G. Vondran. Optimal Use of Mixed Task and Data Parallelism for Pipelined Computations. *Journal of Parallel and Distributed Computing*, 3:297–319, March 2000.

[169] Parallel Patterns Project Glossary. Página Web. `http://www.cise.ufl.edu/research/ParallelPatterns/glossary.htm/`.

[170] K. Taura and A. Chien. A Heuristic Algorithm for Mapping Communicating Task on Heterogeneous Resources. In *Heterogeneous Computing Workshop*, pages 102–115, 2000.

[171] S. Thakkar, M. Dubois, A. T. Laundrie, G. S. Sohi, D. V. James, S. Gjessing, M. Tapar, B. Delagi, M. Carton, and A. Despain. New Directions in Scalable Shared-Memory Multiprocessor Architectures. *IEEE Computer*, 23(6):71–83, June 1990.

[172] K. Thews. Llega la Nueva Medicina Molecular. *MUY Especial: Biotecnología*, 12:34–40, 1993.

[173] R. Thiele, R. Zimmer, and T. Lengauer. Protein Threading by Recursive Dynamic Programming. *Journal of Molecular Biology*, 290(3):757–779, 1999.

[174] W. Thumshirn. La Vida en un Puño. *MUY Especial: Biotecnología*, 12:4–25, 1993.

[175] The tRNA Sequence Database. Página Web.
`http://www.uni-bayreuth.de/departments/biochemie/trna/`.

[176] D. Turgay and Y. Parker. Optimal Scheduling Algorithms for Communication Constrained Parallel Processing. In *Euro-Par 2002 Parallel Processing,* 8^{th} *International Euro-Par Conference (Lecture Notes in Computer Science (LNCS), vol. 2400)*, pages 197–211. Springer-Verlag, August 2000.

[177] K. D. Underwood, R. R. Sass, and W. B. Ligon III. Acceleration of a 2D-FFT on an Adaptable Computing Cluster. In 9^{th} *Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 180–189, April 2001.

[178] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[179] Von Neumann Architecture. Página Web. `http://c2.com/cgi/wiki?VonNeumannArchitecture/`.

[180] Von Neumann Architecture. Página Web.
`http://www.hostgold.com.br/hospedagem-sites/o_que_e/vonNeumann+architecture/+`.

[181] J. VonNewmann. First Draft of a Report on the EDVAC. Página Web, 1945. `http://www.wps.com/projects/EDVAC//`.

[182] D. Wang. A Generic Data Process Pipeline Library. Página Web (Borland Developer Network). `http://bdn.borland.com/article/0,1410,30360,00.html/`.

[183] M. S. Waterman and T. F. Smith. RNA Secondary Structure: A Complete Mathematical Analysis. *Mathematical Bioscience*, 42:257–266, 1978.

[184] J. B. Weissman. Prophet: Automated Scheduling of SPMD Programs in Workstation Networks. *Concurrency: Practice and Experience*, 11(6):301–321, 1999.

[185] B. Wilkinson and M. Allen. *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 1999.

[186] T. L. Williams and R. J. Parson. The Heterogeneous Bulk Synchronous Parallel Model. In *Workshop on Advances in Parallel and Distributed Computational Models*, pages 102–108, 2000.

[187] M. E. Wolf and M. Lam. A Data Locality Optimizing Algorithm. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 30–44, June 1991.

[188] M. Wolfe. Iteration Space Tiling for Memory Hierarchies. *Parallel Processing for Scientific Computing (SIAM)*, pages 357–361, 1987.

[189] P. H. Worley. MPICL: Portable Instrumentation Library. Página Web.
`http://www.csm.ornl.gov/picl/index.html`.

[190] X. Wu. *Performance Evaluation, Prediction and Visualization of Parallel Systems*. Kluwer Academic Publishers, 1999.

[191] Y. Yan, X. Zhang, and Y. Song. An Effective and Practical Performance Prediction Model for Parallel Computing on Non-dedicated Heterogeneuos NOW. *Journal of Parallel and Distributed Computing*, 41(2):63–80, 1997.

[192] X. Zhang and Y. Yan. Modeling and Characterizing Parallel Computing Performance on Heterogeneous Networks of Workstations. In 7^{th} *IEEE Symposium on Parallel and Distributed Processing (SPDP'95)*, pages 25–34, 1995.

[193] M. Zuker. RNA Folding Lectures. Página Web, 1998.
`http://bioinfo.math.rpi.edu/~zukern/BIO-5495/RNAfold.html`.

[194] M. Zuker and D. Sankoff. RNA Secondary Structures and their Prediction. *Bulletin Mathematical Biology*, 46:591–621, 1984.

Apéndice A

Introducción a la biología molecular

En la naturaleza podemos encontrar materia que tiene vida y materia que no la tiene. La investigación de los últimos siglos revela que ambos tipos de materia están compuestas por los mismos átomos y sometidos a las mismas reglas físicas y químicas. Lo que diferencia ambos tipos de materia es el conjunto de reacciones químicas que continuamente tienen lugar en el interior de un ser vivo. Las primeras formas de vida que aparecieron en La Tierra eran muy simples, después de billones de años de evolución, las formas de vida se han diversificado y actualmente existen organismos complejos y organismos simples. Sin embargo, a pesar de las diferencias, todos los organismos poseen una química molecular similar. Los principales elementos que los componen son unas moléculas denominadas proteínas y ácidos nucleicos: ácido desoxirribonucleico (*DNA*) y ácido ribonucleico (*RNA*). Las **proteínas** son las que conforman los seres vivos; mientras que los **ácidos nucleicos** codifican la información necesaria para producir las proteínas y son los responsables de pasar esa información a las siguientes generaciones. El DNA es el depositario de la información genética en el interior de las células. El flujo de información genética en células normales va del DNA al RNA y del RNA a las proteínas.

La ciencia que se encarga del estudio de la base molecular de la vida se denomina **bioquímica**. Algunos de sus logros son el descubrimiento de la estructura doblemente helicoidal del DNA, el mecanismo de acción de muchas moléculas proteicas y la comprensión detallada de las vías metabólicas centrales. La bioquímica influye poderosamente en la medicina: las lesiones moleculares de muchas enfermedades genéticas han sido dilucidadas gracias a esos estudios, permitiendo establecer terapias más adecuadas.

A.1. Las proteínas

La mayoría de las sustancias de nuestro cuerpo son proteínas, necesarias para la regulación estructural y funcional de las células, tejidos y órganos. Una proteína es una molécula compuesta de una o más cadenas de aminoácidos en un orden específico. Los **aminoácidos** son las unidades estructurales básicas que las componen y su orden viene determinado por la secuencia de nucleótidos en la codificación genética de la proteína. Las proteínas se fabrican a partir de una colección de 20 aminoácidos que se unen mediante **enlaces peptídicos** para formar **cadenas polipeptídicas**. Cada uno de los 20 aminoácidos está codificado por una o varias secuencias específicas de 3 nucleótidos y cada unidad de aminoácido en un cadena polipeptídica se denomina **residuo**.

En 1953, Frederick Sanger determinó la secuencia de aminoácidos de la insulina [165]. Fue la primera vez que se demostró que una proteína tiene definida con precisión una secuencia de aminoácidos propia que es codificada por los genes. Esta secuencia es conocida como **estructura primaria** de la proteína.

Sin embargo, las proteínas se pliegan, presentando también estructuras secundarias, terciarias y cuaternarias (sección A.3). Determinar la estructura terciaria de una proteína es una de las principales áreas

de investigación en biología molecular: su forma tridimensional condiciona su función. El hecho de que una proteína pueda estar compuesta de 20 tipos de aminoácidos hace que la estructura tridimensional resultante sea bastante compleja y sin simetría. Actualmente no se conoce ningún método sencillo y preciso con el que obtener esta estructura.

A.2. Funciones de las proteínas

Las proteínas son empleadas por el organismo para la estructuración de tejidos y como material de repuesto de los tejidos que se van gastando en el desarrollo de la vida. También juegan un papel energético, aunque menos importante que el de las grasas o los carbohidratos. Podemos establecer la siguiente clasificación de sus funciones:

- **Función enzimática:** La mayoría de las reacciones metabólicas tienen lugar debido a la presencia de un catalizador específico para cada reacción. Estos biocatalizadores reciben el nombre de **enzimas** y aumentan normalmente las velocidades de reacción al menos un millón de veces.

- **Función hormonal:** Las hormonas son sustancias producidas por una célula y que, una vez secretadas, ejercen su acción sobre otras células dotadas de un receptor adecuado. Algunas hormonas son de naturaleza proteica, como la insulina y el glucagón (que regulan los niveles de glucosa en sangre) o las hormonas segregadas por la hipófisis como la hormona del crecimiento o la calcitonina (que regula el metabolismo del calcio).

- **Reconocimiento de señales químicas:** La superficie celular alberga un alto número de proteínas encargadas del reconocimiento de señales químicas de muy diverso tipo: existen receptores hormonales, de neurotransmisores, de anticuerpos, de virus, de bacterias, etc.

- **Función de transporte:** Muchos iones y moléculas pequeñas son transportados por proteínas. Por ejemplo, la hemoglobina se encarga de transportar el oxígeno que respiramos al resto del organismo a través de la sangre, la transferrina transporta el hierro, etc.

- **Soporte y Estructura:** La tensión de la piel y los huesos se debe a la presencia de una proteína llamada colágeno. Por otro lado, la elastina es una proteína muy elástica que permite a tejidos como arterias y pulmones la posibilidad de estirarse sin causar daño.

- **Protección inmune:** El sistema inmune, responsable de defendernos contra organismos foráneos requiere de la precisa interacción de cientos de proteínas, como los anticuerpos.

- **Función de movimiento:** Todas las funciones de movilidad de los seres vivos están relacionadas con las proteínas; por ejemplo, la contracción del músculo resulta de la interacción entre dos proteínas: la actina y la miosina.

- **Función de reserva:** Mantener una reserva de aminoácidos para el futuro desarrollo del embrión: la ovoalbúmina de la clara de huevo, la lactoalbúmina de la leche, la gliadina del grano de trigo o la hordeína de la cebada.

- **Transducción de señales:** Los fenómenos de transducción (cambio en la naturaleza físico-química de señales) están mediados por proteínas; por ejemplo, durante el proceso de la visión, la rodopsina de la retina convierte un fotón luminoso (una señal física) en un impulso nervioso (una señal eléctrica) y un receptor hormonal convierte una señal química (una hormona) en una serie de modificaciones en el estado funcional de la célula.

A.3. Estructura de las proteínas

La secuencia lineal de aminoácidos de una proteína puede plegarse sobre sí misma, adoptando múltiples estructuras en el espacio (figura A.1):

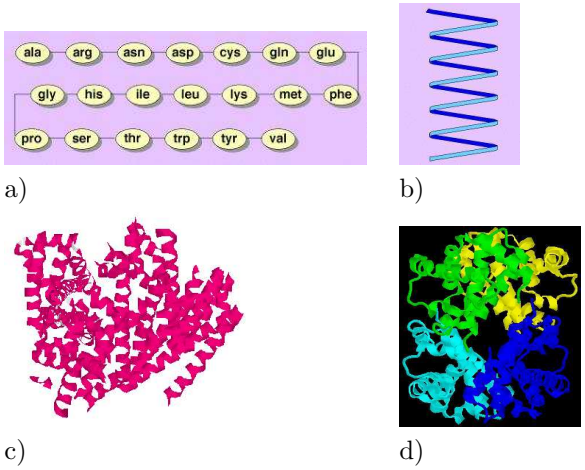

Figura A.1: Estructuras de las proteínas: a) Estructura primaria: secuencia de aminoácidos. b) Estructura secundaria: unión de aminoácidos próximos entre sí. c) Estructura terciaria: unión de aminoácidos alejados entre sí. d) Estructura cuaternaria: unión de proteínas.

- La **estructura primaria** viene determinada por la secuencia de aminoácidos en la cadena proteica, es decir, el número de aminoácidos presentes y el orden en que se encuentran enlazados (figura A.1-a).
- La **estructura secundaria** es el plegamiento que la cadena adopta debido al establecimiento de puentes de hidrógeno entre los átomos (cadena polipeptídica). La encontramos en estructuras regulares y locales, como las hélices que aparecen en el pliegue de la proteína. De esta forma se minimiza la energía liberada y, por lo tanto, se consiguen estructuras más estables. Se puede definir la estructura secundaria como el ordenamiento espacial de los residuos de aminoácidos próximos entre sí (figura A.1-b).
- La **estructura terciaria** es la disposición tridimensional de todos los átomos que componen la proteína y se obtiene como consecuencia del empaquetado de la estructura secundaria en un nivel superior. Es responsable directa de sus propiedades biológicas. Se define como el ordenamiento espacial de los residuos de aminoácidos alejados entre sí (figura A.1-c).
- La **estructura cuaternaria** es la asociación de varias cadenas polipeptídicas: un grupo de proteínas diferentes que se agrupan conjuntamente (figura A.1-d).
- Por último, la asociación de proteínas con otros tipos de biomoléculas para formar asociaciones supramoleculares con carácter permanente dan lugar a la **estructura quinaria**.

Una vez conseguida la proteína pura, es posible averiguar su secuencia de aminoácidos. Estas secuencias suministran mucha información:

- Pueden compararse con otras secuencias conocidas para verificar si existen semejanzas significativas.
- Variando la secuencia de estructuras proteicas conocidas se pueden generar proteínas con nuevas propiedades.

- Comparando las secuencias de la misma proteína en distintas especies de seres vivos se obtiene información sobre sus historias evolutivas.
- Pueden encontrarse repeticiones internas.
- Contienen señales que determinan el destino de la proteína y controlan su maduración.
- Suministran información para preparar anticuerpos específicos para una proteína determinada.
- El análisis de las relaciones entre la secuencia y la estructura tridimensional permite descubrir las leyes que rigen el plegamiento: la secuencia es la conexión entre el mensaje del DNA (los genes) y la configuración tridimensional (que perfila su función biológica).

A.4. El problema del plegado

El problema del plegado es uno de los problemas más importantes de la biología molecular. En 1961, Anfinsen demostró que la ribonucleasa podía desnaturalizarse y replegarse sin perder su actividad enzimática. Este experimento sugirió que toda la información que necesita una proteína para adoptar su configuración se encuentra codificada en la estructura primaria. Siendo así, debería ser posible derivar las reglas para el plegado de las proteínas del análisis en estructuras conocidas y luego aplicar esas reglas disponiendo únicamente de la secuencia lineal de aminoácidos. El problema del plegamiento de las proteínas se basa en 2 preguntas: ¿de qué manera la secuencia de aminoácidos determina su estructura tridimensional? y ¿cómo se forma una proteína nativa a partir de una cadena polipeptídica desnaturalizada?. El plegamiento de las proteínas se produce mediante una progresiva estabilización de intermediarios, en un proceso asistido por enzimas. Las proteínas son muy poco estables: la mayoría de las proteínas pequeñas se desaturalizan de forma reversible al aumentar la temperatura o al añadir agentes desnaturalizadores [165].

Se observa que secuencias de aminoácidos muy diferentes pueden originar plegamientos proteicos con un parecido sorprendente y, a la inversa, la estructura terciaria de dos proteínas puede variar aunque sus secuencias sean prácticamente idénticas. Esto provoca que la predicción sea complicada porque el número de formas posibles de una cadena polipeptídica es enorme y las proteínas son poco estables; esto significa que la diferencia de energía entre los estados nativo y desnaturalizado es reducida.

A.5. El déficit secuencia/estructura

La última década ha abierto las puertas a la investigación y al análisis comparativo de genomas completos (sección A.9). El análisis del genoma consiste en la búsqueda por esclarecer los genes y los productos genéticos de los organismos y se sostiene en un conjunto de conceptos asociados principalmente al proceso de la evolución, al mecanismo del plegado de las proteínas y, fundamentalmente, a las funciones de las proteínas. El uso de los ordenadores para modelizar estos procesos está limitado por los conocimientos alcanzados en estos campos. Por ejemplo, todavía no se conocen exactamente las reglas que rigen el plegado de las proteínas; por este motivo, aún no es posible diagnosticar la función de una proteína con el conocimiento aislado de su secuencia.

Existe un gran contraste entre la facilidad con la que es posible determinar la secuencia o estructura primaria de estas moléculas, y la dificultad para determinar su estructura tridimensional, utilizando técnicas costosas y no siempre factibles, debido a la sofisticación de las estructuras tridimensionales. Como resultado, existe una amplia diferencia entre la cantidad de moléculas de las que se conocen su secuencia de nucleótidos y aquellas de las que se conoce su estructura tridimensional. Por ejemplo, en 1998, los métodos para predecir la estructura secundaria tenían una fiabilidad del orden el 50-60 %, lo que producía un enorme déficit de información (figura A.2).

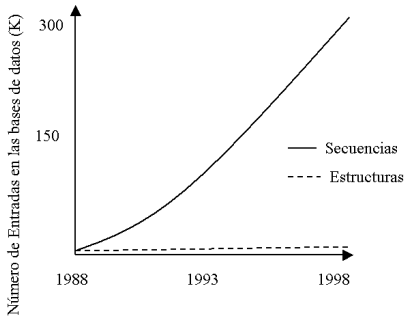

Figura A.2: El déficit secuencia/estructura hasta 1998.

Los datos biológicos (proteínas y ácidos nucleicos) proliferan rápidamente y es necesarios disponer de bases de datos que almacenen toda la información que se genera y que permitan acceder a esos datos con facilidad. Existen multitud de bases de datos, tanto públicas como privadas: el *Banco de Datos de Proteinas* (*Protein Data Bank*, textitPDB) [146], *The Nucleic Acid Database* (*NDB*) [19, 139], *GenBank* [138], la *RNA Structura Database* (*RNABase*) [135], la *Database of Non-canonical Base Pairs* [152], la *RNA Modification Database* [55, 119], la *tRNA Sequence Database* [175], *The European Ribosomal RNA Database* [70]. Desde la llegada de las conexiones rápidas e Internet es posible acceder a esos datos así como a muchos programas, de forma rápida, fácil y barata desde cualquier lugar del mundo. Como consecuencia, las herramientas basadas en computadores juegan un creciente papel crítico en los avances de la investigación biológica. A comienzos de 1998, en las bases de datos no redundantes públicamente disponibles, se habían depositado mas de 300000 secuencias de proteínas y el número de secuencias parciales en las bases de datos públicas y privadas se estimaba en un orden de millones. Por contra, las estructuras tridimensionales son más complejas de obtener, derivar, almacenar y manipular. Esta situación ha empeorado cuando los distintos proyectos genoma que se estaban desarrollando han finalizado. Por supuesto, la actual adquisición de datos estructurales también se está acelerando, en la figura A.3 se muestra el enorme incremento de estructuras que se ha producido en los últimos años en el *Banco de Datos de Proteínas* [146]. Sin embargo, estas cifras son claramente insuficientes en comparación con el número de secuencias que se descifran al año.

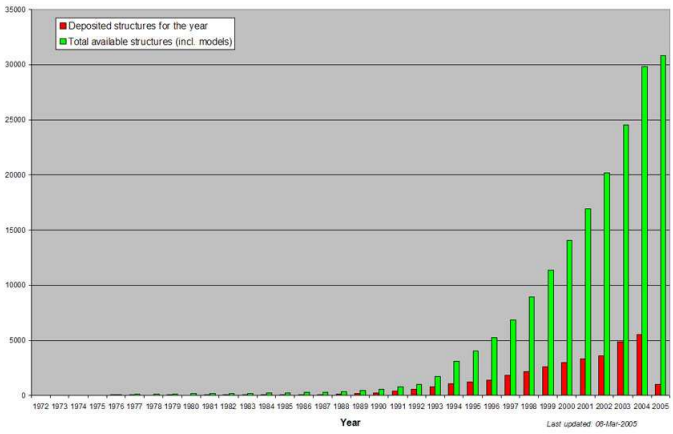

Figura A.3: Incremento de la adquisición de datos estructurales en el Banco de Datos de Proteínas en los últimos años.

A.6. Ácidos nucleicos

La identificación de la desoxirribona en 1929 por Phoebus A. T. Levene estableció dos tipos de ácidos nucleicos: el ribonucleico (RNA) y el desoxirribonucleico (DNA). En 1953, Crick y Watson descubrieron la estructura del DNA, la macromolécula que forma los cromosomas y los genes. Dos años más tarde, el bioquímico español Severo Ochoa logró sintetizar el RNA, la molécula que hace posible la transformación del DNA en proteínas [26]. En la *XI Conferencia Internacional sobre el Origen de la Vida* (1996) los 300 científicos reunidos emitieron una conclusión: sin las moléculas de DNA y RNA la vida no habría emergido en nuestro planeta [50]. Todas las criaturas terrestres portan su información genética en forma de DNA, salvo excepciones que se sirven del RNA.

El DNA es una molécula que se encuentra en el interior del núcleo de las células en unas moléculas llamadas **cromosomas**. En la figura A.4 se observa la imagen de una célula de DNA. Se distinguen algunas de sus partes más importantes: el núcleo contiene los cromosomas, las mitocondrias son responsables de la respiración celular, la función de los ribosomas consiste en sintetizar proteínas y el aparato de Golgi prepara y empaqueta los materiales moleculares para su transporte.

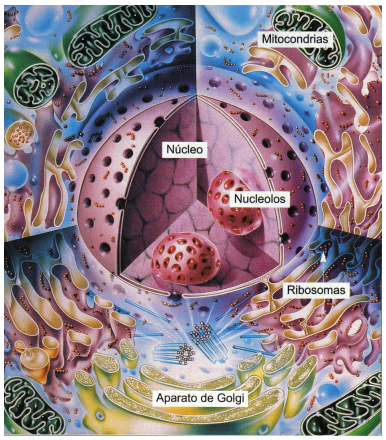

Figura A.4: Imagen de una célula de DNA. Se distinguen algunos de sus orgánulos más importantes: el núcleo contiene los cromosomas, las mitocondrias son responsables de la respiración celular, la función de los ribosomas consiste en sintetizar proteínas y el aparato de Golgi prepara y empaqueta los materiales moleculares para su transporte.

Esta molécula está formada por dos bandas paralelas unidas por peldaños, formando una doble hélice parecida a una escalera de caracol (figura A.5-(1)), similar a una escala de cuerda enrollada en espiral de sólo 2 millonésimas de milímetro de ancho [136, 165, 174]. Los peldaños de esta escalera de caracol lo constituyen los genes, donde están registradas y grabadas las informaciones hereditarias de cada una de las especies vivas del planeta. Además tiene la facultad de desdoblarse, dando lugar a otra molécula totalmente idéntica.

El DNA constituye el material genético de todas las células y de muchos virus (algunos virus utilizan RNA). La cantidad de información almacenada en una molécula de DNA es distinta para diferentes especies. Por ejemplo, las bacterias son organismos muy simples que ni siquiera tienen núcleo, por lo que no necesitan mucha información genética: su molécula de DNA sólo tiene un milímetro de largo. Por el contrario, el contenido de una célula humana, si pudiera desenrollarse y extenderse sobre una mesa, mediría cerca de tres metros (la figura A.6 muestra un fragmento de DNA desenrollado a través del microscopio electrónico).

Del mismo modo que las proteínas, el DNA es una cadena formada por moléculas más simples: cada tramo de la escalera helicoidal (figura A.5-(2)) está constituido por cuatro únicas submoléculas

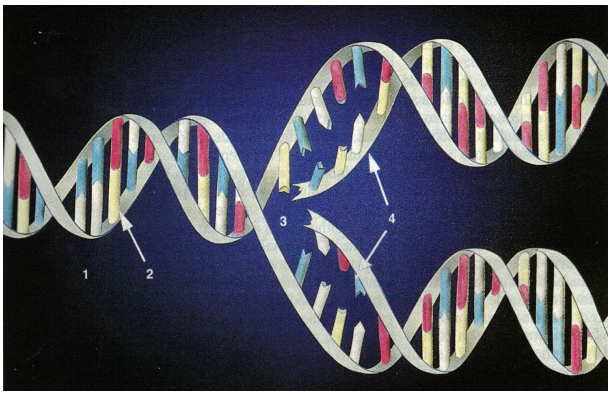

Figura A.5: Vista de una molécula de DNA: (1) La molécula de DNA forma una doble hélice. (2) La molécula de DNA está constituida por nucleótidos. (3) En el proceso de replicación las dos cadenas se separan. (4) Se consigue obtener una copia idéntica a la doble hélice original.

Figura A.6: Fragmento de DNA desenrollado visto a través del microscopio electrónico. El contenido de una célula humana (desenrollado y extendido) mide cerca de tres metros de longitud.

(**nucleótidos**): Adenina (A), Guanina (G), Citosina (C) y Timina (T). Los nucleótidos se unen formando parejas (A con T y C con G). Tres nucleótidos consecutivos forman una palabra (tiplete): las instrucciones para que la célula fabrique un aminoácido. Las combinaciones de muchos tripletes seguidos forman las frases (los genes). En una célula humana, las moléculas de DNA tienen cientos de millones de nucleótidos. En la figura A.7 se muestra la unión de los nucleótidos en una molécula de DNA para formar la doble hélice.

Las bases A y T se dice que son **bases complementarias**, como también lo son C y G. Las parejas de bases constituyen la unidad de longitud más utilizada cuando se habla de moléculas de DNA o RNA y se denotan por *bp*: se dice que una molécula de DNA tiene una longitud de 100000 bp o 100 kbp. Debido a las características de esta molécula (su forma de doble hélice), es posible inferir la secuencia de una cadena dada la otra. La operación que permite realizarlo se conoce como **complementación inversa**. Es precisamente este mecanismo lo que permite al DNA de una célula replicarse, un organismo que comienza su vida como una célula crece hasta millones de células, cada una llevando copias de las moléculas de DNA de la célula original.

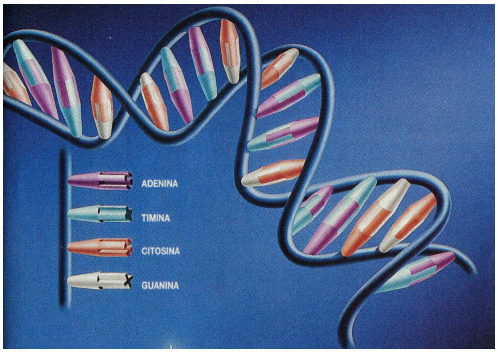

Figura A.7: Estructura de una molécula de DNA: está formada por cuatro moléculas más simples (nucleótidos): Adenina (A), Guanina (G), Citosina (C) y Timina (T).

En el proceso de **replicación del DNA** las dos cadenas se separan (figura A.5-(3)). Este proceso se denomina **fusión** porque tiene lugar bruscamente a una temperatura determinada. Cada cadena actúa entonces como una plantilla para la formación, a partir de ella misma, de una nueva cadena. De esta forma se consigue obtener una copia idéntica de la doble hélice de DNA original (figura A.5-(4)).

La tecnología actual del DNA permite generar mutaciones específicas para construir genes nuevos con propiedades diseñadas a voluntad. También se pueden crear nuevas proteínas juntando fragmentos de genes que codifican dominios proteicos no asociados en la naturaleza.

Las moléculas de RNA son similares a las moléculas de DNA con las siguientes diferencias en composición y estructura:

- En el RNA encontramos Uracilo en vez de Timina. El Uracilo se empareja con Adenina.

- Las moléculas de RNA también pueden ser secuenciadas y se pliegan sobre sí mismas para conformar estructuras secundarias y estructuras terciarias, esenciales en el desarrollo de sus funciones biológicas. Sin embargo, sus estructuras secundarias y terciarias son más variadas que las del DNA. Las moléculas de RNA no forman una doble hélice, poseen una única cadena y no tienen necesidad de mantener determinadas proporciones de bases complementarias. Esto hace que la predicción de la estructura secundaria del RNA sea un problema más complejo e interesante de tratar puesto que el plegamiento de esta molécula, determina su función biológica e influye en la síntesis de las proteínas (sección A.7). En la figura A.8 se muestra un ejemplo de la estructura secundaria de una molécula de RNA.

- El tamaño de las moléculas de RNA es mucho menor que las del DNA.

Existen diferentes tipos de RNA en las células según las funciones que realizan. Las moléculas que forman parte de las subunidades de los ribosomas se les denomina **RNA ribosomal (rRNA)** (figura A.9). Las moléculas encargadas de transportar los aminoácidos activados hasta el lugar de síntesis de proteínas en los ribosomas se les conoce por **RNA de transferencia (tRNA)** (figura A.10) y los RNA que son portadores de la información genética y la transportan a los ribosomas son llamados **RNA mensajero (mRNA)**. Existe una molécula de mRNA para cada gen o grupo de genes que vaya a expresarse. La secuencia de bases de un mRNA es complementaria a la de su DNA molde.

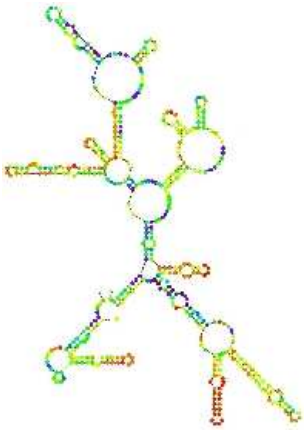

Figura A.8: Estructura secundaria de una molécula de RNA: es más variada que la del DNA.

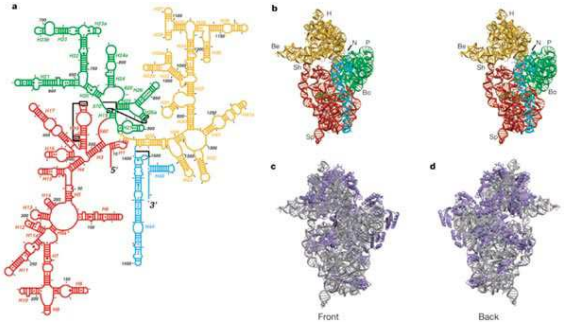

Figura A.9: Estructura secundaria y terciaria del RNA ribosomal (rRNA).

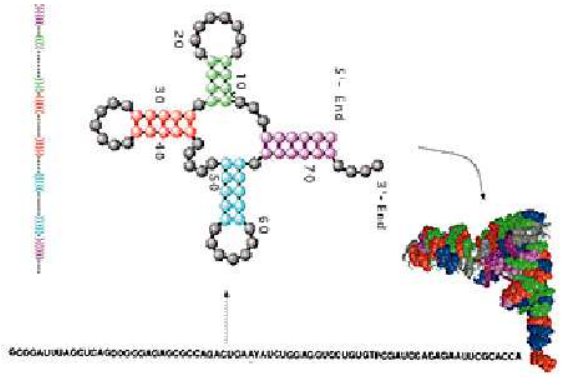

Figura A.10: Estructura primaria, secundaria y terciaria del RNA de transferencia (tRNA).

A.7. Síntesis de proteínas

El DNA contiene secciones en las que codifica información para construir proteínas y a cada proteína le corresponde una única sección en el DNA. Cada una de estas secciones forman un **gen**. Algunos genes pueden originar moléculas de RNA, por lo que se define un gen como una sección contigua de DNA que

contiene la información necesaria para construir una proteína o una molécula de RNA. Los genes varían en longitud pero en el caso de los seres humanos pueden tener un tamaño del orden de 10000 bp. A pesar de que todos los genes están presentes en todas las células, sólo una parte de ellos se utilizan en cada célula específica.

Puesto que una proteína es una cadena de aminoácidos, para especificar una proteína es necesario determinar cada uno de los aminoácidos que contiene; esta es la función que realiza un gen en una molécula de DNA. Para especificar cada aminoácido utiliza tripletas de aminoácidos (conocidos como **codones**). La tabla que determina la correspondencia entre tripleta y aminoácido se conoce como **código genético**.

La fabricación (síntesis o traducción) de una proteína está dirigida por la doble hélice de DNA, según un código genético que es común para todos los organismos y tiene lugar en los ribosomas del citoplasma celular. La síntesis se inicia con el desdoblamiento de la cadena de DNA a la altura del gen responsable de una proteína. Una de las hebras sirve de molde para que el enzima RNA polimerasa transcriba la información almacenada en forma de secuencias de bases (las letras del alfabeto genético) a una cadena de RNA mensajero (mRNA). De esta forma, el mRNA tiene la misma secuencia que una de las hebras del gen pero sustituyendo el Uracilo por Timina. Este proceso se denomina **transcripción**. Por otro lado, el RNA transferente (tRNA) hará la conexión entre el codón y el aminoácido correspondiente. Los aminoácidos son ensamblados uno a uno en un orden estricto (por complementariedad de bases) hasta completar la proteína. En la figura A.11 se muestra de forma esquemática este proceso.

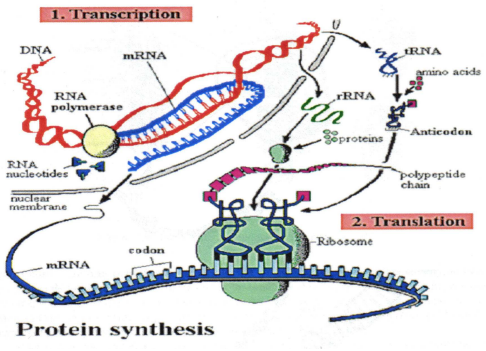

Figura A.11: Proceso de la síntesis de proteínas.

Una vez finalizada la síntesis de una proteína el RNA mensajero queda libre y puede ser leído de nuevo. Es muy frecuente que antes de que finalice una proteína ya está comenzando otra, por lo que una molécula de RNA mensajero, puede estar siendo utilizada por varios ribosomas simultáneamente.

Como se observa en la descripción anterior la clave de la traducción reside en el código genético, compuesto por combinaciones de tres nucleótidos consecutivos (o tripletes) en el mRNA. Los distintos tripletes se relacionan específicamente con tipos de aminoácidos usados en la síntesis de las proteínas. Cada triplete constituye un codón: existen en total 64 codones, 61 de los cuales sirven para cifrar aminoácidos y 3 para marcar el cese de la traducción. Dado que existen más codones que aminoácidos (20), casi todos pueden ser reconocidos por más de un codón. El número de codones en el mRNA determina la longitud de la proteína, existiendo siempre un codón de inicialización y uno de terminación.

A.8. Virus y bacterias

Los virus son parásitos a nivel molecular. No pueden ser considerados una forma de vida, puesto que solamente son capaces de reproducirse cuando infectan a las células adecuadas (*hosts*). No exhiben ningún tipo de metabolismo, no se producen reacciones bioquímicas en su interior; en su lugar, utilizan

el metabolismo del *hosts* para replicarse. La mayoría de los virus consisten en una cápsula de proteína que contiene material genético (DNA o RNA) en su interior. El DNA viral es mucho más pequeño que el DNA de los cromosomas y por tanto más fácil de manipular. En 1977 Frederick Sanger desveló la primera secuencia completa de uno de ellos, un bacteriófago (virus que infectan bacterias) *PhiX174*. Su cadena de DNA tenía 5386 bp, que definían 11 genes. Otro virus más familiar, el causante del resfriado común, tiene unos 7500 bp y el de la viruela unos 186000 [26]. Algunos virus presentan incluso una sola hebra durante parte de su ciclo vital. En la figura A.12 se muestra de forma esquemática el aspecto de un virus.

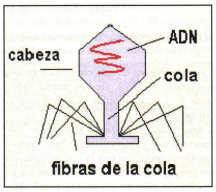

Figura A.12: Aspecto de un virus.

Cuando un virus infecta a una célula el material genético se introduce en el citoplasma. Este material genético es interpretado por el mecanismo celular como su propio DNA, por esta razón las células comienzan a producir proteínas con la codificación del virus como si fueran de la propia célula. Estas proteínas propagan la replicación DNA vírica y generan nuevas cápsulas, de modo que se ensamblan un elevado número de nuevas partículas virales en el interior de una célula infectada. Otras proteínas víricas rompen la membrana de la célula y liberan todas las partículas víricas en el entorno, donde pueden atacar a otras células. La figura A.13 muestra el método utilizado por los virus para penetrar en el interior de la célula.

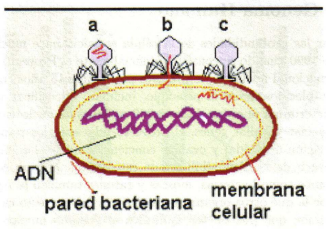

Figura A.13: Proceso de infección viral sobre una célula.

Ciertos virus no atacan al host inicialmente. El DNA viral se inserta en el genoma del host y puede permanecer ahí durante largos períodos de tiempo sin ningún cambio visible en la vida de la célula. Bajo ciertas condiciones, el virus durmiente puede activarse y en ese momento es cuando se libera del genoma del *host* y comienza su actividad de replicación. Los virus son altamente específicos siendo capaces infectar sólo a un tipo de células; por ejemplo, el virus *T2* infecta únicamente a células del tipo *E. coli*. El *HIV*, el virus de inmunodeficiencia humana, está formado por RNA que se replica mediante un DNA intermediario. Este virus infecta sólo a células humanas implicadas en defender al organismo contra los intrusos en general. Las moléculas donde la información genética fluye del RNA al DNA se conocen como **retrovirus**.

Una bacteria es un organismo unicelular que tiene sólo un cromosoma. El genoma de las bacterias es mayor que el de los virus; por ejemplo, el de la bacteria *Escherichia coli* (*E. coli*) tiene 4.64 Mbp (millones de pares de bases). En la figura A.14 se observa el aspecto que presenta una bacteria.

Las bacterias pueden multiplicarse mediante replicación del DNA en períodos muy cortos de tiempo, lo que las hace muy útiles en investigación genética. Al ser invadidas por bacterias, las células de algunos

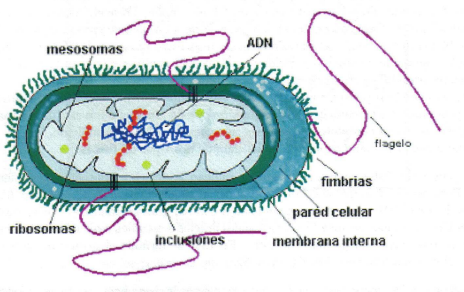

Figura A.14: Aspecto de una bacteria.

organismos inferiores elaboran sustancias llamadas antibióticos para defenderse de la infección. En muchos casos los antibióticos logran sus objetivos interfiriendo la síntesis proteica en los ribosomas de las bacterias, lo que las destruye. Sin embargo, las bacterias también son necesarias; por ejemplo, diferentes especies de bacterias completan el trabajo realizado por la levadura al transformar la leche en requesón.

A pesar de los conocimientos actuales, los límites de la vida aún están por definir: los científicos descubren constantemente nuevas bacterias capaces de prosperar en agua hirviendo, ácido sulfúrico o bajo toneladas de presión. En marzo de 2000, científicos del *Woods Hole Oceanographic Institution*, en Massachusetts y de la Universidad de Winconsin-Madison, presentaron a la comunidad científica uno de los seres vivos más insólitos de nuestro planeta, un microbio recién descubierto que come hierro y que sólo puede desarrollarse en las aguas más ácidas y tóxicas de la naturaleza (aguas con altas concentraciones ácido sulfúrico, arsénico, cobre, cadmio y zinc). Esta bacteria ha sido bautizada con el nombre científico de *Ferroplasma acidarmamus* y fue localizada en las profundidades de una mina de cobre abandonada en Iron Mountain, cerca de Redding, California [51]

A.9. El proyecto genoma humano

Desde 1865 cuando Gregor Mendel estableció las leyes de la herencia hasta el proyecto genoma se ha producido un avance espectacular en el campo de la genética y la biología molecular. En 1902 Ernest Starling y William Bayliss realizaron el descubrimiento de las hormonas. En 1944 Oswald Avery descubrió la molécula del DNA, la molécula responsable de transmitir la herencia a las siguientes generaciones; y en 1953 el Francis Crick y James Watson descubrieron la estructura de doble hélice de la molécula de DNA.

Desde su aparición en 1974, la ingeniería genética se ha impuesto como una herramienta imprescindible no sólo en el campo de la medicina, sino en todas las áreas de la investigación biológica y en ciertos sectores de la industria alimentaria. Mediante las más revolucionarias técnicas de ingeniería genética (llamadas técnicas del DNA recombinante) los investigadores son capaces de reprogramar bacterias, levaduras y células de mamíferos, insectos y vegetales, para que fabriquen a gran escala proteínas escasas o difíciles de extraer del organismo humano, con una pureza prácticamente absoluta [48]. Otras aplicaciones son: la depuración de aguas residuales, la limpieza de los mares afectados por la marea negra y en el sector agrícola, para hacer plantas más resistentes a las sequías.

A mediados de los años setenta los bioingenieros ya tenían la posibilidad de construir microorganismos con características predefinidas. Hoy en día las bacterias producen numerosas proteínas que nunca hubiesen generado de una manera natural. Una *E. coli* es capaz de destilar hasta el 30 % de su volumen proteínico total en forma de proteínas recombinadas: una vez introducido el gen extraño, el sistema de biosíntesis de la *E. coli* trabaja exclusivamente a sus órdenes [174]. Como ejemplos de sustancias producidas por ingeniería genética podríamos citar el interferón, la insulina y la hormona del crecimiento. Menos conocidas son una serie de sustancias que han llegado a adquirir una considerable importancia en

la ciencia médica (anticuerpos monoclonales de distintos tipos para fines diagnósticos).

En octubre de 1990 se constituyó oficialmente el **Proyecto Genoma Humano** (**PGH**), un proyecto internacional inicialmente previsto para durar 15 años (hasta el 2005). Sus objetivos, claramente establecidos desde el principio, incluían: identificar y ubicar todos los genes del ser humano y localizar y averiguar la función de los 3000 millones de pares de bases que forman la cadena de DNA humano. Además, se pretendía almacenar toda la información obtenida y evaluar las implicaciones éticas, legales y sociales que surgen de la disponibilidad de estos datos. Se consideraba que los resultados obtenidos en este proyecto permitirían un amplio campo de alternativas para la medicina, tanto en el ámbito de la prevención como en el del tratamiento. Sin la ayuda de la informática (600 supercomputadores) esta tarea hubiera requerido varias generaciones. El coste de la secuenciación debería reducirse como consecuencia de la nueva y mejorada tecnología.

Como parte del proyecto, genomas de otros organismos como bacterias, moscas y ratones también se han estudiado. Una de las razones por la que otros organismos son parte del proyecto era para perfeccionar los métodos de secuenciación que pueden ser aplicados al genoma humano. Todas las especies consideradas son ampliamente utilizadas en la investigación molecular y genética. Hasta ahora, muchos genomas de virus han sido completamente secuenciados, pero sus tamaños están en el rango de 1kbp y 10 kbp. El primer organismo vivo que fue totalmente secuenciado fue la bacteria *Haemophilus influenzea*, que contenía un genoma de 1800 kbp. En 1996 se obtuvo la secuencia completa del genoma de la levadura (una secuencia de 10 millones de bp).

Un problema adicional es el del genoma medio. Diferentes individuos tienen genomas distintos (esto permite que el DNA sea un indentificador personal). Se estimaba que los genomas de dos personas distintas difieren en una de cada 500 bases. Por lo tanto, era necesario decidir de quién se iba a secuenciar el genoma. Incluso si un individuo se elige de algún modo que su DNA sea el estándar, está el problema de los elementos que se mueven de una parte a otra del genoma, por lo que estaríamos viendo una instantánea del genoma en un momento dado. Se habla de *el* genoma y no de *los* genomas porque, a pesar de que todos los individuos sean distintos entre sí, existe una gran similitud: todos tenemos igual cantidad de cromosomas, idénticos genes situados en las mismas ubicaciones de los respectivos cromosomas y las mismas largas distancias de DNA sin función conocida. Además, en cada célula de nuestro organismo es posible encontrar 23 pares de cromosomas, los 30000 o más genes y unos 3000 millones de pares de bases, la cifra total que se suponía que componía el DNA del ser humano. Cualquier célula con núcleo contiene lo mismo, provenga ésta de sangre, semen, un cabello con raíz o piel.

Entre los beneficios que proporcionaría el genoma humano podemos mencionar:

- El reconocimiento de las proteínas que son producidas por los genes.
- Predecir la aparición de enfermedades. Se considera que gran parte de las enfermedades tiene un componente genético, se podría identificar la predisposición a ciertos males y prevenirlos.
- Intervenir mejor ante la aparición de las enfermedades. Al conocer con más exactitud el funcionamiento de los genes, se podrán desarrollar terapias que los reemplacen o los frenen cuando no funcionen correctamente.
- Aportar tratamiento más personalizado. Se desarrollarán *Chips Genéticos*, una tecnología que posibilitará encontrar el medicamento según el perfil que presente cada paciente.

El Proyecto Genoma Humano es, en definitiva, un trabajo de cartografía y supone la realización de dos tipos de mapas distintos pero complementarios [35]:

- **Mapas físicos:** Consiste en la secuenciación completa de los nucleótidos en la molécula de DNA. Es necesaria la ayuda de ordenadores (figura A.15).
- **Mapas genéticos:** Indican la ubicación precisa de los genes en el interior de los cromosomas. Para hacerlo se estudia la transmisión de caracteres hereditarios entre generaciones. (figura A.16).

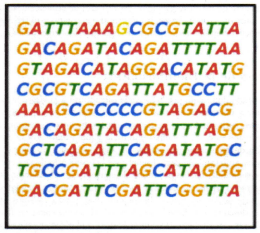

Figura A.15: Mapa físico del genoma humano: secuenciación completa de los nucleótidos en la molécula de DNA.

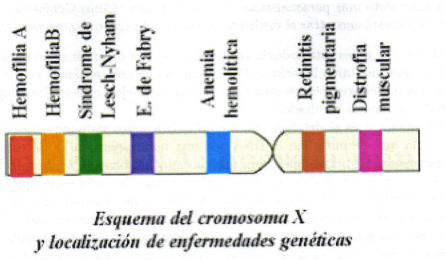

Figura A.16: Mapa genético del genoma humano: ubicación precisa de los genes en el interior de los cromosomas.

En julio de 1991 ya se habían identificado 2114 genes [27]. En 1993 se esperaba que distintos microorganismos, manipulados genéticamente o no, pudieran resultar de gran utilidad en la producción de alimentos, eliminación de basuras, obtención de materias primas y para descontaminar (existen bacterias capaces de *digerir* sustancias ultratóxicas, como las dioxinas, y transformarlas en moléculas totalmente inofensivas) [174]. Otras bacterias naturales sirven para enriquecer yacimientos minerales de vetas pobres, volviendo rentables su explotación.

Cuando los biólogos secuencian un gen (averiguan su lugar en la cadena de DNA y su función), saben al mismo tiempo de qué unidades elementales se compone la correspondiente proteína. Gracias a la ingeniería genética, hoy resulta más fácil encontrar y decodificar genes que estudiar directamente la estructura de una proteína. Así, una vez que han aislado un gen, los biólogos pueden utilizar la técnica del DNA recombinante para fabricar la proteína que desean en la cantidad que quieran. Esto se consigue implantando el gen elegido en bacterias con objeto de convertirlas en diminutas fábricas de la proteína correspondiente. Los microbios alterados genéticamente ya sólo necesitan multiplicarse, algo que no requiere condiciones demasiado complicadas en el laboratorio: algunos recipientes llenos de agua azucarada como alimento para las bacterias, una máquina que agite los recipientes con objeto de que las bacterias no se aglomeren en el fondo, y una incubadora a 37 grados de temperatura. Al cabo de 24 horas, la proteína humana buscada ya puede ser extraída. En 1993 ya se estaban generando proteínas genéticamente alteradas [172].

En ese año ya se consideraba que el proyecto avanzaba gracias a la tecnología de los modernos ordenadores, que permitían leer unas 100000 bases por día. Los sofisticados programas se han encargado de unir los trozos de DNA y recomponer el rompecabezas molecular. Si esta tarea se hubiera realizado de forma manual, se hubiera demorado durante varias generaciones. Gracias a la tecnología, en 1993 ya se había leído completamente el cromosoma 21, que alberga genes cuyas alteraciones tienen que ver con el síndrome de Down y la enfermedad de Alzheimer y el cromosoma Y, que determina el sexo masculino. Los científicos esperaban que el mapa del genoma humano estaría acabado mucho antes de los esperado. Además, secuenciar más bases al día implicaba un evidente ahorro económico.

En octubre de 1995, se habían secuenciado ya el 75 % de todos los genes [27]. A finales del año 1996 se preveía que en menos de 5 años fuera posible conocer el secreto de algunas enfermedades, la manera de

combatirlas y, lo más importante, de prevenirlas. Según el doctor Michael Goettmann, coordinador del proyecto genoma de los Institutos Nacionales de la Salud estadounidenses, la terapia genética (la manera de prevenir y tratar las enfermedades tomando como punto de partida la reparación del DNA alterado) estaba tan cercano que se esperaba que las inyecciones DNA se venderán en las farmacias antes de que finalice la primera década del siglo XXI.

En 1996 se esperaba también que el desarrollo de medicamentos y vacunas genéticas ofrecieran nuevas estrategias para tratar enfermedades incurables, como el SIDA y ciertos tumores [49] y encontrar fármacos capaces de acabar con las enfermedades sin indeseables efectos secundarios. Ya existían más de 200 proteínas recombinantes que se estaban empleando con mayor o menor éxito en alguna fase experimental clínica, para combatir el cáncer, las infecciones víricas y bacterianas, las cicatrización de las heridas, las dolencias cardiovasculares, para atajar las infecciones y tumores, para tratar la diabetes, el enanismo y la osteoporosis. También existen vacunas contra la malaria, la gripe, la hepatitis, el herpes y el sida. Como se indica en [49] la forma de parar el avance del cáncer se basa en la síntesis protaica, donde una molécula del RNA mensajero hace las veces de molde para la creación de la proteína. La forma de hacerlo consiste en diseñar medicinas genéticas que se unan a los segmentos elegidos de DNA e impidan la traducción de la información o al RNA mensajero para que éste quede anulado como el molde. El método consiste en extraer células del paciente para incorporar en su material genético un gen sano que supla la función del defectuoso.

El 26 de junio de 2000 es ya una fecha para la historia de la biomedicina tras 10 años de intensa investigación: se anunció formalmente la secuenciación del genoma humano (el orden de los 3200 millones de letras que componen el DNA) [140]. Este logro fue anunciado consecutivamente en China, Japón, Francia, Alemania, el Reino Unido y Estados Unidos.

Las consecuencias futuras de este sensacional logro científico son todavía difíciles de predecir, pero parece seguro que se podrán vencer al fin las casi 4000 enfermedades hereditarias conocidas.

Con los resultados obtenidos en este proyecto se puede afirmar ahora que los seres humanos son muy semejantes entre sí. Efectivamente, nuestras diferencias a nivel genético no superan el 0.2 %. Por ejemplo, en los 5 genomas descifrados (3 mujeres y 2 hombres de diferentes orígenes étnicos) no hay modo de distinguir una etnia de otra [52]. También se ha determinado que los humanos compartimos con el chimpancé el 98.4 % de los genes; con una ternera de Ávila, el 90 %; y con un ratoncillo, el 75 %. Hay otros animales menos parecidos, como el pequeño gusano nematodo, que no llega a tener ni mil células en su cuerpo y en el que tan sólo un 40 % de sus 8000 genes tienen homólogos humanos [140]. Las personas compartimos genes con todos los seres vivos y, en general, la diferencia genética nos da la medida de la antigüedad de la separación evolutiva entre dos especies.

El genoma humano es aproximadamente 1000 veces mayor que el de *E. coli*, pero el número de genes es probablemente sólo 50 veces mayor, contiene 3000 millones de letras que dice cómo somos a través de decenas de miles de genes o instrucciones [140]. Unos meses antes se estimaba que la cifra era de 100000 a 150000 [53].

Hoy en día, uno de los mayores desafíos de la biología molecular es descifrar la función de esa mayor parte del DNA en el genoma humano que no codifica genes ni proteínas; averiguar las condiciones en que se activan y desactivan y las acciones biológicas que emergen de la interacción de genes distantes en la molécula de DNA. El tamaño de la molécula de DNA de un organismo no es un indicador infalible de su complejidad. Por ejemplo: el DNA de la levadura de cerveza (*Saccharomices cerevisae*) tiene 12 Mbp, la mosca de la fruta (*Drosophila melanogaster*) llega a los 140 Mbp y la mariposa o gusano de seda (*Bombix mori*) unos 490 Mbp. Entre los de genoma mayor que el humano (30000 Mbp) están el ratón (*Mus musculus*) con 3300 Mbp, el guisante (*Pisum sativum*) con 4800 Mbp y la langosta (*Locusta migratoria*) con 5000 Mbp. Existe una variedad de trigo (*Triticum aestivum*) cuyo genoma llega a los 17000 Mbp [140]. La cosa se complica si se tiene en cuenta que entre un gen y el que le sigue en la hebra genética existen segmentos de secuencias interminables y aparentemente sin sentido (DNA basura). Afortunadamente, los ordenadores están facilitando esa tarea. Pero para conocer lo más exactamente posible el funcionamiento del organismo, los científicos también tendrán que fijarse en el producto de

los genes, es decir, las proteínas. Éstas son realmente las que hacen el trabajo. Aunque se estima que únicamente contamos con unos 30000 genes, la cantidad de combinaciones entre ellos hace posible que en una célula se puedan contabilizar millones de proteínas químicamente distintas. Durante el *XVIII Congreso Internacional de Bioquímica y Biológia Molecular*, que se celebró en junio de 2000, la empresa privada *PE Celera Genomics* anunció su nuevo objetivo: realizar el mapa de las proteínas que rigen todas las reacciones químicas del cuerpo humano (el **Proyecto Proteoma Humano**) como complemento del Proyecto Genoma [52].

La lectura del genoma también nos ha permitido saber que tenemos genes suicidas. Las células del cuerpo pueden suicidarse en determinadas circunstancias: tras cumplir su cometido biológico, cuando dejan de recibir las señales bioquímicas que las mantienen operativas y si sufren un daño irreparable en el material genético u otros componentes celulares. Las células suicidas experimentan una fragmentación de sus cromosomas y mueren. Los genes suicidas guardan una inquietante conexión con los procesos cancerosos. Recientes investigaciones demuestran que las células tumorales lo son, en parte, porque han perdido la capacidad de autodestrucción. Ciertas mutaciones genéticas hacen que los programas de suicidio funcionen de forma incorrecta y contribuyan al deterioro celular, que conduce al cáncer y la muerte.

A.10. Importancia de la bioinformática

La bioinformática es un campo que ha estado dominado por la biología estructural a lo largo de los últimos 20-30 años. Con los proyectos genoma y el déficit secuencias/estructuras se está produciendo un cambio espectacular. Su principal reto es la racionalización de la cantidad de información de secuencias, para obtener más y mejores medios de almacenamiento de datos y también para diseñar más herramientas de análisis. Lo más importante de este proceso analítico es la necesidad de convertir la información de las secuencias en conocimiento bioquímico y biofísico, descifrar las claves estructurales, funcionales y evolutivas codificadas en el lenguaje de las secuencias biológicas. Se puede considerar que se trata de decodificar un lenguaje desconocido. El objetivo de la bioinformática es comprender cada parte de la secuencia de una proteína para deducir sus funciones y ser capaces en un futuro de diseñar nuevas proteínas.

Al investigar el significado de las secuencias aparecen dos temas de análisis: en la primera aproximación se aplican técnicas de reconocimiento de patrones para encontrar similitudes entre secuencias e inferir funciones y estructuras relacionadas. En la segunda aproximación, se emplean métodos de predicción para deducir la estructura tridimensional su funcionalidad, directamente desde la secuencia lineal. A día de hoy el número de estructuras determinadas experimentalmente es bastante bajo. Por este motivo en los próximos años el desarrollo de técnicas más potentes para el reconocimiento de patrones y para la predicción de estructuras continuará siendo un tema de investigación dominante en la investigación bioinformática.