Universidad Autónoma de Barcelona
Escuela Técnica Superior de Ingeniería
Departamento de Arquitectura de Computadores y
Sistemas Operativos

Modelización y Asignación de Tareas para Aplicaciones Paralelas con Flujo Continuo de Datos de Entrada

Memoria presentada por Fernando Guirado Fernández para optar al grado de Doctor en Informática por la Universidad Autónoma de Barcelona.

Barcelona, Julio de 2007

Modelización y Asignación de Tareas para Aplicaciones Paralelas con Flujo Continuo de Datos de Entrada

Memoria presentada por Fernando Guirado Fernández para optar al grado de Doctor en Informática por la Universidad Autónoma de Barcelona. Trabajo realizado en el Departamento de Arquitectura de Computadores y Sistemas Operativos (DACSO) de la Escuela Técnica Superior de Ingeniería de la Universidad Autónoma de Barcelona, dentro del programa de Doctorado en Informática, opción A: "Arquitectura de Computadores y Procesamiento Paralelo", bajo la codirección de la Dra. Ana Ripoll Aracil y la Dra. Concepció Roig Mateu.

Barcelona, Julio de 2007

Vo. Bo. Codirectora de Tesis

Dra. Ana Ripoll Aracil

Vo. Bo. Codirectora de Tesis

Dra. Concepció Roig Mateu

A mi familia

Agradecimientos

Cuando se inicia el desarrollo de una tesis doctoral, deben ser pocos los que son conscientes del reto real al que se enfrentan. Cuando el trabajo va tomando forma y se vislumbra el final, es entonces cuando se tiene consciencia del esfuerzo que ha supuesto tanto a nivel intelectual como a nivel personal. Por este motivo quiero dedicar estas líneas a aquellas personas que han hecho posible que este momento llegue.

En primera instancia quiero agradecer a la Dra. Ana Ripoll y a la Dra. Concepció Roig por la dirección de esta tesis. Gracias a sus acertados puntos de vista, especial dedicación y notable esfuerzo se ha podido llevar a cabo este trabajo.

Al Dr. Emilio Luque y a las personas que forman parte del Departamento de Arquitectura de Computadores y Sistemas Operativos de la Universitat Autònoma de Barcelona, por su apoyo e interés en éste trabajo.

A mis compañeros, pero sobre todo amigos, del Grup de Computació Distribuïda de la Universitat de Lleida, Dr. Fernando Cores, Dr. Francesc Giné, Dr. Francesc Solsona y Josep Lluís Lérida, por su ayuda en los momentos complicados ofreciendo siempre una visión positiva. Al resto de compañeros de la Escola Politècnica Superior por los buenos momentos a la hora del café.

A mis amigos en general, que aún sin entender porqué he pasado las vacaciones y los fines de semana trabajando, han sido capaces de soportarme cuando me he dejado ver.

A mi familia, ya que entre todos han sido capaces de darme el apoyo y la fuerza necesaria para poder llegar hasta aquí. Y agradecer de forma muy especial a mi esposa Yolanda por estar a mi lado, por su infinita paciencia y por entenderme aún cuando yo mismo no soy capaz de hacerlo. Y a mis hijas Júlia y Ana, que han sabido hacerme olvidar mis preocupaciones cuando más lo he necesitado.

Propósito y desarrollo de la memoria

En la actualidad, el procesamiento paralelo es una técnica de programación completamente asentada en el desarrollo de aplicaciones de supercomputación. Aún así, la complejidad de su uso continua siendo elevada, por lo que en gran medida es el propio programador el encargado de determinar las principales pautas en el desarrollo de las aplicaciones paralelas. La definición del tipo de paralelismo a explotar, de datos o funcional, o la granularidad de las tareas son sólo dos ejemplos de las decisiones de diseño que debe afrontar. En la actualidad existe una gran variedad de metodologías y de herramientas que facilitan la creación de aplicaciones paralelas y que ayudan al programador en las fases de su programación y posterior ejecución. Este es el caso de la asignación de las tareas a los nodos de procesamiento, acción denominada como *mapping*, para la que la literatura proporciona una gran variedad de heurísticas.

Cada vez más, la computación paralela afronta nuevos retos que la propia ciencia y la sociedad demandan. Este es el caso del conjunto de aplicaciones paralelas que se basan en el procesamiento de un flujo continuo de datos, en las que las tareas se organizan en una estructura de dependencias, en la que cada una de ellas recibe el dato a procesar de sus predecesoras, realizan el cómputo que tienen asignado y envían su resultado a sus tareas sucesoras. Su forma de actuar hace que se denominen comúnmente como *aplicaciones pipeline*. Este es el caso de las aplicaciones de procesamiento de vídeo en tiempo real, procesamiento de señal o sistemas de interacción persona-ordenador en tiempo real, por poner algún ejemplo y que están siendo utilizadas en múltiples ámbitos, como puede ser la industria cinematográfica, el análisis de imágenes médicas, control de calidad en procesos industriales, aplicaciones multimedia, etc.

Si la complejidad en el desarrollo de las aplicaciones paralelas clásicas con el objetivo de obtener un determinado rendimiento es elevada, la problemática añadida de tener presente el procesamiento de un flujo continuo de datos, la incrementa notablemente al añadir nuevos objetivos de rendimiento a alcanzar, como puede ser el mantener un ratio predeterminado en el procesamiento de datos por unidad de tiempo, parámetro

denominado como *productividad* o conseguir un tiempo mínimo de respuesta en el procesamiento individual de un dato, denominado como *latencia*.

En la literatura existen trabajos que enfocan su esfuerzo en la optimización de las aplicaciones pipeline, pero por falta de generalidad, no pueden ser utilizados en todos los ámbitos posibles en los que estas aplicaciones aparecen. En algunos casos, se limita su uso a un tipo concreto de arquitectura paralela, como puede ser el de memoria compartida, en otros se marcan fuertes restricciones en la estructura de la aplicación pipeline, definiendo la forma en que se deben dar las dependencias entre tareas o el patrón que define como se debe distribuir su cómputo.

En el caso de las características que definen a la propia aplicación, es importante no marcar restricciones si se quiere dar una solución global para la optimización de cualquier tipo de aplicación pipeline, independientemente de su ámbito de uso. Por este motivo el presente trabajo se define en la necesidad de presentar una solución para la optimización de este tipo de aplicaciones sin establecer ningún tipo de restricción. En este sentido se ha desarrollado una solución que afronta el problema desde una perspectiva global basada en dos fases. La primera orientada a la definición más adecuada de la estructura de dependencias de tareas de la aplicación, representada mediante un grafo de tareas, que permita alcanzar unos requisitos de rendimiento. Para ello se han desarrollado dos técnicas, en las que se identifican las tareas que actúan como cuello de botella en la obtención del rendimiento deseado y establece los mecanismos más adecuados para redefinir, bien la estructura de la propia aplicación o la forma en que la tarea lleva a cabo su función. La segunda fase lo hace a partir del proceso de mapping, el cual obtiene la asignación de tareas a nodos de procesamiento cumpliendo los requisitos de rendimiento marcados como objetivo.

Así, las soluciones que se proponen ofrecen una herramienta para el desarrollo de aplicaciones pipeline que abarca desde las etapas iniciales en la definición del grafo de tareas de la aplicación que permita alcanzar el rendimiento deseado, hasta las etapas finales en la asignación de las tareas a la arquitectura a utilizar.

La presente memoria se organiza en los siguientes capítulos.

- Capítulo 1. Se enmarcan las aplicaciones paralelas de flujo continuo de datos dentro de la programación paralela, mostrando sus características y las diferencias respecto a las aplicaciones paralelas clásicas que las definen. Este conocimiento es la base para poder afrontar el estudio que se llevará a cabo en este trabajo. A continuación se identifican sus parámetros de rendimiento y clasifican los diferentes tipos de flujos de datos de entrada. Para finalizar se presenta el estado del

arte que trata este tipo de aplicaciones y los objetivos que se desarrollan en este trabajo.

- Capítulo 2. En este capítulo se profundiza en el estudio de las aplicaciones paralelas pipeline, determinando las características que definen su comportamiento y los factores que afectan a los parámetros de rendimiento y que deben ser tenidos en cuenta a la hora de marcarse unos objetivos a alcanzar en el rendimiento.

- Capítulo 3. A partir del conocimiento obtenido de las aplicaciones pipeline, en este capítulo se presentan un conjunto de heurísticas encaminadas a obtener el rendimiento deseado. Para ello la solución propuesta se plantea en dos fases. La primera orientada a la definición del modelo de grafo que define la estructura de tareas de la aplicación. Para ello se ha desarrollado dos técnicas: (a) Técnica de Paralelización, la cual explota el paralelismo implícito de las tareas de la aplicación y (b) Técnica de Replicación, con la que se aumenta el grado de concurrencia en el procesamiento de los datos del flujo de entrada. En la segunda fase se trata el problema desde el punto de vista del mapping, aportando dos heurísticas de diferente complejidad, MPASS (*Mapping of Pipeline Applications based on Synchronous Stages*) y MPART (*Mapping of Pipeline Applications based on Reduced Tree*). Ambas actúan bajo un criterio de rendimiento obteniendo una asignación que minimiza el número de nodos de procesamiento utilizados.

- Capítulo 4. Se presenta un estudio experimental, en el que se comprueba la capacidad de optimizar el rendimiento de las aplicaciones pipeline para las heurísticas presentadas en el capítulo previo. La experimentación se desarrolla en dos ámbitos. El primero evalúa de forma individual cada una de las heurísticas desarrolladas, utilizando un conjunto de aplicaciones pipeline sintéticas. En el segundo se analiza la efectividad de las heurísticas de forma conjunta, desde la perspectiva de la definición del grafo de tareas de la aplicación y posterior mapping, sobre tres aplicaciones pipeline dedicadas al procesamiento de secuencias de imágenes.

Estos capítulos se complementan con el Apéndice A en el que se muestran los grafos de tareas que modelan las aplicaciones pipeline utilizadas en el capítulo de experimentación.

Índice general

Capítulo 1

Introducción

Para la resolución de problemas computacionales complejos el paradigma de programación utilizado hoy en día es el de la programación paralela. Las aplicaciones paralelas clásicas procesan un conjunto de datos que por lo general ha sido obtenido y almacenado antes de su ejecución. En la actualidad, existe un amplio rango de situaciones en las que el procesamiento de la información se debe realizar a medida en que ésta se va generando. Este es el caso de las aplicaciones de procesamiento de vídeo en tiempo real, procesamiento de señal o sistemas de interacción persona-ordenador en tiempo real, por poner algún ejemplo y que están siendo utilizadas en múltiples ámbitos, como puede ser la industria cinematográfica, el análisis de imágenes médicas, control de calidad en procesos industriales, etc.[DSAW99][LLP01][RNR+03][YKS03][KMP06][ZK07][LHC07].

Estas aplicaciones, que se basan en el procesamiento de un flujo continuo de datos, demandan unos compromisos de rendimiento que difieren de los habituales en las aplicaciones paralelas clásicas que solo procesan un único dato, como puede ser mantener un ratio de procesamiento de datos por unidad de tiempo constante.

En este capítulo se da una visión global de este tipo de aplicaciones, definiendo sus características, como se enmarcan dentro de la programación paralela y el estado del arte en su estudio. Para finalizar se darán a conocer los objetivos que se desarrollan en este trabajo.

1.1. Aplicaciones paralelas con flujo continuo de datos

Las aplicaciones paralelas que se basan en el procesamiento de un flujo continuo de datos de entrada están formadas por un conjunto de tareas que poseen un comportamiento formado por tres fases que repiten en el procesamiento de cada uno de los datos

del flujo de entrada, en las que las acciones que se llevan a cabo son:

1. Espera y recepción de la información a procesar por parte de las tareas predecesoras.

2. Procesamiento de la información recibida.

3. Transmisión del resultado obtenido a cada una de sus tareas sucesoras.

Esta forma de actuar se puede considerar como iterativa, siendo el número de iteraciones a realizar el correspondiente al número de datos a procesar. Se ha de tener presente que la primera y última tarea pueden tener un comportamiento diferente al carecer de una tarea predecesora y sucesora respectivamente.

En la Figura 1.1 se muestra un ejemplo de aplicación con flujo continuo de datos que representa la estructura de tareas de una aplicación STAP (*Space-Time Adaptive Processing*) [CkLW+00]. Las aplicaciones STAP son utilizadas en el control de radar aéreo, para la detección de objetivos pequeños en entornos saturados de información y con gran cantidad de interferencias y ruido ambiental. La información de entrada es capturada por un array de radares formando un bloque de datos con estructura en tres dimensiones denominado CPI (*Coherent Processing Interval*). Cada bloque se mueve por cada una de las tareas, que realizan un cómputo diferente hasta alcanzar la última tarea que genera un informe de los posibles objetos detectados.

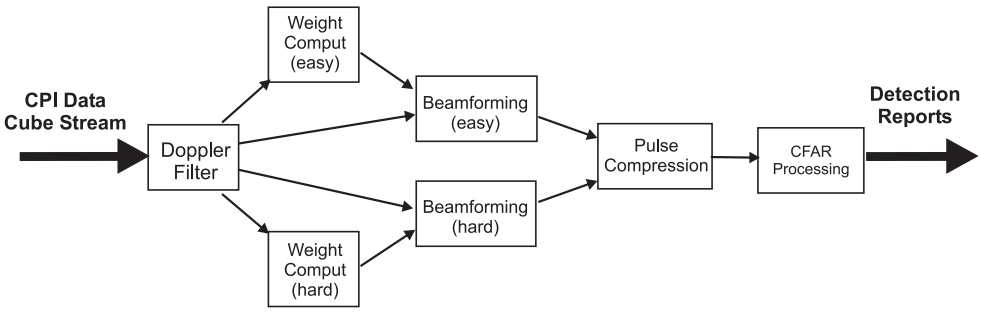


Figura 1.1: Estructura de tareas de una aplicación STAP (*Space-Time Adaptive Processing*).

La estructura que define las dependencias entre tareas de este tipo de aplicaciones, es una estructura acíclica, donde las dependencias entre tareas hace que su forma de actuar se pueda definir como una ejecución pipeline, en analogía a la a estructura pipeline utilizada en el proceso de ejecución de las instrucciones por parte de la CPU. Este término, utilizado en la literatura para identificar las aplicaciones con flujo de

datos de entrada, servirá en este trabajo para referenciarlas de una forma más simple [SV00][YKS03][DA05].

El procesamiento de un flujo continuo de datos es el motivo que determina la diferencia con respecto a las aplicaciones paralelas clásicas desde el punto de vista del rendimiento que se desea obtener de ellas. Si en las aplicaciones paralelas clásicas el objetivo principal es el de obtener el menor tiempo de ejecución global, en las aplicaciones pipeline se definen dos parámetros que se encuentran relacionados con la capacidad de procesar el flujo de entrada. Estos parámetros son:

- Latencia. Determina el tiempo necesario para procesar de forma completa un dato del flujo de entrada.
- Productividad. Indica el ratio en que es procesado el flujo de datos de entrada expresado en datos procesados por unidad de tiempo. La productividad se mide también en términos de su valor inverso, denominado *Iteration Period* (IP), que representa el intervalo de tiempo existente entre la obtención de dos resultados consecutivos.

Debido a la gran variedad de ámbitos científicos en los que se utilizan las aplicaciones pipeline, existen diferentes necesidades a la hora de optimizar estos parámetros de rendimiento. En aquellas aplicaciones en las que el objetivo consiste en minimizar el tiempo de respuesta, como puede ser el caso de obtener el resultado en tiempo real, se deberá minimizar la latencia. Éste es el caso de las aplicaciones de procesamiento de señal o estimación de movimiento [LLP98][SV00][ZCA06]. Por el contrario, en aplicaciones como las del tipo *Adaptive Signal Processing*, en las que se recibe de forma continua el flujo de datos de entrada, los resultados se han de obtener manteniendo el mismo ratio de entrada, en cuyo caso es la productividad el parámetro principal a tener presente en la optimización [LLP01][CA06][ZK07].

En la literatura los principales objetivos que se marcan a la hora de optimizar los parámetros de rendimiento de las aplicaciones pipeline, se enfocan principalmente sobre dos aspectos diferentes [CNNS94]:

- Minimizar la latencia.
- Maximizar la productividad.

El proceso de optimización del rendimiento de cualquier aplicación paralela, se plantea en la literatura a través de establecer un modelo que capture de forma resumida las

características de las dependencias entre las tareas, su cómputo y comunicaciones. Este modelo se representa mediante un grafo, en el que cada tarea de la aplicación aparece como un nodo y mediante líneas que los unen las dependencias entre ellas. En la literatura, para definir el grafo que representa a la aplicación paralela se puede encontrar tres opciones: (a) Grafo de Precedencia de Tareas (TPG: *Task Precedence Graph*) [NT93], (b) Grafo de Interacción de Tareas (TIG: *Task Interacion Graph*) [Pel98] y (c) Grafo de Interacción de Tareas Temporal (TTIG: *Temporal Task Interaction Graph*) [RRG07].

Cada uno de los grafos se utiliza en función de las características de la aplicación paralela, por lo que no todos ellos pueden ser usados en los mismos casos. Con este objetivo a continuación se presenta de forma resumida cada uno de ellos, para poder más adelante justificar la elección realizada para las aplicaciones pipeline:

(a) *Grafo de Precedencia de Tareas*

Mediante el Grafo de Precedencia de Tareas (TPG: Task Precedence Graph), la aplicación se representa mediante un grafo dirigido acíclico, $G = (N, E)$, donde N es el conjunto de nodos del grafo, cada uno representando una tarea y E es el conjunto de arcos entre tareas adyacentes. Dichos arcos representan tanto las comunicaciones como las relaciones de precedencia entre tareas e indican un orden parcial en la ejecución de las mismas.

A cada tarea $T_i \in N$ se le asocia un valor no negativo $\mu(T_i)$, que representa su tiempo de cómputo estimado y a cada arco $(T_i, T_j) \in E$ se le asocia un valor $com(T_i, T_j)$, que representa el volumen de comunicación que se transmite de T_i a T_j durante la ejecución. La Figura 1.2 ilustra un ejemplo de grafo TPG de seis tareas $\{T_0,..,T_5\}$. Como ejemplo de orden parcial en la ejecución, puede observarse en el grafo, que las tareas T_0 y T_1 son tareas iniciales y pueden empezar su ejecución desde el primer momento, en cambio la tarea T_3 debe esperar a que T_0 y T_2 hayan finalizado y transmitido sus resultados para iniciar su ejecución.

(b) *Grafo de Interacción de Tareas*

Mediante el Grafo de Interacción de Tareas (TIG: Task Interaction Graph), la aplicación paralela se representa mediante un grafo no dirigido $G = (V, E)$, donde V es el conjunto de vértices, cada uno representando a una tarea del programa y E es el conjunto de aristas que representan las interacciones entre tareas adyacentes (i.e. tareas que se comunican entre ellas). Al igual que sucede en el modelo TPG, en el TIG se asocian pesos a los vértices y a las aristas, representando

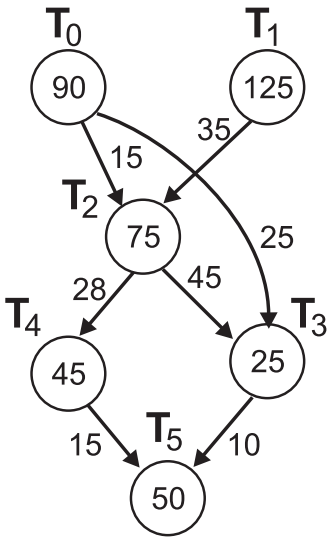


Figura 1.2: Ejemplo de Grafo de Precedencia de Tareas.

los tiempos de cómputo de las tareas y los volúmenes de comunicación que se intercambian respectivamente. El TIG no incluye ninguna información relativa al orden de ejecución de las tareas, y de igual manera las aristas, al no ser dirigidas, no representan las precedencias existentes. Debido a ello, en la literatura los autores que usan el grafo TIG para modelar la aplicación asumen que todas las tareas pueden ejecutarse concurrentemente.

El grafo de la Figura 1.3 es un ejemplo de grafo TIG de cuatro tareas $\{T_0, .., T_3\}$, en el que de cada tarea solo se conoce el tiempo de cómputo y cuales son sus tareas adyacentes con el volumen de comunicación que se transmite entre ellas, pero no el instante en que se efectúa esta transmisión.

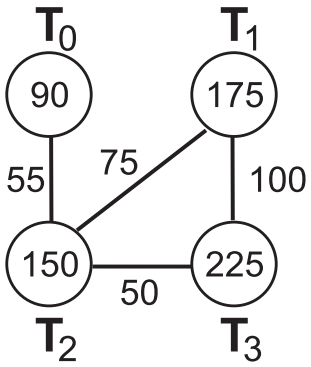


Figura 1.3: Ejemplo de Grafo de Interacción de Tareas.

(c) Grafo de Interacción de Tareas Temporal

El Grafo de Interacción de Tareas Temporal (TTIG: Temporal Task Interaction Graph) es un modelo propuesto dentro de nuestro grupo de investigación que engloba las ventajas ofrecidas por los dos grafos anteriores y que puede ser utilizado como una solución unificada de los mismos.

El modelo representa la aplicación paralela mediante un grafo dirigido $G = (N, E)$, donde N es el conjunto de nodos que representan a las tareas de la aplicación, y E es el conjunto de arcos que determinan las comunicaciones entre tareas adyacentes. Cada nodo posee asociado un valor que representa el tiempo de cómputo de la tarea a la que hace referencia y los arcos tienen asociados dos parámetros: el volumen de datos involucrado en la comunicación y el grado de paralelismo, que es un valor normalizado en el rango $[0, 1]$ que indica el máximo porcentaje de tiempo de cómputo que una tarea puede realizar en paralelo con su adyacente, considerando sus dependencias mutuas.

En la Figura 1.4 se muestra un ejemplo de grafo TTIG. En él se puede observar como para las tareas T_0 y T_1, existe una comunicación bidireccional, representada por dos arcos. Cada uno de ellos posee un valor diferente para el volumen de datos transferidos, en el caso del arco $T_0 \rightarrow T_1$, corresponde a 45 unidades y para el arco $T_1 \rightarrow T_0$ 10 unidades. De igual forma los valores asociados al grado de paralelismo son diferentes ya que están expresados en función del tiempo de cómputo de la tarea receptora de la comunicación, siendo 0.57 y 0.43 respectivamente, lo que implica que ambas tareas podrán ejecutar en paralelo 86 unidades de tiempo como máximo.

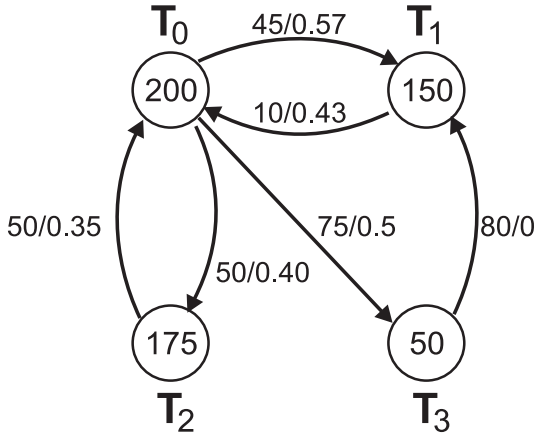


Figura 1.4: Ejemplo de Grafo de Interacción de Tareas Temporal.

De los tres modelos de grafo presentados, es el modelo TPG el que permite tener presente las principales características de comportamiento de las tareas de las aplicaciones pipeline, siendo los motivos que fundamentan su elección los siguientes:

- Las tareas poseen una estructura estricta de dependencias, que representan la secuencia temporal que se debe seguir en su ejecución.
- En la estructura de las aplicaciones pipeline no se dan ciclos derivados de las dependencias entre tareas, por lo que la información siempre discurre en el mismo sentido desde la tarea inicial hasta la tarea final.
- No existen comunicaciones bidireccionales entre tareas.
- Las comunicaciones solo se realizan al inicio de cada tarea, en espera de la información del dato a procesar, y al final de cada tarea para transmitir los resultados generados.

1.1.1. Optimización de las aplicaciones pipeline

Una forma de abordar la optimización de la ejecución de las aplicaciones pipeline es mediante el proceso de mapping, en el que se determina que tareas deben ser asignadas a cada nodo de procesamiento para cumplir unos objetivos de rendimiento predeterminados.

En la literatura existen múltiples propuestas de heurísticas de mapping de aplicaciones paralelas basadas en el grafo TPG de la aplicación [MLS94][KA96][VKS+06]. Las heurísticas propuestas en dichos trabajos no pueden ser aplicadas directamente sobre las aplicaciones pipeline ya que no tienen en cuenta la característica de ejecución iterativa de sus tareas. La problemática añadida asociada a la ejecución iterativa radica en el hecho de que cada tarea al finalizar el procesamiento de un dato del flujo de entrada, no da por finalizada su ejecución si no que la tarea permanece activa mientras existan datos a procesar. En esta situación los recursos de cómputo utilizados en la ejecución de estas tareas, no son liberados como se presupone en las heurísticas clásicas de mapping de TPGs.

Para adecuar las heurísticas clásicas de mapping de TPGs al mapping de aplicaciones pipeline, existen trabajos de la literatura que partiendo de la asignación obtenida, aplican técnicas de refinamiento que permitan tener en cuenta la naturaleza iterativa de las tareas [YKS03]. Este tipo de soluciones, al no tomar como punto de partida el

comportamiento de la aplicación pipeline, no suelen obtener una asignación eficiente de los recursos de cómputo utilizados, además de añadir mayor complejidad a la heurística de mapping. Esta situación hace que se apunte como una solución más efectiva el desarrollo de nuevas heurísticas que desde un principio tengan presente las características del comportamiento de las aplicaciones pipeline.

Para la optimización de los parámetros de latencia y de productividad se ha de tener presente que éstos no están relacionados directamente, lo cual provoca que en múltiples situaciones intentar optimizar la aplicación para uno de ellos produzca la reacción contraria sobre el otro, yendo en detrimento de su rendimiento.

Para el caso de intentar reducir la latencia, los métodos existentes en la literatura se basan en dos objetivos: (a) eliminar las comunicaciones [RA01][DA04] o (b) reducir el tiempo de cómputo de las tareas [CNNS94][SV00].

La primera opción, eliminar las comunicaciones, se puede asumir en la fase de mapping, mediante el uso de heurísticas que se encarguen de asignar sobre el mismo nodo de procesamiento aquellas tareas cuyas comunicaciones afecten en mayor medida a la latencia. Por el contrario la segunda opción, reducir el cómputo de las tareas, requiere actuar sobre la forma en que han sido implementadas. Dejando de lado la opción que involucra tener que rediseñarlas y reprogramarlas totalmente, existe la posibilidad de aplicar sobre ellas heurísticas de paralelización basadas en la aplicación del paralelismo de datos, siempre que sea posible por la funcionalidad que tienen asignada. Mediante esta técnica, cada uno de los datos será dividido en partes que pueden ser procesadas de forma concurrente, reduciendo en consecuencia el tiempo de procesamiento necesario originalmente para un dato individual [HR93][SV00][GRR+03].

Si el parámetro a optimizar es la productividad, la solución pasa por dos opciones: (a) reducción del tiempo de cómputo de las tareas y (b) aumentar la capacidad de la aplicación para procesar un mayor número de datos de forma concurrentemente.

La opción de reducir el tiempo de cómputo, aplicada en la misma forma que en el caso de la optimización de la latencia, permite conseguir que las tareas puedan generar su resultado en un menor tiempo, lo que redunda en un mayor ratio de datos procesados por unidad de tiempo. En el caso de la solución basada aumentar el número de datos procesados de forma concurrentemente, se consigue a partir de la creación de nuevas líneas de ejecución en el grafo de tareas de la aplicación, que discurren de forma paralela y en las que se añaden las tareas que forman el cuello de botella en la obtención del rendimiento deseado. Esta técnica recibe el nombre de replicación [LLP01][AA04][GRRL05b].

La utilización de la paralelización o de la replicación para la optimización del rendimiento de la aplicación paralela está supeditada a las características del flujo de datos de entrada y a como éste debe ser procesado, determinando en gran medida el comportamiento de las aplicaciones pipeline. Por este motivo es importante definir cuales son sus principales características, siendo el aspecto principal a tener en cuenta el que hace referencia a la existencia o no de dependencias entre los datos que forman parte del flujo de entrada. En la literatura se plantean los siguientes tipos de dependencias [CkLW+00]:

- *Flujos de datos independientes*

  Cada dato del flujo de entrada, no posee relación con ningún otro, por lo que las tareas no precisan mantener información específica del procesamiento de ese dato para comenzar a procesar un dato posterior. Hay que remarcar que en este tipo de flujo el orden en el que se procesan los datos no es relevante, pudiéndose hacer en un orden diferente al de llegada.

  Un ejemplo para este tipo de flujo de datos es el formado por la secuencia de imágenes capturadas desde una cámara de control de calidad en un proceso de producción. En este caso cada imagen hace referencia a un producto concreto a evaluar y el resultado de esta evaluación es independiente del resto [DSAW99].

- *Flujos de datos dependientes*

  En este caso los datos mantienen una relación entre sí que es utilizada para el procesamiento de cada uno de ellos en las diferentes tareas de la aplicación. De esta forma, el resultado de procesar un dato concreto depende de los resultados obtenidos al procesar otros datos del flujo de entrada. Ante esta situación, el orden de llegada se convierte en el orden en el que los diferentes datos han de ser procesados para asegurar que el resultado sea el correcto.

  Un ejemplo se da en el algoritmo de compresión de video MPEG2, en el que para la compresión de una imagen específica, se requiere la información referente a otras imágenes presentes en la secuencia que forma el flujo de entrada [IS04].

1.2. Estado del arte

El estado del arte en el estudio de las aplicaciones pipeline, se remonta a la década de los 90, durante la cual se sentaron las bases principales para su optimización. Así en-

contramos autores clásicos como pueden ser P. Hoang y J. Rabey que expusieron en su trabajo publicado en 1993, *Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput* [HR93], como obtener la máxima productividad para aplicaciones DSP (*Digital Signal Processing*) sobre arquitecturas multiprocesador. Desde entonces hasta la actualidad han sido múltiples las aportaciones de la comunidad científica, de entre las cuales a continuación se exponen algunas de las más relevantes.

P. Hoang y J. Rabey en [HR93] presentan las bases para maximizar la productividad sobre arquitecturas de memoria compartida bajo unas restricciones en el número de nodos de procesamiento y de la memoria presente. Su modelo de aplicación parte de un grafo TPG en el que las tareas poseen como característica el que, para reducir su tiempo de ejecución, se les puede asignar más de un procesador que tendrán en exclusividad para su ejecución. Para conseguir su objetivo, presentan una heurística *top-down* que intenta explotar las capacidades del uso de la estructura pipeline, asignando las tareas una a una y evaluando los beneficios de esta asignación. Este proceso se repite hasta que obtiene un resultado satisfactorio. Hay que destacar que en este trabajo aparecen unas primeras bases sobre el concepto de paralelización de tareas para mejorar la productividad, y que recibe el nombre de *Node Decomposition*. Para ello se aplica el paradigma de paralelismo de datos, en aquellas tareas que pueden convertirse en un cuello de botella de forma que se disminuye su tiempo de cómputo.

Más tarde, Alok N. Choudhary, añade al estudio de las aplicaciones pipeline la problemática de la latencia. Para ello se basa en aplicaciones de visión por computador y que también son representadas mediante un grafo TPG [CNNS94]. Su estudio plantea dos objetivos principales: (a) *Minimizar la latencia bajo unas restricciones de productividad* y (b) *maximizar la productividad bajo unas restricciones de latencia*. Al igual que en el trabajo previo de Hoang y Rabey, las arquitectura de procesamiento es de memoria compartida, de forma que se puede asignar, si es necesario, más de un procesador a aquellas tareas que son un cuello de botella en el rendimiento de la aplicación. Los autores de dicho trabajo, asumen que su método no es válido si la red de interconexión no tiene un gran ancho de banda y carece de contenciones. Posteriormente los mismos autores extienden su propuesta a las arquitecturas distribuidas con paradigma de programación basado en paso de mensajes, ampliando de esta forma el ámbito de utilización de su trabajo [CkLW+00]. El estudio de las aplicaciones pipeline por lo general ha estado centrado a casos concretos, como es el caso del trabajo realizado por W. Liu y V. Prasanna sobre las aplicaciones ESP (*Embedded Signal Processing*) y STAP (*Space-Time Adaptive Processing*), que tienen como principal característica la

necesidad de proporcionar una respuesta en tiempo real [LLP98] [LLP01]. Su trabajo difiere sustancialmente de los previos en la elección de arquitecturas distribuidas como plataforma de ejecución en la que las tareas de la aplicación solo pueden ejecutarse en un único nodo de procesamiento, aunque puede asignarse más de una tarea a cada uno de ellos. La solución propuesta en su trabajo requiere que las tareas sean definidas con granularidad gruesa para asegurar que los problemas originados por las comunicaciones no afecten a su método. Otra restricción importante y que no permite generalizar su metodología hacia cualquier tipo de aplicación pipeline es el que la estructura de la aplicación, deja de ser una estructura arbitraria representada por un grafo TPG para pasar a ser una estructura estricta formada por etapas bien definidas, en las que cada una de ellas puede tener un número variable de tareas pero forzosamente idénticas.

Finalmente su trabajo presenta la posibilidad de usar la replicación de tareas, que actúa sobre grupos de éstas, predefinidos previamente, con el fin de reducir la latencia y mejorar la productividad.

En el año 2000, J. Subhlok y G. Vondram presentan una recopilación de sus trabajos titulada *Optimal Use of Mixed Task and Data Parallelism for Pipelined Computations* [SV00]. El problema que plantean estos autores es similar al planteado por Alok N. Choudhary aunque posee como principal diferencia el que su método se basa en encontrar la asignación óptima acotando el problema a aplicaciones tengan una estructura con una única línea ejecución, lo que significa que todas las tareas de la aplicación poseen una única tarea predecesora y una única tarea sucesora, a excepción de la primera y la última tarea. Para obtener el mapping óptimo, su método realiza una búsqueda exhaustiva. La solución propuesta, es implementada el compilador paralelo *Fx* para HPF (*High Performance Fortran*) [RSB94]. En su trabajo, además de la heurística de mapping desarrollada presentan una forma de optimizar la productividad y reducir la latencia basada en la replicación de las agrupaciones de tareas obtenidas en la fase de mapping.

El creciente uso de arquitecturas distribuidas, hace que el problema asociado a las comunicaciones empiece a ser un factor importante en el desarrollo de un tipo de aplicaciones en las que la transferencia de datos, y en consecuencia la sincronización de las diferentes tareas, es un factor clave. Por ese motivo S. Ranaweera añade el factor asociado a las comunicaciones dentro de su definición del concepto de etapa en una aplicación pipeline [RA01]. A su vez plantea como utilizar la idea de replicación de tareas con el objetivo de reducir las comunicaciones y a su vez la latencia. De igual manera M.T. Yang basándose en el algoritmo ETF (*Earliest Task First*) para el

scheduling de grafos TPG, desarrolla una heurística en la que el objetivo principal es la reducción de la latencia. Para ello, a partir de la asignación obtenida del algoritmo ETF, aplica una fase de refinamiento que tiene presente la característica de iteratividad propia de las aplicaciones pipeline [YKS03]. Una aproximación diferente al problema de la minimización de la latencia para las aplicaciones de procesamiento de flujo de datos, se basa en el tratamiento de las comunicaciones y de los factores que las caracterizan. Wei Du hace un estudio en este sentido bajo la perspectiva de la elección del tamaño óptimo de los paquetes que forman los mensajes [DA04], mientras que Hsiao-Kuang, en [WLTC04], las trata desde el punto de vista de la gestión de las colas del sistema de comunicación. En ambos trabajos, por el contrario, no se tiene en cuenta la estructura de la aplicación.

El concepto de que sea el propio compilador el encargado de explotar las características propias de las aplicaciones pipeline, comentado en el trabajo realizado por J. Subhlok [SV00], no es único, si no que en la literatura se pueden encontrar varias aportaciones en ese sentido. De esta forma en [TKA02] se propone un lenguaje específico orientado al desarrollo de aplicaciones que actúan sobre flujos de datos, denominado *StreamIt*, cuya motivación principal es implementar optimizaciones para las aplicaciones pipeline. La justificación para crear este lenguaje se basa en que los lenguajes de programación clásicos no tienen una forma natural de tratar el procesamiento de flujo de datos, motivo por el que se requiere la realización de optimizaciones adicionales en las aplicaciones a posteriori. Siguiendo esta corriente el autor Wei Du realiza aportaciones en éste sentido, otorgando al propio compilador la capacidad de determinar la granularidad adecuada en las tareas [DFA03][DA05].

Existen también aportaciones desde el punto de vista del uso de *frameworks* que tengan en cuenta las características de la ejecución de las aplicaciones pipeline. E. Cesar en [CSL05] presenta el entorno POETRIES, que a partir de aplicaciones pipeline de una línea de ejecución, define el número de replicaciones a realizar de forma dinámica para cada tarea, con el objetivo de mejorar la productividad a alcanzar. De forma parecida M. Nijhuis en [NBB06], presenta un entorno basado para aplicaciones multimedia que explota las capacidades del procesamiento paralelo temporal y espacial del flujo de datos de entrada.

El uso creciente de las aplicaciones pipeline, hace que cada vez más se tengan presente para su ejecución en las plataformas multi-cpu y multi-core. Los trabajos presentados en [LDWL06] y [KSJ05] son algunos ejemplos de ello. En el primero se trata el problema del particionamiento de los datos y de la granularidad de las tareas,

mientras que en el segundo se estudia como afecta en el rendimiento la organización de los procesadores.

Para finalizar este apartado comentar como la investigación realizada en otras áreas de la ciencia, que utilizan la supercomputación como herramienta, se plantean la utilización de aplicaciones pipeline como una forma de alcanzar el rendimiento que de otra forma no es posible, así podemos encontrar aportaciones en el campo de la medicina [ZFE02], en el estudio de la secuencia del DNA [CB04], en el análisis de resultados en las ciencias de la tierra [YMW04] o en el ámbito de *Data-Mining* [ZCA06], por poner algunos ejemplos.

1.3. Objetivos del trabajo

El tratamiento de las aplicaciones pipeline ha evolucionado con el tiempo, desde la definición de los objetivos de rendimiento, hasta el desarrollo de compiladores y lenguajes específicos para su creación. Esta evolución hay que enmarcarla en la necesidad, cada vez más importante, de tratar problemas que se basan en el procesamiento de un flujo de datos, característica que diferencia a estas aplicaciones de las que podemos encontrar en la computación paralela clásica.

Como se ha presentado anteriormente, en la literatura es posible encontrar soluciones, en el desarrollo y posterior ejecución, pero por lo general acotados a aplicaciones pipeline de ámbitos concretos o para plataformas de ejecución determinadas. Hoy en día, en la sociedad de la información, el procesamiento de flujos de datos es cada vez más habitual siendo los clusters de computadores, plataformas de procesamiento paralelo de uso cotidiano. Por este motivo el presente trabajo se enmarca en la optimización de la ejecución de aplicaciones pipeline en plataformas distribuidas tipo cluster.

En los siguientes capítulos, partiendo de la información correspondiente al grafo de dependencias de tareas de la aplicación, el tipo y características del flujo de datos a tratar y las capacidad de las tareas para ser paralelizadas o no, se trata el problema de la optimización de las aplicaciones pipeline. Esta optimización se plantea desde dos perspectivas complementarias entre sí:

1. Determinar la estructura de dependencias de tareas más adecuada para una aplicación pipeline tomando como base unos criterios de rendimiento. En este apartado, se crearán técnicas, que partiendo del comportamiento de la aplicación, identificarán las tareas que son el cuello de botella que impide alcanzar el ren-

dimiento deseado y ofrecen como solución el grafo de tareas que representa la estructura idónea para esta aplicación. Las técnicas desarrolladas permitirán:

   a. Aplicar paralelismo de datos, sobre aquellas tareas cuya funcionalidad lo permita, para disminuir su granularidad.
   b. Obtener un mayor ratio de datos procesados por unidad de tiempo a partir de la creación de nuevas líneas de ejecución que permitan su procesamiento de forma concurrente.

2. Realizar el proceso de mapping de las tareas de la aplicación pipeline, obteniendo la asignación más adecuada de tareas a procesadores y que optimizando su utilización permitan:

   a. Alcanzar una productividad prefijada.
   b. Obtener la mínima latencia bajo un requisito de productividad.
   c. Minimizar el número de nodos de procesamiento necesarios.

Estos dos objetivos se complementan entre sí ofreciendo una solución completa que permite afrontar el desarrollo de una aplicación pipeline, desde la definición del grafo de tareas, que identifica la estructura más adecuada para la aplicación pipeline para obtener el rendimiento deseado, hasta la asignación de sus tareas sobre la plataforma de ejecución optimizando el uso de recursos.

Capítulo 2

Análisis de las Aplicaciones Pipeline

Este capítulo profundiza en el estudio de las características de las aplicaciones pipeline con el fin de llegar al conocimiento necesario para abordar el desarrollo de estrategias para su optimiazción. Para ello inicialmente, se caracterizará el modelo utilizado para su representación y a continuación, tomando como base dicho modelo, se analiza con detalle su comportamiento.

El objetivo principal del conocimiento adquirido en este estudio, es el de poder determinar que factores son los que influyen en el rendimiento de estas aplicaciones, y así desarrollar heurísticas que exploten dicha información tanto desde el punto de vista de la definición más adecuada para el grafo de tareas de la aplicación como de la forma en que se debe llevar a cabo la asignación de estas tareas a los nodos de procesamiento.

2.1. Caracterización del modelo para las aplicaciones pipeline

Las aplicaciones pipeline están formadas por una secuencia de tareas $\{T_0,T_1,..,T_n\}$, en la que cada una de ellas recibe datos de sus tareas predecesoras, los procesa y envía el resultado a sus tareas sucesoras. La tarea inicial es la que se encarga de capturar el conjunto de datos del flujo de entrada, mientras que la tarea final ofrece el resultado del procesamiento de éstos. Las dependencias estrictas de la relación existente entre las tareas, derivadas de este comportamiento, se representarán mediante un grafo TPG, cuyo modelo ha sido presentado en el apartado 1.1.

El objetivo de este trabajo se enmarca en la optimización de los parámetros de rendimiento de las aplicaciones pipeline: la latencia que indica el tiempo necesario para poder procesar de forma completa un dato individual, y la productividad, que representa el número de datos procesados por unidad de tiempo. Ambos parámetros

poseen en común el uso del tiempo como factor de evaluación, por lo que se ha escogido éste como métrica en su representación, homogeneizando de esta forma la información presente en el grafo. Así el valor asignado a cada nodo, T_i, determina el tiempo de cómputo, $\mu(T_i)$, necesario para que la tarea a la que representa ejecute la función asociada sobre un único dato de entrada. Y el valor asociado a un arco de la tarea T_i a la tarea T_j, representa el tiempo requerido para transmitir la información generada en esa comunicación, $com(T_i, T_j)$. Para que esta representación sea válida, se supone que las aplicaciones poseen un comportamiento estable y los valores que se asignan son representativos para cualquier ejecución de la aplicación, no presentando variaciones significativas en el procesamiento individual de cada dato del flujo de entrada.

La métrica escogida posee como particularidad el ser dependiente de la plataforma sobre la que se realiza la ejecución de la aplicación, y la obtención de los valores de cómputo y de comunicación es representativa de esta situación concreta. Sin animo de restringir la generalidad de este trabajo y tomando como base el comportamiento estable de la aplicación, se da por supuesto que las modificaciones en la plataforma, siempre que se den de forma homogénea, afectan en la misma medida a todas las tareas y comunicaciones de la aplicación. De esta forma una tarea con mayor tiempo de cómputo respecto a otra, mantendrá esta característica si los procesadores sobre los que se evalúa la ejecución de ambas tareas, cambian en igual medida, y en el mismo sentido para las comunicaciones, siendo más costosas aquellas que transmiten un mayor volumen de datos.

Las plataformas de ejecución en las que se centra este trabajo son las arquitecturas distribuidas, y en concreto los clusters de computadores. Así en la literatura se presentan diversas técnicas que permiten caracterizar los valores para el tiempo de cómputo y de las comunicaciones, comentándose a continuación las más comunes.

Para de la caracterización de los tiempos de cómputo se presentan tres opciones:

1. Utilizar herramientas de *profiling* o técnicas de *tracing* en la ejecución sobre una plataforma real, y más tarde aplicar factores de corrección para encuadrar los resultados sobre cualquier otra plataforma de estudio. Esta opción requiere que la aplicación haya sido previamente programada y ejecutada, así como ponderar correctamente el factor de corrección necesario [LCM+05].

2. Mediante *estimaciones* del tiempo cómputo, basadas en la caracterización de las librerías de funciones usadas en el desarrollo de las tareas. Estas estimaciones pueden formar parte de una base de datos que permita, mediante parámetros

específicos de la plataforma a estudio, ajustar el valor asignado a cada tarea [HMM05].

3. Desarrollando y evaluando una *expresión analítica* que sea capaz de representar el tiempo de cómputo de las tareas, ajustándola a los parámetros específicos que caractericen la arquitectura. Esta posibilidad es compleja ya que requiere de un exhaustivo estudio previo de las funciones a implementar que en la mayoría de las veces no es trivial, así como tener presente un gran número de factores de la propia arquitectura [RBL04].

En el caso de las comunicaciones, se ha de tener presente que el punto de partida en su estudio corresponde por lo general al volumen de datos que son transferidos entre las tareas involucradas, este valor se obtiene en la fase de modelización de la aplicación. Para adaptarlo a la nueva métrica, existe la posibilidad de realizar un análisis de la ejecución de la aplicación, capturando los tiempos involucrados en las comunicaciones o bien mediante la utilización de un modelo que tenga la capacidad de representar el sistema de interconexión utilizado en la arquitectura. En la literatura existen múltiples modelos con esta capacidad, y que poseen una mayor o menor complejidad en función del número de parámetros que utilizan, como pueden ser el ancho de banda de la red, el número de enlaces presentes, el mecanismo de encaminamiento entre nodos, etc. Por este motivo el modelo escogido ha de balancear el nivel de abstracción utilizado al representar los diferentes parámetros con la precisión que se desea obtener.

Algunos de los modelos que se pueden encontrar en la literatura son los siguientes:

- *LoGPC* [MF01]. Este modelo es una mejora de los modelos previos LogP [CKP+93] y LogGP [AISS97]. El modelo se define en tres capas, en las que de forma incremental se añaden más detalles que caracterizan a la arquitectura y a la aplicación en estudio. En la capa superior se representa la información referente a los mecanismos de comunicación utilizados por la aplicación. En la segunda capa, se añaden detalles sobre la arquitectura y sus capacidades. Y finalmente en la tercera capa se modeliza los retardos producidos por las contenciones de la red.

- *P-3PC* [Sei02]. Este modelo está orientado a las comunicaciones bloqueantes MPI (MPI_Send y MPI_Recv), tratándolas a un nivel en el que no se tiene en cuenta la forma en que han sido implementadas las primitivas, siendo de esta forma independiente de las diferentes implementaciones del estándar MPI. En el modelo se tiene en cuenta la sobrecarga de las propias primitivas en el emisor

y receptor, el coste propio de la transmisión, así como las diferencias existentes entre la transmisión de información ubicada de forma contigua o fragmentada en el espacio de memoria. El modelo propuesto se aplica a partir del uso de una biblioteca que caracteriza de forma precisa las funciones utilizadas en la aplicación a estudio.

- log_nP [CGS07]. Este modelo al igual que el modelo LoGPC, es una evolución del modelo LogP. En éste las comunicaciones se diferencian en dos ámbitos, *comunicaciones implícitas* que hacen referencia a todas las acciones que involucran a las comunicaciones, como puede ser el acceso a la información teniendo en cuenta la jerarquía de memoria, etc., y *comunicaciones explícitas* que corresponden al "middleware" utilizado y que es el que determina la implementación final de las primitivas de comunicación. La complejidad de este modelo depende del nivel de detalle que se desee obtener al ser aplicado, siendo este nivel de detalle el parámetro n incluido en su nombre. Una implementación que ofrece una buena relación entre complejidad y precisión es el denominado log_3P, en la que se tienen presente tres factores: la sobrecarga propia de la implementación de las primitivas de comunicación para el emisor y el receptor, la latencia en el acceso a la información a transmitir según se encuentre o no fragmentada en memoria y finalmente la propia red de interconexión. Este modelo posee una mayor complejidad que el modelo LoGPC aunque ofrece unas mejores estimaciones.

En todos los modelos presentados, previamente a su utilización se requiere de un proceso de "benchmarking" de la arquitectura que obtenga la información necesaria para determinar los parámetros utilizados en su caracterización.

Con la información obtenida para los valores asociados al tiempo de cómputo de las tareas y al tiempo necesario para realizar las comunicaciones se procede a crear el grafo TPG que será utilizado en las siguientes secciones en el estudio del comportamiento de las aplicaciones pipeline.

2.2. Comportamiento de las aplicaciones pipeline

Para abordar cualquier objetivo relacionado con el rendimiento de las aplicaciones pipeline, es necesario conocer cuales son los factores que rigen su comportamiento. Las bases de dicho comportamiento se basan principalmente en las siguientes características:

- Para un dato concreto las precedencias entre las tareas son *estrictas*. Esto hace que una tarea no puede iniciar el cómputo sobre el dato actual, hasta que todas sus tareas predecesoras hayan finalizado y transmitido sus resultados del procesamiento para ese mismo dato.

- La aplicación, por su naturaleza iterativa, hace que las tareas *no finalicen su ejecución tras procesar y transmitir los resultados del dato actual*, si no que por el contrario y mientras existan datos provenientes del flujo de entrada, éstas se mantienen activas.

- *El nivel de concurrencia entre tareas adyacentes no depende directamente de las precedencias* que existen entre ellas ya que éstas pueden estar procesando datos diferentes del flujo de entrada y en consecuencia pueden estar ejecutándose simultáneamente.

La estructura que podemos encontrar en las aplicaciones pipeline puede ser arbitraria en función del número de tareas predecesoras y sucesoras existentes para una tarea concreta. Por este motivo el estudio del modelo de comportamiento que se realiza en los siguientes apartados se enfocará en dos situaciones. La primera, más simple, en el que se supondrá una única línea de ejecución, en la que cada tarea de la aplicación posee una única tarea predecesora y sucesora. Este primer caso de estudio dará las bases para la segunda situación más genérica, en el que el número de tareas predecesoras y sucesoras de una tarea puede ser arbitrario, y en consecuencia dar lugar a más de una línea de ejecución desde la primera a la última tarea de la aplicación.

2.2.1. Aplicaciones pipeline de una línea

Las aplicaciones pipeline de una línea, son el caso más simple que se puede plantear en éste tipo de aplicaciones. En ellas, las tareas aparecen organizadas formando una cadena simple, en la que cada tarea posee una única predecesora y una única sucesora, a excepción de la tarea inicial y final. En ésta estructura existe un único camino desde la primera tarea hasta la última por el que se mueven los datos del flujo de entrada.

La Figura 2.1(a) muestra como ejemplo el grafo de tareas de una aplicación pipeline con una línea de ejecución formada por seis tareas, con los respectivos tiempos de cómputo y comunicación representados en el grafo.

La Figura 2.1(b) muestra el diagrama de Gantt de la ejecución de la aplicación teniendo en cuenta las precedencias de las tareas, al procesar los primeros cuatro datos

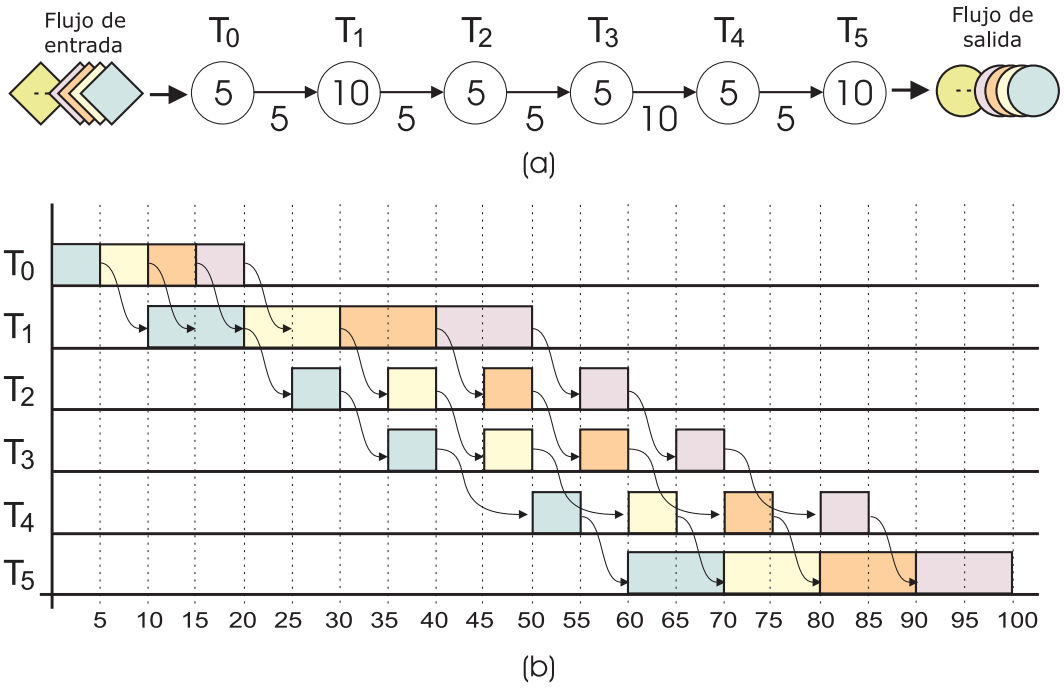


Figura 2.1: (a) Ejemplo de grafo de tareas para una aplicación pipeline de una línea. (b) Diagrama de Gantt de su ejecución atendiendo a las dependencias entre tareas.

del flujo de entrada. Suponemos que los datos de entrada llegan de forma continua y sin esperas por parte de la tarea inicial T_0, y que las comunicaciones pueden solaparse con el cómputo de las tareas.

Del diagrama de Gantt obtenido se extraen las siguientes situaciones:

- En la pareja de tareas T_0 y T_1, la tarea receptora T_1, es incapaz de seguir el ritmo de procesamiento de la tarea T_0. Esta situación se debe a la diferencia de cómputo existente entre ellas, ya que la tarea T_0 posee un tiempo de cómputo menor y en consecuencia genera sus resultados a un ritmo superior al que la tarea T_1 es capaz de asumir.

- En el caso de las tareas T_1 y T_2, la situación es la inversa a la anterior. En este caso la tarea T_2, con un menor tiempo de cómputo, se encuentra con fases de inactividad debidos a que la tarea T_1 no es capaz de generar resultados con el ritmo que T_2 puede asumir.

- En el caso de las tareas T_3 y T_4, como se puede observar, poseen periodos de inactividad, que vienen propagados desde la tarea T_2. Hay que destacar que aunque bien es cierto que la comunicación asociada al par de tareas T_3 y T_4 es mayor que

la proveniente desde T_2, ésta no tiene incidencia sobre duración de las fases de inactividad, excepto en su propagación.

- Para finalizar, en el par de tareas T_4 y T_5, se observa como ésta última no presenta inactividades como se da en su predecesora, manteniéndose activa de forma continua. Esto se debe a que su mayor tiempo de cómputo hace que mientras la tarea T_4 se encuentra inactiva en espera del próximo dato, ella por el contrario se encuentra procesando el dato previo.

Las principales conclusiones que se pueden extraer de este caso es que las diferencias de cómputo pueden provocar que las tareas adyacentes de menor cómputo sufran fases de inactividad. Además estas inactividades se propagan en la línea de ejecución siempre que el cómputo de las tareas en dicha línea sea inferior al cómputo de la tarea que las generó.

2.2.2. Aplicaciones pipeline de más de una línea

Las aplicaciones pipeline de más de una línea son el caso más genérico que se puede encontrar y engloba a cualquier tipo de aplicación pipeline. Éstas tienen como característica, que el grafo de tareas que las representa posee múltiples caminos que permiten llegar desde la tarea inicial a la tarea final de la aplicación, debido a que existen tareas con múltiples tareas sucesoras o predecesoras. En el caso de que una tarea posea más de una sucesora, se denominará como *tarea divergente*, siendo el *grado de divergencia* el número de tareas sucesoras que posee. El resultado que genera una tarea divergente es requerido por cada una de sus tareas sucesoras, que no pueden iniciar su procesamiento hasta que dicho resultado sea recibido. Cada una de las tareas sucesoras abre una nueva línea de ejecución dentro del grafo y todas ellas pueden avanzar de forma concurrente entre sí.

Cuando por el contrario una tarea posee más de una predecesora, ésta se denomina *tarea convergente*, y el número de tareas que posee como predecesoras corresponde a su *grado de convergencia*. Una tarea convergente, antes de iniciar una nueva iteración deberá recibir de cada una de sus predecesoras su resultado del dato a procesar. Dichos resultados pueden no llegar al mismo tiempo, debido a la posibilidad de que exista una estructura arbitraria, tanto para los cómputos como para las comunicaciones en las diferentes líneas de ejecución. Finalmente puede darse el caso de que haya tareas que sean divergentes y convergentes al mismo tiempo.

La Figura 2.2(a) muestra un ejemplo de aplicación pipeline con dos líneas de ejecución, $\{T_0, T_1, T_3, T_5, T_6\}$ y $\{T_0, T_2, T_4, T_5, T_6\}$. Para facilitar el estudio de dicho ejemplo se ha asignado a todas las comunicaciones el mismo valor para incidir en el comportamiento de las tareas. En el ejemplo, la tarea divergente T_0 posee dos tareas sucesoras, T_1 y T_2, por lo que su grado de divergencia es 2. La tarea T_5, por el contrario posee dos predecesoras, T_3 y T_4, siendo su grado de convergencia igual a 2. Como se observa en el diagrama de Gantt de la Figura 2.2(b), el cual muestra el comportamineto de la aplicación exclusivamente a partir de las dependencias entre las tareas, los resultados que genera T_0, son enviados a sus sucesoras, que inician de forma concurrente su procesamiento. Finalmente las líneas de ejecución que se han abierto, convergen sobre la tarea T_5, la cual permanece en espera de los resultados de cada una de sus predecesoras para poder iniciar su cómputo.

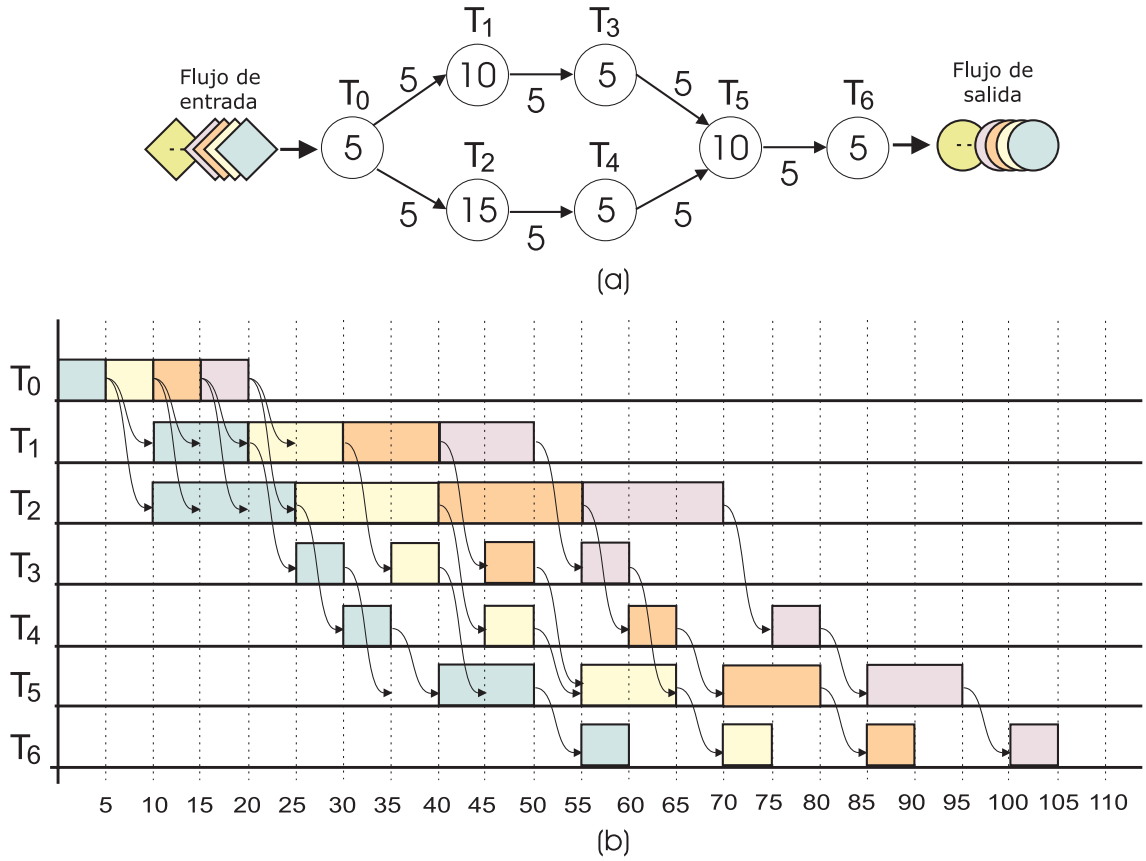


Figura 2.2: (a) Ejemplo de grafo de tareas para una aplicación pipeline de más de una línea. (b) Diagrama de Gantt de su ejecución atendiendo a las dependencias entre tareas.

El hecho de que se puedan dar varias líneas de ejecución de forma simultanea, hace

que la evaluación del comportamiento global de la aplicación sea más complejo que el anterior, debido a las múltiples dependencias que se pueden dar entre las tareas. Aún así se pueden definir unas características básicas para este tipo de aplicaciones que facilite este análisis, y que se pueden observar analizando el diagrama de Gantt de la Figura 2.2(b).

- Las tareas T_1 y T_2 dependen directamente de la tarea T_0, por lo que en la primera iteración han de esperar a tener el resultado proveniente de ésta para poder iniciarla. A partir de aquí procesan de forma ininterrumpida ya que su cómputo es superior al de la tarea T_0 y poseen la información con antelación al inicio de cada nueva iteración.

- Una vez iniciadas las tareas T_1 y T_2, las líneas de ejecución que inician, que incluyen a las tareas T_3 y T_4 respectivamente, actúan de forma independiente. Cada una de estas líneas propaga un periodo de inactividad propio debido a las diferencias de cómputo que hay entre las tareas de cada una de ellas. Así en el caso de la tarea T_3 esta inactividad corresponde a 5 unidades y en el caso de T_4 es de 10 unidades.

- Sobre la tarea T_5 convergen las dos líneas de ejecución, y esta tarea se ve afectada por una inactividad que viene propagada desde dichas líneas de ejecución. En cada una de ellas, existe una inactividad propagada, pero solo la que proviene desde la tarea T_2 es la que realmente le afecta, ya que es la mayor de ellas. El periodo de inactividad que finalmente afecta a la tarea T_5, viene dado por su diferencia de cómputo con T_2.

  Se puede observar que realmente la línea de ejecución formada por las tareas $\{T_0, T_2, T_4, T_5, T_6\}$ es la que está generando inactividades que se propagan hasta la tarea final T_6, mientras que las tareas T_1 y T_3 no afectan en este sentido al estar procesando de forma concurrente con ellas.

2.2.3. Conclusiones en el comportamiento de las aplicaciones pipeline

Una vez analizado el comportamiento en función de la estructura de las aplicaciones pipeline se pueden extraer las siguientes conclusiones:

1. El mínimo grado de convergencia y de divergencia que pueden tener las tareas de una aplicación pipeline es de uno, a excepción de la tarea inicial que tendrá grado de convergencia cero y la tarea final que tendrá grado de divergencia cero.

2. Sea una secuencia de tareas $\{T_i,T_{i+1},..,T_{i+n}\}$, cada una de ellas con grado de divergencia y de convergencia con valor uno. Y sea el tiempo de cómputo evaluado para cada una de ellas $\mu(T_i)$, $\mu(T_{i+1})$, ..., $\mu(T_{i+n})$. Suponiendo que la tarea T_i es aquella tarea que posee el mayor de todos los valores de cómputo en la secuencia definida, entonces cada tarea T_j de la secuencia tendrá un tiempo de inactividad de valor $\mu(T_i) - \mu(T_j)$, entre cada nueva iteración.

3. Suponiendo la misma secuencia de tareas anterior. Si existe una tarea T_k, posterior a esta secuencia, en la línea de procesamiento, cuyo cómputo $\mu(T_k)$ es mayor o igual a $\mu(T_i)$, la tarea T_k, no se verá afectada por inactividades entre cada iteración.

4. Cada línea de ejecución, en su análisis individual, puede ser considerada de forma independiente como un subgrafo, tomando como tarea inicial del mismo la tarea divergente que la inicia, y como tarea final la tarea convergente sobre la que confluye.

5. Las tareas *divergentes* trasladan sus fases de inactividad a cada una de las nuevas líneas de ejecución que abren.

6. Las tareas *convergentes* son afectadas por la mayor de las fases de inactividad que se propagan por cada una de las líneas de ejecución en las que aparece, solo si su cómputo es menor o igual a dicha inactividad.

2.3. Análisis de los parámetros de rendimiento

Los principales parámetros de rendimiento de las aplicaciones pipeline son dos; la latencia y la productividad. El valor que se puede obtener de ambos depende del comportamiento de la aplicación y en consecuencia de la estructura de dependencias de sus tareas. Por éste motivo a partir del conocimiento obtenido de la sección previa se evaluará los factores que los determinan.

A continuación se estudian cada uno de ellos con más detalle.

2.3.1. Latencia

La latencia, indica el tiempo necesario para procesar de forma completa un dato del flujo de entrada. Para el cálculo de la latencia se deberá tener en cuenta la estructura de la aplicación, según tenga una o varias líneas de ejecución.

1. *Aplicaciones con una línea de ejecución*

   La latencia corresponde a la suma de los tiempos de cómputo y de comunicación de las tareas de la aplicación.

$$latencia = \sum_{T_i \in Aplicación} \mu(T_i) + \sum_{T_i, T_j \in Aplicación} com(T_i, T_j) \tag{2.1}$$

   La optimización de la latencia pasa por obtener el mínimo valor para la expresión (2.1). Desde el punto de vista de la aplicación esto equivale a reducir alguno de los dos factores que intervienen, o ambos si esto es posible. El primero de ellos, el tiempo de cómputo, es un valor estático que hemos supuesto estable. Por el contrario el factor correspondiente a las comunicaciones entre tareas adyacentes, puede ser eliminado entre algunas de ellas, si éstas agrupan su cómputo formando una única tarea, acción que repercute en un aumento de la granularidad de las tareas, o se asignan al mismo nodo de procesamiento en la fase de mapping. El caso extremo en la reducción de las comunicaciones sería el correspondiente a eliminar todas ellas haciendo que la aplicación estuviese formada por una única tarea que englobase todo el cómputo, o que la aplicación completa se asigne a un único nodo de procesamiento. Ambas opciones eliminan la posibilidad de procesar múltiples datos del flujo de entrada al eliminarse la estructura pipeline.

   Tomando como ejemplo la Figura 2.1(a), y aplicando la ecuación (2.1), el valor obtenido para la latencia es de 70 unidades de tiempo. Agrupando todas las tareas en una única, este valor quedaría reducido a 40 unidades de tiempo.

2. *Aplicaciones con varias líneas de ejecución*

   En este ámbito, para determinar la latencia hay que evaluar cada una de las líneas de ejecución de forma individual, siendo el valor final de la latencia el valor máximo obtenido para cada una de ellas, como se refleja en la expresión (2.2), donde m es el conjunto de líneas de ejecución diferentes y C_k es una en particular de éstas. Esta expresión incluye a la expresión (2.1) para una única línea de ejecución, siendo de esta forma válida para cualquier situación.

$$latencia = \max_{1 \leq k \leq m} \left(\sum_{T_i \in C_k} \mu(T_i) + \sum_{T_i, T_j \in C_k} com(T_i, T_j) \right) \tag{2.2}$$

Usando como ejemplo el caso de la Figura 2.2(a), las líneas de ejecución que se identifican son dos, $\{T_0, T_1, T_3, T_5, T_6\}$ y $\{T_0, T_2, T_4, T_5, T_6\}$, con latencias de 55 y 60 unidades respectivamente. A la hora de intentar reducir la latencia se puede optar por agrupar tareas adyacentes de las líneas de ejecución eliminando sus comunicaciones. Hay que tener presente en este caso que las agrupaciones creadas en una línea de ejecución pueden afectar a la latencia de las restantes si comparten alguna de las tareas involucradas en dichas agrupaciones. Este sería el caso de una posible agrupación de la tarea T_0 con alguna de sus sucesoras, T_1 o T_2. Si se escogiera realizar la agrupación (T_0, T_1), esta añadiría el cómputo de la tarea sucesora T_1 sobre la línea en la que inicialmente no actúa, incrementando así su latencia.

2.3.2. Productividad

La productividad determina el ratio de datos procesados por unidad de tiempo del flujo de datos de entrada y puede ser expresado por su valor inverso, denominado *Iteration Period* (IP) que indica el intervalo de tiempo transcurrido en obtener el resultado de procesar dos datos consecutivos.

$$Productividad = \frac{1}{IP}$$

En términos ideales, el valor de IP de una aplicación, corresponde al mayor tiempo de cómputo de las tareas de la aplicación, determinando ésta la productividad máxima alcanzable [CNNS94][SV00][LLP01]. Esta relación se puede observar en la Figura 2.2(b), en la que el valor del IP que se obtiene corresponde al valor del cómputo de la tarea mayor, T_2, dando una productividad de un dato procesado cada 15 unidades de tiempo.

Partiendo de la premisa de que las comunicaciones pueden solaparse entre sí, esta aproximación es correcta para cualquier tipo de aplicación pipeline, independientemente de su estructura, como se demostrará a continuación.

Dada una tarea T_i de la aplicación, el valor de su $IP(T_i)$, corresponde al intervalo de tiempo necesario por esa tarea para finalizar el procesamiento de dos datos consecutivos,

es decir:

$$IP(T_i) = f(T_i^n) - f(T_i^{n-1}) \tag{2.3}$$

donde $f(T_i^n)$ y $f(T_i^{n-1})$ son los tiempos de finalización de la tarea T_i para los datos de entrada n y $n-1$ respectivamente.

El momento de finalización del procesamiento del dato n, $f(T_i^n)$, viene dado por la suma del momento de inicio en el procesamiento de éste, $i(T_i^n)$, y el cómputo asociado a la tarea $\mu(T_i)$.

$$f(T_i^n) = i(T_i^n) + \mu(T_i)$$

Para que la tarea T_i pueda iniciar su cómputo se requiere que se cumplan las dos condiciones siguientes:

1. La tarea T_i debe haber finalizado el procesamiento del dato previo $n-1$ para así poder iniciar la iteración correspondiente al procesamiento del dato n. En este caso se tiene que:

   $$i(T_i^n) = f(T_i^{n-1})$$

   Esta situación puede observarse en el diagrama de Gantt de la Figura 2.2(b), en el que la tarea T_1, aún habiendo recibido los datos necesarios de su tarea predecesora T_0, no puede iniciar la nueva iteración hasta que no ha finalizado el procesamiento del dato que lleva a cabo en ese momemto.

2. Todas las tareas predecesoras de T_i deben haber finalizado el procesamiento correspondiente al dato n y todos los mensajes provenientes de éstas deben haber llegado. En este caso el momento de inicio se puede expresar como:

   $$i(T_i^n) = \max_{T_j \in pred(T_i)} (f(T_j^n) + com(T_j, T_i))$$

   Donde $pred(T_i)$ es el conjunto de tareas predecesoras de T_i. De esta forma el tiempo de inicio viene dado por el mayor de los tiempos de llegada de los mensajes en espera. Denotamos como T_p la tarea que cumple la condición de valor máximo, quedando la expresión de la forma:

   $$i(T_i^n) = f(T_p^n) + com(T_p, T_i)$$

   De nuevo tomando como ejemplo la Figura 2.2(b), la tarea T_5 ilustra esta situación puesto que debe esperar a que sus tareas predecesoras, T_3 y T_4, hayan finalizado

y sus mensajes hayan sido recibidos. En particular es la tarea T_4 la que marca el momento de inicio de la nueva iteración identificándose ésta como la tarea T_p de la expresión anterior para este ejemplo.

Teniendo en cuenta las dos situaciones anteriores, el momento de inicio, $i(T_i^n)$, corresponderá al mayor de los valores anteriores.

$$i(T_i^n) = \text{máx}[f(T_i^{n-1}), f(T_p^n) + com(T_p, T_i)] \tag{2.4}$$

Considerando de forma individual cada una de las dos componentes de la expresión (2.4), tenemos dos posibles casos para el cálculo de $IP(T_i)$ de la expresión (2.3):

1. Si el valor máximo es el primer parámetro, $f(T_i^{n-1})$, la expresión queda de la forma:

$$\begin{aligned} IP(T_i) &= i(T_i^n) + \mu(T_i) - f(T_i^{n-1}) \\ &= f(T_i^{n-1}) + \mu(T_i) - f(T_i^{n-1}) \\ &= \mu(T_i) \end{aligned}$$

   Esta situación indica que el valor de $IP(T_i)$ es el cómputo de la propia tarea y en consecuencia éste es el mayor en la línea de ejecución.

2. Si el valor máximo corresponde al segundo parámetro, los valores de finalización para dos datos consecutivos, $f(T_i^{n-1})$ y $f(T_i^n)$, vienen dados por el momento de finalización de la tarea predecesora T_p. Hay que tener presente que en nuestro estudio consideramos que las aplicaciones poseen un comportamiento estable para los diferentes datos del flujo de entrada y en consecuencia la tarea T_p siempre es la misma. De esta forma se obtiene que:

$$\begin{aligned} IP(T_i) &= f(T_p^n) + com(T_p, T_i) - f(T_p^{n-1}) - com(T_p, T_i) \\ &= f(T_p^n) - f(T_p^{n-1}) \\ &= IP(T_p) \end{aligned} \tag{2.5}$$

   En donde se puede observar que el valor de $IP(T_i)$ es el de $IP(T_p)$.

   Evaluando de forma recursiva la expresión (2.5) se llegaría a encontrar alguna tarea predecesora T_{p_i} tal que $IP(T_{p_i}) = \mu(T_{p_i})$ que determina la productividad

debido a su cómputo mayor. El caso extremo sería aquel en el que la tarea T_{p_i} corresponda a la tarea inicial.

Finalmente se llega a la conclusión de que el valor mínimo de IP en la aplicación pipeline corresponde exclusivamente al mayor tiempo de cómputo de las tareas que forman parte de ella.

$$\text{mín } IP = \max_{T_i \in Aplicación} \mu(T_i)$$

siendo la máxima productividad el inverso de este valor.

$$\text{máx } Productividad = \frac{1}{\text{mín } IP(Aplicación)}$$

Para el caso del ejemplo de la Figura 2.2, se deduce que el valor mínimo de su IP es $\mu(T_2) = 15$ unidades de tiempo.

Debido a que el mayor tiempo de cómputo de las tareas es el que determina la productividad, la elección de una granularidad adecuada, se convierte en un factor importante a tener en cuenta, siendo mayor la productividad alcanzable cuanto más fina sea la granularidad.

2.3.3. Latencia y productividad de forma conjunta

De lo expuesto en el análisis individual de los parámetros de latencia y productividad, se puede deducir que ambos entre sí poseen características antagónicas. La latencia se beneficia de una granularidad gruesa, que reduzca comunicaciones entre tareas. Esto repercute negativamente sobre el parámetro de productividad, que se beneficia de granularidades finas donde el cómputo de las tareas es menor. Por este motivo intentar optimizar un parámetro incide negativamente sobre el resultado en el otro.

Esta situación hace que se requiera alcanzar un compromiso en la optimización:

- Minimizar la latencia marcando un requisito de productividad.
- Maximizar la productividad intentando obtener la mínima latencia posible.

Alcanzar alguno de estos objetivos en la fase de diseño de la aplicación es complicado por la cantidad de factores que intervienen. En este trabajo, la optimización de las aplicaciones pipeline se afronta partiendo de una estructura para la aplicación ya desarrollada y representada mediante un grafo de tareas TPG. A partir de aquí se evaluará la necesidad o no de modificar dicha estructura para alcanzar el rendimiento deseado

y posteriormente se realizará la asignación de las tareas a los nodos de procesamiento que cumpla con los requisitos de rendimiento definidos.

Capítulo 3

Heurísticas para la optimización de aplicaciones pipeline

En el capítulo anterior se han dado a conocer las bases que definen el comportamiento de las aplicaciones pipeline así como el modelo de representación escogido para ellas y que será utilizado en este trabajo. Además se ha realizado un análisis de los factores que actúan sobre sus parámetros de rendimiento, productividad y latencia.

Basándose en este conocimiento, en el presente capítulo se presentan las heurísticas que se han desarrollado para la optimización de las aplicaciones pipeline, y que se centran principalmente en dos aspectos:

- Dar una estructura adecuada para el grafo de tareas, con la finalidad de que la aplicación pipeline alcance unos objetivos de rendimiento predefinidos desde el punto de vista de la productividad.
- Definir la asignación para las tareas de la aplicación sobre los nodos de procesamiento presentes en la arquitectura, que cumpla unos requisitos de rendimiento preestablecidos.

El escenario de partida para el desarrollo de las heurísticas que se presentan en este tema, se define a partir de las siguientes consideraciones:

- La arquitectura sobre la cual se realiza la ejecución es un cluster de computadores dedicado.
- Se dispone del grafo de tareas de la aplicación basado en el modelo TPG. Dicho grafo habrá sido obtenido mediante alguna de las técnicas comentadas en el capítulo previo. La estructura del grafo TPG puede ser totalmente arbitraria.

- Se conoce la funcionalidad de las tareas de tal forma que pueden ser identificados los siguientes elementos: (a) qué tareas pueden ser paralelizadas y (b) cómo es el flujo de datos de entrada desde el punto de vista de las dependencias o no de sus datos.

- Las aplicaciones pipeline que se consideran poseen una granularidad gruesa que haga que el ratio cómputo/comunicación sea alto, de forma que tenga sentido el uso de plataformas de ejecución distribuidas del tipo cluster.

3.1. Definición de la estructura del grafo de tareas de la aplicación pipeline

Como se ha comentado en el capítulo anterior, el rendimiento que se puede obtener de las aplicaciones pipeline, tanto en latencia como en productividad, está condicionado por la estructura de dependencias entre sus tareas y el tiempo de cómputo de éstas. Esto hace que el programador de la aplicación deba afrontar, además del difícil proceso de la paralelización, la problemática añadida de obtener una estructura adecuada que alcance el rendimiento deseado. Por esta razón la posibilidad de utilizar técnicas que de forma automática permitan ajustar el diseño obtenido, facilitan el desarrollo de estas aplicaciones.

En esta sección se presentan dos técnicas diferentes, que tienen como objetivo localizar y actuar sobre aquellas tareas que son un cuello de botella en el procesamiento del flujo de entrada e impiden alcanzar el rendimiento deseado. La primera de estas técnicas, que denominaremos *Técnica de Paralelización*, aplica mecanismos basados en el paralelismo de datos sobre las tareas identificadas como "problemáticas", redefiniéndolas como un conjunto de subtareas, a las que se les asigna una parte del dato original a procesar, reduciendo de esta forma el tiempo de cómputo de las tareas [GRRL05b]. Esto redunda en beneficio tanto de la productividad como de la latencia.

La segunda técnica, que denominaremos *Técnica de Replicación*, por el contrario, redefine la estructura del grafo de tareas de la aplicación, realizando copias exactas de las tareas cuello de botella, a las que se les asignará de forma cíclica, cada uno de los datos provenientes del flujo de entrada. De esta forma esos datos que inicialmente serían procesados en orden, y de forma secuencial, podrán ser procesados de forma concurrente sobre las tareas replicadas. En este caso el tiempo de computo de procesar cada dato individual no se disminuye, pero al procesarse varios de ellos de forma concurrente se

consigue aumentar el ratio de datos procesados por unidad de tiempo y en consecuencia se aumenta la productividad final [GRR+06].

La Figura 3.1 ilustra estas dos técnicas. En la Figura 3.1(a) se muestra la tarea original T_i con un tiempo de cómputo de 40 unidades y que se encarga de procesar el flujo de datos de entrada. En este caso se obtendría unos valores de latencia en IP de 40 unidades de tiempo respectivamente. Aplicando la Técnica de Paralelización en la Figura 3.1(b), se obtienen cuatro subtareas, $\{T_i', T_i'', T_i''', T_i''''\}$, cada una de ellas con un tiempo de cómputo de 10 unidades de tiempo y que procesan solo una cuarta parte de cada uno de los datos, alcanzando una latencia de 10 y una productividad con IP=10 unidades de tiempo. En el caso de la Figura 3.1(c), se muestra la estructura obtenida al aplicar la Técnica de Replicación, en la que se obtienen cuatro copias exactas de la tarea original, $\{T_i^1, T_i^2, T_i^3, T_i^4\}$, que procesarán cada una de ellas un dato completo del flujo de entrada. Esta ejecución mantendrá una latencia de 40 unidads de tiempo como la original, pero permite incrementar la productividad ya que se tiene 4 datos procesados cada 40 unidades de tiempo (equivalente a IP=10).

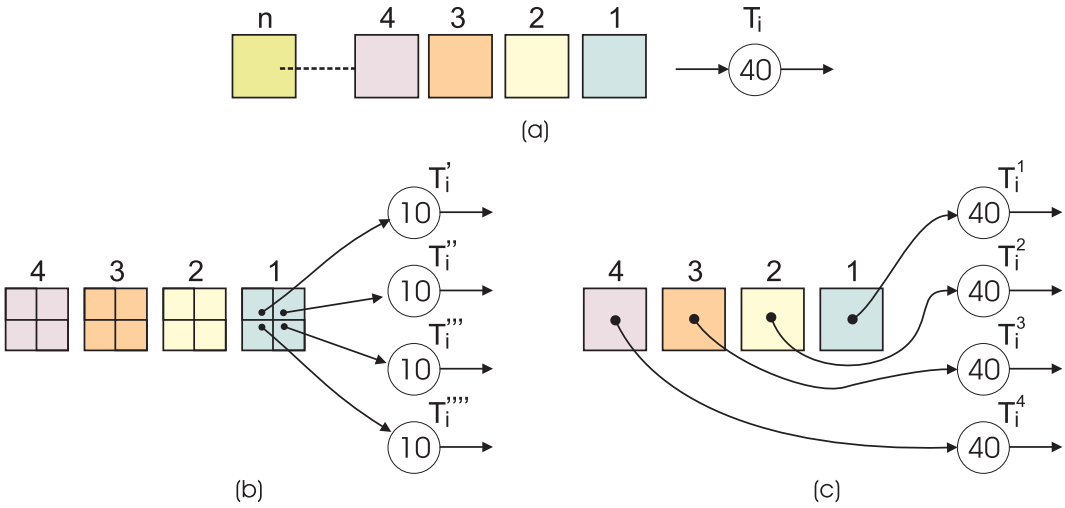


Figura 3.1: Ejemplo de paralelización y replicación de tareas: (a) situación original, (b) tarea paralelizada, (c) tarea replicada.

Cada técnica posee sus ventajas e inconvenientes, que hacen que sea factible su aplicación en diferentes situaciones.

En el caso de tener que procesar un flujo de datos dependendiente, como puede ser el caso de una secuencia de vídeo MPEG2, la Técnica de Replicación plantea el problema de como mantener dichas dependencias, ya que es importante que todos los datos sean procesados por las mismas tareas para mantener la información compartida entre

los datos que son consecutivos. Por este motivo, esta técnica solo podrá ser aplicada en el caso de procesar flujos de datos independientes. Por el contrario la Técnica de Paralelización, puede ser utilizada en cualquier tipo de flujo de datos ya que éstos, aún siendo divididos discurren por las mismas tareas o subtareas pudiendo mantenerse las relaciones de dependencias entre ellos. Por el contrario se ha de tener presente que la funcionalidad asociada a las tareas involucradas en el uso de la Técnica de Paralelización, ha de permitir que pueda ser redefinida mediante el uso del paralelismo de datos, situación que no siempre será posible.

Sea cual sea la técnica a aplicar, ambas se rigen por dos acciones que se deben llevar a cabo:

1. Identificar correctamente aquellas tareas sobre las que se debe aplicar la técnica escogida.

2. Determinar, para estas tareas, cual es el número de subtareas o de copias que se debe de obtener al aplicar la Técnica de Paralelización o de Replicación respectivamente.

Una vez que se han aplicado dichas técnicas la nueva estructura que se obtiene para el grafo de tareas de la aplicación contendrá un conjunto de nuevas tareas, bien como subtareas de la tarea original o como copias de ésta. Esta situación puede dar lugar a que las nuevas dependencias que se crean generen una sobrecarga debido al mayor número de comunicaciones a realizar. La Figura 3.2 ejemplifica esta situación para la Técnica de Replicación.

La Figura 3.2(a) muestra una parte del grafo de tareas de la aplicación pipeline que contiene dos tareas, T_i y T_j, que previamente han sido identificadas para ser replicadas. En el ejemplo suponemos que tras evaluar las características de ambas tareas se ha determinado que el número de veces en las que se debe replicar cada una de ellas es de 2 y 4 veces respectivamente, obteniendo de esta forma los conjuntos de tareas $\{T_i^1, T_i^2\}$ y $\{T_j^1, T_j^2, T_j^3, T_j^4\}$. La estructura de dependencias que se obtiene a partir de estos conjuntos de tareas se muestra en la Figura 3.2(b).

En esta nueva estructura la forma de procesar el flujo de entrada es de la siguiente forma. Las tareas T_i^1 y T_i^2 reciben de forma alternada los datos desde su tarea predecesora, y a su vez reparten el resultado que generan a las tareas sucesoras T_j^1, T_j^2, T_j^3 y T_j^4, también de forma alternada. Como se puede observar, las dependencias existentes entre las tareas en esta nueva estructura, hace que el patrón de comunicaciones para

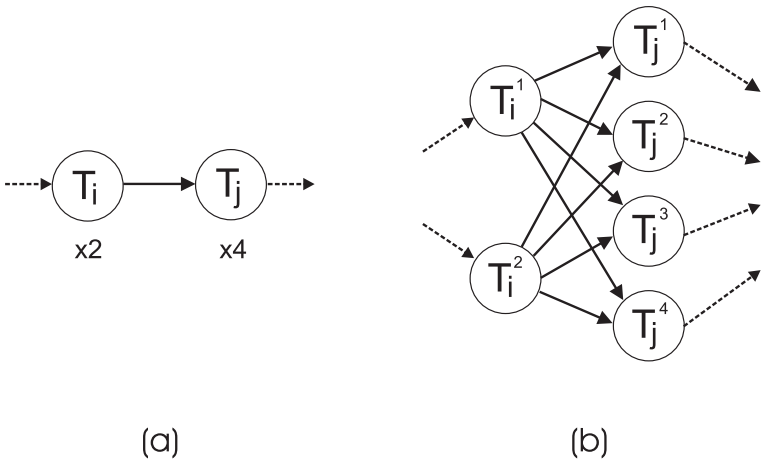


Figura 3.2: (a) Tareas identificadas como cuello de botella en el rendimiento. (b) Estructura de dependencias obtenida al replicar las tareas.

mantener la secuencia en el flujo de datos posea una gran complejidad y genera una gran sobrecarga en el proceso de las comunicaciones.

Este problema se da tanto en la Técnica de Paralelización como en la Técnica de Replicación en las que las tareas que deben ser tratadas son adyacentes comunicándose entre ellas. Por este motivo, en este trabajo se propone la definición de una nueva estructura basada en el concepto de *subgrafo*, cuyo objetivo es simplificar al máximo el patrón de comunicaciones eliminando la sobrecarga ocasionada.

En sí, un subgrafo es una estructura formada por el conjunto de las tareas adyacentes, identificadas como aquellas a las que se les debe aplicar alguna de las técnicas. Una vez obtenido, el subgrafo será utilizado como una única entidad que puede aparecer múltiples veces, repitiendo su propia estructura de dependencias entre las tareas que forman parte de él.

La Figura 3.3 presenta el concepto de subgrafo partiendo del ejemplo previo.

Para definir el subgrafo se parte del hecho de que ambas tareas, T_i y T_j, son adyacentes y en consecuencia poseen una dependencia que debe ser tenida en cuenta en el patrón de comunicaciones de la nueva estructura a definir. Tras evaluar el número de veces en que las tareas deben ser replicadas, 2 y 4 respectivamente, se toma como base que ambas lo deben ser de forma conjunta y no individualmente. Así el subgrafo aparecerá replicado el número de veces determinado como el menor número de replicaciones de las tareas que forman parte de él, que para este ejemplo será 2. Sabiendo esto, el número de copias de las tareas presentes en el subgrafo, se recalculan dividiendo su valor original por el número de veces en que aparecerá el subgrafo, obteniendo finalmente la

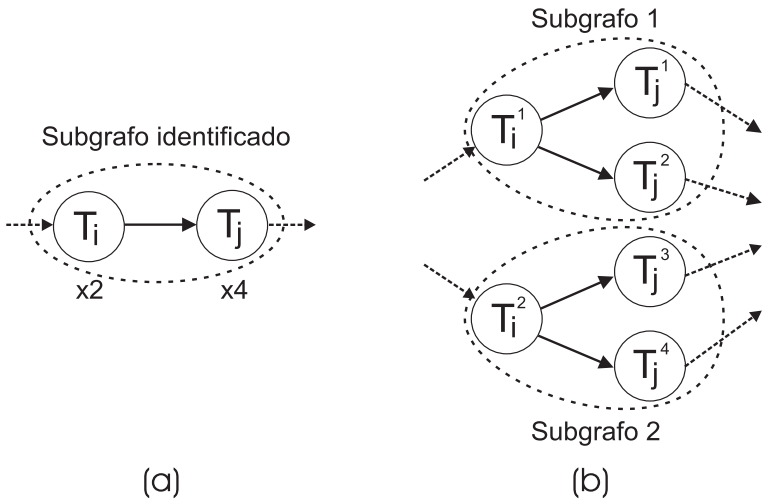


Figura 3.3: (a) Tareas identificadas como subgrafo. (b) Nueva estructura obtenida tras aplicar la replicación basada en subgrafos.

estructura de la Figura 3.3(b). Como puede observase el número de copias de ambas tareas se mantiene respecto al valor definido originalmente, pero esta nueva estructura simplifica el grado de dependencias entre estas copias al quedar las comunicaciones acotadas dentro del propio subgrafo.

A partir del concepto de subgrafo, la segunda acción a realizar por las heurísticas se redefine de la forma: "*Determinar los subgrafos definiendo el número en que aparecerán, así como el número de subtareas o de copias presentes para las tareas que forman parte de ellos.*"

La nueva estructura que se obtiene para el grafo de tareas de la aplicación, deberá trasladarse a ésta modificando y añadiendo el código necesario para que se tenga presente. Esta fase puede ser automatizada tomando como base, el código fuente de la aplicación pipeline y el nuevo grafo de dependencias obtenido.

Un aspecto adicional, enfocado a las comunicaciones, que se ha tenido en cuenta en este trabajo y que añade una diferencia significativa respecto a heurísticas similares presentes en la literatura, es tomar como suposición de partida que el número de enlaces de entrada y salida en los nodos de cómputo de la arquitectura está limitado, de forma que para comenzar una nueva transmisión previamente se debe haber finalizado la actual. Esta característica no se contempla en los trabajos de otros autores que simplifican el problema al dar por supuesto que existe un número ilimitado de estos enlaces permitiendo que múltiples comunicaciones se puedan dar de forma simultanea [CkLW+00][SV00]. Al tener en cuenta esta restricción se conseguirá caracterizar de una

forma más realista el comportamiento de las arquitecturas distribuidas y concretamente los clusters de computadores usados como plataforma en este estudio.

Tomando como premisas, el concepto de subgrafo y la restricción de las comunicaciones, a continuación se expone con detalle cada una de de las heurísticas desarrolladas para implementar las Técnicas de Paralelización y de Replicación.

3.1.1. Técnica de Paralelización

La Técnica de Paralelización pretende identificar aquellas tareas que suponen un cuello de botella en la obtención de una productividad determinada y redefinirlas mediante la aplicación de técnicas de paralelismo de datos. De esta forma el procesamiento de un dato individual se agiliza al dividirse el cómputo inicial en cómputos menores que se dan de forma concurrente [GRRL05b]. La aplicación de esta técnica también permite una reducción de la latencia, aunque no ha sido éste el principal objetivo en su desarrollo.

Para poder usar esta técnica se requiere que previamente las tareas que forman parte de la aplicación, hayan sido clasificadas como paralelizables o no, bien por parte del programador o mediante técnicas automáticas que analicen las librerías utilizadas en su desarrollo. Partiendo de esta información y del grafo de la aplicación se procede a aplicar los siguientes pasos:

Paso 1 - Para aquellas tareas que pueden ser paralelizables, determinar cuales actúan como cuello de botella, definiendo a partir de éstas los subgrafos asociados.

Hay que tener en cuenta que aunque una tarea pueda ser paralelizable, no quiere decir que se deba paralelizar. Esto es debido a que bajo el criterio de obtener un determinado valor rendimiento para la productividad, expresada en nuestro caso como el valor de IP, si el cómputo asociado de estas tarea es inferior a dicho valor, ésta no actúa como cuello de botella.

Paso 2 - Determinar cual es el grado de paralelismo a implementar definiendo el número de subtareas a obtener para cada tarea dentro de los subgrafos identificados, así como el número de veces en el que aparecerán estos subgrafos.

A continuación se desarrolla cada uno de estos pasos:

Paso 1 - Determinar los subgrafos de tareas paralelizables

En este paso, se toma como parámetro de evaluación de las tareas el valor de IP que representa la productividad a alcanzar. Los subgrafos, tal y como se ha comentado estarán formados por aquellas tareas que siendo adyacentes y paralelizables poseen un cómputo que excede al valor de IP deseado.

El pseudo-código del algoritmo que implementa este paso se muestra en la Figura 3.4. El cuerpo principal del algoritmo determina que tareas de las que han sido etiquetadas como paralelizables, son aptas para formar parte de un subgrafo. Para cada una de estas tareas T_i, se llama a la función recursiva $grupo(T_i)$, que obtiene el conjunto de tareas adyacentes a ella que deben agruparse para formar parte del mismo subgrafo.

Una vez que se ha obtenido el subgrafo para una tarea concreta, se comprueba si existe algún otro subgrafo, obtenido previamente, con alguna tarea en común. Si se da esta situación, ambos subgrafos son unidos formando uno único, ya que lo que se pretende es que los subgrafos mantengan todas aquellas tareas que por sus dependencias son adyacentes.

Para ilustrar el funcionamiento del algoritmo se usará el grafo de ejemplo de la Figura 3.5. En él, se supone que las tareas que pueden ser paralelizadas han sido identificadas previamente, siendo éstas T_1, T_2 y T_4. El valor de IP que se desea alcanzar será de 50 unidades. Bajo estas suposiciones en el Cuadro 3.1 se muestra el desarrollo del algoritmo.

El algoritmo recorre el conjunto de tareas paralelizables, $M=\{T_1, T_2, T_4\}$, buscando para cada una de ellas las que, por sus dependencias, son adyacentes. Así al evaluar la tarea T_1 mediante la llamada $grupo(T_1)$, se identifica como subgrafo a la que pertenece el conjunto $\{T_1, T_2\}$. Debido a que la tarea T_4 no es adyacente a otras tareas paralelizables, forma por sí misma un subgrafo.

Paso 2 - Cálculo del número de copias de cada subgrafo y del número de subtareas para las tareas que forman parte de ellos

En este paso se calcula el número de veces en que cada subgrafo definido debe aparecer, así como el número de subtareas de cada tarea dentro de él.

A la hora de determinar el número de subtareas, se debe tener presente la sobrecarga producida por el incremento en el número de comunicaciones. Presuponemos, como se ha comentado previamente, que las comunicaciones salientes desde un mismo nodo de

```
1  Conjunto_subgrafos = ∅
2  M = ∀ T_i ∈ Aplicación identificada como tarea paralelizable y μ(T_i) > IP
3  Para cada T_i ∈ M
4   SG = grupo(T_i)
5   Para cada H ∈ Conjunto_subgrafos
6      Si H ⋂ SG ≠ ∅ entonces
7         SG = SG ⋃ H
8         Conjunto_subgrafos = Conjunto_subgrafos − {H}
9      fin_si
10  fin_para
11  Conjunto_subgrafos = Conjunto_subgrafos ⋃ SG
12 fin_para
```

```
1  Función grupo(T_i)
2   tareas_adyacentes ={T_i}
3   M = M − T_i
4   Para cada tarea T_j sucesora de T_i y T_j ∈ M
5      tareas_adyacentes = tareas_adyacentes ⋃ grupo(T_j)
6   fin_para
7   retorna(tareas_adyacentes)
8  fin_función_grupo
```

Figura 3.4: Pseudo-código para la creación de los subgrafos de tareas paralelizables.

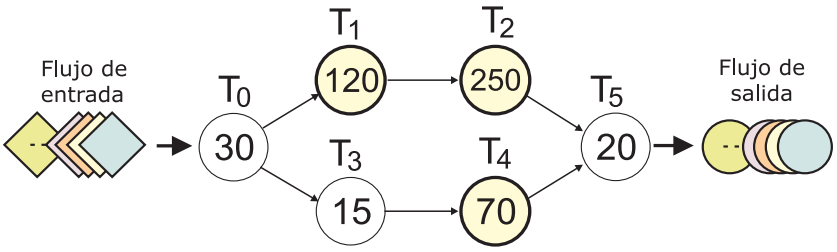


Figura 3.5: Grafo de tareas de la aplicación donde T_1, T_2 y T_4 son paralelizables.

Algoritmo principal

$Conjunto\_subgrafos$	M	T_i	$SG = grupo(T_i)$	H	$H \bigcap SG$
$\emptyset$	$\{T_1, T_2, T_4\}$	T_1	grupo$(T_1) = \{T_1, T_2\}$	$\emptyset$	-
$\{T_1, T_2\}$	$\{T_4\}$	T_4	grupo$(T_4) = \{T_4\}$	$\{T_1, T_2\}$	$\emptyset$
$\{T_1, T_2\}\{T_4\}$	$\emptyset$	-	-	-	-

Llamada a función grupo(T_1)

$tareas\_adyacentes$	M	T_j $sucesora$	grupo(T_j)	Retorna
$\{T_1\}$	$\{T_2, T_4\}$	T_2	grupo$(T_2) = T_2$	$\{T_1, T_2\}$
$\{T_1, T_2\}$	$\{T_4\}$	$\emptyset$	-	

Llamada a función $grupo(T_2)$

$tareas\_adyacentes$	M	T_j $sucesora$	grupo(T_j)	Retorna
$\{T_2\}$	$\{T_4\}$	$\emptyset$	-	$\{T_2\}$

Cuadro 3.1: Desarrollo del pseudo-código de la Figura 3.4 y de la llamada a las funciones $grupo(T_1)$ y grupo(T_2).

procesamiento no pueden darse de forma simultanea, si no que se debe finalizar la comunicación que se esté realizando para poder comenzar una nueva. Esta situación provoca que algunas de las subtareas deban esperar más que otras para recibir su parte del dato a procesar, como queda de manifiesto en la Figura 3.6, en la que se supone el caso en el que los datos se han dividido y repartido entre cuatro subtareas. En la figura se han marcado las fases de comunicación involucradas en la recepción y transmisión como IN y OUT respectivamente.

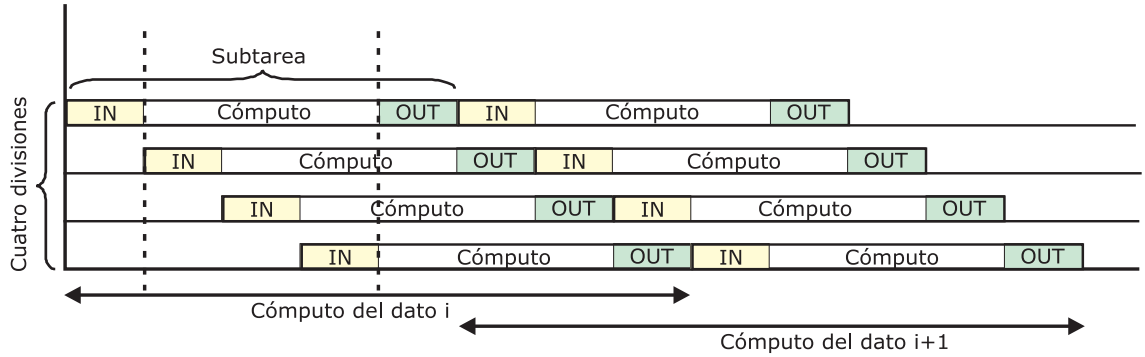


Figura 3.6: Solapamiento de las fases de cómputo y de comunicaciones.

Bajo esta premisa, el número máximo de subtareas a obtener no deberá provocar que las esperas que padecen éstas excedan al tiempo de cómputo que cada una de

ellas deba realizar. El caso óptimo será el que permita solapar perfectamente las fases de cómputo con las comunicaciones, tal y como se muestra en la Figura 3.6. Este caso ideal, es el que obtiene la mejor relación cómputo/comunicación en el procesamiento de los datos. Un menor número de divisiones en los datos, no permite obtener el máximo nivel de concurrencia para la tarea, mientras que un mayor número forzará a periodos de inactividad en espera de la parte del dato a procesar.

El número óptimo de subtareas a obtener puede ser caracterizado en función de su tiempo de cómputo y el tiempo involucrado en la transmisión de cada división del dato según la expresión:

$$(n\_subtareas[T_i] - 1) \times coste\_comIN(T_{i\_sub}) \leq \mu(T_{i\_sub}) \tag{3.1}$$

En la expresión $n\_subtareas[T_i]$ indica el número subtareas calculado para la tarea T_i, T_{i_sub} hace referencia a una de las subtareas obtenidas para la tarea T_i, $coste\_comIN(T_{i\_sub})$ es el tiempo asociado a las comunicaciones de entrada para esa subtarea y $\mu(T_{i\_sub})$ es el tiempo de computo de la misma.

El método que se propone en este trabajo, aplica el paradigma de paralelismo de datos sobre las tareas siempre y cuando éstas excedan en su tiempo de cómputo al valor de IP a alcanzar, en cuyo caso se determinará el número de subtareas más adecuado a obtener, procurando de esta forma que la sobrecarga ocasionada por el aumento en el número de comunicaciones a realizar no suponga un problema en la obtención del rendimiento deseado.

En la técnica es importante que se pueda modelar correctamente el tiempo de cómputo de las subtareas T_{i_sub}, por lo que presuponemos que las aplicaciones poseen un comportamiento estable que permite mediante un estudio previo del tiempo de cómputo de la tarea original, para diferentes tamaños en la división de los datos, y a partir de las técnicas presentadas en la sección 2.1, obtener una referencia en el número de divisiones necesarias para alcanzar el valor de IP.

La Figura 3.7 muestra el pseudo-código del algoritmo que se debe aplicar a cada subgrafo para determinar el número de subtareas y de subgrafos a obtener.

Para cada tarea T_i, que forma parte del subgrafo SG, se calcula el número de subtareas que debe obtenerse de cada una de ellas teniendo en cuenta el valor de IP a alcanzar, este número se obtiene a partir del estudio previo comentado anteriormente. El número de copias para el subgrafo se determina a partir de la tarea con el menor

```
1  Para cada subgrafo SG ∈ Conjunto_subgrafos
2   Para cada T_i ∈ SG
3      Calcular n_subtareas[T_i]
4   fin_para
5   R_min = mín(n_subtareas[T_i]) en SG
6   n_copias[SG] = R_min
7   Para cada T_i ∈ SG
8      n_subtareas[T_i]=⌈ n_subtareas[T_i] / R_min ⌉
9   fin_para
10 fin_para
```

Figura 3.7: Pseudo-código para determinar el número de subtareas y de subgrafos a obtener en la Técnica de Paralelización.

número de subtareas a obtener, R_{min}. Este valor se utiliza para ajustar el total de subtareas para cada una de las tareas del subgrafo.

El Cuadro 3.2 muestra el desarrollo del algoritmo para el grafo mostrado en la Figura 3.5, asumiendo a modo de ejemplo, que el valor $n\_subtareas[T_i]$, para T_1, T_2 y T_4 es de 2, 4 y 2 respectivamente. Con esta configuración, se presupone que las subtareas obtenidas poseen un tiempo de cómputo asociado de 50, 45 y 25 unidades de tiempo respectivamente.

$Conjunto\_subgrafos$	SG	T_i	n_subtareas[T_i]	R_{min}	$n\_copias[SG]$
$\{T_1,T_2\}\{T_4\}$	$\{T_1,T_2\}$	T_1	$n\_subtareas[T_1]$=2	-	-
		T_2	$n\_subtareas[T_2]$=4	mín{2,4}=2	2
		T_1	$\lceil \frac{n\_subtareas[T1]}{R_{min}} \rceil$=1		
		T_2	$\lceil \frac{n\_subtareas[T_2]}{R_{min}} \rceil$=2		
	$\{T_4\}$	T_4	$n\_subtareas[T_4]$=2	mín(2)=2	2
		T_4	$\lceil \frac{n\_replications[T_4]}{R_{min}} \rceil$=1		

Cuadro 3.2: Desarrollo del cálculo del número de copias para cada subgrafo y de subtareas en él para la Técnica de Paralelización.

El resultado que se obtiene determina que el subgrafo $\{T_1,T_2\}$ debe aparecer dos veces y con una y dos subtareas para T_1 y T_2 respectivamente. Para el subgrafo $\{T_4\}$, que contiene una única tarea, éste debe aparecer dos veces con una única subtarea de T_4. Por lo tanto tras aplicar la Técnica de Paralelización propuesta, el nuevo grafo de tareas para la aplicación la Figura 3.5, quedaría tal y como se muestra en la Figura 3.8(a). En ella se puede observar como al realizarse la paralelización a nivel del subgrafo, las

dependencias afectan exclusivamente a las subtareas que lo forman, eliminándose las comunicaciones cruzadas entre subtareas y simplificando el patrón de comunicaciones de la aplicación resultante.

En la Figura 3.8(b) se muestra el diagrama de Gantt para el procesamiento de tres datos del flujo de entrada, suponiendo que cada tarea está en un nodo de procesamiento y que las comunicaciones desde un mismo nodo no pueden darse de forma simultanea, si no que se deben serializar. Para simplificar el ejemplo, se ha supuesto que todas las comunicaciones poseen el mismo coste temporal, de forma que en el mismo se pueda analizar sin distorsiones el comportamiento de las tareas. Se puede observar como los subgrafos definidos mediante la Técnica de Paralelización actúan de forma independiente, ejecutándose de forma concurrente. Con la nueva estructura obtenida, los resultados se obtienen cada 50 unidades de tiempo tal y como se marcó como objetivo.

3.1.2. Técnica de Replicación

La Técnica de Replicación, a diferencia de la técnica anterior, presupone que a las tareas que se identifican como cuello de botella para el rendimiento de la aplicación, no se les puede aplicar paralelismo de datos debido la función que implementan y en consecuencia su tiempo de cómputo no puede reducirse por esta vía. Otro aspecto a tener presente, es que el flujo de entrada a procesar no posee dependencias entre sus datos, por lo que éstos podrán ser procesados en cualquier orden. Bajo estas premisas, la Técnica de Replicación toma las tareas identificadas como cuello de botella y realiza copias exactas de ellas para que vayan procesando de forma cíclica los datos del flujo de entrada. Debido a la característica que se presupone en las comunicaciones, en la que antes de iniciar una nueva debe haber finalizado la actual, la elección del número de copias a realizar depende directamente de la capacidad de transmitir la información entre los nodos así como de los tiempos de cómputo de las tareas asignadas [GRR+06].

A modo de ejemplo la Figura 3.9(a) muestra esta situación, para dos tareas, T_0 y T_1 con 5 y 20 unidades de tiempo de cómputo respectivamente. El ejemplo supone que el coste de las comunicaciones entre ellas es de 10 unidades de tiempo y que ambas tareas se asignan a nodos de procesamiento diferentes.

La Figura 3.9(b) muestra el diagrama de Gantt que se obtiene de la ejecución de la aplicación, suponiendo que no existen restricciones en el acceso a los datos del flujo de entrada, obteniéndolos en el momento en el que la tarea T_0 los necesita. En la figura se remarca el efecto producido por la serialización en las comunicaciones, de forma

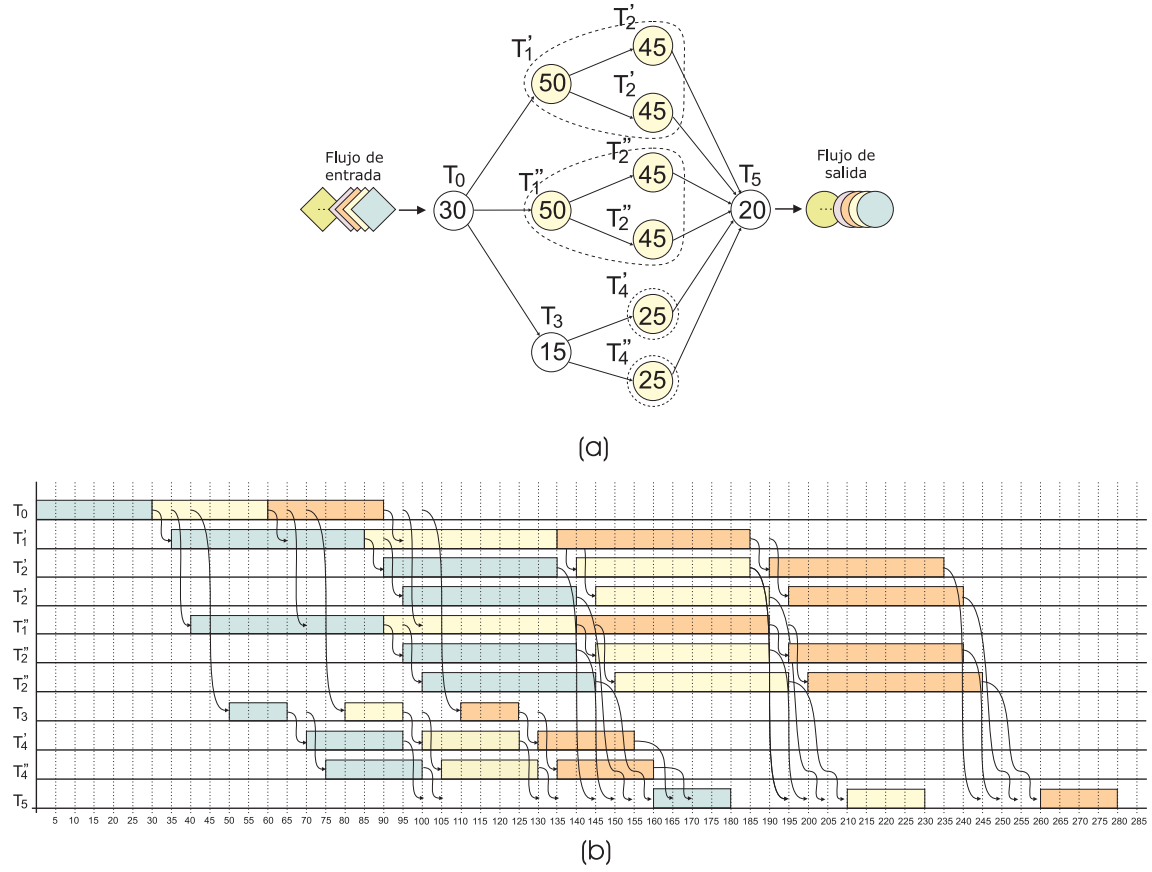


Figura 3.8: (a) Grafo de tareas obtenido tras aplicar la Técnica de Paralelización. (b) Diagrama de Gantt del procesamiento de tres datos de entrada .

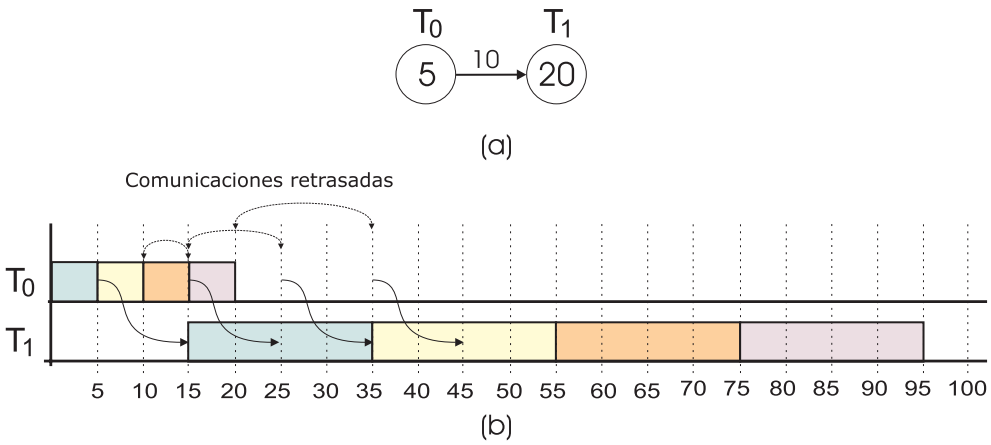


Figura 3.9: (a) Grafo de tareas. (b) Diagrama de Gantt del procesamiento de cuatro datos.

que no puede iniciarse una hasta que la actual no ha finalizado. Así, la información generada en T_0 en el procesamiento del segundo dato, no puede ser transmitido hasta el momento 15 a pesar de estar disponible en el momento 10. Aún bajo esta restricción en las comunicaciones, la productividad obtenida depende directamente del cómputo de la tarea T_1, debido que su tiempo de cómputo es muy superior, permitiendo que aún que se retrasen las comunicaciones, los datos que transmiten estén disponibles cuando son requeridos. Debido que la tarea T_1 marca el ritmo de procesamiento esto la convierte en una candidata a aplicarle la Técnica de Replicación.

La Figura 3.10(a) muestra la estructura que se obtiene para el mismo ejemplo, suponiendo que se decide replicar la tarea T_1 dos veces, obteniendo T_1^1 y T_1^2.

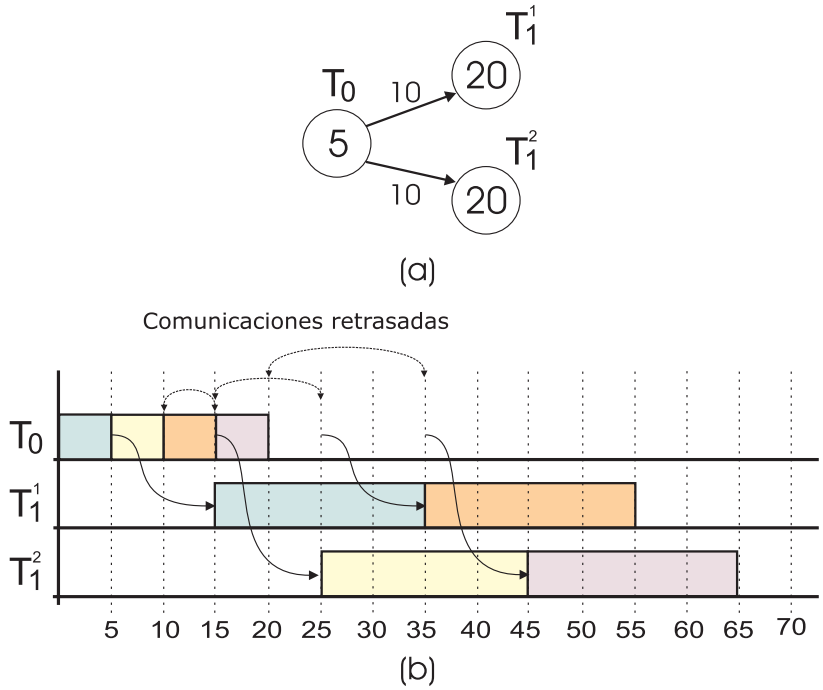


Figura 3.10: (a) Grafo de tareas obtenido tras replicar la tarea T_1 dos veces. (b) Diagrama de Gantt del procesamiento de cuatro datos.

El diagrama de Gantt que representa la ejecución de esta nueva estructura se puede observar en la Figura 3.10(b). La tarea T_0 va repartiendo los datos que recibe desde el flujo de entrada a cada una de las copias de T_1, de forma que éstos son procesados concurrentemente lo que permite obtener un aumento significativo en la productividad. El número de veces en que la tarea T_1 puede ser replicada depende directamente de la capacidad que tiene la tarea T_0 de repartir sus resultados. Si se replica un número excesivo de veces, se daría la situación en la que algunas de las replicaciones finalizarían

su ejecución antes de recibir el siguiente dato a procesar, por lo que el número de replicaciones a realizar está determinado tanto por el tiempo de cómputo concurrente de las tareas replicadas como por la capacidad de recibir los datos necesarios para dicho cómputo. Por este motivo, las tareas que no pueden alcanzar el ritmo requerido de entrega de los datos entran dentro del conjunto de tareas que deben ser replicadas.

Para que una tarea de la aplicación sea candidata a ser replicada debe cumplir una de las dos condiciones siguientes:

1. Su tiempo de cómputo es mayor al valor de IP a alcanzar. Esta condición hace que se pueda procesar más de un dato de forma concurrente por cada una de las tareas replicadas.

2. El tiempo involucrado en la comunicación con alguna de sus tareas sucesoras es mayor al valor de IP definido. En este caso la tarea encargada de repartir los datos es la que actúa como cuello de botella al no poder entregarlos con el ratio adecuado, por lo que si se replica se consigue aumentar el ratio de entrega.

Para ilustrar la forma en que la heurística actúa se utilizará la Figura 3.11 que representa una aplicación pipeline formada por una única línea de ejecución compuesta por cinco tareas. De lo expuesto en el capítulo previo, la productividad máxima teórica que se puede obtener para esta aplicación viene determinada por la tarea T_3 que es la que posee un mayor tiempo de cómputo asociado, que es de 30 unidades de tiempo, siendo éste el valor teórico de IP que puede ofrecer la aplicación. Para el ejemplo, se tomará como objetivo a alcanzar un valor de IP de 10 unidades de tiempo.

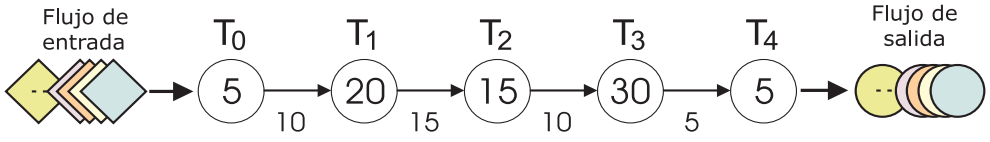


Figura 3.11: Grafo de tareas de la aplicación de ejemplo para la Técnica de Replicación.

La Técnica de Replicación se divide en los dos pasos siguientes:

Paso 1 - Identificar las tareas candidatas a ser replicadas agrupadas en subgrafos.

Paso 2 - Determinar el número de veces en el que cada subgrafo y cada tarea dentro de él, deben ser replicados.

A continuación se desarrolla cada paso:

Paso 1 - Obtención de los subgrafos de tareas a replicar

La primera parte de la heurística, siguiendo el concepto de subgrafo presentado anteriormente, se basa en determinar que tareas deben ser replicadas y que subgrafos deberán ser definidos. El pseudo-código del algoritmo que implementa esta parte de la heurística se muestra en la Figura 3.12.

El algoritmo evalúa mediante la función $es\_replicable(T_i)$, cada una de las tareas de la aplicación comprobando si cumple alguna de las dos condiciones necesarias para ser replicada. Si cumple alguna de ellas se llama a la función recursiva $grupo(T_i)$, que comprueba si existen tareas adyacentes que a su vez deban ser replicadas formando parte de ese mismo subgrafo.

Al final del algoritmo se obtiene un conjunto, $Conjunto\_SG$, que contiene todos los subgrafos identificados y que contienen las tareas a replicar. El Cuadro 3.3 muestra el desarrollo del algoritmo sobre el grafo de tareas propuesto como ejemplo.

Como resultado de aplicar la técnica sobre el ejemplo se obtiene un único subgrafo a replicar formado por el conjunto de tareas $\{T_1, T_2, T_3\}$, que actúan como cuello de botella para alcanzar el valor de IP deseado.

Paso 2 - Cálculo del número de replicaciones de los subgrafos y de sus tareas

Para cada tarea presente en cada uno de los subgrafos, identificados en el paso previo, se define su número de replicaciones, $n\_rep[T_i]$, a partir de la evaluación de la siguiente expresión:

$$n\_rep[T_i] = \left\lceil \frac{\max(\mu(T_i), \max_{\forall T_j \ sucesora \ de \ T_i} com(T_i, T_j))}{IP} \right\rceil$$

En esta expresión el número de replicaciones de una tarea, es el mayor valor entre el tiempo de cómputo de la tarea y el mayor de los tiempos involucrados en las comunicaciones asociadas con sus tareas sucesoras, divididos por el valor de IP a alcanzar.

De entre todos los valores $n\_rep[T_i]$ obtenidos para las tareas de un subgrafo, el menor de ellos, $R_{\text{mín}}$, es el que define el número de replicaciones globales del subgrafo. Así para las tareas que forman parte de él, se debe recalcular el total de replicaciones teniendo presente que el subgrafo puede aparecer en múltiples copias.

El pseudo-código que implementa el Paso 2 de esta heurística se muestra en la Figura 3.13.

```
1  Conjunto_SG = ∅
2  M = {T_i ; T_i ∈ Aplicación}
3  Para cada T_i ∈ M
4   Si es_replicable(T_i) entonces
5      SG = grupo(T_i)
6      Para cada H ∈ Conjunto_SG
7         Si (H ⋂ SG) ≠ ∅ entonces
8            SG = SG ⋃ H
9            Conjunto_SG = Conjunto_SG - {H}
10        fin_si
11        Conjunto_SG=Conjunto_SG ⋃ SG
12     fin_para
13    si no
14      M = M − T_i
15  fin_si
16 fin_para
```

```
1  Función grupo(T_i)
2   tareas_adyacentes = ∅
3   M = M − T_i
4   tareas_adyacentes = T_i
5   Para cada tarea T_j sucesora de T_i y es_replicable(T_j)
6      tareas_adyacentes = tareas_adyacentes ⋃ grupo(T_j)
7   fin_para
8   retorna(tareas_adyacentes)
9  fin_función_grupo
```

```
1  Función es_replicable(T_i)
2   Si (μ(T_i) > IP) o (∃ T_j sucesora de T_i tal que com(T_i → T_j) > IP) entonces
3      retorna(true)
4    si no
5      retorna(false)
6   fin_si
7  fin_función_es_replicable
```

Figura 3.12: Pseudo-código para identificar los subgrafos formados por las tareas a ser replicadas.

Algoritmo principal

M	T_i	$es\_replicable(T_i)$	$grupo(T_i)$	$Conjunto\_SG$
$\{T_0, T_1, T_2, T_3, T_4\}$	T_0	**false** $\mu(T_0) = 5 < 10$ $com(T_0 \rightarrow T_1) = 10 \leq 10$	-	$\emptyset$
$\{T_1, T_2, T_3, T_4\}$	T_1	**true** $\mu(T_1) = 20 > 10$ $com(T_0 \rightarrow T_1) = 15 > 10$	$grupo(T_1) = \{T_1, T_2, T_3\}$	$\{T_1, T_2, T_3\}$
$\{T_4\}$	T_4	**false** $\mu(T_5) = 5 < 10$	-	$\{T_1, T_2, T_3\}$

Llamada a función grupo(T_1)

$tareas\_adyacentes$	T_j	$es\_replicable(T_j)$	$grupo(T_j)$	M
$\{T_1\}$	T_2	**true** $\mu(T_2) = 15 > 10$ $com(T_2 \rightarrow T_3) = 10 \leq 10$	$grupo(T_2) = \{T_3\}$	$\{T_2, T_3, T_4\}$

Llamada a función grupo(T_2)

$tareas\_adyacentes$	T_j	$es\_replicable(T_j)$	$grupo(T_j)$	M
$\{T_1, T_2\}$	T_3	**true** $\mu(T_3) = 35 > 10$ $com(T_3 \rightarrow T_4) = 5 \leq 10$	$grupo(T_3) = \{T_3\}$	$\{T_3, T_4\}$

Llamada a función grupo(T_3)

$tareas\_adyacentes$	T_j	$es\_replicable(T_j)$	$grupo(T_j)$	M
$\{T_1, T_2, T_3\}$	T_4	**false** $\mu(T_4) = 5 < 0$	-	$\{T_4\}$

Cuadro 3.3: Desarrollo de la obtención de los subgrafos formados por las tareas que deben ser replicadas para el ejemplo de la Figura 3.11.

1 **Para cada** $SG \in Conjunto\_subgrafos$
2 **Para cada** $T_i \in SG$
3 n_rep[T_i]= $\left\lceil \frac{\text{máx}(\mu(T_i), \underset{\forall T_j \ sucesora \ de \ T_i}{\text{máx}}(comm(T_i \rightarrow Tj)))}{IP} \right\rceil$
4 **fin_para**
5 $R_{min} = \underset{\forall T_i \in SG}{\text{mín}}(n\_rep[T_i])$
6 $n\_copias[SG] = R_{min}$
7 **Para cada** $T_i \in SG$
8 n_rep[T_i]= $\left\lceil \frac{n\_rep[T_i]}{R_{min}} \right\rceil$
9 **fin_para**
10 **fin_para**

Figura 3.13: Pseudo-código para determinar el número de copias de los subgrafos y de las tareas en ellos para la Técnica de Replicación.

En el Cuadro 3.4 se muestra desarrollado el algoritmo sobre el subgrafo identificado previamente, $\{T_1, T_2, T_3\}$. El resultado que se obtiene, determina que el subgrafo debe aparecer replicado dos veces, y el número de copias de las tareas dentro de cada replicación del subgrafo será de 1, 1 y 2 veces para las tareas T_1, T_2 y T_3 respectivamente.

SG	T_i	n_rep[T_i]	R_{min}	$n\_copias[SG]$
$\{T_1, T_2, T_3\}$	T_1	n_rep[T_1]=$\lceil \frac{\text{máx}(20,15)}{10} \rceil = 2$		
	T_2	n_rep[T_2]=$\lceil \frac{\text{máx}(15,10)}{10} \rceil = 2$		
	T_2	n_rep[T_3]=$\lceil \frac{\text{máx}(30,5)}{10} \rceil = 3$	mín{2,2,3}=2	2
	T_1	$\lceil \frac{n\_rep[T_1]}{R_{min}} \rceil = \lceil \frac{2}{2} \rceil = 1$		
	T_2	$\lceil \frac{n\_rep[T_2]}{R_{min}} \rceil = \lceil \frac{2}{2} \rceil = 1$		
	T_3	$\lceil \frac{n\_rep[T_3]}{R_{min}} \rceil = \lceil \frac{3}{2} \rceil = 2$	2	2

Cuadro 3.4: Desarrollo del cálculo del número de copias para cada subgrafo y de replicaciones de las tareas.

Tras aplicar este método la nueva estructura que se obtiene para el grafo de tareas de la aplicación es la mostrada en la Figura 3.14(a).

La Figura 3.14(b) muestra el diagrama de Gantt de la ejecución de la aplicación para seis datos consecutivos, una vez replicada y suponiendo que cada tarea es asignada a un único nodo de procesamiento. Se observa que los datos que se obtienen desde el flujo de entrada van siendo entregados de forma alternada a cada una de las copias de las tareas replicadas que se han añadido, y como dentro de cada subgrafo al que se

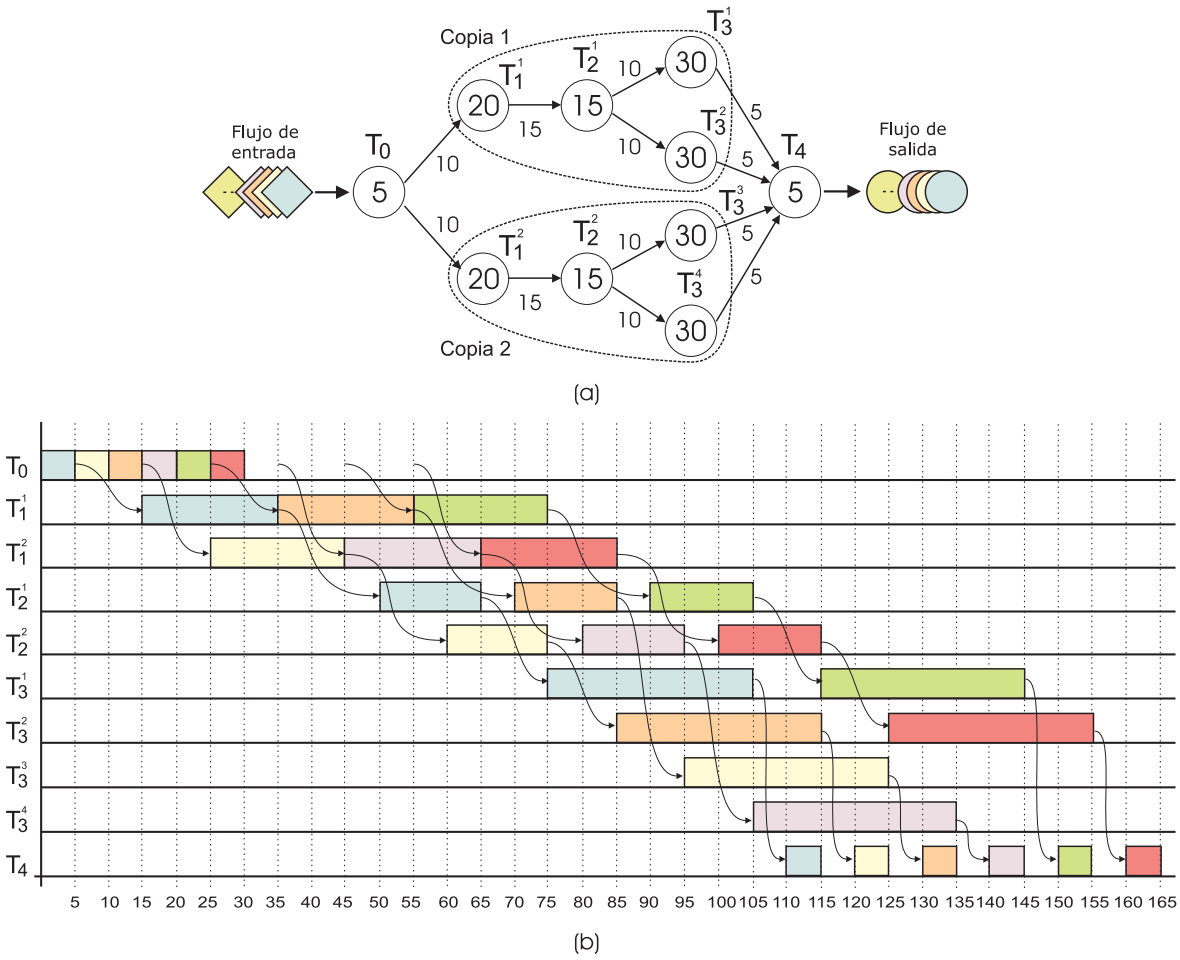


Figura 3.14: (a) Grafo de tareas obtenido tras la replicación. (b) Diagrama de Gantt del procesamiento de seis datos de entrada.

asignan éstas, los datos avanzan de forma independiente.

Mediante esta nueva estructura de depencias de tareas para la aplicación se consigue el procesamiento de múltiples datos de forma concurrente tal y como se deseaba en un principio, mejorándose así el ratio de datos procesados por unidad de tiempo y en este caso alcanzándose el valor marcado como objetivo para el IP de 10 unidades de tiempo.

3.2. Heurísticas de mapping

Las heurísticas de mapping aparecen como una segunda fase en la optimización de las aplicaciones pipeline. En la fase previa, se han definido las técnicas que permiten redefinir el grafo de tareas que representa a la estructura de dependencias de la aplicación con el fin de alcanzar un rendimiento determinado. Por el contrario, en esta

segunda fase, se pretende que esos objetivos de rendimiento se mantengan al realizar la asignación de las tareas a los nodos de procesamiento sin estar forzado para ello a usar tantos nodos de procesamiento como tareas tiene la aplicación.

Para poder hacer la asignación de forma adecuada, se parte del concepto de *ejecución síncrona*. Una de las principales características de las aplicaciones pipeline, es la capacidad que tienen de procesar múltiples datos, provenientes del flujo de entrada, de forma concurrente. Esta característica en el procesamiento se explota al máximo cuando se consigue que los datos a procesar avancen de forma síncrona por las tareas de la aplicación.

Este avance síncrono es más fácil de conseguir si la estructura de la aplicación es totalmente homogénea, tanto para el tiempo de cómputo de las tareas como para las comunicaciones involucradas. Una estructura ideal como ésta es difícil de implementar cuando el desarrollo de la aplicación se realiza a un alto nivel de abstracción, mediante el uso de librerías que por lo general no han sido diseñadas para que todas sus funciones posean un tiempo de cómputo similar. También es posible que las dependencias entre tareas no posean una estructura regular o no requieran del mismo volumen de información a transmitir, lo que puede provocar que las comunicaciones tampoco se den de forma homogénea.

La fase de mapping ha sido tratada en la literatura dando lugar a múltiples heurísticas. Tanto éstas como las desarrolladas en el presente trabajo, se basan en intentar agrupar las tareas de la aplicación, con la finalidad de homogeneizar el cómputo o las comunicaciones, consiguiendo la ejecución síncrona. Las diferencias de las propuestas estriban en como se define y obtiene ese sincronismo.

Algunos autores han optado por definir el sincronismo a nivel de balancear el volumen de cómputo asignado a los nodos de procesamiento. Para ello sin tener presente el comportamiento iterativo de las tareas, realizan una asignación previa basándose en algoritmos clásicos de mapping de TPGs para más tarde, aplicando procesos de refinamiento, obtener la ejecución síncrona [YKS03].

Otros autores por el contrario consideran desde el principio el comportamiento iterativo de las aplicaciones pipeline pero parten de fuertes restricciones en la estructura de la aplicación, forzándola a un único y exclusivo patrón de dependencias. En estos casos, las tareas se agrupan en lo que denominan etapas que son utilizadas como base para la ejecución síncrona. En algún trabajo se obliga a que exista una única tarea por etapa [SV00], en otros por el contrario, se permite que el número de éstas sea variable pero bajo la condición de que todas ellas tengan el mismo volumen de cómputo y que

todas las tareas presentes en etapas adyacentes sean dependientes entre sí [LLP01]. Ambas propuestas acotan el problema predefiniendo una estructura de dependencias de tareas que facilita la obtención de la ejecución síncrona.

El presente trabajo no impone ninguna restricción desde el punto de vista del tiempo de cómputo de las tareas, del volumen de datos en las comunicaciones o de las dependencias existentes entre ellas, como es el caso de las propuestas comentadas de la literatura. Por este motivo, antes de afrontar el problema de desarrollar las heurísticas de mapping se definirá el concepto de *etapa síncrona* [GRRL04b][GRRL05a].

El concepto de etapa síncrona toma como referencia un valor de tiempo que define su duración y que se identifica con el valor de IP. Este intervalo de tiempo corresponde al tiempo máximo en el que los datos han de estar disponibles antes de realizar su intercambio dentro del avance de la ejecución pipeline. Así, dentro de una etapa síncrona se encontrarán aquellas tareas, que por sus dependencias puedan ejecutarse, o bien de forma concurrente en el mismo intervalo de tiempo, o de forma secuencial debido a sus dependencias, siempre que su tiempo de cómputo acumulado no exceda de ese valor de tiempo prefijado. La etapa síncrona engloba de una forma más flexible, la idea de etapa utilizada en la literatura, añadiendo la posibilidad de que un conjunto de tareas que por sus dependencias son adyacentes, puedan incluirse de forma conjunta en la misma etapa síncrona sin perjuicio en el ratio de procesamiento.

Para ilustrar el concepto de etapa síncrona y como ésta es identificada se usará el grafo de la Figura 3.15. En el se muestra una aplicación de seis tareas formada por una única línea de ejecución y en la que se desea una ejecución síncrona con un intervalo de tiempo, o valor de IP, de 15 unidades de tiempo. Para simplificar el ejemplo se ha supuesto que todas las comunicaciones requieren el mismo tiempo de realización.

Para definir las etapas síncronas se evalua las tareas desde la primera de ellas, y recorriendo el grafo de tareas hasta llegar a la última tarea de la aplicación, de forma que se asignan a cada nueva etapa síncrona el conjunto de tareas cuyo tiempo de cómputo acumulado cumpla menor o igual al valor a alcanzar de IP. Cada etapa síncrona se identifica con un índice de profundidad, siendo la primera aquella que engloba a la primera tarea de la aplicación e incrementándose hasta alcanzar la última de ellas.

Para el ejemplo propuesto se obtienen tres etapas síncronas, mostradas en la Figura 3.15(a) sobre el grafo de la aplicación, siendo las tareas que forman parte de cada una de ellas los conjuntos $\{T_0, T_1\}$, $\{T_2, T_3, T_4\}$ y $\{T_5\}$. El cómputo acumulado de cada conjunto es de 15, 15 y 10 unidades de tiempo respectivamente.

La Figura 3.15(b) muestra el diagrama de Gantt del comportamiento de la aplica-

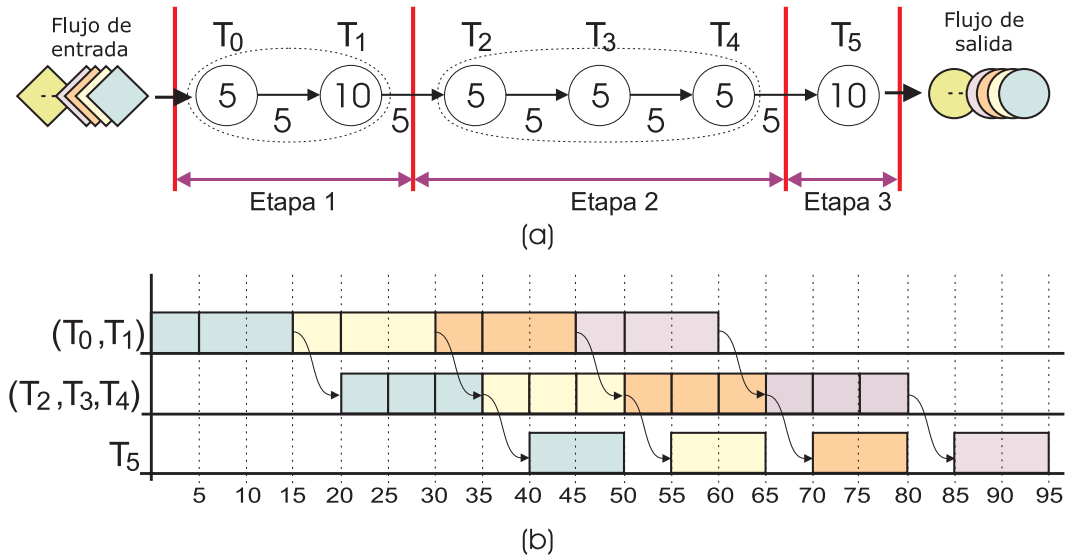


Figura 3.15: (a) Grafo de tareas y etapas síncronas obtenidas. (b) Diagrama de Gantt del procesamiento de cuatro datos de entrada.

ción suponiendo que el conjunto de tareas que forma cada etapa síncrona es considerado como una única entidad y ejecutada en un mismo procesador. Se puede observar como las etapas síncronas solapan el cómputo de sus tareas avanzando de forma síncrona tal y como se desea.

A la hora de determinar que tareas forman parte de las diferentes etapas síncronas, el cómputo acumulado nunca debe ser superior al valor preestablecido del IP. Esta situación puede ocasionar que alguna de ellas no alcance exactamente dicho valor, dando lugar a que no todas las etapas síncronas tengan la misma duración, produciéndose fases de inactividad debido a las dependencias entre las tareas que forman parte de ellas. En el ejemplo se puede observar esta situación, en el que la tercera etapa síncrona, formada exclusivamente por la tarea T_5 posee un tiempo de cómputo inferior a 15 unidades provocando fases de inactividad de 5 unidades de tiempo.

Bajo la suposición de que las aplicaciones pueden tener una estructura arbitraria, el número de líneas de ejecución que pueden existir puede ser variable. La obtención de las etapas síncronas en este caso, parte de evaluar individualmente cada una de las líneas de ejecución. La Figura 3.16(a) muestra como ejemplo el caso de una aplicación con dos líneas de ejecución formadas por las tareas $\{T_0, T_1, T_3, T_5\}$ y $\{T_0, T_2, T_4, T_5\}$ respectivamente, y en las que el tiempo de cómputo de sus tareas es diferente. Para desarrollar el ejemplo se ha escogido un valor de IP de 15 unidades.

En la Figura 3.16(b) se muestran las etapas síncronas que se obtienen al evaluar

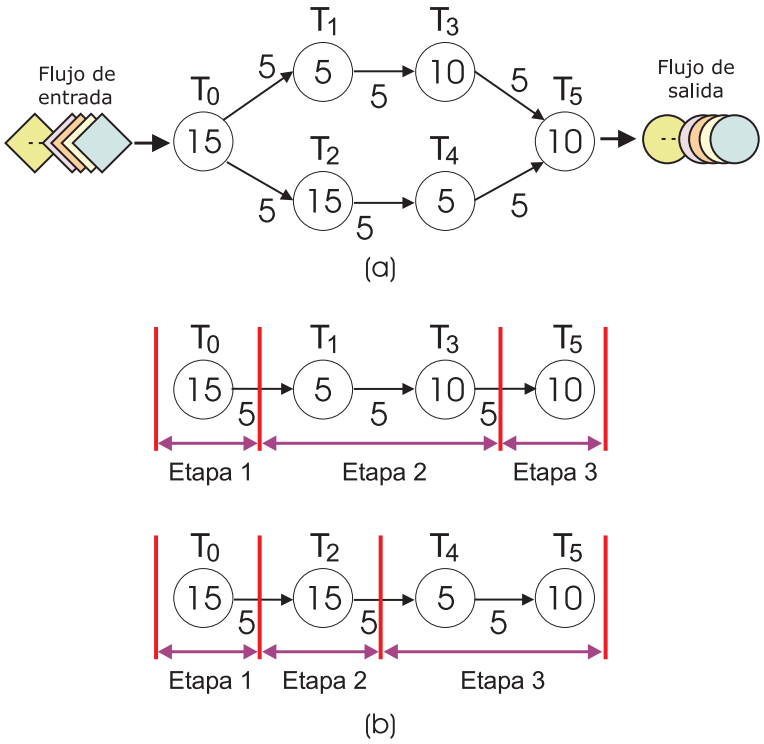


Figura 3.16: (a) Grafo de tareas. (b) Etapas síncronas obtenidas en las líneas de ejecución.

cada una de las líneas de ejecución. La primera línea de ejecución se distribuye en tres etapas de la forma $\{T_0\}$, $\{T_1, T_3\}$ y $\{T_5\}$ y la segunda de la forma $\{T_0\}$, $\{T_2\}$ y $\{T_4, T_5\}$. Ambos resultados se han conjuntar para obtener una única definición de las etapas síncronas. Para ello, finalmente cada etapa síncrona i, contendrá las tareas ubicadas en cada una de las etapas síncronas parciales de igual índice.

En la Figura 3.17(a), se muestra la distribución de las etapas síncronas sobre el grafo original, y en 3.17(b) mediante el diagrama de Gantt, el comportamiento que se obtiene para los conjuntos de tareas obtenidos en cada línea de ejecución. Se puede observar como estas agrupaciones solapan su cómputo dentro de la etapa síncrona, como es el caso de $\{T_1, T_3\}$ y $\{T_2\}$.

Al evaluar de forma independiente cada una de las líneas de ejecución, se puede dar la situación de que el número de etapas síncronas que se obtiene en cada una de ellas sea diferente. Esto es debido a la posibilidad de que el número de tareas o el tiempo de cómputo asociado a ellas sean también diferentes. De igual forma también es posible que una tarea pueda estar asignada a diferentes etapas síncronas, en función de la línea de ejecución que está siendo evaluada. El criterio para asignar finalmente estas tareas es el de escoger las etapas síncronas con un menor índice de profundidad.

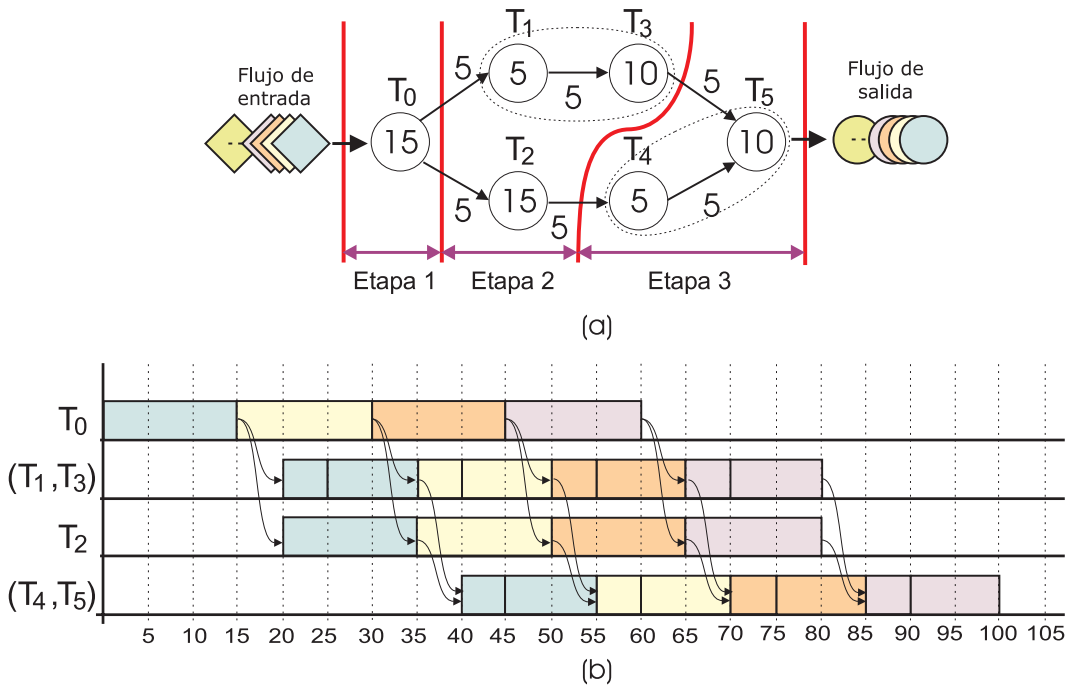


Figura 3.17: (a) Grafo de tareas y etapas síncronas obtenidas. (b) Diagrama de Gantt del procesamiento de cuatro datos de entrada.

La conclusión que se obtiene del concepto de etapa síncrona, es que si se puede definir de forma correcta se convierten en un concepto útil a la hora de obtener la asignación de tareas a nodos de procesamiento.

A continuación se presentan las heurísticas de mapping desarrolladas en este trabajo, denominadas MPASS (*Mapping of Pipeline Applications based on Synchronous Stages*) y MPART (*Mapping of Pipeline Applications based on Reduced Tree*). Ambas utilizan la definición de etapa síncrona, para determinar el conjunto de tareas que se asignará a cada nodo de procesamiento. Los objetivos de optimización que se definen para ambas heurísticas son:

- *Obtener la menor latencia posible, para un valor de productividad dado, utilizando el menor número de nodos de procesamiento.*
- *Obtener una productividad minimizando el número de nodos de procesamiento.*

3.2.1. Heurística de mapping - MPASS (*Mapping of Pipeline Applications based on Synchronous Stages*)

La heurística MPASS se basa en tres pasos:

Paso 1 - Identificar las líneas de ejecución de la aplicación y definir las etapas síncronas.

Paso 2 - Determinar que tareas dentro de una misma etapa síncrona se deben asignar al mismo nodo de procesamiento.

Paso 3 - Minimizar el número de nodos de procesamiento utilizados.

A continuación se desarrolla cada uno de ellos por separado.

Paso 1 - Identificar las líneas de ejecución de la aplicación y definir las etapas síncronas

Como se ha comentado anteriormente, es necesario conocer las líneas de ejecución de la aplicación con la finalidad de determinar las diferentes etapas síncronas. Para ello, basta con recorrer el grafo TPG que modela la aplicación, desde la primera tarea de ésta, por los diferentes caminos existentes hacia la última tarea de la aplicación.

En la heurística, cada línea de ejecución será evaluada tanto para definir las etapas síncronas como para determinar las agrupaciones de tareas que se deben realizar. En función del parámetro de rendimiento escogido, latencia o productividad, el orden en el que éstas son evaluadas varía, utilizando para ello un valor de coste que determina su prioridad en el orden de evaluación. Este valor de coste se calcula basándose en los siguientes criterios en función del objetivo a alcanzar:

- *Minimizar la latencia.* La función de coste se obtiene según la siguiente expresión:

$$coste(línea\_ejecución) = \sum_{T_i \in línea\_ejecución} \mu(T_i) + \sum_{T_i, T_j \in línea\_ejecución} com(T_i, T_j)$$

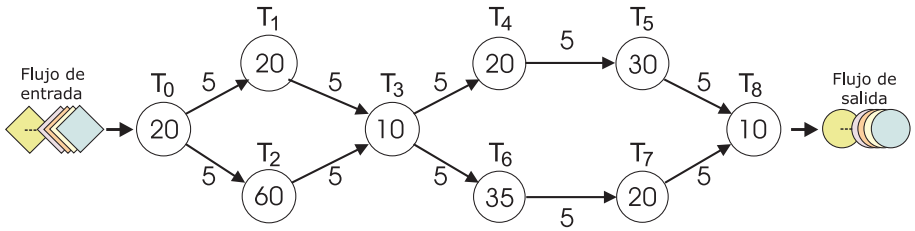
El valor de coste en sí representa la latencia individual de cada línea de ejecución. En este sentido, el criterio de ordenación de los valores será de mayor a menor, priorizando la evaluación de aquellas líneas de ejecución que poseen una latencia mayor, y permitiendo de esta forma que el proceso de mapping agrupe sus tareas eliminando parte de sus comunicaciones, lo cual redunda en una reducción de la latencia global.

- *Obtener una productividad dada.* En este caso la función de coste sólo tiene en cuenta el cómputo de las tareas presentes en la línea de ejecución, según la siguiente expresión:

$$coste(línea\_ejecución) = \sum_{T_i \in línea\_ejecución} \mu(T_i)$$

  El orden de evaluación será de menor a mayor coste, priorizando las agrupaciones de tareas en aquellas líneas de ejecución con menor cómputo global asociado. De esta forma se consigue una mayor concurrencia con las tareas de las líneas con mayor cómputo asociado.

Como ejemplo de aplicación de la heurística se utilizará el grafo de la Figura 3.18(a) el cual posee cuatro líneas de ejecución tal y como se muestran en la Figura 3.18(b), en la que se ha añadido el coste asociado calculado según el criterio de latencia o de productividad.



(a)

Líneas de ejecución	Coste
$\{T_0, T_2, T_3, T_6, T_7, T_8\}$	180
$\{T_0, T_2, T_3, T_4, T_5, T_8\}$	175
$\{T_0, T_1, T_3, T_6, T_7, T_8\}$	140
$\{T_0, T_1, T_3, T_4, T_5, T_8\}$	135

Criterio de latencia

Líneas de ejecución	Coste
$\{T_0, T_1, T_3, T_4, T_5, T_8\}$	110
$\{T_0, T_1, T_3, T_6, T_7, T_8\}$	115
$\{T_0, T_2, T_3, T_4, T_5, T_8\}$	150
$\{T_0, T_2, T_3, T_6, T_7, T_8\}$	155

Criterio de productividad

(b)

Figura 3.18: (a) Grafo TPG de una aplicación pipeline. (b) Ordenación de las líneas de ejecución por valor de coste según el criterio de latencia y de productividad.

Se puede observar como la elección de uno u otro criterio, hace que el orden en la evaluación de las líneas de ejecución sea diferente.

Una vez identificadas las líneas de ejecución se procederá a examinarlas con la finalidad de obtener las etapas síncronas y determinar las tareas que deben ser incluidas

en cada una de ellas. Para llevar a cabo este paso se requiere como parámetro el valor que define el tiempo de cómputo asociado a la etapa síncrona o IP. Este valor, determina la productividad que se obtendrá en la ejecución de la aplicación.

En el caso de querer obtener la máxima productividad posible de la aplicación, el valor escogido para el parámetro IP será el mayor tiempo de cómputo presente, de entre todas las tareas de la aplicación tal y como se comento en el capítulo anterior.

El pseudo-código que representa el algoritmo encargado de definir las etapas síncronas se muestra en la Figura 3.19.

```
1  Para cada línea de ejecución L ∈ Lista_líneas
2    i=1
3    T_ini=Primera tarea ∈ L
4    etapa_síncrona(i) = etapa_síncrona(i) ∪T_ini
5    cómputo_acumulado=μ(T_ini)
6    T_suc=Tarea sucesora de T_ini ∈ L
7    Mientras no se llegue al final de L
8      Mientras (cómputo_acumulado+μ(T_suc)) ≤ IP) y (T_suc ≠ NULL)
9        Si (T_suc ∈ etapa_síncrona(k))
10         Si (k>i) entonces
11           etapa_síncrona(k)=etapa_síncrona(k) - {T_suc}
12           etapa_síncrona(i)=etapa_síncrona(i) ∪T_suc
13         fin_si
14       si no
15         etapa_síncrona(i)=etapa_síncrona(i) ∪T_suc
16       fin_si
17       cómputo_acumulado=cómputo_acumulado + μ(T_suc)
18       T_suc=Tarea sucesora de T_suc ∈ L
19     fin_mientras
20     i=i+1
21     cómputo_acumulado=0
22   fin_mientras
23 fin_para
```

Figura 3.19: Pseudo-código para definir las etapas síncronas.

La forma de proceder del algoritmo, se basa en recorrer cada una de las líneas de ejecución buscando aquellos grupos de tareas cuyo tiempo de cómputo acumulado sea menor o igual al valor de IP a alcanzar. En este paso las comunicaciones no se tienen en cuenta. En el caso de que una tarea pueda pertenecer a más de una etapa síncrona se escogerá aquella que tenga un número de etapa menor.

En el Cuadro 3.5 se muestra el desarrollo del algoritmo para el ejemplo presentado en la Figura 3.18. En él se ha usado un valor de IP de 80 unidades de tiempo y el criterio de minimización de la latencia.

Línea de ejecución	i	T_{ini}:$\mu(T_{ini})$	T_{suc}:$\mu(T_{suc})$	*cómputo_acum.*	*etapa_síncrona(i)*
$\{T_0, T_2, T_3, T_6, T_7, T_8\}$	1	T_0:20	-	20	$\{T_0\}$
			T_2:60	80	$\{T_0, T_2\}$
	2		T_3:10	10	$\{T_3\}$
			T_6:35	45	$\{T_3, T_6\}$
			T_7:20	65	$\{T_3, T_6, T_7\}$
			T_8:10	75	$\{T_3, T_6, T_7, T_8\}$
$\{T_0, T_2, T_3, T_4, T_5, T_8\}$	1	T_0:20	-	20	$\{T_0, T_2\}$
			T_2:60	80	
	2		T_3:10	10	$\{T_3, T_6, T_7, T_8\}$
			T_4:20	30	$\{T_3, T_4, T_6, T_7, T_8\}$
			T_5:30	60	$\{T_3, T_4, T_5T_6, T_7, T_8\}$
			T_8:10	70	$\{T_3, T_4, T_5T_6, T_7, T_8\}$
$\{T_0, T_1, T_3, T_6, T_7, T_8\}$	1	T_0:20	-	20	$\{T_0, T_2\}$
			T_1:20	40	$\{T_0, T_1, T_2\}$
			T_3:10	50	$\{T_0, T_1, T_2, T_3\}$
	2		T_6:35	35	$\{T_4, T_5, T_6, T_7, T_8\}$
			T_7:20	55	$\{T_4, T_5, T_6, T_7, T_8\}$
			T_8:10	65	$\{T_4, T_5, T_6, T_7, T_8\}$
$\{T_0, T_2, T_3, T_4, T_5, T_8\}$	1	T_0:20	-	20	$\{T_0, T_1, T_2, T_3\}$
			T_1:20	40	$\{T_0, T_1, T_2, T_3\}$
			T_3:10	50	$\{T_0, T_1, T_2, T_3\}$
			T_4:20	70	$\{T_0, T_1, T_2, T_3, T_4\}$
	2		T_5:30	30	$\{T_5, T_6, T_7, T_8\}$
			T_8:10	40	$\{T_5, T_6, T_7, T_8\}$

Cuadro 3.5: Desarrollo del pseudo-código para la definición de las etapas síncronas con un IP de 80 unidades de tiempo.

Se puede observar que las tareas se distribuyen sobre dos etapas síncronas:
$etapa\_síncrona(1) = \{T_0, T_1, T_2, T_3, T_4\}$
$etapa\_síncrona(2) = \{T_5, T_6, T_7, T_8\}$
tal y como se muestra de forma gráfica, sobre el grafo de la aplicación, en la Figura 3.20.

Paso 2 - Identificar los clusters de tareas para su asignación

Una vez que se ha identificado la etapa síncrona a la que pertenece cada tarea, el siguiente paso consiste en crear las agrupaciones de tareas dentro de la etapa síncrona, que denominaremos *clusters*, y que serán asignados de forma indivisible a la misma unidad de procesamiento, dando lugar al mapping.

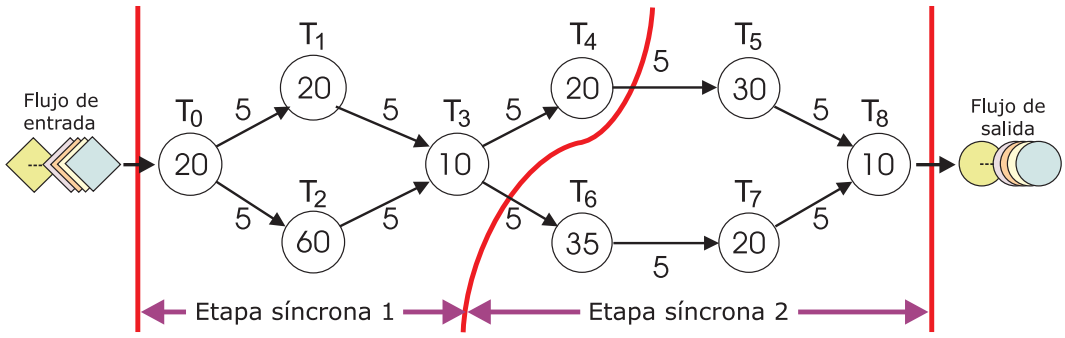


Figura 3.20: Etapas síncronas identificadas.

En este paso, nuevamente se recorre cada una de las líneas de ejecución, manteniendo la prioridad en su evaluación según el criterio de mapping escogido. Los clusters que se definen han de cumplir que el tiempo de cómputo acumulado de las tareas que los forman no exceda al valor de IP escogido. A pesar de las similitudes con el paso previo, en la definición de las etapas síncronas, éste paso se debe realizar por separado debido a que los clusters solo pueden formarse a partir de las tareas que se encuentran en la misma etapa síncrona. Así para asegurar con certeza la ubicación de cada tarea en la etapa síncrona a la que pertenece, se debe haber evaluado la totalidad de las líneas de ejecución.

La Figura 3.21 muestra el pseudo-código correspondiente al algoritmo que implementa este paso. Y el Cuadro 3.6 muestra el algoritmo anterior desarrollado sobre el ejemplo propuesto, manteniendo el valor de IP de 80 unidades de tiempo.

Debido a que las tareas pueden formar parte de más de una línea de ejecución, como es el caso de T_0, T_3 y T_8, en el algoritmo cada vez que se obtiene un nuevo cluster, esta agrupación se tiene en cuenta a la hora de evaluar las siguientes líneas de ejecución en las que se comparte alguna de estas tareas. La razón es que los clusters son indivisibles, por lo que cualquier nueva agrupación que involucre alguna de sus tareas afecta a todas las que forman el cluster. Esta situación se puede observar sobre el Cuadro 3.6, en la evaluación de la segunda línea de ejecución. En ésta la tarea T_5 podría formar un cluster con la tarea T_8, ya que el tiempo de cómputo acumulado de ambas es inferior a 80, pero esta agrupación no puede crearse ya que T_8 ya forma parte del cluster (T_6, T_7, T_8), cuyo cómputo acumulado es de 65 unidades de tiempo.

La Figura 3.22 muestra de forma gráfica los clusters obtenidos sobre el grafo de la aplicación.

```
1  Para cada línea de ejecución L ∈ Lista_líneas
2    i=1
3    Para cada etapa_síncrona(i)
4      Sea CL el primer elemento (cluster o tarea) tal que
                 (CL ∈ etapa_síncrona(i) y CL ∈ L)
5      cómputo_acumulado=μ(CL)
6      cluster=CL
7      Mientras exista un sucesor CL_j de CL tal que
                 (CL_j ∈ etapa_síncrona(i)) y (CL_j ∈ L)
8        Si (cómputo_acumulado+μ(CL_j)) ≤ IP entonces
9          cluster=cluster ∪ CL_j
10         cómputo_acumulado=cómputo_acumulado + μ(CL_j)
11       si no
12         mapping = mapping ∪ cluster
13         Si CL_j ≠ ∅ entonces
14           CL=CL_j
15           cómputo_acumulado=μ(CL_j)
16           cluster=CL_j
17         fin_si
18       fin_si
19     fin_mientras
20     Si cluster ≠ ∅ entonces
21       mapping=mapping ∪ cluster
22     fin_si
23     actualizar los clusters etapa_síncrona(i)
24     i=i+1
25   fin_para
26   Actualizar Lista_líneas con los nuevos clusters identificados
26 fin_para
```

Figura 3.21: Pseudo-código para la agrupación de tareas en clusters dentro de las etapas síncronas.

Línea de ejecución	etapa síncrona	CL: $\mu(CL)$	CL_j: $\mu(CL_j)$	cómputo acumulado	cluster	mapping
$\{T_0, T_2, T_3, T_6, T_7, T_8\}$	1	T_0:20	-	20	(T_0)	
			T_2:60	80	(T_0, T_2)	(T_0, T_2)
		T_3:10	-	10	(T_3)	$(T_0, T_2)(T_3)$
	2	T_6:35	-	35	(T_6)	
			T_7:20	55	(T_6, T_7)	$(T_0, T_2)(T_3)$
			T_8:10	65	(T_6, T_7, T_8)	(T_6, T_7, T_8)
$\{(T_0, T_2), T_3, T_4, T_5, (T_6, T_7, T_8)\}$	1	(T_0, T_2):80	-	80	(T_0, T_2)	$(T_0, T_2)(T_3)$ (T_6, T_7, T_8)
		T_3:10	-	10	(T_3)	$(T_0, T_2)(T_3, T_4)$
		-	T_4:20	30	(T_3, T_4)	(T_6, T_7, T_8)
	2	T_5:30	-	30	(T_5)	$(T_0, T_2)(T_3, T_4)$ $(T_5)(T_6, T_7, T_8)$
		(T_6, T_7, T_8):65	-	65	(T_6, T_7, T_8)	$(T_0, T_2)(T_3, T_4)$ $(T_5)(T_6, T_7, T_8)$
$\{(T_0, T_2), T_1, (T_3, T_4), (T_6, T_7, T_8)\}$	1	(T_0, T_2):80	-	80	(T_0, T_2)	$(T_0, T_2)(T_3, T_4)$ (T_6, T_7, T_8)
		T_1:20	-	20	(T_1)	(T_0, T_2)
		-	(T_3, T_4):30	50	(T_1, T_3, T_4)	$(T_1, T_3, T_4)(T_5)$ (T_6, T_7, T_8)
	2	(T_6, T_7, T_8):65	-	65	(T_6, T_7, T_8)	(T_0, T_2) $(T_1, T_3, T_4)(T_5)$ (T_6, T_7, T_8)
$\{(T_0, T_2), (T_1, T_3, T_4), T_5, (T_6, T_7, T_8)\}$	1	(T_0, T_2):80	-	80	(T_0, T_2)	(T_0, T_2) $(T_1, T_3, T_4)(T_5)$ (T_6, T_7, T_8)
		(T_1, T_3, T_4):50	-	50	(T_1, T_3, T_4)	(T_0, T_2) $(T_1, T_3, T_4)(T_5)$ (T_6, T_7, T_8)
	2	(T_5):30	-	30	(T_5)	(T_0, T_2) $(T_1, T_3, T_4)(T_5)$ (T_6, T_7, T_8)
		(T_6, T_7, T_8):65	-	65	(T_6, T_7, T_8)	(T_0, T_2) $(T_1, T_3, T_4)(T_5)$ (T_6, T_7, T_8)

Cuadro 3.6: Desarrollo de la composición de los clusters de tareas dentro de las etapas síncronas.

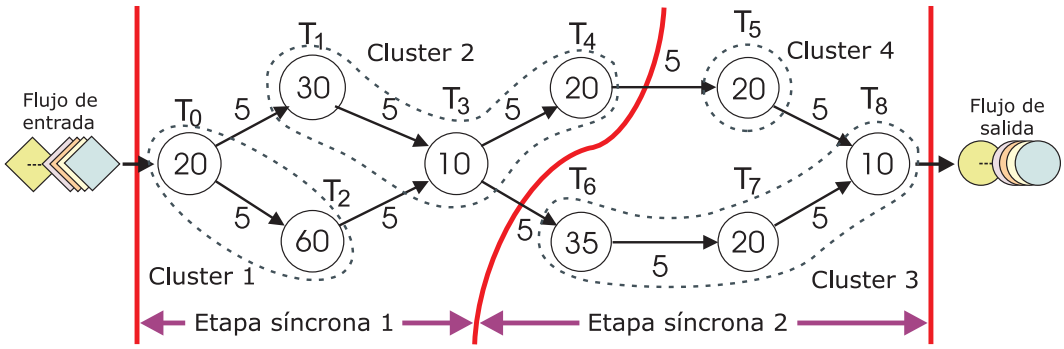


Figura 3.22: Asignación de tareas a clusters.

Paso 3 - Reducción del número de clusters y optimización de los recursos

En los pasos anteriores, se obtiene una asignación de tareas a procesadores que permite alcanzar los objetivos de rendimiento marcados. Esta asignación se ha creado basándose en la obtención de clusters de tareas, que pueden ser ejecutadas de forma concurrente en los diferentes nodos de procesamiento. Los clusters obtenidos, poseen exclusivamente tareas que se encuentran identificadas dentro de la misma etapa síncrona y bajo el criterio de que su tiempo de cómputo acumulado sea inferior o igual al del valor de IP a alcanzar. Estas dos restricciones provocan que puedan darse situaciones en las que el cómputo asignado a algún cluster no alcance el valor de IP definido, como puede observarse en la Figura 3.22 para los clusters 2, 3 y 4, lo que da lugar a que los nodos de procesamiento a los que son asignados tengan periodos de inactividad provocados por las dependencias existentes con respecto de otros clusters con cómputo acumulado mayor.

Si se da la situación de que los clusters con tiempo de cómputo menor al valor de IP son adyacentes y que la suma del tiempo de cómputo de ambos es menor o igual al valor de IP, éstos pueden ser agrupados formando un único cluster que consiga que las inactividades en los clusters originales se compensen entre si. Esta nueva agrupación a su vez permite una reducción en el número de nodos de procesamiento necesarios haciendo un mejor uso de ellos. Bajo esta idea actúa el algoritmo mostrado en el pseudo-código de la Figura 3.23.

Debido a que solo se pueden tener en cuenta aquellos clusters que son adyacentes entre sí, se hace necesario recorrer las líneas de ejecución para asegurar que las dependencias entre ellas se mantienen. El Cuadro 3.7 muestra el desarrollo del algoritmo,

```
1  Para cada línea de ejecución L ∈ Lista_líneas
2    CL_i=cluster inicial ∈ L
3    Mientras CL_i ≠ último cluster ∈ L
4      cómputo_acumulado=0
5      cluster=∅
6      Mientras ((cómputo_acumulado+μ(CL_i)) ≤ IP) y
                  (CL_i posee sucesores)
7        cómputo_acumulado=cómputo_acumulado + μ(CL_i)
8        cluster=cluster ∪ CL_i
9        CL_i=sucesor de CL_i
10     fin_mientras
11     mapping=mapping ∪ cluster
12   fin_mientras
13   actualizar la Lista_líneas con los nuevos clusters identificados
14 fin_para
```

Figura 3.23: Pseudo-código para la obtención de agrupaciones entre etapas síncronas adyacentes.

para la evaluación de las dos primeras líneas de ejecución, ya que la evaluación de las dos restantes no añaden ninguna modificación sobre el mapping que se obtiene.

Línea de ejecución	CL_i	$\mu(CL_i)$	cómputo acumulado	cluster
$\{(T_0,T_2),(T_1,T_3,T_4),$ $(T_6,T_7,T_8)\}$	(T_0,T_2)	80	80	(T_0,T_2)
	(T_1,T_3,T_4)	50	50	(T_1,T_3,T_4)
	(T_6,T_7,T_8)	65	115	(T_6,T_7,T_8)
$\{(T_0,T_2),(T_1,T_3,T_4),T_5,$ $(T_6,T_7,T_8)\}$	(T_0,T_2)	80	80	(T_0,T_2)
	(T_1,T_3,T_4)	50	50	(T_1,T_3,T_4)
	T_5	30	80	(T_1,T_3,T_4,T_5)
	(T_6,T_7,T_8)	65	145	
	(T_6,T_7,T_8)	65	65	(T_6,T_7,T_8)

Cuadro 3.7: Desarrollo del paso 3 para las dos primeras líneas de ejecución de la lista.

Se puede observar, que al evaluar la segunda línea de ejecución, es posible combinar los clusters 2 y 4 que contienen a las tareas (T_1,T_3,T_4) y (T_5) respectivamente, dando lugar al cluster (T_1,T_3,T_4,T_5). Al realizar esta unión, se reduce en una unidad el número de nodos de procesamiento necesarios. El resultado final del conjunto de clusters obtenidos se muestra en la Figura 3.24 sobre el grafo de la aplicación.

En la Figura 3.25(a) se muestra el grafo que representa las dependencias entre los clusters creados, y en la Figura 3.25(b) el mapping resultante.

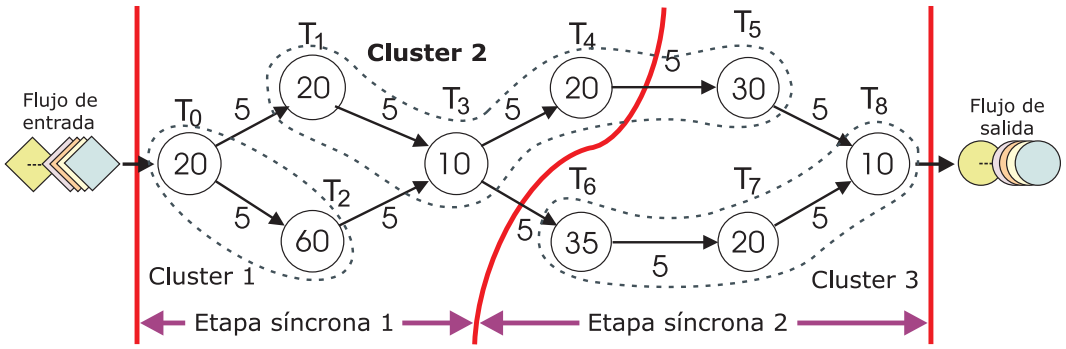


Figura 3.24: Agrupación de clusters entre etapas síncronas adyacentes.

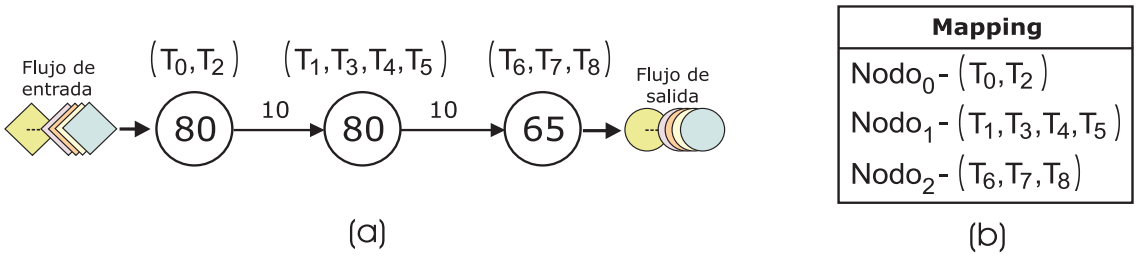


Figura 3.25: (a) Grafo que representa las dependencias entre los clusters de tareas obtenidos. (b) Mapping obtenido con la heurística MPASS.

En la Figura 3.26 se muestra la ejecución de la aplicación para el procesamiento de dos datos del flujo de entrada, utilizando el mapping indicado. El valor de la latencia que se consigue es de 165 unidades de tiempo, el cual solo excede en 10 unidades respecto al tiempo de cómputo acumulado del camino crítico de la aplicación, formado por las tareas $\{T_0, T_2, T_3, T_6, T_7, T_8\}$, el cual marca la cota mínima de latencia para este ejemplo. Además, a partir de la obtención del primer resultado, el ratio en el procesamiento de los datos corresponde a uno cada 80 unidades de tiempo tal y como se deseaba.

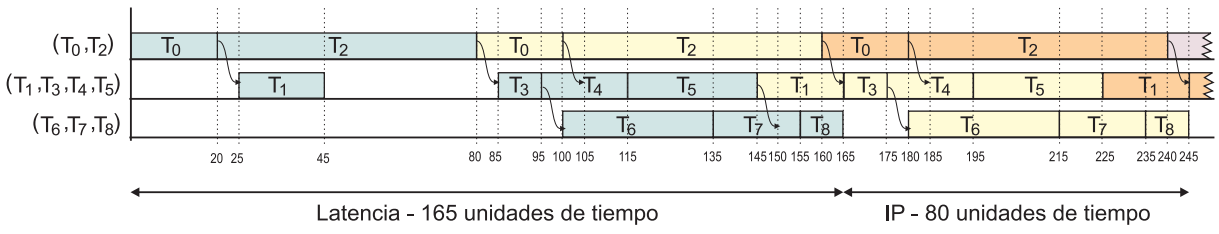


Figura 3.26: Diagrama de Gantt del procesamiento de dos datos con el mapping obtenido.

3.2.2. Complejidad de la heurística

La complejidad de la heurística está directamente relacionada con el número de líneas de ejecución que hay que evaluar. Como se ha comentado en secciones previas, dicho número depende directamente de la propia estructura de dependencias de la aplicación.

Por lo general las aplicaciones pipeline suelen poseer un patrón de dependencias estructurado, en el que las tareas se distribuyen de forma regular en etapas, de forma que solo existen dependencias entre aquellas tareas que se encuentran en etapas adyacentes [CkLW+00][SV00][LLP01][DA05]. Teniendo en cuenta esta característica, el número de líneas de ejecución depende en gran medida en como se distribuyen las tareas en las etapas. El menor número posible de etapas sea cual sea el número de tareas es de tres; la primera etapa en la que se encontrará la tarea inicial, una segunda etapa con la totalidad de tareas restantes, exceptuando la última tarea, que se ubicará en la tercera y última etapa. Para este tipo de estructura, el número total de líneas de ejecución posibles es de $t-2$, siendo t el número total de tareas de la aplicación, tal y como se muestra en la Figura 3.27(a).

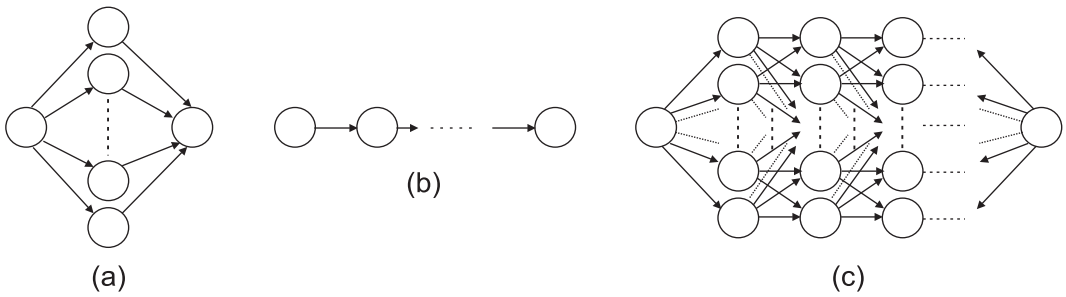


Figura 3.27: Combinaciones de distribución de tareas en etapas.

Por el contrario, cuando cada tarea forma por si misma una etapa, sólo existe una única línea de ejecución, situación que se representa en la Figura 3.27(b). Finalmente, el caso más complejo es aquel en el que las tareas se distribuyen de forma homogénea entre las etapas, existiendo una total dependencia entre cada una de las tareas presentes en etapas adyacentes, como se refleja en la Figura 3.27(c). En esta situación siendo e el número total de etapas, cada una de ellas tendrá asignada $(t-2)/(e-2)$ tareas, y el número total de líneas de ejecución se puede obtener a partir de la expresión:

$$número\_de\_líneas\_de\_ejecución = \left(\frac{t-2}{e-2}\right)^{e-2} \tag{3.2}$$

En la Figura 3.28(a) y (b) se muestra de forma gráfica dos casos en la evaluación de la expresión anterior variando el número de tareas y de etapas, el primer caso en el rango $[3, 10]$ y el segundo en el rango $[3, 50]$.

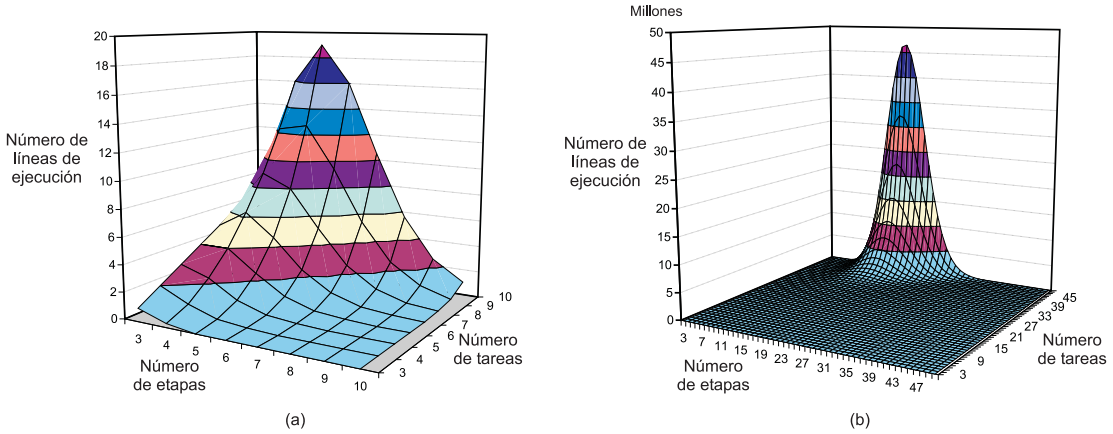


Figura 3.28: (a) Rango de tareas y etapas $[3, 10]$. (b) Rango de tareas y etapas $[3, 50]$.

Observando el primer caso , Figura 3.28(a), el valor máximo de líneas de ejecución se obtiene en la distribución de 5 etapas, dos etapas reservadas para las tareas inicial y final, y tres que albergan 3, 3 y 2 tareas respectivamente, dando lugar a un total de 18 líneas de ejecución.

En la Figura 3.28(b), en el que el número de tareas se ha ampliado a 50 el valor máximo se alcanza para 20 etapas, siendo el número de líneas de ejecución del orden de $46{,}5 \times 10^6$. Hay que destacar que sólo un número muy reducido de combinaciones de tareas/etapas realmente alcanza valores tan extremos. La Figura 3.29 muestra la distribución del número de líneas de ejecución. Para este ejemplo un 63 % de todas las posibles combinaciones genera un número inferior al centenar de líneas de ejecución y solo un 7 % es superior a 10^6. Se ha de tener presente que esta última situación representa el caso más extremo posible en el patrón de dependencias de una aplicación pipeline y que por lo general no representa a la mayoría de éstas, las cuales suelen poseer una estructura de dependencias mucho más simple que reduce considerablemente el número total de líneas de ejecución.

La heurística está formada por tres pasos. En estos se evalúan todas las líneas de ejecución pasando por cada una de sus tareas. En una distribución equitativa de tareas en etapas como la mostrada en al Figura 3.27(c), que corresponde al caso más desfavorable, y basándose en la expresión (3.2), el número de evaluaciones que realiza el algoritmo en cada paso será:

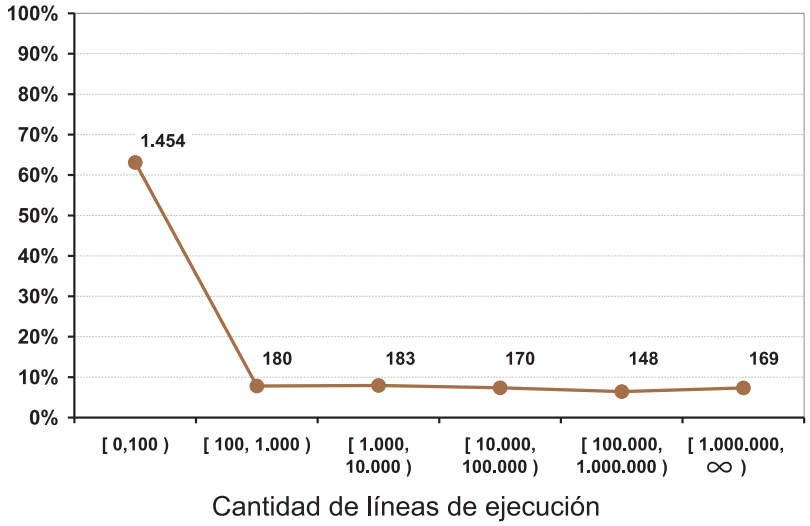


Figura 3.29: Porcentaje en la distribución de la cantidad de líneas de ejecución y el número de éstas para el caso de la Figura 3.28(b).

$$evaluaciones\_en\_cada\_paso = e \times \left(\frac{t-2}{e-2}\right)^{e-2} \cong e \times \left(\frac{t}{e}\right)^{e-2} \cong \frac{t^{e-2}}{e^{e-1}} \tag{3.3}$$

Finalmente, este número de evaluaciones se realiza tres veces, una para cada paso dando lugar a una complejidad de $O\left(3 \times \frac{t^{e-2}}{e^{e-1}}\right)$. Esta expresión depende en gran medida de la estructura de dependencias de la aplicación. Así el caso de menor complejidad es aquel en el que cada tarea se encuentre ubicada en una etapa, dando lugar a una complejidad $O(3t) \cong O(t)$, muy inferior a heurísticas que presuponen esta estructura para la aplicación pipeline [SV00].

3.2.3. Heurística de mapping - MPART (*Mapping of Pipeline Applications based on Reduced Tree*)

En esta sección se presenta una heurística de menor complejidad basada también en el concepto de etapas síncronas. La idea toma como base el que una aplicación pipeline con un elevado grado de dependencias entre sus tareas, posee un gran número de líneas de ejecución que comparten entre sí múltiples de estas tareas, por lo que la evaluación de todas las líneas de ejecución puede incurrir en redundancia al procesarlas.

Esta situación queda de manifiesto en el ejemplo de la Figura 3.18 utilizado en la heurística previa. En él, las cuatro líneas de ejecución existentes se crean a partir de

combinaciones de las sublíneas $\{T_0, T_1, T_3\}$, $\{T_0, T_2, T_3\}$, $\{T_4, T_5, T_8\}$ y $\{T_6, T_7, T_8\}$. De esta forma, si en la valoración de una de las líneas de ejecución más prioritarias se ha determinado que se debe crear el cluster (T_6, T_7, T_8), volver a evaluar de nuevo esta sublínea puede que no aporte grandes beneficios en la evaluación del resto de líneas de ejecución. Este conocimiento, es posible utilizarlo en el momento de la obtención de las líneas de ejecución y creación de las etapas síncronas, reduciendo de esta forma el número total de evaluaciones a realizar.

Para desarrollar la nueva heurística MPART, se utiliza una estructura de árbol que representa las dependencias entre tareas y en la que cada tarea aparece como nodo hijo de una de sus tareas predecesoras. En esta estructura, una tarea solo puede aparecer en un único nodo del árbol, motivo que da lugar a que en la creación del mismo no se tenga presente todas las posibles líneas de ejecución existentes en la aplicación. Una vez que el árbol ha sido creado, existe una única rama que lleva desde la tarea inicial, nodo raíz del árbol, a la tarea final correspondiente a un nodo hoja del mismo. La creación de esta estructura se realiza con el objetivo de que esta rama represente la línea de ejecución más prioritaria a evaluar según el criterio escogido para el mapping. El resto de tareas se irán distribuyendo en los nodos del árbol dando lugar a las ramas que representan las sublíneas presentes en el resto de líneas de ejecución eliminado su repetición.

Debido a que cada tarea aparece en el árbol exclusivamente una vez, no es posible representar todas las dependencias que estas tareas puedan tener. Esta situación da lugar a que no se evalúen agrupaciones que se hubiera podido llevar a cabo si se hubieran tenido en cuenta. Por esta razón, y para poder tener presente cada una de las dependencias de las tareas cuyo grado de convergencia es mayor a uno, la heurística MPART se ha definido a partir de un proceso iterativo, en el que cada nueva iteración tenga presente la elección de una nueva tarea predecesora para aquellas en las que sea posible hacerlo. De esta forma el número máximo de iteraciones de la heurística no viene determinado por la cantidad de líneas de ejecución si no que viene acotado por el mayor grado de convergencia presente en las tareas de la aplicación pipeline.

Para poder realizar cada nueva iteración, se define para cada tarea T_i con grado de convergencia mayor a uno la lista $L_{pred}(T_i)$, en la que se indica cada tarea T_j que es predecesora suya. En cada nueva iteración se tomará como tarea predecesora una tarea diferente de esta lista. En el caso de que una tarea T_k haya sido evaluada en todas sus tareas predecesoras, y se requiera realizar nuevas iteraciones, se tomará la tarea predecesora utilizada en la primera iteración.

Los pasos principales en la heurística son los siguientes:

Repetir Pasos 1 y 2 hasta evaluar todas las dependencias entre tareas

Paso 1 - Creación del árbol de dependencias e identificación de las etapas síncronas.

Paso 2 - Asignación de tareas a clusters dentro de una misma etapa síncrona.

Paso 3 - Reducción del número de clusters y optimización de los recursos.

A continuación se explica en detalle cada una de los pasos.

Paso 1 - Creación del árbol de dependencias e identificación de las etapas síncronas

El primer paso se inicia con la creación del árbol que representa las dependencias existentes entre las tareas. En este árbol los nodos son las tareas de la aplicación, siendo el nodo raíz la tarea inicial. Las ramas del árbol se abren a partir de las tareas sucesoras de cada nodo, de forma que cada tarea T_i aparece una única vez en el árbol y como nodo hijo de un único nodo padre. En la situación en la que sea posible para una tarea T_i la elección de más de una tarea predecesora, se decidirá ésta mediante el uso de una función de peso, $w(T_i)$, definida a partir del criterio escogido para el mapping como se indica a continuación:

- *Minimizar latencia*. La función de peso se define como la suma del tiempo de cómputo y del coste de las comunicaciones del conjunto de las tareas en la rama que va, desde la tarea representada en el nodo raíz hasta la posible ubicación del nodo. En el caso de tener que escoger entre más de una posible ubicación, se elegirá aquella que maximice el valor del peso.

$$w(T_i) = \max_{\forall T_j \ padre \ de \ T_i} (w(T_j) + \mu(T_i) + com(T_j, T_i))$$

Siguiendo este criterio, la rama que une el nodo raíz y el nodo terminal, que representan a la tarea inicial y final de la aplicación respectivamente, corresponde a la línea de ejecución con mayor tiempo de cómputo y de comunicación acumulado.

- *Obtener una productividad determinada.* En este caso, el peso de las tareas se obtiene exclusivamente a partir de la suma de los tiempos de cómputo de las tareas representadas en la misma rama. Así la elección de una tarea predecesora, se basará en aquel que minimice el valor del peso.

$$w(T_i) = \min_{\forall T_j \ padre \ de \ T_i} (w(T_j) + \mu(T_i))$$

  En este caso la línea de ejecución representada en la rama que une la tarea inicial con la tarea final de la aplicación es la que posee un menor tiempo de cómputo acumulado.

Es posible que al suponer una estructura arbitraria para las aplicaciones pipeline, la evaluación de la función de peso coincida dando el mismo valor en algunas situaciones en las que se deba escoger entre varios nodos padre. Ante esta situación se escogerá como tarea predecesora, aquella que posea una menor profundidad, entendiéndose ésta, como el menor número de nodos desde el nodo raíz hasta ella. Si aún así el empate persiste, la elección podrá ser tomada de forma arbitraria.

En la Figura 3.30(a) y 3.30(b), se muestran los árboles que se obtienen aplicando los criterios de latencia y productividad sobre la aplicación pipeline de la Figura 3.18(a). En cada nodo/tarea aparece representado su cómputo individual, junto con el valor de la función de peso obtenido ($\mu(T_i)/w(T_i)$), también se ha resaltado los nodos que representan a la tarea inicial y final de la aplicación. Se puede observar que la rama que contiene a ambas tareas coincide con la línea de ejecución prioritaria definida en la heurística MPASS para los mismos criterios en el mapping, Figura 3.18(b).

Como se ha comentado anteriormente, la heurística permite realizar varias iteraciones en las que las tareas T_i, que tienen múltiples predecesoras puedan variar la tarea escogida como nodo padre de la lista $L_{pred}(T_i)$. Así en el caso de la Figura 3.30(a), con criterio basado en la latencia, la tarea T_3, que posee como lista $L_{pred}(T_3) = \{T_1, T_2\}$, y para la que en esta primera iteración se ha escogido como tarea predecesora T_2, en la próxima iteración sera la tarea T_1 la escogida como tarea predecesora. Para el caso de la Figura 3.30(b) con criterio basado en la productividad, la primera tarea predecesora escogida corresponde a la tarea T_1, siendo la próxima elección la tarea T_2. De esta forma se tiene presente, en las sucesivas iteraciones, las diferentes líneas de ejecución sobre las que aparece cada tarea.

En la Figura 3.31 se muestra el pseudo-código del algoritmo encargado de obtener el árbol de dependencias de la aplicación. Para facilitar el seguimiento del algoritmo se

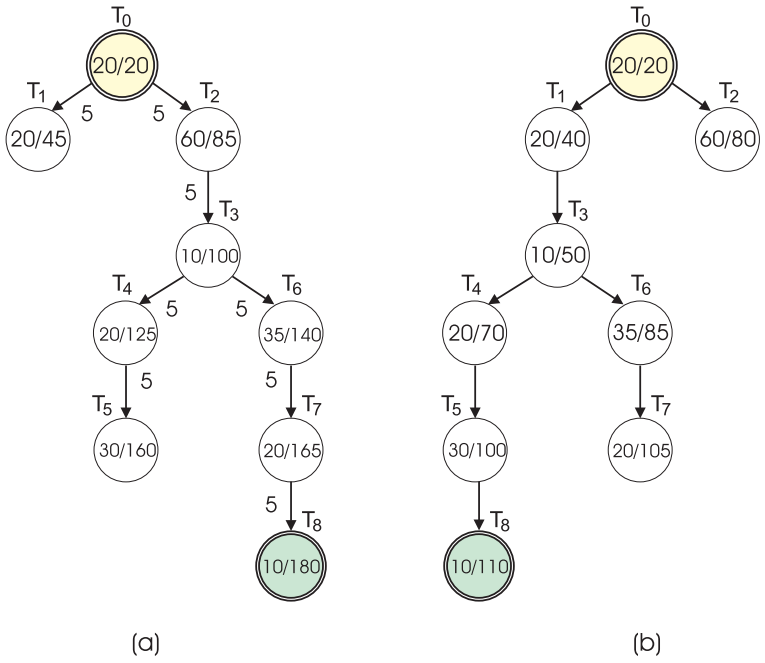


Figura 3.30: Árbol de dependencias basado en: (a) criterio de latencia y (b) criterio de productividad.

ha incluido el conjunto de tareas sucesoras $L_{suc}(T_p)$ para la tarea predecesora escogida.

El Cuadro 3.8 muestra el desarrollo del algoritmo, para la obtención del árbol en la primera iteración de la heurística MPART, utilizando como criterio el de minimización de la latencia.

Una vez creado el árbol se procede a la creación de las etapas síncronas identificando aquellas tareas que forman parte de ella. Aunque esta parte se muestra por separado para facilitar la comprensión de la heurística, puede llevarse a cabo a medida que se crea el árbol conjuntamente con el algoritmo de la Figura 3.31.

A cada tarea T_i se le añade un nuevo valor denominado *cómputo acumulado*, $cómp\_acum(T_i)$, que se obtiene exclusivamente a partir de la suma de los tiempos de cómputo de las tareas presentes en la rama en la va desde el nodo raíz del árbol hasta ella. Debido a que el cómputo acumulado no incluye la información correspondiente al coste de las comunicaciones, éste no tiene porqué coincidir con el valor del peso $w(T_i)$.

Con la información que nos proporciona el valor de $cómp\_acum(T_i)$, para un valor de IP en la definición de las etapas síncronas, una tarea T_i pertenecerá a la etapa síncrona j, si $cómp\_acum(T_i)$ está dentro del intervalo $((j-1) \times IP, j \times IP]$.

```
1  Para cada tarea T_i ∈ Aplicación
2    Si T_i es la tarea inicial entonces
3      w(T_i) = μ(T_i)
4     si no
5      Si L_pred(T_i) = ∅ entonces
6        L_pred(T_i) = T_predecesora(T_i) utilizada en la primera iteración de la heurística
7      fin_si
8      Si el criterio es minimizar latencia entonces
9        Sea T_p tal que w(T_i) = máx_{T_p ∈ L_pred(T_i)} (w(T_p) + μ(T_i) + com(T_p, T_i))
10       w(T_i) = w(T_p) + μ(T_i) + com(T_p, T_i)
11      si no
12       Sea T_p tal que w(T_i) = mín_{T_p ∈ L_pred(T_i)} (w(T_p) + μ(T_i))
13       w(T_i) = w(T_p) + μ(T_i)
14     fin_si
15     L_pred(T_i) = L_pred(T_i) − T_p
16     L_suc(T_p)=L_suc(T_p) ∪ {T_i}
17   fin_si
18 fin_para
```

Figura 3.31: Pseudo-código para la obtención del árbol de dependencias de la aplicación.

T_i:$\mu(T_i)$	$L_{pred}(T_i)$	$T_p : w(T_p)$	$w(T_p) + \mu(T_i)$+ $+com(T_p, T_i)$	$w(T_i)$	$L_{suc}(T_p)$
T_0:20	-	-	-	20	-
T_1:20	$\{T_0\}$	T_0:20	45	45	$\{T_1\}$
T_2:60	$\{T_0\}$	T_0:20	85	85	$\{T_1, T_2\}$
T_3:10	$\{T_1, T_2\}$	T_1:45 T_2:85	60 **100**	100	$\{T_3\}$
T_4:20	$\{T_3\}$	T_3:100	125	125	$\{T_4\}$
T_5:30	$\{T_4\}$	T_4:130	160	160	$\{T_5\}$
T_6:35	$\{T_3\}$	T_3:100	140	140	$\{T_4, T_6\}$
T_7:20	$\{T_6\}$	T_6:140	165	165	$\{T_7\}$
T_8:10	$\{T_5, T_7\}$	T_5:160 T_7:165	175 **180**	180	$\{T_8\}$

Cuadro 3.8: Desarrollo de la creación del árbol de precedencias basándose en el criterio de minimización de latencia para la primera iteración de la heurística MPART.

En la Figura 3.32 se muestra el pseudo-código del algoritmo define para cada tareas la etapa síncrona a la que pertenece y en el Cuadro 3.9 el resultado obtenido tras aplicarlo sobre el árbol mostrado en la Figura 3.30(a).

```
1  Para cada tarea T_i ∈ Aplicación
2    etapa_T_i = ⌈ cómp_acum(T_i) / IP ⌉
3    etapa_síncrona(etapa_T_i) = etapa_síncrona(etapa_T_i) ∪ {T_i}
4  fin_para
```

Figura 3.32: Asignación de las tareas a las etapas síncronas.

$T_i : cómp\_acum(T_i)$	$\left\lceil \frac{cómp\_acum(T_i)}{IP} \right\rceil = etapa\_T_i$	Etapas Síncronas
$T_0 : 20$	$\lceil 20/80 \rceil = 1$	$etapa\_síncrona(1) = \{T_0\}$
$T_1 : 40$	$\lceil 40/80 \rceil = 1$	$etapa\_síncrona(1) = \{T_0, T_1\}$
$T_2 : 60$	$\lceil 60/80 \rceil = 1$	$etapa\_síncrona(1) = \{T_0, T_1, T_2\}$
$T_3 : 90$	$\lceil 90/80 \rceil = 2$	$etapa\_síncrona(1) = \{T_0, T_1, T_2\}$ $etapa\_síncrona(2)$ - $\{T_3\}$
$T_4 : 110$	$\lceil 110/80 \rceil = 2$	$etapa\_síncrona(1) = \{T_0, T_1, T_2\}$ $etapa\_síncrona(2)$ - $\{T_3, T_4\}$
$T_5 : 140$	$\lceil 140/80 \rceil = 2$	$etapa\_síncrona(1) = \{T_0, T_1, T_2\}$ $etapa\_síncrona(2)$ - $\{T_3, T_4, T_5\}$
$T_6 : 125$	$\lceil 125/80 \rceil = 2$	$etapa\_síncrona(1) = \{T_0, T_1, T_2\}$ $etapa\_síncrona(2) = \{T_3, T_4, T_5, T_6\}$
$T_7 : 145$	$\lceil 145/80 \rceil = 2$	$etapa\_síncrona(1) = \{T_0, T_1, T_2\}$ $etapa\_síncrona(2) = \{T_3, T_4, T_5, T_6, T_7\}$
$T_8 : 155$	$\lceil 155/80 \rceil = 2$	$etapa\_síncrona(1) = \{T_0, T_1, T_2\}$ $etapa\_síncrona(2) = \{T_3, T_4, T_5, T_6, T_7, T_8\}$

Cuadro 3.9: Desarrollo del algoritmo de asignación de las tareas a las etapas síncronas con valor de IP de 80 unidades de tiempo.

En el ejemplo escogido, el conjunto de tareas asignadas a las etapas síncronas no coincide con el de la heurística MPASS. Esta situación se debe a que en la heurística MPART no se realiza un estudio exhaustivo de todas las líneas de ejecución, lo cual aseguraría que las tareas formen parte de las etapas síncronas de menor índice. Si no

que por el contrario, evalúa las tareas bajo la perspectiva de la función de peso, que no tiene presente la estructura de dependencias global de la aplicación.

En la Figura 3.33 se muestra gráficamente la asignación de las tareas a las etapas síncronas obtenida del Cuadro 3.9. Al lado de cada tarea se ha añadido entre paréntesis el valor del cómputo acumulado correspondiente a cada una de ellas.

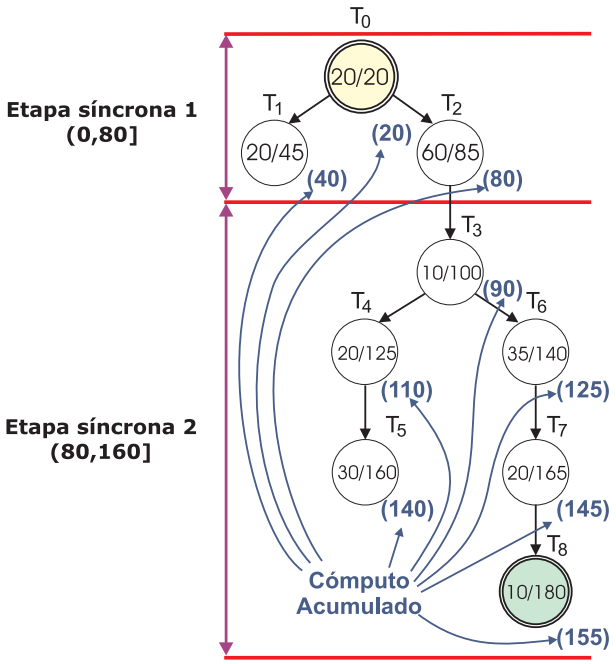


Figura 3.33: Etapas síncronas obtenidas y cómputo acumulado de cada tarea.

Paso 2 - Asignación de tareas a clusters dentro de una misma etapa síncrona

En este paso se procede a definir los clusters de tareas, que serán asignados a cada nodo de procesamiento. Para ello se recorren las diferentes ramas, acotando el análisis a las tareas que forman parte de la misma etapa síncrona y cuya suma de su tiempo de cómputo es menor o igual al del valor de IP a alcanzar. Se ha de tener presente que inicialmente cada tarea por si misma forma un cluster al que pertenece ella sola.

Debido a que la heurística MPART realizará varias iteraciones, teniendo presente las diferentes opciones a la hora de escoger las tareas predecesoras, presentes en las listas $L_{pred}(T_i)$, algunas tareas cambiarán de ubicación en la estructura de árbol que se generará. Esto se debe tener en cuenta, evitando que una tarea que forma parte de un cluster definido en iteraciones previas, cuyo valor de cómputo acumulado de todas sus tareas ha alcanzado el valor deseado, se reagrupe en un nuevo cluster. Por este

motivo, a las tareas se les asocia el valor del cómputo del cluster del que forman parte, $cómp\_CL(T_i)$, el cual es utilizado para determinar si podrá o no agruparse con otras tareas en las sucesivas iteraciones.

En el desarrollo de este paso se hace un recorrido de las ramas del árbol, comenzando desde los nodos hoja hacia el nodo raíz del árbol, definiendo el orden a la hora de escoger la rama a analizar a partir de una lista, denominada L_{tareas}, en la que aparecen ordenadas las tareas según los siguientes criterios:

1. La primera tarea es la última tarea de la aplicación. De esta forma se asegura que la primera rama que se analiza corresponde a la línea de ejecución prioritaria para el mapping escogido.

2. El resto de tareas T_i se ordenan según su peso $w(T_i)$, de forma que si el criterio de la heurística es minimizar la latencia, dicho orden será de mayor a menor. Por el contrario si el criterio es el de productividad, este orden será de menor a mayor.

Basándose en estos criterios, y para el ejemplo utilizado la lista L_{tareas} es $\{T_8, T_5, T_1\}$. El pseudo-código que representa este paso se muestra en la Figura 3.34.

El Cuadro 3.10 muestra el algoritmo desarrollado para el ejemplo suponiendo un valor para IP de 80 unidades de tiempo.

$T_i : cómp\_CL(T_i)$	$Etapa\ Síncrona(T_i)$	$T_j : cómp\_CL(T_j)$	$Etapa\ Síncrona(T_j)$	CL_i inicial	CL_j inicial	CL_i final
$T_8 : 10$	2	$T_7 : 20$	2	$\{T_8\}$	$\{T_7\}$	$\{T_7, T_8\}$
$T_7 : 30$	2	$T_6 : 35$	2	$\{T_6\}$	$\{T_7, T_8\}$	$\{T_6, T_7, T_8\}$
$T_6 : 65$	2	$T_3 : 10$	2	$\{T_3\}$	$\{T_6, T_7, T_8\}$	$\{T_3, T_6, T_7, T_8\}$
$T_2 : 60$	1	$T_0 : 20$	1	$\{T_2\}$	$\{T_0\}$	$\{T_0, T_2\}$
$T_0 : 20$	1	NULL	-	$\{T_0, T_2\}$	-	$\{T_0, T_2\}$
$T_5 : 30$	2	$T_4 : 10$	2	$\{T_5\}$	$\{T_4\}$	$\{T_4, T_5\}$
$T_3 : 75$	2	$T_4 : 40$	2	$\{T_3, T_6, T_7, T_8\}$	$\{T_4, T_5\}$	$\{T_4, T_5\}$
$T_4 : 10$	2	$T_2 : 15$	1	$\{T_4\}$	$\{T_2\}$	$\{T_4\}$
$T_1 : 20$	1	$T_0 : 80$	1	$\{T_1\}$	$\{T_0, T_2\}$	$\{T_1\}$
$T_0 : 80$	1	NULL	-	$\{T_0, T_2\}$	-	$\{T_0, T_2\}$

Cuadro 3.10: Desarrollo de la creación de clusters de tareas con un valor IP de 80 unidades de tiempo.

```
1  Sea L_tareas la lista de tareas terminales del árbol ordenado por w(T_i) según
   el criterio de optimización
2  Para cada tarea T_i ∈ L_tareas
3   Sea T_j la tarea predecesora de T_i
4   Mientras T_j ≠ NULL hacer
5      Si (etapa_síncrona(T_i) == etapa_síncrona(T_j)) entonces
6         Si (cómp_CL(T_i) + cómp_CL(T_j)) ≤ IP entonces
7            Sea CL_i el cluster tal que T_i ∈ CL_i
8            Sea CL_j el cluster tal que T_j ∈ CL_j
9            Para cada tarea T_h ∈ CL_i hacer
10              cóm_CL(T_h) = cóm_CL(T_h) + cóm_CL(T_j)
11           fin_para
12           Para cada tarea T_h ∈ CL_j hacer
13              cóm_CL(T_h) = cóm_CL(T_h) + cóm_CL(T_i)
14           fin_para
15           CL_i = CL_i ∪ CL_j
16        fin_si
17     fin_si
18     T_i = T_j
19     Sea T_j la tarea predecesora de T_i
20  fin_mientras
21  Sea T_i la siguiente tarea de L_tareas
22 fin_para
```

Figura 3.34: Pseudo-código para la creación de clusters de tareas dentro de las etapas síncronas.

Los clusters obtenidos siempre se mantienen dentro de la misma etapa síncrona, por lo que la tarea T_1, que no puede agruparse con ninguna otra tarea, forma por sí misma un cluster. En la Figura 3.35 se muestra de forma gráfica los clusters de tareas obtenidos en este paso y entre paréntesis al lado de cada nodo el valor de $cómp\_CL(T_i)$ de cada tarea.

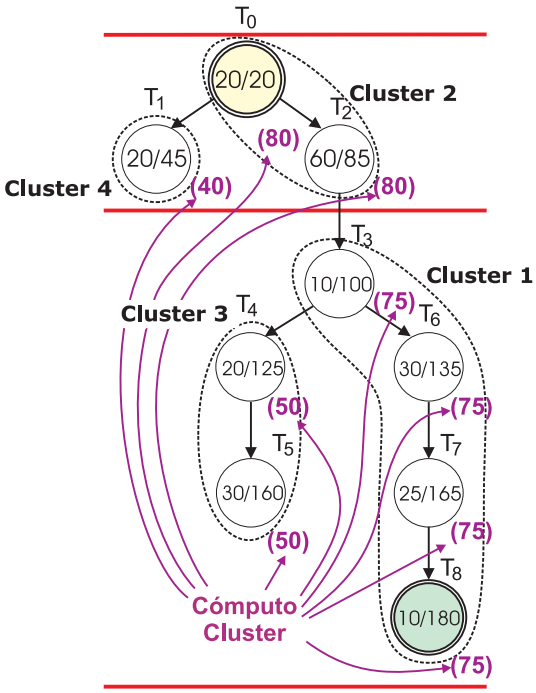


Figura 3.35: Asignación de tareas a clusters y cómputo asociado al cluster de cada uno de ellos.

Tras este último paso, ya se tiene una primera asignación de tareas a nodos de procesadores que es refinada repitiendo los pasos anteriores en los que se actualiza la ubicación de las tareas que tienen más de una tarea predecesora. Sobre el mismo ejemplo, el nuevo árbol de dependencias que se obtiene para la siguiente iteración se muestra en la Figura 3.36. En él las tareas T_3 y T_8 cambian su ubicación pasando a ser nodos hijos de las tareas T_1 y T_5 respectivamente. En la figura se ha incluido la distribución de las etapas síncronas que se obtiene en esta iteración, así como para cada tarea T_i el valor de cómputo acumulado, $cómp\_acum(T_i)$ y del cómputo del cluster al que pertenece, obtenido de la iteración previa $cómp\_CL(T_i)$. También aparece indicado mediante líneas discontinuas los clusters obtenidos en la iteración previa.

A la hora de proceder con la segunda iteración, no se realiza ningún cambio respecto a las agrupaciones ya existentes, ya que las tareas que podrían ser candidatas a agruparse, poseen un coste asociado para $cómp\_acum(T_i)$, que lo impide. Esto situación

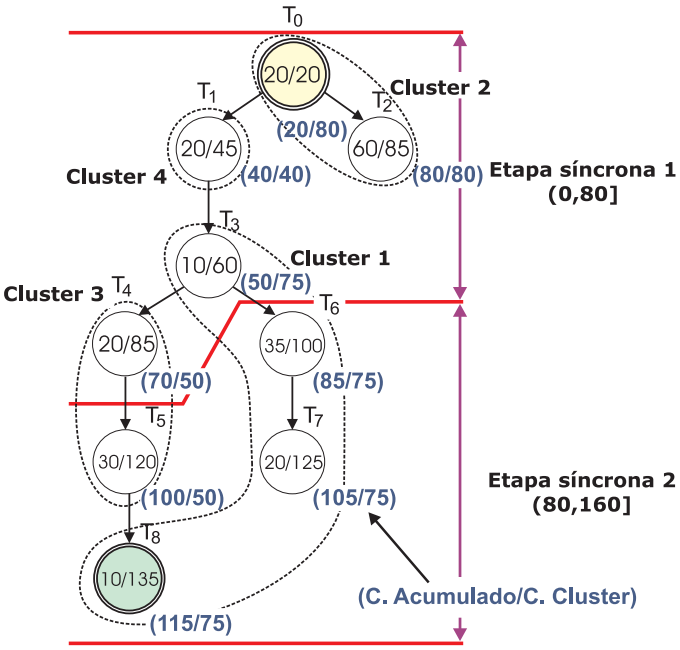


Figura 3.36: Árbol de dependencias obtenido en una segunda evaluación.

que esta heurística es posible alcanzar la asignación final sin tener que realizar todas las posibles iteraciones.

Paso 3 - Reducción del número de clusters y optimización de los recursos

Tras haber realizado todas las iteraciones, es posible al igual que en la heurística previa, intentar reducir el número de clusters y en consecuencia el número de nodos de procesamiento, agrupando aquellos clusters que se encuentran en la frontera de las etapas síncronas y cuya suma de su tiempo de cómputo no exceda el valor de IP a alcanzar.

El pseudo-código que implementa este paso se muestra en la Figura 3.37.

La forma de proceder es similar a la del paso 2, pero sin tener en cuenta las fronteras entre etapas síncronas. En el caso del ejemplo utilizado, este paso no obtiene ninguna modificación en las agrupaciones. La Figura 3.38(a), se muestran las dependencias entre los clusters obtenidos y en la Figura 3.38(b) el mapping a realizar, el cual como se puede observar difiere del generado por la heurística MPASS, requiriendo éste un nodo de procesamiento más.

En la Figura 3.39 se muestra el diagrama de Gantt de la asignación obtenida en el procesamiento de dos datos del flujo de entrada. El resultado de latencia y de pro-

```
1  Sea L_tareas la lista de tareas terminales del árbol ordenado por w(T_i) según
   el criterio de optimización
2  Para cada tarea T_i ∈ L_tareas
3   Sea T_j la tarea predecesora de T_i
4   Mientras T_j ≠ NULL hacer
5      Si (cómp_CL(T_i) + cómp_CL(T_j)) ≤ IP entonces
6         Sea CL_i el cluster al que pertenece T_i
7         Sea CL_j el cluster al que pertenece T_j
8         Para cada tarea T_h ∈ CL_i hacer
9           cóm_CL(T_h) = cóm_CL(T_h) + cóm_CL(T_j)
10        fin_para
11        Para cada tarea N_h ∈ CL_j hacer
12          cóm_CL(T_h) = cóm_CL(T_h) + cóm_CL(T_i)
13        fin_para
14        CL_i = CL_i ∪ CL_j
15     fin_si
16     T_i = T_j
17     Sea T_j la tarea predecesora de T_i
18  fin_mientras
19  Sea T_i la siguiente tarea de L_tareas
20 fin_para
```

Figura 3.37: Pseudo-código para la creación de clusters de tareas dentro de las etapas síncronas.

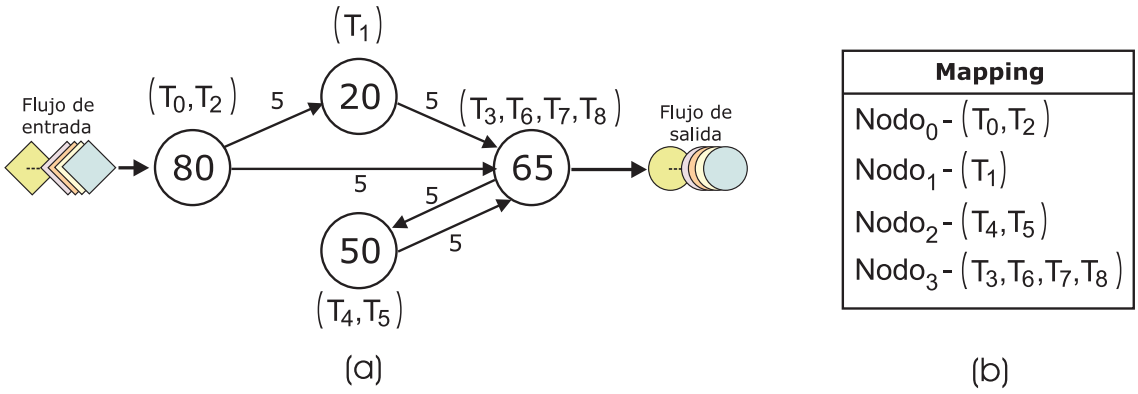


Figura 3.38: (a) Grafo que representa los clusters de tareas obtenidos. (b) Mapping obtenido.

ductividad cumple con los requisitos fijados, aunque por el contrario, la utilización de los nodos de procesamiento es inferior al de la heurística MPASS, al tener éstos un mayor tiempo promedio de inactividad. Esta situación viene como consecuencia de que en el proceso de creación de las agrupaciones, no se ha tenido en cuenta la estructura de dependencias global de la aplicación y no ha sido capaz de identificar de forma tan rigurosa las etapas síncronas a las que pertenecen las tareas.

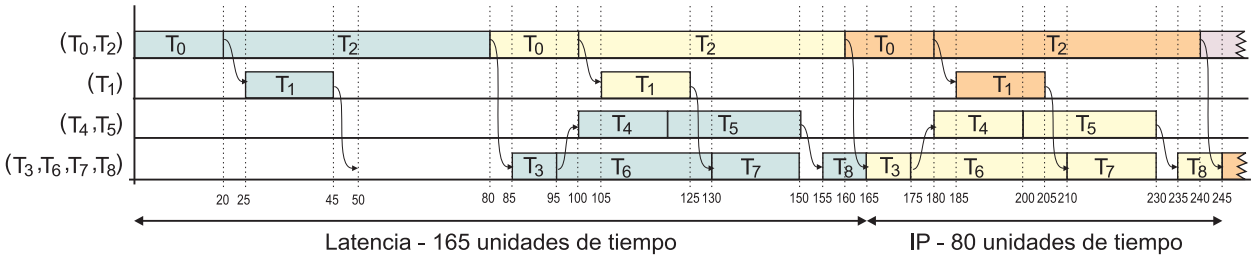


Figura 3.39: Diagrama de Gantt del procesamiento de dos datos con el mapping obtenido por la heurística MPART.

3.2.4. Complejidad de la heurística

Para determinar la complejidad de esta heurística se ha de evaluar de forma individual cada unos de los pasos que intervienen en ella.

El Paso 1, evalúa cada una de las tareas que forman parte de la aplicación y para cada una de ellas, todas sus predecesoras para determinar cual de éstas será tomada como nodo padre dentro del árbol, así como identificar a que etapa síncrona pertenecen. El caso más desfavorable en este paso se da cuando, suponiendo que las tareas se distribuyen de forma homogénea en las diferentes etapas de cómputo, todas ellas poseen precedencias con las tareas de la etapa previa. Suponemos que, siendo el número total de tareas de la aplicación t, y el número de etapas e, el número de tareas distribuidas de forma uniforme en cada etapa es $\left(\frac{t-2}{e-2}\right) \cong \left(\frac{t}{e}\right)$. De esta forma en este paso se deben evaluar para cada etapa, todas las posibles sucesoras de cada tarea, dando lugar a una complejidad de $O(e \times \left(\frac{t}{e}\right)^2) = O\left(\frac{t^2}{e}\right)$.

El Paso 2, evalúa cada nodo del árbol y sus nodos padres en la misma etapa síncrona resiguiendo una rama determinada, con el fin de determinar la posibilidad de agrupar o no las tareas a las que representan. Cada nodo, posee un único nodo padre, por lo que no existe ambigüedad a la hora de escoger los nodos a evaluar ya que coinciden con el número de tareas de la aplicación. Así la complejidad es del orden $O(t)$.

Finalmente, el Paso 3 repite las mismas acciones que el paso previo eliminando la restricción de que los nodos aparezcan en la misma etapa síncrona. Esto implica que la complejidad es la misma en ambos casos, $O(t)$.

Se ha de tener presente que el Paso 1 y el Paso 2, se pueden repetir tantas veces como dependencias máximas se puedan dar en las tareas. Suponiendo la situación extrema, en la que todas las tareas de una etapa son predecesoras de todas las tareas de la etapa posterior, el número de iteraciones corresponde al total de tareas presentes en una etapa.

De esta forma la máxima complejidad de la heurística se obtiene como:

$$O\left(\frac{t}{e} \times \frac{t^2}{e}\right) + O\left(\frac{t}{e} \times t\right) + O(t) \cong O\left(\frac{t^3}{e^2}\right)$$

Se puede observar, como en esta heurística, la complejidad depende directamente del número de tareas de la aplicación y en como estas se distribuyen en las etapas que la forman, siendo este valor acotado e inferior a la complejidad máxima posible de la heurística MPASS, aunque por el contrario, el resultado obtenido, puede no ser tan optimo como se mostrará en el capítulo de experimentación.

Capítulo 4

Experimentación

En este capítulo se presenta el estudio realizado, junto con los resultados obtenidos, tras evaluar las diferentes técnicas y heurísticas propuestas en este trabajo. En el estudio experimental se ha tenido en cuenta los diferentes ámbitos en los que las propuestas han sido desarrolladas. El primero de ellos orientado a la definición de la estructura para el grafo de tareas de la aplicación y el segundo determinando la asignación de tareas a los nodos de procesamiento que permita optimizar el rendimiento de las aplicaciones pipeline.

A continuación se presenta el entorno de experimentación utilizado, para más adelante mostrar los resultados obtenidos.

4.1. Entorno de experimentación

Las pruebas experimentales realizadas en este trabajo se han estructurado en dos bloques. El primero de ellos, presentado en las secciones 4.2 y 4.3, se orienta al estudio individual de las Técnicas de Paralelización y Replicación, así como de las heurísticas de mapping. Para ello se han escrito un conjunto de aplicaciones sintéticas en lenguaje C junto con la librería de paso de mensajes MPI que responden a distintas estructuras de dependencias de tareas.

El segundo bloque, sección 4.4, se basa en la utilización conjunta de ellas, definiendo tanto la estructura para el grafo de la aplicación como el mapping, utilizando para ello tres aplicaciones pipeline orientadas al procesamiento de secuencias de imágenes, de diferentes ámbitos: (a) compresor de vídeo mediante el algoritmo MPEG2, (b) IVUS (*IntraVascular UltraSound*) utilizada el procesamiento de imágenes médicas y (c) BASIZ (*Bright and Saturated Image Zones*) para la detección de regiones en secuencias de vídeo.

Para el estudio experimental se han utilizado tres plataformas diferentes:

(a) Cluster de supercomputación

- 1 servidor Frontend HP Proliant DL145-G2, con dos procesadores AMD Opteron (2.2GHz/1MB), 512MB de memoria RAM y un disco duro de 120GB de capacidad.
- 80 nodos HP Proliant DL145-G1. Cada uno de ellos con 2 procesadores AMD Opteron (1.6GHz/1MB), 1GB de memoria RAM y 2 discos duros de 80GB de capacidad cada uno de ellos.
- Red de producción Ethernet Gigabit.
- Red de gestión Ethernet 100Mbps.
- Sistema operativo Linux, distribución para cluster RocksCluster 4.0.0 (Fuji).

Cabe remarcar que en el proceso de experimentación solo se ha usado un procesador en cada nodo. Esto es así debido a que las comunicaciones realizadas entre las CPU que se encuentran en el mismo nodo de procesamiento, se realizan sin el uso de la red de interconexión, por lo que la estimación del tiempo necesario para la comunicación entre las tareas asignadas en cada una de ellas, no correspondería a los datos utilizados por las heurísticas de mapping.

Este cluster ha sido utilizado en el primer bloque experimental, encargado de la evaluación de las propuestas de este trabajo ya que posee un número de nodos adecuado a las necesidades de las aplicaciones utilizadas. También se ha utilizado en el estudio de la aplicación pipeline que implementa el compresor de vídeo MPEG2.

(b) Cluster de producción

- Servidor Frontend Compaq DC 5100 con un procesador Intel Pentium 4 (3GHz/2MB), 1GB de memoria RAM y un disco duro de 80GB de capacidad.
- 18 nodos Compaq DC 7100 con procesador Intel Pentium 4 (3GHz/2MB), 1GB de memoria RAM y un disco duro de 80GB de capacidad.
- Red de producción y gestión Ethernet Gigabit.
- Sistema operativo Linux 2.4.22-ac4, distribución RedHat 9.

Este cluster ha sido utilizado en la ejecución de la aplicación IVUS ya que ésta requiere una configuración a nivel de librerías que no ha sido posible disponer en el cluster de supercomputación.

(c) Entorno de simulación

En el estudio experimental se ha incluido un entorno de simulación, denominado *p*MAP (*Predicting the best Mapping of pArallel aPplications*) [GRRL04a], que ha permitido evaluar las técnicas propuestas en la situación en la que el número de procesadores necesarios excede al número de ellos que los clusters anteriores ofrecen. Esta es la situación de la aplicación IVUS, en unos casos de estudio concretos, o en el estudio de la aplicación BASIZ que por la implementación disponible no ha sido posible ejecutarla en el cluster de producción.

El entorno *p*MAP permite simular la ejecución de aplicaciones paralelas sobre entornos distribuidos, caracterizando la mayoría de sus parámetros, como puede ser la capacidad de procesamiento de cada nodo, el tipo de scheduling de CPU utilizado, el tipo de red de interconexión, etc., en nuestro caso se ha caracterizado la configuración del cluster de producción.

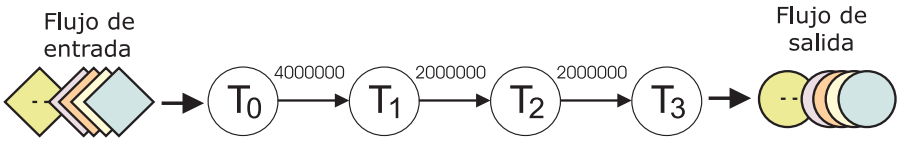
En las siguientes secciones se presentan los resultados obtenidos en cada uno de los bloques experimentales.

4.2. Definición de la estructura de dependencias de tareas de la aplicación

En esta sección, la experimentación se ha centrado en determinar si las nuevas estructuras de dependencias de tareas que se derivan de aplicar las Técnicas de Paralelización y Replicación, presentadas en las secciones 3.1.1 y 3.1.2, son capaces de cumplir con los objetivos de rendimiento que se les impone como parámetro.

Para ello se ha creado una aplicación utilizada como caso de estudio, desarrollada en lenguaje C junto con la librería de paso de mensajes MPI, formada por cuatro tareas, cada una de ellas con diferentes funcionalidades. El grafo de tareas que representa a la aplicación se muestra en la Figura 4.1(a), en ella se puede observar como la estructura para el grafo de tareas de la aplicación pipeline está formada por una única línea de ejecución.

La función implementada en cada tarea es la siguiente:



(a)

	Función	Tiempo (seg.)
T_0	Entrada de matrices	0,7974
T_1	Transformación de los datos	9,1427
T_2	Multiplicación de matrices	25,2993
T_3	Obtención de resultados	0,0187

Volumen de datos	Tiempo (seg.)
4.000.000 Bytes	0,033683
2.000.000 Bytes	0,016866

(b)

Figura 4.1: (a) Grafo de tareas de la aplicación. (b) Tiempo de cómputo de cada tarea y tiempo necesario para la comunicación.

- T_0 - Capta los datos del flujo de entrada desde el exterior. Cada dato es independiente entre sí y corresponde a un par de matrices, de dimensión 1000x1000, de números en formato ASCII dentro en el rango $[0, 99]$. Las dos matrices ocupan 4.000.000 bytes. El flujo de datos se encuentra almacenado y es accesible en el momento en que lo requiere esta tarea.

- T_1 - Recibe el par de matrices, y convierte cada uno de sus datos de formato ASCII a un número representado en un byte. Tras ello, aplica varias transformaciones sobre ambas matrices, la primera es una transformada DCT (*Discrete Cosinus Transform*) y la segunda corresponde a una ordenación por regiones de los valores por filas y columnas. El resultado obtenido ocupa 2.000.000 bytes.

- T_2 - Realiza la multiplicación de las dos matrices recibidas, obteniendo una nueva matriz de números enteros representados mediante dos bytes. El resultado ocupa 2.000.000 bytes.

- T_3 - Sobre la matriz recibida, determina cuantos elementos en ella superan el 30 % del valor ubicado en la celda [0,0].

A la hora de desarrollar esta aplicación se ha definido, para las tareas T_1 y T_2 una funcionalidad y una implementación que permite que sean paralelizadas y en consecuencia susceptibles de ser tenidas en cuenta en la Técnica de Paralelización.

En la Figura 4.1(b) se muestra el tiempo de cómputo de cada tarea en segundos, para el procesamiento de un único dato, a partir del promedio obtenido tras múltiples

ejecuciones en el cluster de supercomputación, asignando cada tarea a un único nodo de procesamiento. También se muestra el tiempo necesario para la transmisión del volumen en bytes en la comunicación de las tareas.

Tras la ejecución de la aplicación, la productividad que se ha obtenido es 0,0393 datos por segundo, correspondiente a un valor de IP de 25,45 segundos, y una latencia de 35,40 segundos. Estos resultados indican que es la tarea T_2 el cuello de botella de la aplicación, como era de esperar ya que es la tarea con mayor tiempo de cómputo.

En los dos próximos apartados, el primero dedicado a la Técnica de Paralelización, y el segundo a la Técnica de Replicación, se muestra los resultados obtenidos al usar ambas técnicas con la finalidad de obtener una nueva estructura para el grafo de tareas para la aplicación pipeline que permita aumentar la productividad. Esta nueva estructura ha sido trasladada a la propia aplicación pipeline.

El requerimiento que se ha marcado como objetivo de productividad a alcanzar corresponde a 2, 4, 8, 16 y 32 veces superior al valor de productividad que la aplicación es capaz de ofrecer. Estos valores aparecerán en los diferentes gráficos y cuadros etiquetados como x2, x4, x8, x16 y x32 respectivamente.

4.2.1. Técnica de Paralelización

La aplicación de la Técnica de Paralelización requiere que se identifiquen previamente aquellas tareas que son susceptibles de ser paralelizadas aplicando paralelismo de datos. En este sentido, son las tareas T_1 y T_2 las que cumplen esta característica. De esta forma al ser paralelizadas, obteniendo subtareas con un menor tiempo de cómputo que el de la tarea original, se consigue disminuir la granularidad en la aplicación aumentando la productividad a alcanzar.

Para determinar el número de subtareas de T_1 y T_2, es necesario previamente caracterizar su tiempo de cómputo en función de la dimensión de la matriz que deberá procesar cada subtarea. Para ello se ha ejecutado la aplicación variando la dimensión de la matriz procesada 2, 4, 8, 10 y 20 veces inferior a la de la matriz original, obteniendo los tiempos de cómputo mostrados en escala logarítmica en la Figura 4.2.

Hay que destacar que debido a la funcionalidad asignada a ambas tareas, el tiempo de cómputo no se reduce de igual forma para cada una de ellas. Si para la matriz original, T_1 posee un tiempo de cómputo inferior al de T_2, cuando la dimensión de la matriz disminuye esta característica se invierte. Esto es así debido a que las transformadas utilizadas en la tarea T_1 no escalan en igual medida que la multiplicación de matrices realizada por la tarea T_2.

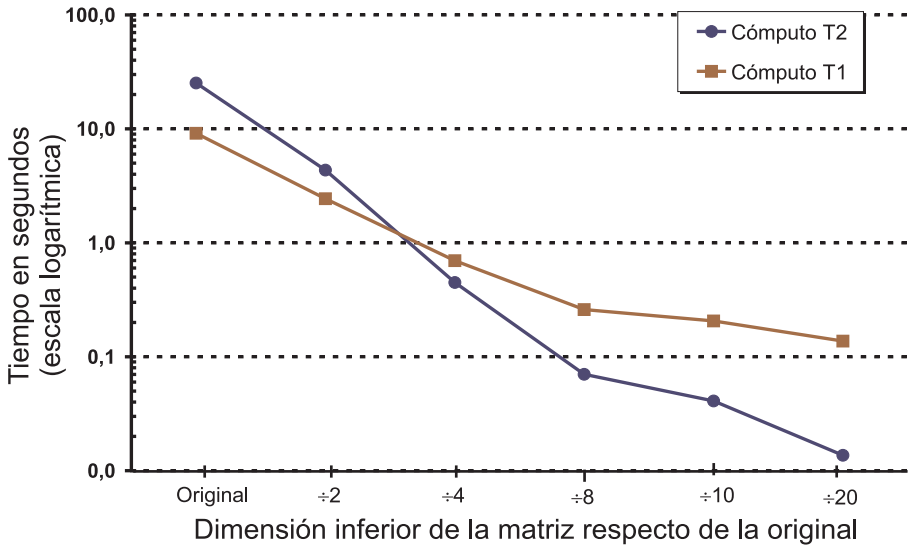


Figura 4.2: Variación del tiempo de cómputo de las tareas T_1 y T_2 al disminuir el tamaño de la matriz de entrada.

Basándose en estos datos y mediante un proceso de interpolación usando un calculador simbólico, se ha obtenido las expresiones mostradas en (4.1) que definen el comportamiento de ambas tareas en función del número de divisiones de la matriz, o lo que es lo mismo en función del número de subtareas.

$$\begin{aligned} tiempo\_cómputo(T_1) &= 9{,}1498e^{-1{,}245\log_2(n\_subtareas)} \\ tiempo\_cómputo(T_2) &= 25{,}2827e^{-1{,}895\log_2(n\_subtareas)} \end{aligned} \tag{4.1}$$

Para determinar el número adecuado de subtareas de T_1 y T_2 que permitan alcanzar una productividad concreta, basta con determinar el valor de $n\_subtareas$ que cumpla par el valor de IP deseado la igualdad:

$$IP = tiempo\_cómputo(T_i) \tag{4.2}$$

. En el Cuadro 4.1, se detalla el resultado obtenido para los valores de IP que corresponden a la productividad a alcanzar en cada uno de los casos que se evalúan, para cada tarea el número $n\_subtareas$ obtenido de evaluar la expresión (4.2), así como el número de subtareas final a partir del redondeo al alza del valor calculado.

A partir de estos datos y aplicando la técnica propuesta en la sección 3.1.1, se obtiene para cada caso: los subgrafos con sus tareas, el número de copias de cada subgrafo y el número de subtareas de cada tarea que aparecerá dentro del subgrafo.

Caso	IP	$n\_subtareas$ T_1	$n\_subtareas$ T_2	subtareas T_1	subtareas T_2
x2	12,723109	0,8316	1,2855	1	2
x4	6,361554	1,2252	1,6565	2	2
x8	3,180777	1,8051	2,1345	2	3
x16	1,590389	2,6593	2,7505	3	3
x32	0,795194	3,9179	3,5442	4	4

Cuadro 4.1: Número de subtareas a obtener para T_1 y T_2 según las expresiones 4.1.

Estos datos se muestran en el Cuadro 4.2. En el Apéndice A.0.4 aparecen los grafos que representan a estas estructuras una vez aplicada la Técnica de Paralelización.

Caso	$Subgrafo$	Copias del $Subgrafo$	$n\_subtareas(T_i)$
x2	$\{T_2\}$	2	(1)
x4	$\{T_1, T_2\}$	2	(1,1)
x8	$\{T_1, T_2\}$	2	(1,2)
x16	$\{T_1, T_2\}$	3	(1,1)
x32	$\{T_1, T_2\}$	4	(1,1)

Cuadro 4.2: Relación de subgrafos y subtareas obtenidas en la Técnica de Paralelización.

Como se puede observar, en el caso x2, es la tarea T_2 la única que debe ser paralelizada, ya que es ella la que actúa como cuello de botella. Por el contrario cuando se requiere una mayor productividad, la heurística agrupa las tareas T_1 y T_2 formando un subgrafo y ajustando el número de subtareas en que éstas tareas deben ser divididas. Hay que destacar como el diferente comportamiento del tiempo de cómputo de las tareas hace que el número de subtareas obtenidas se ajuste en función del requerimiento de productividad. Así para el caso x8, en cada subgrafo la relación entre las tareas T_1 y T_2 es de 1:2, mientras que para el resto es de 1:1.

En la Figura 4.3(a) se muestran los resultados de productividad obtenidos al ejecutar utilizando en la aplicación la nueva estructura propuesta mediante la Técnica de Paralelización y asignando una tarea en cada nodo de procesamiento para poder evaluar sin distorsiones los resultados obtenidos. En ella se compara la productividad teórica a alcanzar respecto de la productividad que se ha obtenido, mostrando el porcentaje de desviación entre ambos valores. Para cada caso se ha añadido el número de nodos de procesamiento utilizados, que se corresponde con el número de tareas de la aplicación.

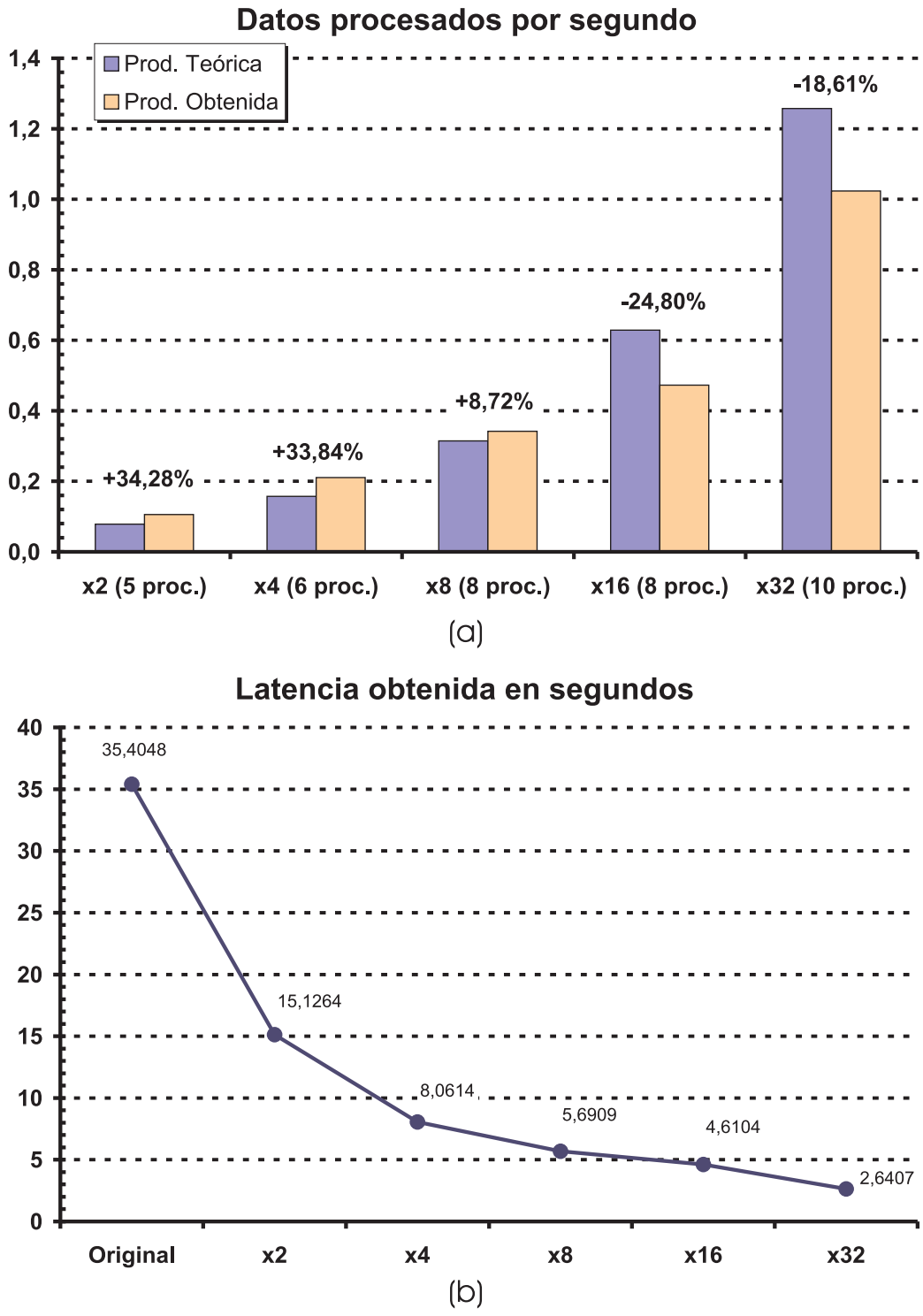


Figura 4.3: (a) Productividad y, (b) latencias obtenidas tras aplicar la Técnica de Paralelización.

Se puede observar, como en las pruebas x2 y x4, la productividad obtenida es mayor a la teórica marcada como objetivo. Esto es así, debido a que el número de subtareas se obtiene a partir de un redondeo al alza, sobre-explotando el paralelismo de éstas. Por el contrario, al aumentar el número de subtareas y en consecuencia disminuir el tamaño de la dimensión de la matriz, el valor para el tiempo de cómputo de las subtareas admite un menor margen de diferencia provocando que el resultado obtenido, sea inferior al teórico como se da en los casos x16 y x32.

Hay que destacar que al ir disminuyendo el valor de IP al alcanzar, se llega a la situación en la que aparece una nueva tarea que actúa como cuello de botella, siendo ésta la tarea T_0, la cual no puede ser paralelizada y que marca la cota máxima de productividad alcanzable para esta aplicación pipeline. Esta situación indica que la Técnica de Paralelización llega al límite de sus posibilidades para esta aplicación concreta, debido a que aunque se aumente el número de subtareas para T_1 y T_2 no será posible aumentar la productividad.

En la Figura 4.3(b) se muestra la latencia obtenida en cada caso de estudio. Como era de esperar, al reducir la granularidad de las tareas la latencia se reduce significativamente, reafirmando que este método además de permitir una mejora de la productividad es capaz también de reducir la latencia.

4.2.2. Técnica de Replicación

La Técnica de Replicación no impone a las tareas de la aplicación pipeline, ninguna restricción sobre su funcionalidad para que pueda ser aplicada sobre ellas, sino que cualquier tarea es susceptible de ser replicada si cumple las condiciones para ello. Por este motivo es una técnica idónea para aumentar la productividad en aquellas aplicaciones pipeline en las que no es posible utilizar la Técnica de Paralelización. Por el contrario posee como handicap el que solo puede ser utilizada cuando los datos a procesar del flujo de entrada no presenten dependencias entre ellos.

Aplicando el método desarrollado en la sección 3.1.2 bajo los mismos criterios de productividad que los usados en el apartado anterior, se obtienen los subgrafos, el número de copias a realizar de cada uno, así como las tareas que forman parte de ellos además del número de veces en que éstas deben ser replicadas en cada subgrafo, que se muestra en el Cuadro 4.3. En el Apéndice A.0.5 aparecen los grafos de tareas que representan a estas estructuras de la aplicación una vez realizadas las replicaciones.

Se puede observar como inicialmente, en el caso x2, sólo es la tarea T_2 la que debe ser replicada al ser ella la única que actúa como cuello de botella. A medida que el

Caso	IP	$Subgrafo$	$n\_copias[SG]$	$(n\_rep[T_i])$
x2	12,723109	$\{T_2\}$	2	(1)
x4	6,361554	$\{T_1, T_2\}$	2	(1,2)
x8	3,180777	$\{T_1, T_2\}$	3	(1,3)
x16	1,590389	$\{T_1, T_2\}$	6	(1,3)
x32	0,795194	$\{T_0, T_1, T_2\}$	2	(1,6,16)

Cuadro 4.3: Relación de número de replicaciones para los subgrafos y las tareas obtenidas en la Técnica de Replicación.

valor de IP a alcanzar se reduce, otras tareas, cuyo tiempo de cómputo excede al IP, pasan a formar parte del subgrafo. Hay que destacar, como en esta técnica para el caso x32, la tarea T_0, que en el estudio previo no era tenida en cuenta al no ser posible paralelizarla, en éste pasa a formar parte del subgrafo siendo replicada.

La Figura 4.4 muestra los resultados de productividad obtenidos, de la ejecución de la aplicación con la nueva estructura propuesta, asignando una tarea a cada nodo de procesamiento. En la figura se compara, incluyendo el porcentaje de desviación, la productividad obtenida con la teórica a alcanzar. Además se ha añadido el número de procesadores utilizados en la ejecución.

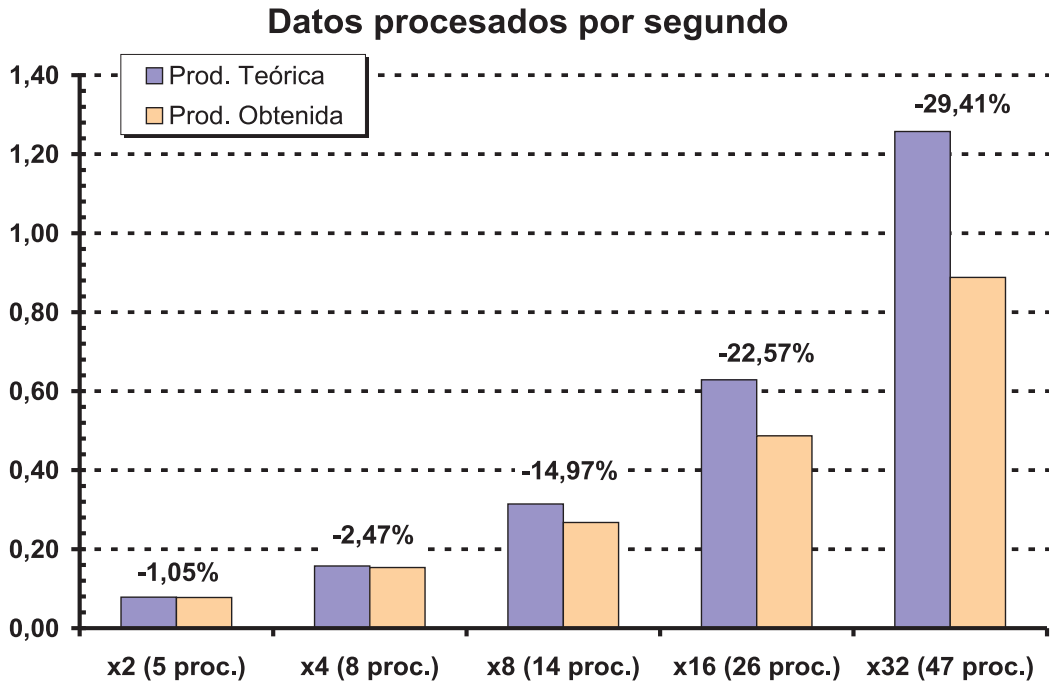


Figura 4.4: Productividad obtenida tras aplicar la Técnica de Replicación.

El resultado muestra como la productividad aumenta a medida que la técnica replica

las tareas, de forma que éstas procesan concurrentemente múltiples datos del flujo de entrada. A medida que el requerimiento de productividad a alcanzar crece, la diferencia con respecto a la productividad demandada es mayor. Esta situación, se debe a que el cuello de botella ya no lo es tanto la capacidad de cómputo de las tareas, si no la capacidad de repartir los datos a cada nueva línea de ejecución que se crea. Esta situación queda de manifiesto, en el momento en el que la tarea T_0 pasa a formar parte del subgrafo, forzando a obtener un mayor ratio de datos del flujo de entrada.

Si se aplica la Técnica de Replicación forzando a alcanzar una productividad mucho mayor, se podría llegar a la situación en la que cada dato del flujo de entrada se procese en una línea de ejecución propia. En este caso, la diferencia sustancial respecto a una implementación de la aplicación basada en el paralelismo de datos puro, en el que toda ella forma una única tarea y que aparecería copiada n veces sobre n nodos de procesamiento, correspondería a que una estructura pipeline de múltiples líneas de ejecución, permite explotar el paralelismo funcional de las tareas reduciendo de esta manera la latencia.

Cabe señalar que la latencia que se obtiene en la nueva estructura de dependencias de la aplicación, es la misma que la obtenida de la aplicación original ya que las tareas no han sido modificadas y cada dato concreto es procesado de la misma forma.

Para finalizar, en la Figura 4.5 se compara la productividad obtenida en las Técnicas de Paralelización y de Replicación. En ella se puede observar como ambas técnicas poseen un comportamiento parecido, debido a que se les ha aplicado los mismos objetivos de rendimiento. Esto indica que las dos pueden ser utilizadas para definir la estructura del grafo de tareas que siendo trasladada a la aplicación pipeline permite obtener una productividad que inicialmente no es alcanzable por la aplicación.

Un factor que diferencia a ambas técnicas, es el número de nodos de procesamiento que ha sido necesario utilizar. En el caso de la Técnica de Replicación se ha requerido un mayor número de ellos, debido a que en esta técnica la forma en que se consigue aumentar la productividad es a partir de crear nuevas líneas de ejecución. En el caso de la Técnica de Paralelización, al disminuir la granularidad de las tareas, se aumenta la productividad con un requerimiento de nodos de procesamiento muy inferior.

Otra diferencia significativa, estriba en que en el caso de la Técnica de Paralelización existe un límite en la productividad que es posible alcanzar y que viene determinado por el tiempo de cómputo de las tareas que no pueden ser paralelizadas. Este límite no aparece en el caso de la Técnica de Replicación, la cual solo depende del ratio máximo de entrega de datos del flujo de entrada.

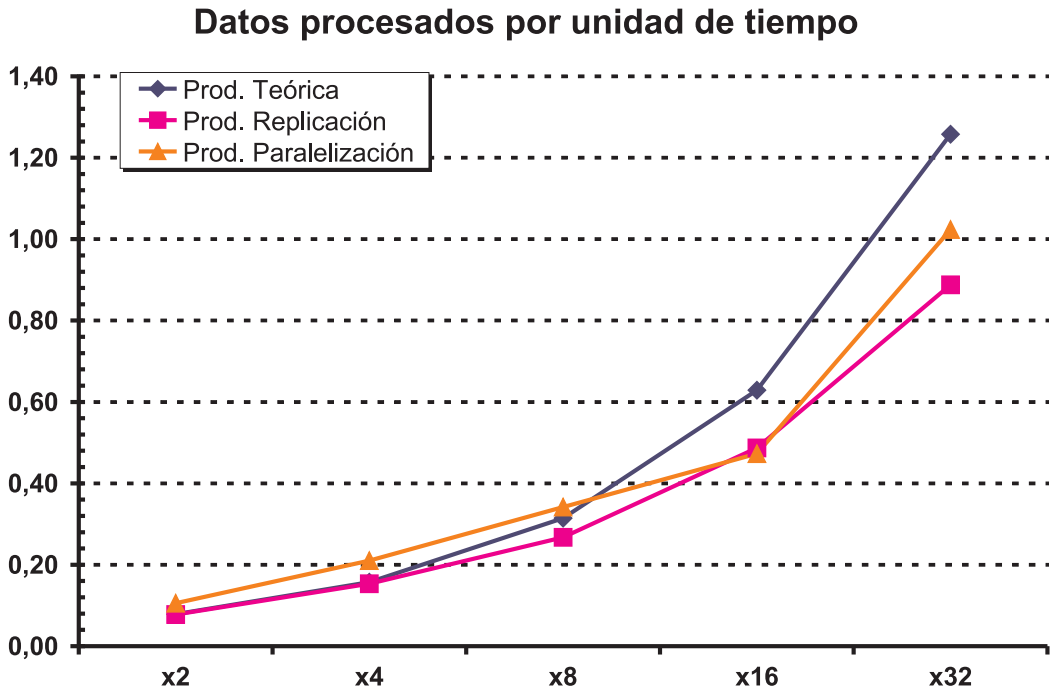


Figura 4.5: Comparación de la productividad obtenida mediante la Técnica de Paralelización y la Técnica de Replicación.

4.3. Experimentación con las heurísticas de mapping

En esta sección se mostrará los resultados obtenidos al utilizar las heurísticas de mapping MPASS y MPART presentadas en este trabajo.

Los resultados obtenidos se han comparado, además de con los valores marcados como objetivo a alcanzar, con los que se obtienen al utilizar dos heurísticas de la literatura con objetivos de rendimiento similares a los propuestos en este trabajo. Las heurísticas escogidas son las presentadas en [LLP01] y [YKS03] y que denotaremos por Lee01 y Yang03 respectivamente en los resultados que se muestran más adelante.

Muy brevemente se comentan ambas heurísticas. La primera de ellas, Lee01, ha sido desarrollada para el mapping de aplicaciones STAP (*Space-Time Adaptive Processing*) ejecutándose sobre un cluster de computadores y con la finalidad de obtener su máxima productividad. Las aplicaciones STAP poseen una estructura de dependencias rígida, definida por etapas, en las que todas las tareas que forman parte de una etapa en particular, poseen el mismo tiempo de cómputo. Así mismo, cada una de las tareas en una etapa, es sucesora de todas las tareas de la etapa previa y a su vez es predecesora de todas las tareas de la etapa posterior. Todas las tareas de la misma etapa transmiten

el mismo volumen de datos.

La forma de proceder de la heurística Lee01, se basa en una primera asignación utilizada como base, y que va siendo reajustada a partir del movimiento de las tareas, de un nodo de procesamiento a otro, siempre que sean nodos a los que se les ha asignado tareas predecesoras o sucesoras a la tarea sobre la que se decide su asignación, si con ello se consigue obtener una mejora en la productividad. Esta forma de proceder se realiza en todas las posibles combinaciones de asignaciones de tareas a nodos de procesamiento hasta obtener aquella que maximiza la productividad.

La segunda heurística, Yang03, permite una estructura más flexible para las aplicaciones pipeline, requiriendo exclusivamente que éstas puedan ser representadas mediante un grafo TPG. La heurística se basa en dos pasos, el primero se encarga de obtener una asignación inicial basada en la heurística clásica ETF (*Earliest Task First*) de asignación de grafos TPGs [HCAL89]. El segundo paso se encarga de refinar la asignación previa mediante el uso de dos técnicas, TDA (*Top-Down Approach*) y *Look Ahead*, con el fin de obtener grupos de tareas cuyo tiempo de cómputo acumulado no exceda del valor de IP a obtener, creando una pseudo-estructura de etapas que puedan ejecutarse de forma solapada.

Debido a las diferencias en los objetivos de las heurísticas y en la estructura de las aplicaciones en las que se basan, se ha creado dos conjuntos de experimentación, formado cada uno de ellos por diez aplicaciones desarrolladas en lenguaje C junto a la librería de paso de mensajes MPI. El primero, que denominaremos como aplicaciones homogéneas y que están etiquetadas como {ho01,..,ho10}, han sido creadas basándose en la estructura de las aplicaciones STAP, necesaria para el uso de la heurística Lee01. De esta forma todas las tareas en una misma etapa poseen el mismo tiempo de cómputo y transmiten el mismo volumen de datos. El segundo conjunto de aplicaciones, denominadas aplicaciones arbitrarias y etiquetadas como {ar01,..,ar10}, poseen una mayor heterogeneidad en los tiempos de cómputo y en el volumen de datos a transmitir, de forma que permiten generalizar en mayor medida los resultados para cualquier tipo de aplicación pipeline. En este segundo conjunto la heurística Lee01 no puede ser utilizada.

Las aplicaciones desarrolladas emulan el flujo de entrada, a partir de que en cada una de las tareas se repite un total de 200 veces la secuencia de acciones: recepción de datos, cómputo y envío de datos, a excepción de la primera y última tarea. El cómputo a realizar se ha implementado a partir de una función que realiza como proceso base un millón de operaciones aritméticas simples con números de doble precisión. Esta función recibe como parámetro el número de veces en se debe repetir el proceso base, así una

llamada con valor del parámetro de 10, equivale a realizar 10 millones de operaciones. Para las comunicaciones, el volumen de datos transmitidos varía entre cien mil y un millón de bytes. En el caso del conjunto de aplicaciones homogéneas, todas las tareas de una misma etapa envían el mismo volumen de datos, manteniendo la estructura requerida para la heurística Lee01, mientras que en el caso del conjunto de aplicaciones arbitrarias, el volumen asignado para cada una de las tareas de la aplicación, ha sido escogido de forma arbitraria en el rango indicado.

En los Cuadros 4.4 y 4.5 se muestra para cada aplicación el número de tareas, el rango tomado para el parámetro de la función de cómputo en las tareas, así como el número de líneas de ejecución. En el Apéndice A.0.6 aparecen los grafos de tareas que representan a estas aplicaciones.

Aplicación	Núm. tareas	Rango del cómputo	líneas de ejecución
ho01	17	[25,100]	50
ho02	17	[50,125]	125
ho03	22	[125]	625
ho04	17	[25,100]	128
ho05	32	[25,125]	59.049
ho06	13	[50,100]	36
ho07	29	[25,100]	5.040
ho08	11	[30,100]	24
ho09	29	[25,100]	5.040
ho10	13	[25,100]	36

Cuadro 4.4: Caracterización del conjunto de aplicaciones homogéneas.

Aplicación	Núm. tareas	Rango del cómputo	líneas de ejecución
ar01	67	[300,1000]	39.916.800
ar02	73	[300,1000]	80.640
ar03	31	[300,1000]	240
ar04	38	[300,1000]	1.176
ar05	67	[300,1000]	39.916.800
ar06	32	[50,200]	15.625
ar07	32	[30,150]	480
ar08	29	[60,150]	5.040
ar09	29	[50,200]	5.040
ar10	48	[50,600]	1.920

Cuadro 4.5: Caracterización del conjunto de aplicaciones arbitrarias.

El estudio experimental se ha realizado bajo los siguientes objetivos:

- Aplicación de las políticas de mapping con el objetivo de alcanzar la máxima productividad teórica que permite la aplicación.
- Obtención de una productividad prefijada.
- Obtención de la mínima latencia bajo una restricción de productividad a alcanzar.

En todos los casos cada prueba se ha ejecutado cinco veces, utilizando para ello el cluster de supercomputación y mostrando en las figuras y en los cuadros siguientes el valor promedio de los resultados obtenidos.

4.3.1. Obtener la máxima productividad

Las heurísticas MPASS, MPART y Yang03, en su definición, requieren un parámetro de productividad a alcanzar, que en este apartado debe corresponder a la máxima que se puede obtener de la aplicación pipeline. Como se ha comentado en los capítulos previos, la productividad máxima de una aplicación pipeline, viene acotada por el mayor de los tiempos de cómputo de sus tareas. Bajo esta premisa, ha sido este valor el utilizado como parámetro en la heurísticas. En el caso de la heurística Lee01, por su propia definición ya corresponde este su objetivo principal el alcanzar la máxima productividad.

Para el conjunto de aplicaciones homogéneas los resultados obtenidos se muestran en la Figura 4.6(a), en forma de porcentaje de desviación en valor absoluto de la productividad obtenida con respecto al valor de IP marcado como objetivo.

Se puede observar como la mayoría de los resultados se mantienen con una desviación por debajo del 5 %, a excepción de la heurística MPART en las aplicaciones *ho*08 y *ho*10. Esto es debido principalmente a que en esta heurística no se tiene en cuenta la estructura global de líneas de ejecución de la aplicación, perdiendo información importante a la hora de realizar las agrupaciones de tareas.

Los resultados obtenidos mediante la heurística Lee01, son los que por lo general ofrecen una menor desviación con respecto a la productividad máxima. Esto es así, debido a que en el mapping obtenido cada tarea es ubicada en un nodo de procesamiento. Esta situación puede observarse en la Figura 4.6(b) que muestra el número de procesadores utilizado, el cual para Lee01 siempre es superior al resto de heurísticas. En el resto de heurísticas, el resultado obtenido utiliza un menor número de procesadores, indicando que son capaces de reducir los periodos de inactividad al detectar y asignar las tareas con fuertes dependencias.

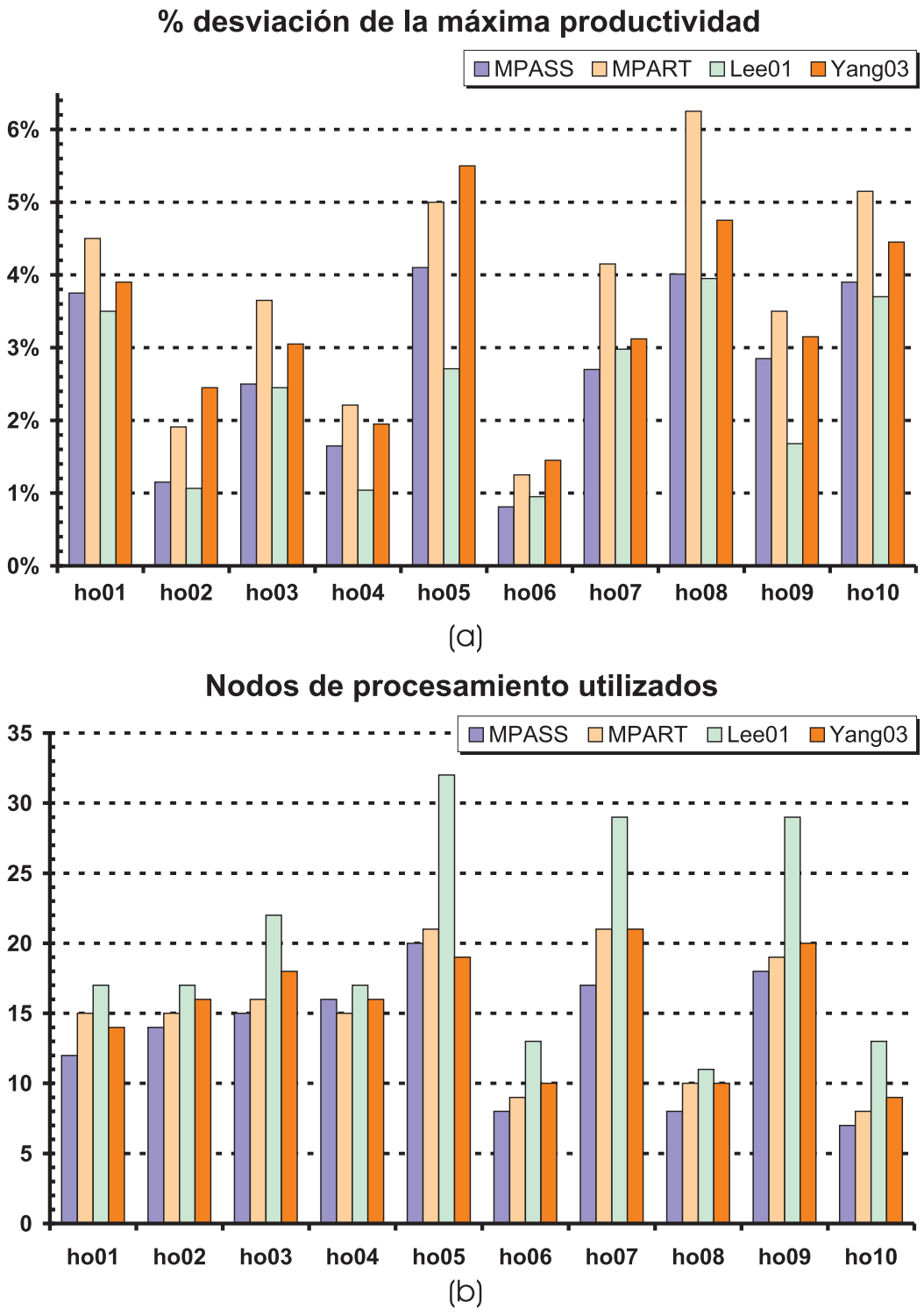


Figura 4.6: Aplicaciones homogéneas: (a) Porcentaje de desviación respecto de la productividad máxima. (b) Nodos de procesamiento utilizados.

En el caso de la heurística Yang03, ésta parte de una asignación basada en conseguir que las tareas se inicien lo antes posible. Esta asignación, debe refinarse para tener presente el comportamiento pipeline de la aplicación, motivo por el que al deshacer algunas de las asignaciones previas obtenidas del mapping ETF, requiere de un mayor número de procesadores que en las heurísticas propuestas en este trabajo.

Las heurísticas MPASS y MPART, poseen un comportamiento que no difiere del resto. En concreto, la heurística MPASS suele ser la que utiliza el menor número de procesadores. Hay que tener presente, que las aplicaciones de este conjunto de pruebas tienen una estructura de dependencias muy rígida, motivo por el que la definición de las etapas síncronas y la evaluación ordenada de las líneas de ejecución ofrece un buen resultado. En el caso de la heurística MPART, por el contrario, al no tener presente el conjunto global de líneas de ejecución, requiere por lo general de un mayor número de procesadores y es la que obtiene una mayor desviación respecto de la productividad máxima.

Para el conjunto de aplicaciones arbitrarias, cuyo resultado se muestra en la Figura 4.7(a), la desviación obtenida respecto a la productividad máxima es mucho mayor que en el estudio previo debido a que el tiempo de cómputo de las tareas y el volumen de datos transmitido entre ellas no posee un patrón preestablecido, dando opción a que cada heurística explote en mayor medida sus capacidades.

En esta situación la heurística MPART, es la que obtiene un peor resultado, como era de esperar debido a que la evaluación de las líneas de ejecución no tiene presente la ubicación correcta de algunas tareas en las etapas síncronas. Por el contrario, las heurísticas MPASS y Yang03, mucho más complejas, alcanzan mejores resultados, ya que son capaces de crear agrupaciones de tareas en las que el tiempo de cómputo está más balanceado, en el caso de las heurística MPASS al definir correctamente las etapas síncronas, y en el caso de Yang03 gracias a las fases de refinamiento TDA y Look-Ahead.

Si bien la productividad alcanzada por la heurística MPASS produce unos resultados ligeramente mejores que Yang03, aunque ciertamente comparables, la mayor diferencia aparece al evaluar el número de procesadores utilizados, como se puede observar en la Figura 4.7(b). Las heurísticas MPASS y MPART utilizan por lo general un menor número de éstos ya que la evaluación que hacen parte desde un principio en la naturaleza iterativa de las tareas, cosa que no tiene presente la heurística Yang03.

Como conclusión para esta prueba, destacar que las heurísticas propuestas, ofrecen unos buenos resultados con el menor número nodos de procesamiento.

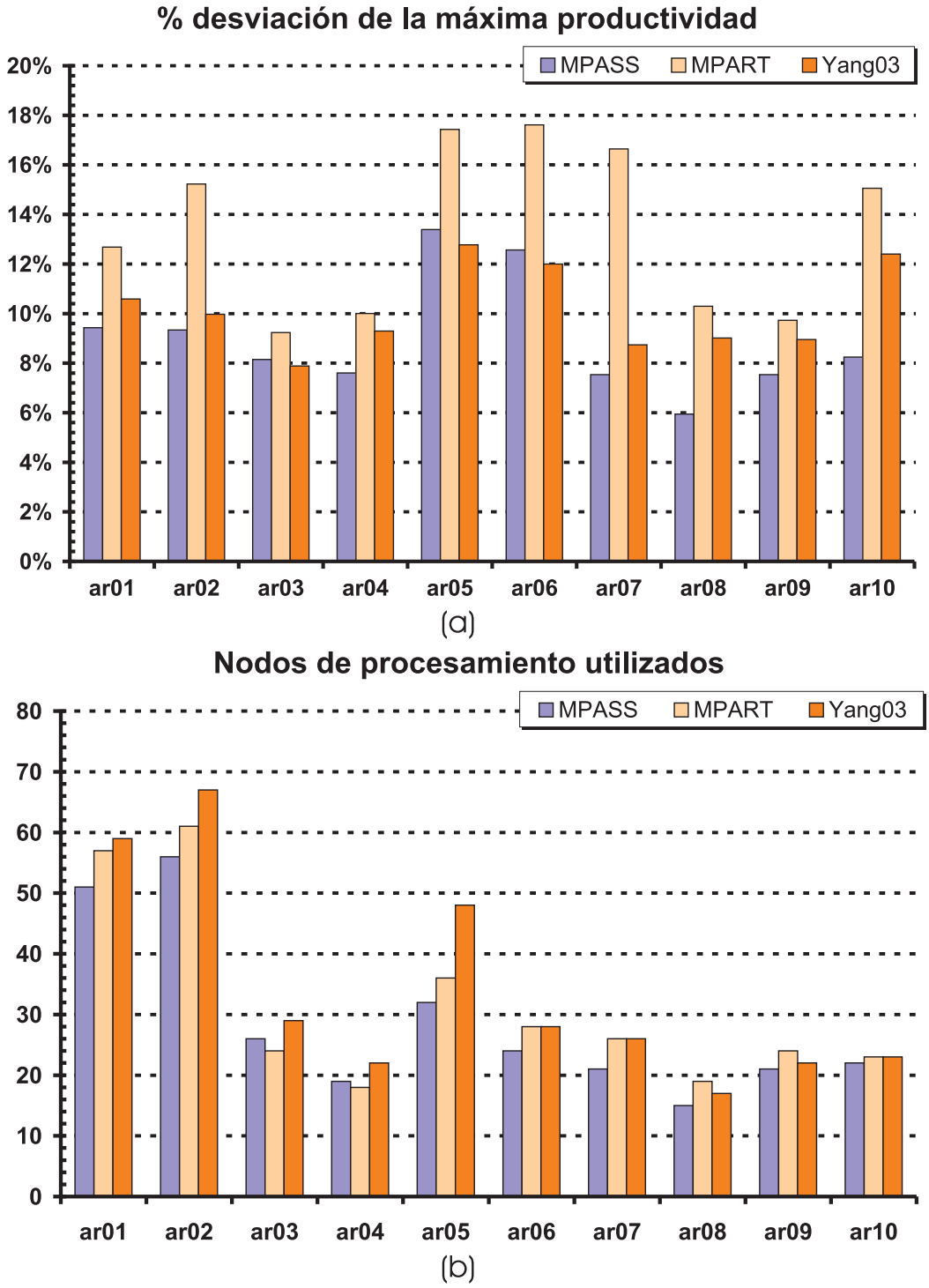


Figura 4.7: Aplicaciones arbitrarias: (a) Porcentaje de desviación respecto de la productividad máxima. (b) Nodos de procesamiento utilizados.

4.3.2. Alcanzar una productividad prefijada

En este apartado se ha tomado como valor de IP a alcanzar el mostrado en el Cuadro 4.6. Estos valores están dimensionados a los valores utilizados en el parámetro de la función de cómputo implementada en las tareas.

Aplicación	IP alcanzar	Aplicación	IP alcanzar
ho01	100	ar01	3000
ho02	200	ar02	2500
ho03	250	ar03	2500
ho04	200	ar04	2500
ho05	200	ar05	2500
ho06	150	ar06	1000
ho07	150	ar07	500
ho08	150	ar08	500
ho09	150	ar09	500
ho10	150	ar10	2000

Cuadro 4.6: Valor de IP a alcanzar establecido para cada aplicación.

La Figura 4.8(a) muestra el porcentaje de desviación respecto a la productividad a alcanzar para las aplicaciones homogéneas.

Se puede observar que los porcentajes de desviación obtenidos varían notablemente, con valores menores al 1 % o cercanos al 8 %. Por lo general, esto es debido a la elección del valor de IP a alcanzar. Si es posible obtener agrupaciones de tareas cuya suma de sus tiempos de cómputo sea próximo al valor de IP, el resultado final es a su vez próximo al deseado, como son los casos *ho*02, *ho*06 y *ho*08, en el caso contrario el resultado se alejará del deseado, como son los casos *ho*03 y *ho*10. Esto da lugar a la conclusión de que, la elección del valor de productividad a alcanzar, se ha de realizar teniendo en cuenta las características propias de la aplicación, y que puede no ser adecuado cualquier valor arbitrario.

Para este conjunto de pruebas, la heurística MPASS es la que por lo general ha obtenido una resultado mejor, debido a su capacidad de poder crear agrupaciones de tareas que pueden ejecutarse síncronamente. Para la heurística MPART, se observa que el resultado obtenido es algo peor, por los motivos expuestos en el apartado previo.

En la Figura 4.8(b) se compara el número de nodos de procesamiento utilizados para este conjunto de pruebas, siendo por lo general la heurística MPASS la que menos requiere de ellos para la ejecución de las aplicaciones.

En el estudio de las aplicaciones arbitrarias, los resultados son los mostrados en

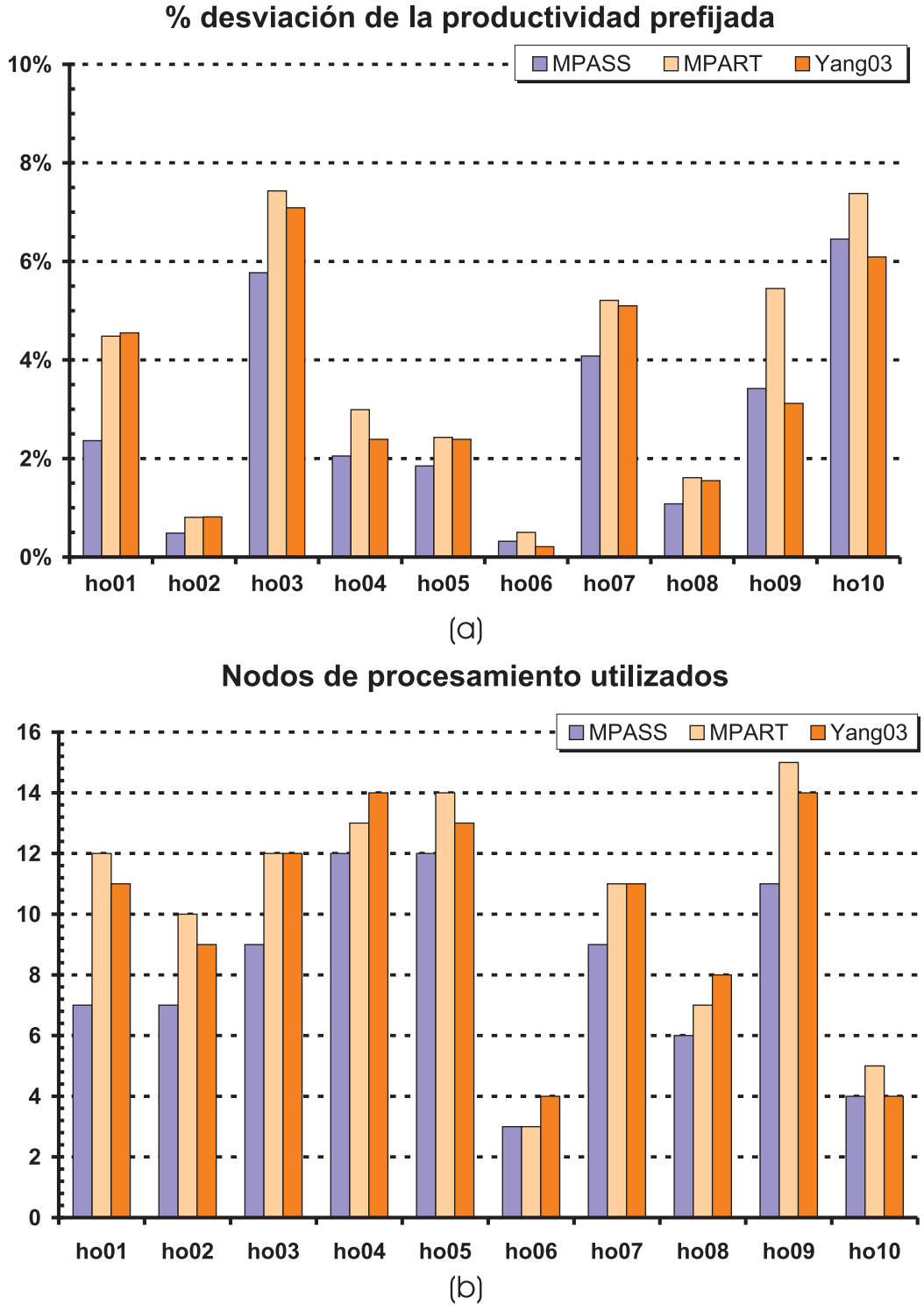


Figura 4.8: Aplicaciones homogéneas: (a) Porcentaje de desviación respecto de la productividad prefijada. (b) Nodos de procesamiento utilizados.

la Figura 4.9(a). En ella se observa una mayor desviación respecto al valor deseado, producida por la dificultad en obtener agrupaciones de tareas dentro de las etapas síncronas en las que el cómputo acumulado corresponda al valor de IP a alcanzar. Aún así los resultados siguen la misma tónica que la del primer conjunto de pruebas.

En la Figura 4.9(b) se muestra el número de nodos de procesamiento utilizados. Estos comparado con respecto a los utilizados en el estudio de la obtención de la máxima productividad se reducen significativamente gracias a que el valor de IP a alcanzar permite una mayor agrupación de tareas, destacando la heurística MPASS como la que es capaz de utilizar el mínimo número de nodos de procesamiento, lo cual redunda en una mayor utilización de los mismos.

Como conclusión se puede decir que las heurísticas MPASS y MPART, ofrecen una asignación que es capaz de acercarse en gran medida al valor de productividad deseado a la vez que utiliza un menor número de nodos de procesamiento. También hay que destacar que la elección de la productividad objetivo utilizada como parámetro, ejerce una gran influencia en la efectividad del resultado final debido a las propias características de la aplicación.

4.3.3. Obtener la mínima latencia bajo un requisito de productividad

En este apartado se han evaluado las heurísticas, utilizando como criterio en el mapping el de obtener la mínima latencia y usando como requisito de productividad a alcanzar el mismo que el usado en el apartado anterior (Cuadro 4.6).

Para poder valorar el resultado de latencia obtenido, se ha comparado el resultado con la suma de los tiempos de cómputo de las tareas que forman parte del camino crítico de la aplicación, ya que no será posible obtener una latencia inferior a este valor.

En la Figura 4.10(a) se muestran los resultados de latencia obtenidos para las aplicaciones homogéneas junto con el valor de latencia mínima.

La heurística Yang03, es la que ofrece por lo general un resultado menor, debido a que el mapping que obtiene se basa sobre el resultado de la heurística ETF, la cual está orientada exclusivamente a la minimización del tiempo de ejecución de una aplicación. Aún así, la heurística MPASS se aproxima bastante utilizando un menor número de procesadores, como se observa en la Figura 4.10(b), en algunos casos igualando el valor de latencia obtenido por la heurística Yang03. Por el contrario los valores obtenidos por

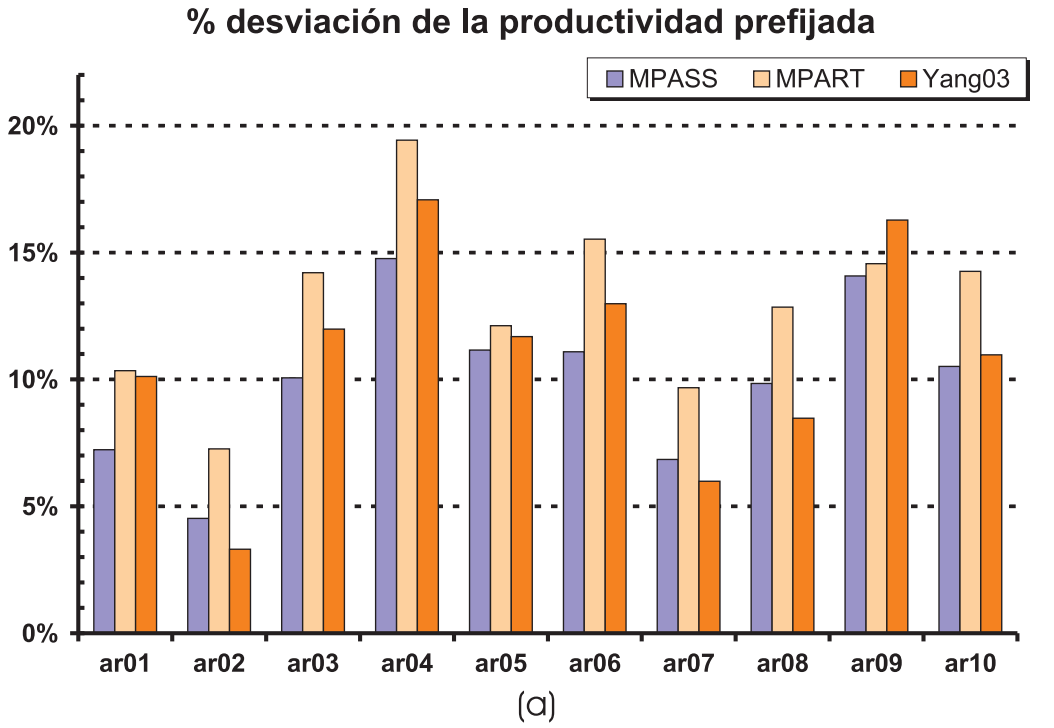


Figura 4.9: Aplicaciones arbitrarias: (a) Porcentaje de desviación respecto de la productividad prefijada. (b) Nodos de procesamiento utilizados.

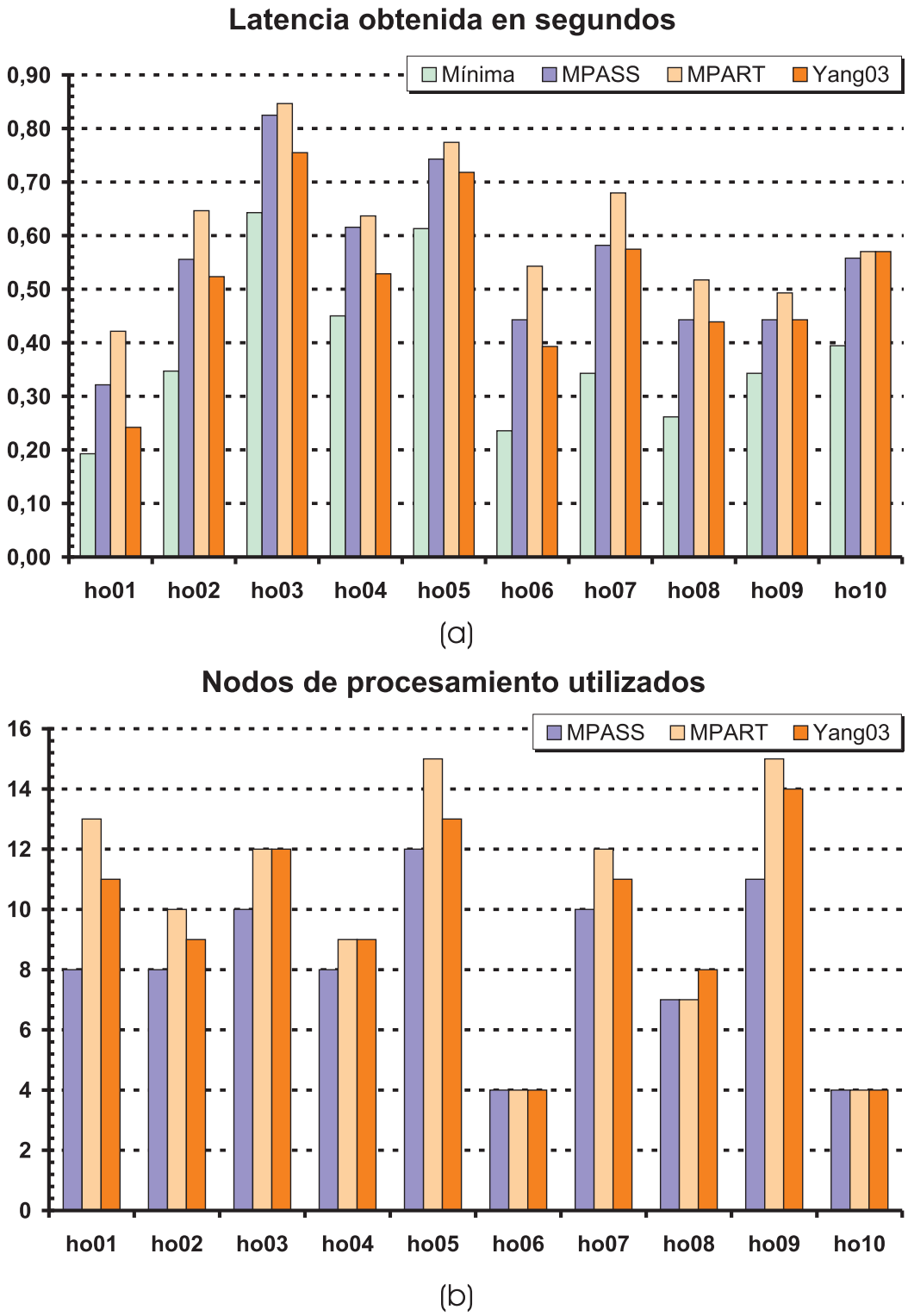


Figura 4.10: Aplicaciones homogéneas: (a) Valor de latencia obtenida. (b) Nodos de procesamiento utilizados.

MPART son algo peores, debido por lo general a que al evaluar de forma parcial las líneas de ejecución no es capaz de realizar una agrupación que reduzca significativamente la latencia.

Los resultados para las aplicaciones arbitrarias, mostrados en la Figura 4.11, tienen una tendencia similar a los resultados obtenidos en las pruebas anteriores.

Como conclusión se puede decir que las heurísticas propuestas son capaces de obtener una latencia aceptable, con un bajo número de nodos de procesamiento.

4.4. Estudio sobre aplicaciones pipeline orientadas al procesamiento de secuencias de imágenes

En este apartado de la experimentación, se ha utilizado de forma conjunta las técnicas orientadas a la definición de la estructura de dependencias de tareas de la aplicación, mediante la Técnica de Paralelización y la Técnica de Replicación, así como la heurística de mapping MPASS, por ser la que ha demostrado que ofrece la mejor asignación, sobre tres aplicaciones orientadas al procesamiento de imágenes, con el objetivo de incrementar la productividad a alcanzar por cada una de ellas. La primera aplicación corresponde a una implementación del compresor de vídeo MPEG2 mediante una estructura pipeline, la segunda aplicación, IVUS (*IntraVascular UltraSound*), es utilizada en el procesamiento de imágenes médicas y la tercera, BASIZ (*Bright and Saturated Image Zones*), que se encarga de la detección de regiones en secuencias de vídeo, que representan a las zonas que focalizan la atención primaria del ojo humano.

Debido a las diferentes características particulares de cada aplicación, la técnica de definición de la estructura de dependencias de tareas para la aplicación varía, como se indica a continuación.

(a) Compresor de vídeo MPEG2. Esta aplicación procesa una secuencia de video que corresponde a un flujo de datos con dependencias entre ellos. Por este motivo, en este caso experimental se ha utilizado la Técnica de Paralelización.

El entorno de ejecución utilizado ha sido el cluster de supercomputación.

(b) IVUS (*IntraVascular UltraSound*). La aplicación IVUS, procesa una secuencia de imágenes sin dependencias entre ellas, además sus tareas no pueden ser paralelizadas . Por esta razón se ha utilizado en este caso la Técnica de Replicación.

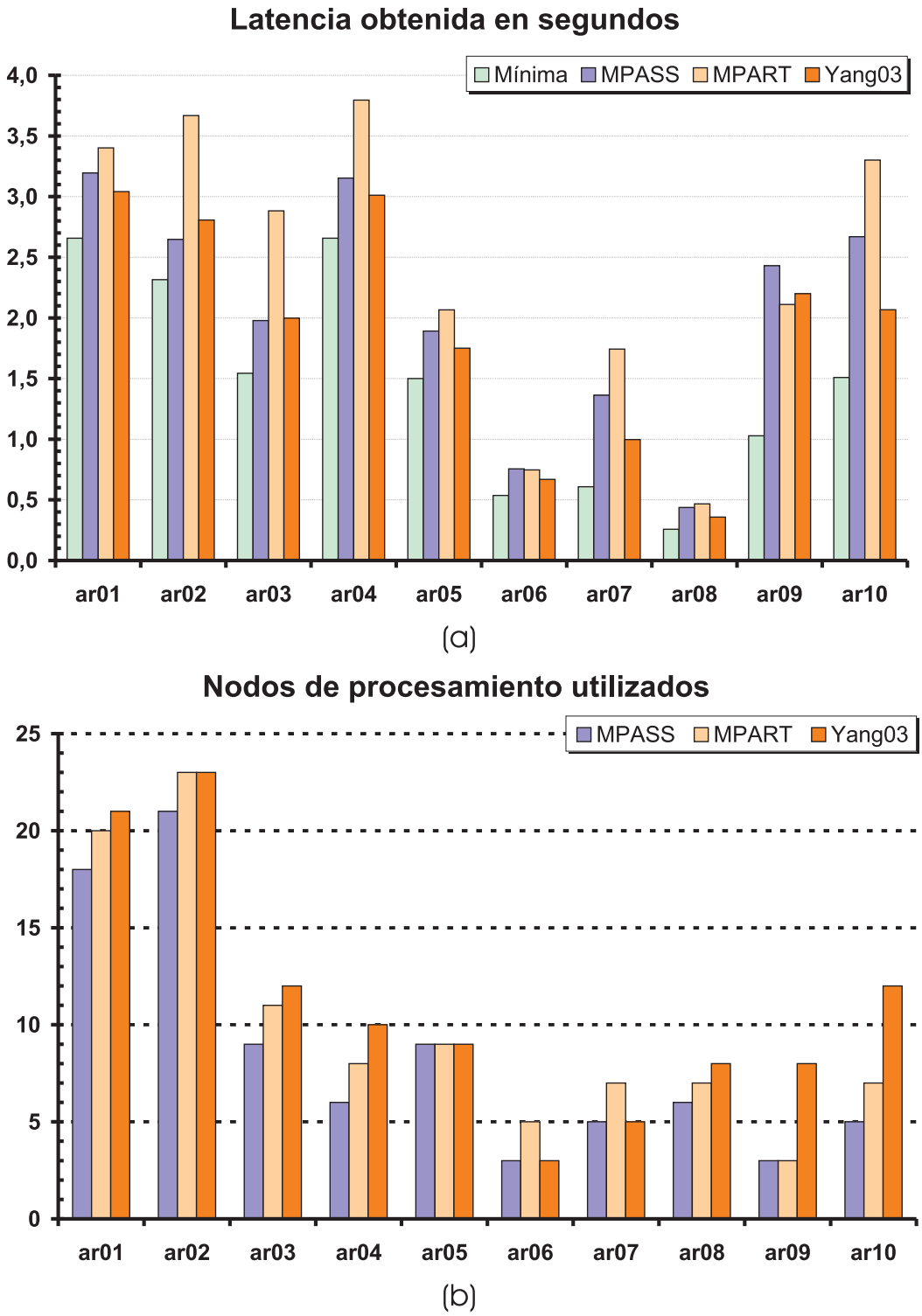


Figura 4.11: Aplicaciones arbitrarias: (a) Valor de latencia obtenida. (b) Nodos de procesamiento utilizados.

La aplicación ha sido ejecutada en el cluster de producción, debido a los requisitos de librerías que necesita para su ejecución. También se ha utilizado el entorno de simulación *p*MAP en aquellos casos en los que el número de nodos de procesamiento necesarios ha excedido del disponible en el cluster de producción.

(c) BASIZ (*Bright and Saturated Image Zones*). BASIZ es una aplicación que procesa una secuencia de imágenes en la que no existe dependencias entre ellas, y en la que algunas de sus tareas, por la función que implementan, pueden ser paralelizadas. En este caso se ha podido aplicar de forma conjunta la Técnica de Paralelización y de Replicación.

El estudio de esta aplicación se ha realizado sobre el entorno de simulación *p*MAP debido a que el número de procesadores necesarios tras obtener la nueva estructura y definir el mapping da lugar a unos requisitos de nodos de procesamiento que los clusters utilizados en los apartados de experimentación previos no son capaces de aportar.

En los siguientes apartados se presenta brevemente cada una de las aplicaciones a estudio así como los resultados obtenidos en la experimentación.

4.4.1. Compresor de video MPEG2

Las aplicaciones más habituales en las que se tratan flujos de datos, son aquellas que procesan secuencias de vídeo. El uso de estas aplicaciones es variado, siendo alguno de ellos, la compresión y descompresión, la transcodificación, la edición en tiempo real, los quioscos interactivos, el seguimiento y reconocimiento de objetos, etc. [GYK+00][YKS03][RNR+03][LVK05].

Este apartado de la experimentación, se centrará en el caso del compresor de video MPEG2 según la norma ISO/IEC 13818, la cual define la estructura del flujo de datos de la secuencia de vídeo sin entrar en la forma en que ésta debe ser generada [IS04]. Este tipo de compresión en la actualidad tiene una gran relevancia debido a su uso en múltiples plataformas de transmisión de video, DVD (*Digital Video Disc*), TDT (*Terrestrial Digital Television*), SAT-TV (*Satellite Television*), CATV (*Cable Television*), así como en un futuro próximo, con mayores prestaciones con la implantación de la televisión de alta definición HDTV (*High Definition Television*).

Una secuencia de video MPEG2 está formada por un flujo de datos que representa, de forma multiplexada, la información de las imágenes que forman la secuencia de video

y las pistas de audio correspondientes, utilizando una menor cantidad de información de la que la secuencia original requiere. En nuestro estudio solo se ha tratado la secuencia de video.

La codificación de las imágenes en movimiento se representa mediante el uso de una estructura denominada GOP (*Group of Pictures*). Cada GOP engloba una breve secuencia de imágenes denominadas cuadros, siendo 15 el número habitual de cuadros que forman cada GOP de 15. Los cuadros, en función de como son codificados, pueden ser de tres tipos diferentes:

- Cuadros I (*Intra frames*): Son los únicos estrictamente necesarios. Cada cuadro se comprime utilizando la compresión "*Intra-frame DCT coding*" usada en las imágenes JPEG. En el caso de que la estructura GOP esté formada exclusivamente por cuadros I, ésta correspondería a una secuencia codificada según el formato MJPG (*Motion JPEG*). Esta codificación no tiene pérdida de información y como resultado genera un gran volumen de datos.

- Cuadros P (*Predictive frames*): En éstos se almacenan, de forma comprimida, los cambios existentes entre la imagen que se procesa en este momento y la representada en el cuadro I o P previo. De esta forma se reduce la cantidad de datos necesaria para representarla.

- Cuadros B (*Bidirectionally-predictive frames*): En éstos, al igual que en el caso de los cuadros P, sólo se almacenan las diferencias existentes entre cuadros, pero en este caso usando como referencia cuadros I o P anteriores y posteriores. Estos cuadros son los que alcanzan un mayor ratio de compresión.

Dentro de la estructura GOP, los diferentes cuadros se colocan según un orden preestablecido, siendo el más habitual I-BB-P-BB-P-BB-P-BB, tal y como se muestra en la Figura 4.12 en la que se ha indicado las dependencias en la codificación de los cuadros. Con un aumento en el número de cuadros B se reduce significativamente el tamaño de la secuencia comprimida, aumentando la posibilidad de la aparición de errores visibles en la secuencia de vídeo. Además, el cómputo necesario para obtener esta secuencia se incrementa sustancialmente.

El diseño del compresor de video MPEG2 está formado principalmente por dos elementos bien diferenciados: el apartado compresor y el corrector. Estos pueden observarse en la Figura 4.13 en la que se muestra el diagrama de bloques básico de un compresor MPEG2. A continuación se comenta brevemente su funcionamiento.

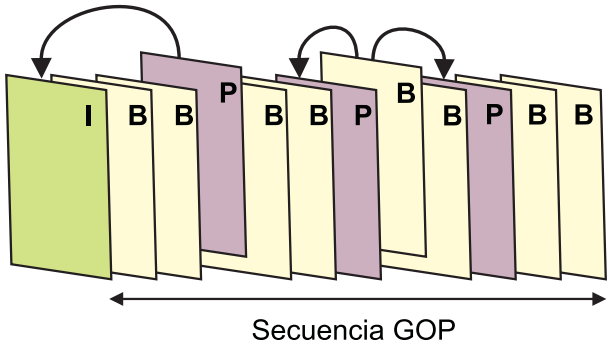


Figura 4.12: Estructura de una secuencia GOP y dependencias entre los diferentes cuadros.

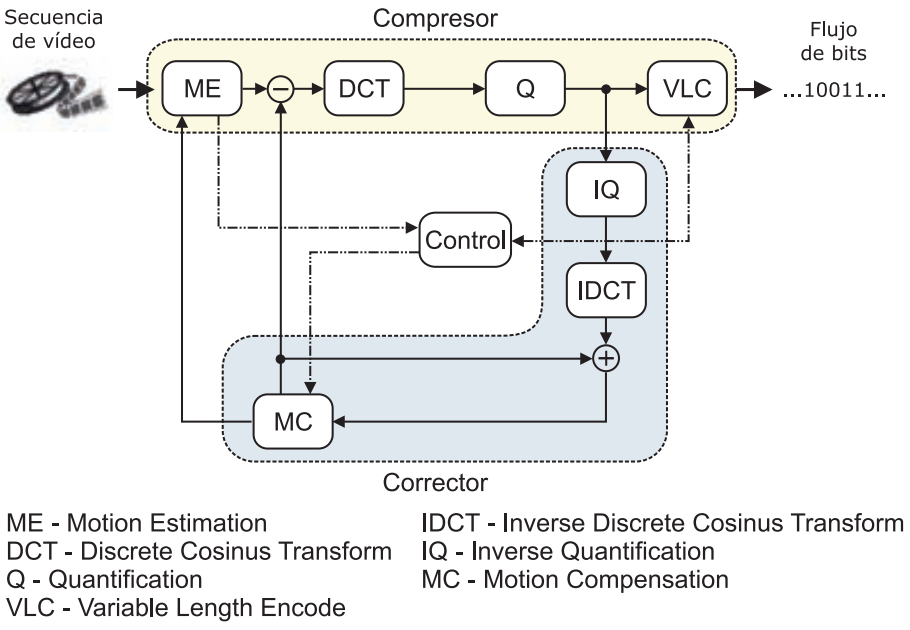


Figura 4.13: Diagrama de bloques de un compresor MPEG2.

El bloque compresor se encarga de tratar los cuadros de la secuencia eliminando la información redundante en ella tratada ésta a dos niveles, el primero de ellos como *redundancia temporal*, en el que se localizan los elementos repetidos en las imágenes adyacentes, y el segundo como *redundancia espacial*, en la que se trata la información repetida en la propia imagen.

El caso de la redundancia temporal, presupone que la imagen actual se ha formado a partir de la modificación de las imágenes previas, en la que los elementos presentes en éstas han cambiado de ubicación hasta alcanzar la posición actual. De esta forma lo que se intenta es localizar éstos elementos y determinar que vector de movimiento representa su desplazamiento, siendo este vector la información que se transmitirá en el flujo de datos. Para determinar los vectores de movimiento, cada imagen se divide en tiras denominadas *slices* y estas a su vez en regiones de tamaño prefijado, denominadas macrobloques, tal y como muestra la Figura 4.14. Cada macrobloque es tratado en el modulo ME (*Motion Estimation*), a partir del frame anterior o del siguiente, según el tipo de cuadro a generar, estimando la cantidad de movimiento en el intervalo entre frames.

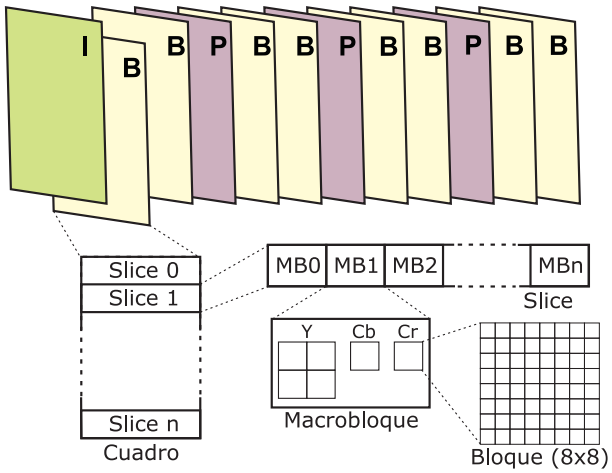


Figura 4.14: Estructura de un cuadro en la secuencia de vídeo.

En el caso de la redundancia espacial, se utilizan técnicas basadas en la transformada DCT (*Discrete Cosinus Transform*), cuya operación básica es la de transformar el conjunto de puntos de la imagen del dominio espacial a una representación idéntica en el dominio frecuencial. La DCT obtiene una matriz de coeficientes ordenados que pasan por una fase de cuantificación, en el modulo Q (*Quantification*) del compresor, en la que se mantienen los valores más significativos y se eliminan aquellos que no son necesarios para almacenar la imagen.

Finalmente en la fase de codificación VLC, se realiza una compresión de toda la información obtenida en los pasos previos, basándose en el algoritmo RLE (*Run Length Encode*) que reduce el tamaño final de la secuencia de bits.

La parte correspondiente al corrector, sin ser obligatoria ni necesaria para el funcionamiento del compresor, se utiliza con la finalidad de aumentar la calidad final de la secuencia de video comprimida. Para ello a medida que se van procesando los cuadros, el resultado que se obtiene desde el compresor, vuelve a ser descomprimido, emulando la acción que realizará el receptor de la secuencia de vídeo. El resultado se compara con la imagen original reajustando los valores de predicción tomados en el módulo de predicción de movimiento. Este reajuste genera unos valores de compensación de error que se añaden a la información para la predicción.

Debido a las características del compresor, se observa que el orden en que se evalúan las imágenes en la secuencia de vídeo es muy importante, debido a que la codificación de una imagen en particular requiere de datos de otras imágenes, situación que hace que la secuencia de vídeo esté dentro de la definición de flujo de datos dependiente.

Resultados experimentales

La implementación paralela utilizada en este apartado de la experimentación, para el compresor MPEG2, se ha basado en la implementación secuencial y de libre distribución realizada por el MPEG Software Simulation Group [Gro], adaptándola mediante la librería de paso de mensajes MPI, a una estructura en forma de pipeline. En esta estructura, mostrada en la Figura 4.15, las diferentes funciones del compresor se han definido como tareas individuales, a excepción de las funciones Q y VLC que se han agrupado en una única tarea. Esta nueva estructura, elimina la vuelta atrás del bloque corrector, por lo que se ha incorporado a esta estructura la tarea MC que hace la función de corrección tomando como referencia la imagen original y la imagen obtenida de la tarea ME.

Debido a la existencia de dependencias entre las imágenes a procesar, las tareas almacenan de forma local la información necesaria para poder llevar a cabo correctamente la codificación. La secuencia de video utilizada en la experimentación está formada por 399 imágenes de dimensión 720x576 pixels, almacenadas en disco y codificadas cada una de ellas mediante tres archivos que representan el formato de color YUV (4:1:1)[1].

[1]El modelo YUV define un espacio de color en términos de una componente de luminancia y dos componentes de crominancia.

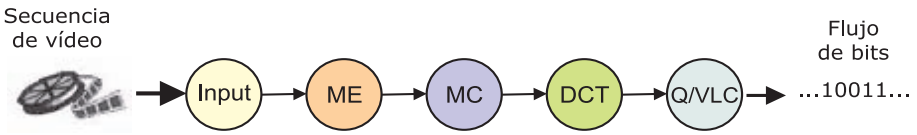


Figura 4.15: Estructura en forma de pipeline para el compresor MPEG2.

En la literatura existen diversos trabajos relacionados con la paralelización del compresor MPEG2. En la mayoría de ellos, la paralelización del compresor se trata desde la perspectiva del paralelismo de datos, mediante una implementación *Master-Worker*, en la que la imagen es dividida en diferentes franjas, que engloban varios *slices* y que son entregados a cada worker para ser procesados. [AAL95][IO98][AHL02].

Esta idea del procesamiento por franjas de las imágenes ha sido utilizada en la adaptación de las funciones a la estructura pipeline. De esta forma cada una de las tareas puede ser dividida en varias subtareas a las que se les puede entregar para procesar una franja de la imagen. Esta característica permite aplicar la Técnica de Paralelización comentada en la sección 3.1.1 y que requiere de la caracterización del tiempo de cómputo de las tareas implementadas. Para ello se ha ejecutado la aplicación evaluando el tiempo de cómputo de éstas en el procesamiento de la imagen original, franjas de dimensión 720×288, 720×192 y 720×96, equivalentes a 2, 3 y 4 veces inferior al tamaño de la imagen original. Los tiempos de cómputo obtenidos en segundos para cada tarea se muestran en el Cuadro 4.7.

Función	720×576	720×288	720×192	720×96
Input	0,017521	0,007774	0,003757	0,001047
ME	**0,483594**	**0,238346**	**0,123112**	**0,072527**
MC	0,013021	0,003581	0,000794	0,000522
DCT	0,002669	0,001098	0,000537	0,000153
Q/VLC	0,060677	0,030208	0,017383	0,009766

Cuadro 4.7: Tiempo de cómputo, en segundos, de las funciones según la dimensión de la franja a procesar.

Como se puede observar la función ME, que aparece remarcada en la tabla, es la que posee un mayor tiempo de cómputo, y por lo tanto es ésta la que marca el ritmo de procesamiento de la aplicación pipeline.

A partir de los datos anteriores se ha obtenido mediante interpolación las siguientes funciones que representan el tiempo de cómputo de las tareas según el número de divisiones de la imagen original y que equivale al número de subtareas, $n\_subtareas$,

a obtener.

$$\begin{aligned} tiempo\_cómputo(Input) &= 0,0477e^{-0,918\times n\_subtareas} \\ tiempo\_cómputo(ME) &= 0,8767e^{-0,6352\times n\_subtareas} \\ tiempo\_cómputo(MC) &= 0,0341e^{-1,1154\times n\_subtareas} \\ tiempo\_cómputo(DCT) &= 0,0072e^{-0,9298\times n\_subtareas} \\ tiempo\_cómputo(Q/VLC) &= 0,1067e^{-0,6033\times n\_subtareas} \end{aligned}$$

Con esta información se ha aplicado la Técnica de Paralelización utilizando como parámetro de IP el valor obtenido de dividir el tiempo de cómputo de la tarea ME por 2, 4, 8, 16, 32, 64 y 128, lo que permite aumentar aproximadamente en igual medida la productividad. Estos casos aparecen etiquetados en el resto del apartado como x2, x4, x8, x16, x32, x64 y x128 respectivamente. De esta forma, para obtener el número de subtareas, para cada tarea se ha de encontrar el valor $n\_subtareas$ que hace que se cumpla la igualdad:

$$IP = tiempo\_cómputo(tarea) \tag{4.3}$$

Los valores obtenidos de la evaluación de la expresión (4.3) para cada tarea se muestran en el Cuadro 4.8(a). Se debe tener presente que no es válida cualquier dimensión para las franjas a crear, si no que éstas han de tener una dimensión adecuada que permita que sean procesadas, por este motivo el número de subtareas reales se redondea por exceso al valor que permite utilizar esta dimensión, tal y como se muestra en el Cuadro 4.8(b). Así puede observarse como la tarea ME en el caso x8, cuyo valor inicial de subtareas a obtener es de 4,2102 finalmente serán seis, procesando cada una de ellas una franja con 96 líneas de altura.

En la sección 3.1.1 se comentó el efecto negativo de las comunicaciones debido a la obtención de un excesivo número de subtareas. Para determinar si se da esta situación, se ha analizado la red de interconexión presente en la arquitectura utilizada para la ejecución de la aplicación. Para ello se ha evaluado los tiempos de transmisión para tamaños de mensaje equivalentes a los que se transmiten en cada uno de los casos de estudio. En este análisis, se ha determinado que el tiempo involucrado en las comunicaciones y la sobrecarga añadida no afecta al número de subtareas a obtener debido a que el tiempo de ejecución de las subtareas es mucho mayor.

Con estos datos, y tras aplicar la Técnica de Paralelización, se ha obtenido el

Caso	IP	*n_división* Input	*n_división* ME	*n_división* MC	*n_división* DCT	*n_división* Q/VLC
x2	0,241797	-	2,0278	-	-	-
x4	0,120898	-	3,1190	-	-	-
x8	0,060449	-	4,2102	-	-	0,9418
x16	0,030225	-	5,3015	-	-	2,0908
x32	0,015112	1,2520	6,3927	-	-	3,2397
x64	0,007556	2,0072	7,4839	1,3510	-	4,3886
x128	0,003778	2,7622	8,5751	1,9724	-	5,5375

(a)

Caso	IP	*subtareas* Input	*subtareas* ME	*subtareas* MC	*subtareas* DCT	*subtareas* Q/VLC
x2	0,241797	1	3	1	1	1
x4	0,120898	1	4	1	1	1
x8	0,060449	1	6	1	1	1
x16	0,030225	1	6	1	1	3
x32	0,015112	2	8	1	1	4
x64	0,007556	3	8	2	1	6
x128	0,003778	3	12	2	1	6

(b)

Cuadro 4.8: (a) Valores obtenidos de la expresión (4.3) para cada tarea. (b) Número de subtareas a obtener.

conjunto de subgrafos, tareas y subtareas en que se deben paralelizar mostrados en el Cuadro 4.9. En el Apéndice A.0.7 aparecen los grafos que representan estas estructuras.

Caso	$Subgrafo$ - SG	n_copias[SG]	$(n\_subtareas(T_i))$
x2	{ME}	3	(1)
x4	{ME}	4	(1)
x8	{ME}	6	(1)
x16	{ME},{Q/VLC}	6,3	(1),(1)
x32	{Input,ME},{Q/VLC}	2,1	(1,4),(4)
x64	{Input,ME,MC},{Q/VLC}	2,1	(2,4,1),(6)
x128	{Input,ME,MC},{Q/VLC}	2,1	(2,6,1),(6)

Cuadro 4.9: Relación de subgrafos y subtareas obtenidos tras aplicar la Técnica de Paralelización.

Sobre la nueva estructura obtenida en cada caso para la aplicación, se ha aplicado la heurística de mapping MPASS, basada en el criterio de productividad.

Con el fin de tener una estimación de como de buenos son los resultados obtenidos, se ha ejecutado una implementación clásica *Master-Worker* [AHL02], en la que a cada nodo de procesamiento se le asigna las funciones ME, MC y DCT, tal y como se muestra en la Figura 4.16. En esta comparación, la ejecución de la implementación master-worker no ha seguido el criterio de obtener una productividad en concreto, si no que solo se ha pretendido comparar el rendimiento obtenido utilizando el mismo número de nodos de procesamiento que en la versión implementada con estructura pipeline.

La elección del número workers es el mismo número de procesadores que los utilizados en cada caso en la ejecución de la versión pipeline, reservando uno de ellos para la tarea master. Al igual que en la versión pipeline, el número de workers se ha ajustado para que coincida con una dimensión para la franja a procesar válida, situación que ha provocado que en algún caso el número de workers sea superior al que se toma como base. En el Cuadro 4.10 se muestra el número de procesadores utilizados en la versión pipeline y master-worker así como la dimensión de las franjas procesadas en cada uno de los workers.

El resultado obtenido se muestra en la Figura 4.17, indicando el número de imágenes procesadas por segundo para la implementación pipeline y master-worker así como el valor teórico deseado.

Se puede observar como la versión pipeline sigue la tendencia del valor de productividad teórico demandado en la Técnica de Paralelización y aplicando el mapping

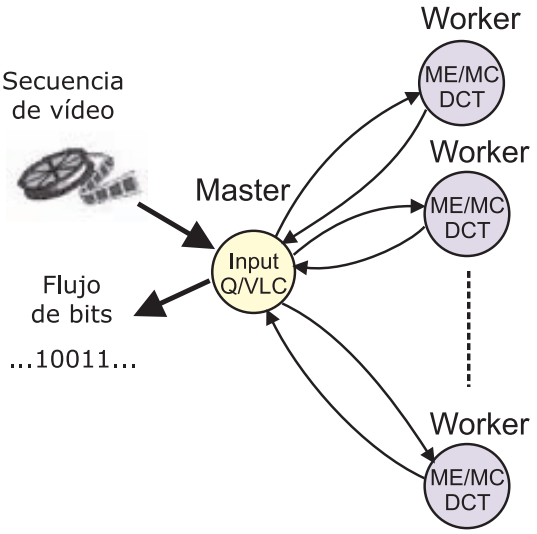


Figura 4.16: Estructura master-worker para el compresor MPEG2.

Caso	Proc. Pipeline	Nodos Worker	Dimensión de la franja
x1	3	2	720×576
x2	5	4	720×144
x4	6	6	720×96
x8	9	8	720×64
x16	11	12	720×48
x32	15	16	720×36
x64	16	16	720×96
x128	22	24	720×24

Cuadro 4.10: Número de nodos worker utilizados en la versión master-worker.

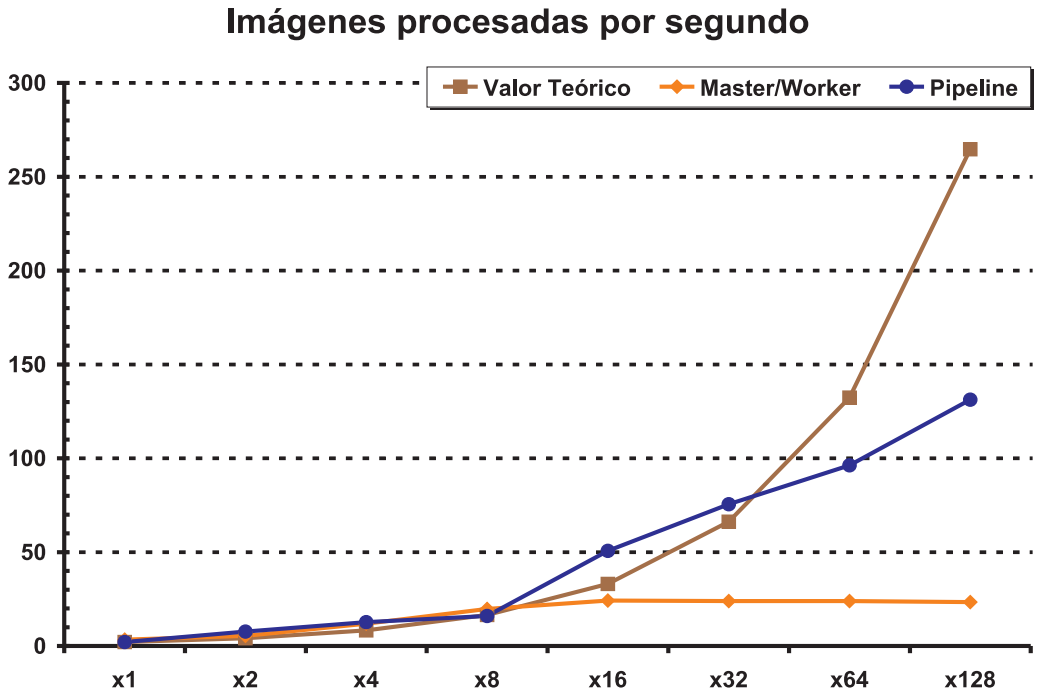


Figura 4.17: Productividad obtenida para el compresor MPEG2.

posterior. Hay que destacar como en algunos de los casos incluso supera a este valor, debido a que usa un mayor número de subtareas de las que eran necesarias inicialmente, por la necesidad de adecuar el número de subtareas a la dimensión de la franja a procesar. Cuando el valor de productividad exigido aumenta, como son los casos x64 y x128, la técnica se aleja de los valores teóricos, debido a que la granularidad de las tareas es muy pequeña y comienzan a notarse los efectos de las comunicaciones.

En la implementación mastre-worker, a medida que se aumenta el número de workers, esta implementación escala correctamente hasta el caso x16. A partir de este punto se usan 12, 16 y 24 workers (Cuadro 4.10), lo que hace que la tarea master se comporte como un cuello de botella, ya que debe esperar a recibir cada uno de los resultados del procesamiento de los workers, antes de poder comenzar a iniciar el procesamiento de la siguiente imagen. Esto unido con la baja granularidad de las tareas worker, hace que la utilización de los nodos de procesamiento sea muy baja. En el caso de la versión pipeline, esta situación no se da debido a que por su forma de procesar, en un momento dado existe más de una imagen que está siendo procesada en las diferentes tareas de la aplicación avanzando sus resultados por la estructura pipeline. De esta forma es capaz de escalar mejor alcanzando una mayor productividad.

Tras evaluar los resultados obtenidos se puede llegar principalmente a dos conclu-

siones:

a. La Técnica de Paralelización, conjuntamente con el mapping utilizado, es capaz de obtener unos resultados de productividad que se aproximan a los deseados, convirtiéndose en una herramienta muy útil para las aplicaciones de procesamiento de flujo de datos dependientes.

b. Las implementación en forma de pipeline del compresor MPEG2, es capaz de escalar mejor que la implementación clásica master-worker, lo que indica que una estructura pipeline es una buena alternativa en la definición de la estructura para las aplicaciones que procesan flujos de datos de entrada, para la obtención de una mayor productividad.

4.4.2. IVUS (*IntraVascular UltraSound*)

En el comienzo de los años noventa, la técnica IVUS cobra un creciente protagonismo en el estudio de las enfermedades cardiovasculares y en particular en las patologías coronarias. A diferencia de otras técnicas, como puede ser la coronariografía, éste es un método que, por medio de un pequeño transductor montado en un catéter que se introduce en las arterias y que emite ultrasonido, permite ver "in vivo" la pared de los vasos y las alteraciones que las patologías coronarias pueden producir. Por lo tanto, con este método es posible "ver" la enfermedad coronaria y no inferir su presencia como sucede con otras técnicas.

El conjunto de imágenes capturadas por el transductor de ultrasonidos es obtenido a medida que el catéter se retira del vaso sanguíneo para ser más tarde procesadas, cada unas de ellas de forma individual, mediante técnicas semiautomáticas o en algunos casos de forma manual por parte de especialistas en la materia. Al realizar el análisis sobre la secuencia completa de imágenes de esta forma, se hace lento y sin la capacidad de tener una visión del conjunto de la porción del vaso sanguíneo en estudio. Es por ello que es necesaria la introducción de herramientas informáticas que ofrezcan la posibilidad de procesar de la forma más rápida posible la información adquirida.

Las herramientas existentes se basan en algoritmos de visión por computador cuya misión consiste en detectar, en función del tipo de análisis a realizar, las regiones de interés para cada una de las imágenes de la secuencia de entrada. Estas regiones corresponden a capas, denominadas túnicas, que definen los contornos de las fronteras en la estructura vascular, la adventicia y la íntima. El principal requerimiento de la comunidad médica para estas herramientas consiste en obtener una alta velocidad en

la respuesta y el que el proceso sea independiente del especialista que realiza el estudio de las imágenes.

En este apartado experimental, la aplicación utilizada se encarga de la caracterización de la túnica adventicia y ha sido desarrollada en el CVC (*Centre de Visió per Computador*) de la Universitat Autònoma de Barcelona en la línea de investigación de imágenes médicas [GHC+05][GHR+06]. La implementación original de la aplicación se basa en un programa serie y programado en el entorno MatLab, el cual procesa una secuencia de entrada formada por bloques independientes entre sí de 20 imágenes cada uno de ellos.

La aplicación se puede descomponer en cuatro etapas básicas que se comentan a continuación y que ha servido como base para la creación de la versión paralela utilizando una estructura pipeline.

1. *Caracterización de la zona de interés.* Cada una de las imágenes capturadas desde el transductor, representa un corte transversal del vaso sanguíneo, el cual se observa en forma circular. Así en una fase de preprocesamietno, se reconvierte la imagen al sistema de coordenadas polar lo cual facilita la forma en que será tratada en las etapas posteriores. En esta nueva representación, la adventicia aparece como una franja horizontal obscura.

   A partir de aquí y mediante el uso de algoritmos RAD (*Restricted Anysotropic Difussion*) se limpia la imagen eliminando el ruido presente, realzando la zona de interés, la cual queda enmarcada con la finalidad de reducir el área de estudio. La Figura 4.18 muestra estos primeros pasos y el resultado que se obtiene sobre una imagen obtenida desde el catéter.

2. *Caracterización de la adventicia.* Tras preparar la imagen, se aplican tres filtros que actúan sobre diferentes aspectos: realzar los bordes horizontales, desviación radial estándar y media acumulativa radial. Estos filtros obtienen la información necesaria para discriminar los elementos que aparecen en la imagen y que representan a los cuatro componentes que el transductor ofrece tras capturar la imagen: la adventicia, la placa cálcica, la estructura fibrosa y otros elementos no útiles en el estudio a realizar.

   El resultado de los tres filtros es puesto en común para definir la máscara que corresponde a la adventicia y a la placa cálcica. La Figura 4.19 muestra el resultado obtenido de aplicar cada uno de los filtros y las máscaras que se definen.

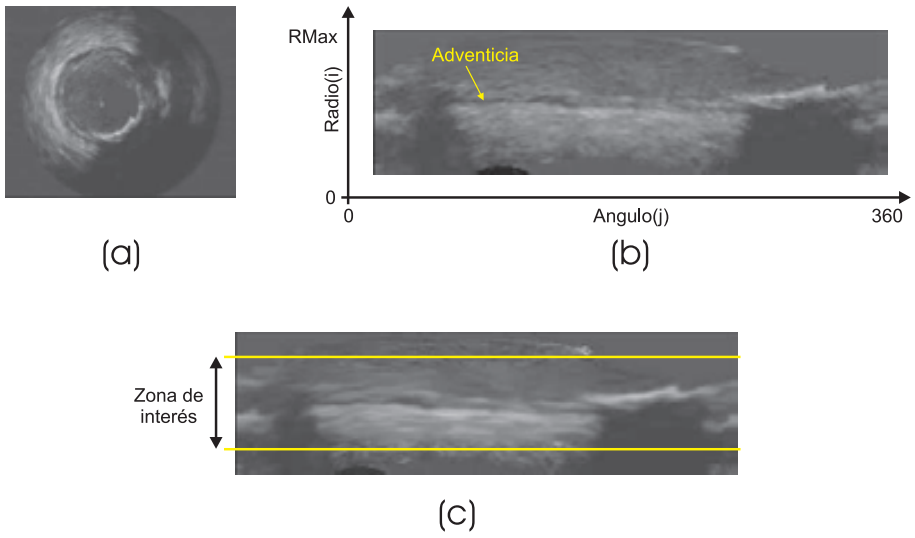


Figura 4.18: (a) Imagen original, (b) imagen en coordenadas polares y (c) zona de interés.

3. *Interpolación de los segmentos de adventicia.* Mediante el uso de la técnica de interpolación ACC (*Anysotropic Contour Closing*), se obtiene la unión de algunos de los segmentos de la máscara de adventicia, eliminando los que forman la placa cálcica. El resultado de esta fase corresponde a un conjunto de segmentos individuales que representan partes inconexas de la adventicia. Para finalizar la etapa se aplica un filtro con el fin de dotar al modelo continuidad 3D. En la Figura 4.20(a) se muestra el resultado de aplicar la técnica ACC.

4. *Unión de los segmentos de adventicia.* A continuación, la imagen se procesa mediante la técnica B-Snake, que permite unir los segmentos de adventicia, mediante el uso de curvas B-Splines.

   El resultado obtenido se retorna al sistema de coordenadas original, obteniendo finalmente la imagen en la que aparece remarcada la túnica adventicia. La Figura 4.20(b) muestra el resultado de la unión de los diferentes segmentos, y en la Figura 4.20(c) se observa el resultado final del procesamiento de la imagen en la que aparece remarcada la túnica adventicia.

Para caracterizar la aplicación se ha configurado el cluster de producción con el entorno de ejecución MatLab, ejecutando sobre uno de sus nodos la versión secuencial de la aplicación tras haberla instrumentalizado. Con la información obtenida se ha creado el grafo representa el comportamiento de la aplicación, desglosándola en las nueve tareas que implementan las funciones de las etapas comentadas anteriormente.

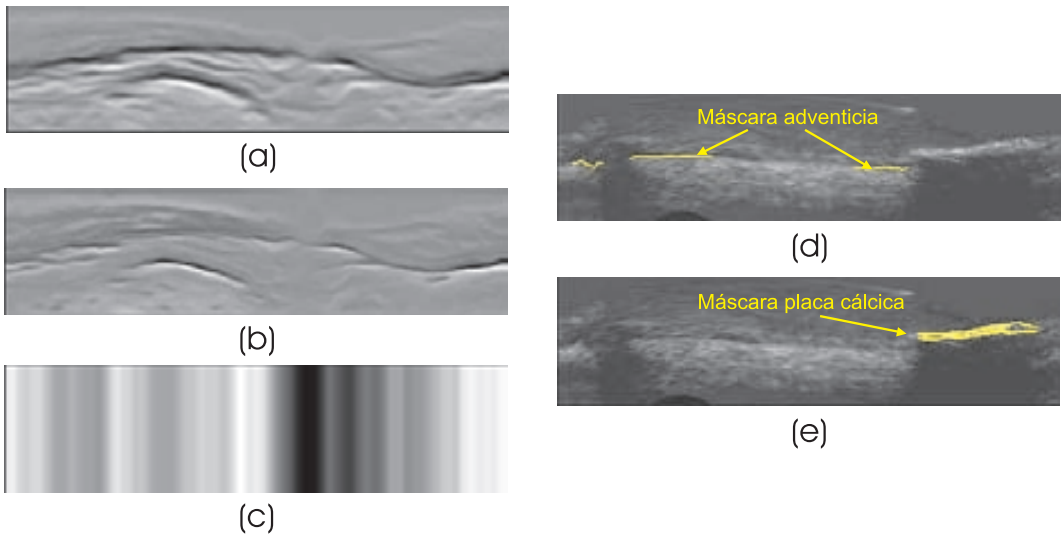


Figura 4.19: Filtros aplicados: (a) Bordes horizontales, (b) desviación radial estándar y (c) media acumulativa radial. Resultado obtenido: (d) Máscara de la adventicia y (e) máscara de la placa cálcica.

La Figura 4.21(a), muestra el grafo obtenido en el que aparecen las dependencias entre tareas y el volumen de datos, medidos en bytes, que intercambian entre ellas. En la Figura 4.21(b) se detalla el tiempo de cómputo asociado, medido en segundos, necesario para cada una de estas tareas para procesar un único bloque de 20 imágenes. La implementación secuencial ofrece una productividad de 0,006 bloques por segundo con una latencia promedio de 164,4 segundos.

Resultados experimentales

En la implementación paralela de la aplicación IVUS se ha tenido presente las dependencias de tareas mostrada en el grafo de tareas de la Figura 4.21 y el hecho de que su procesamiento se basa en un flujo continuo de datos, por lo que se ha optado por una estructura de aplicación pipeline, utilizando para ello la librería de paso de mensajes MPITB, sobre MatLab [FARB06]. De esta forma se aprovecha la capacidad de procesar más de un bloque de imágenes dentro de la estructura pipeline, así como el poder ejecutar más de una tarea de forma simultanea. La productividad obtenida de la implementación paralela pipeline, y utilizando la heurística de mapping MPASS tomando como criterio de optimización la obtención de la máxima productividad, ha sido de 0,011 bloques por segundo y una latencia de 171,5 segundos, que es solo un 4,3 % superior a la de la versión secuencial.

La productividad que se puede llegar a obtener de esta aplicación se restringe a la

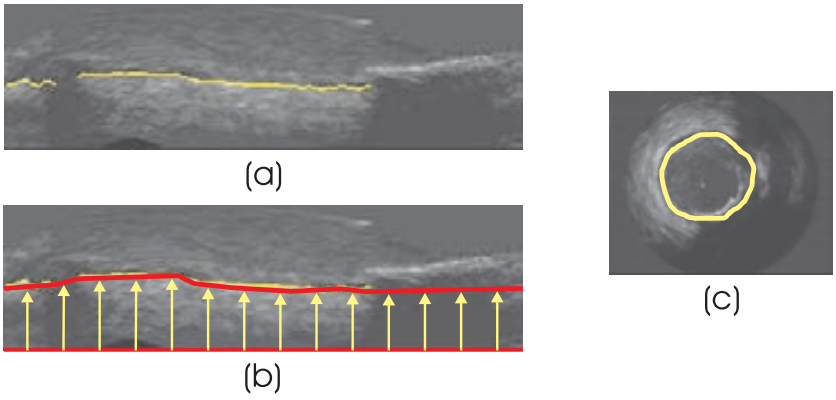


Figura 4.20: (a) Resultado tras aplicar ACC, (b) resultado tras aplicar B-Snake y (c) resultado final del proceso.

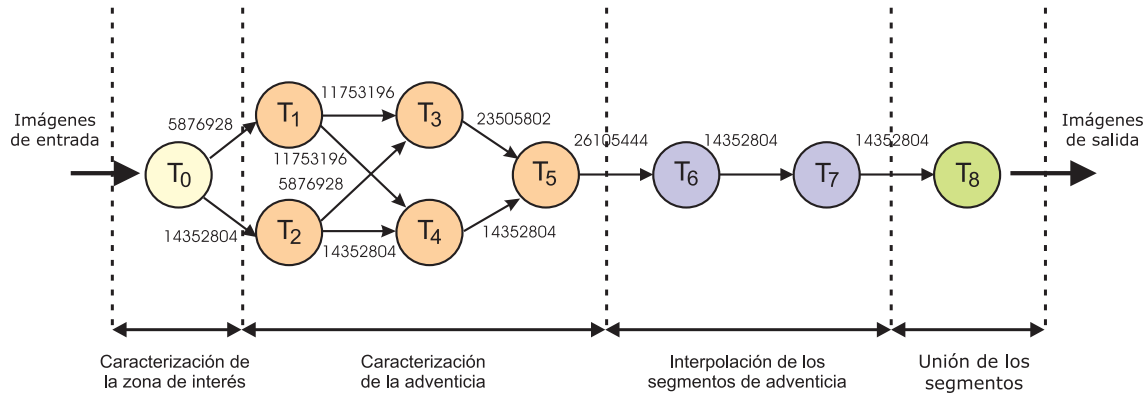
capacidad de procesamiento de la tarea T_0 encargada de aplicar la restricción anisotrópica, ya que es la que posee un mayor tiempo de cómputo, convirtiéndose en el cuello de botella de la aplicación.

La información facilitada desde el CVC, indica que las funciones de cada tarea no pueden ser paralelizadas, esto junto con el hecho de que la secuencia de bloques de imágenes a procesar forman flujo de datos sin dependencias hace que sea la Técnica de Replicación el mecanismo más adecuado para aumentar la productividad de la aplicación.

Por ello se ha aplicado dicha técnica junto con la heurística de mapping MPASS, marcando como objetivo la obtención de valores de productividad 2, 4, 8, 16, 32, 64 y 128 veces superior al que originalmente la aplicación es capaz de ofrecer, y que se mostrarán etiquetados en las figuras y cuadros de este apartado como x1, x2, x4, x8, x16, x32, x64 y x128 respectivamente. Se ha incluido también el caso etiquetado como x1, en el que aparece la aplicación paralela original sobre la que sólo se ha aplicado la heurística de mapping MPASS.

En el Cuadro 4.11 se muestra el resultado de aplicar la Técnica de Replicación, mostrando los subgrafos obtenidos, las veces en que éstos aparecen, así como las tareas que forman parte de ellos y su número de replicaciones. En el Apéndice A.0.8 aparecen los grafos de tareas que representan las estructuras obtenidas.

Para las nuevas estructuras de la aplicación IVUS se ha obtenido la asignación de tareas a nodos de procesamiento tras utilizar la heurística de mapping MPASS tomando como criterio de optimización el de productividad. El mapping obtenido para los casos x16, x32, x64 y x128 requiere de un número de nodos de procesamiento mayor del disponible en el cluster de producción, por este motivo para evaluar los resultados de



(a)

Tarea	Función	Tiempo(seg.)
T_0	Restricción anisotrópica	98,443
T_1	Bordes horiz. & Desv. estándar radial	0,475
T_2	Media acumulativa radial	0,301
T_3	Máscara de la placa cálcica	1,044
T_4	Máscara de la adventicia	0,111
T_5	Filtro/selección de continuidad 3D	2,903
T_6	ACC - Anisotropic Contour Closing	11,898
T_7	Selección de continuidad	2,826
T_8	B-Snake	53,283

(b)

Figura 4.21: IVUS: (a) Grafo de tareas de la aplicación IVUS. (b) Funciones asociadas a las tareas y tiempo de cómputo de éstas.

Caso	IP	$Subgrafo$ - SG	n_copias[SG]	$(n\_rep(T_i))$
x1	98,443	-	-	-
x2	49,222	$\{T_0\}$,$\{T_8\}$	2,2	(1),(1)
x4	24,611	$\{T_0\}$,$\{T_8\}$	4,3	(1),(1)
x8	12,305	$\{T_0\}$,$\{T_8\}$	8,5	(1),(1)
x16	6,153	$\{T_0\}$,$\{T_6\}$,$\{T_8\}$	16,2,9	(1),(1),(1)
x32	3,076	$\{T_0\}$,$\{T_6\}$,$\{T_8\}$	32,4,18	(1),(1),(1)
x64	1,538	$\{T_0\}$,$\{T_5, T_6, T_7, T_8\}$	64,2	(1),(1,4,1,18)
x128	0,769	$\{T_0\}$,$\{T_3, T_5, T_6, T_7, T_8\}$	64,2	(1),(1,2,8,2,36)

Cuadro 4.11: Técnica de Replicación: Subgrafos obtenidos, copias de éstos y tareas que los forman.

estos casos se ha utilizado el entorno de simulación *p*MAP, en el que se ha caracterizado la arquitectura del cluster de producción.

Los resultados obtenidos se presentan en dos bloques diferenciados. El primero, mostrado en la Figura 4.22(a), en el que aparecen los casos x2, x4 y x8, en el que se indica los valores de productividad obtenidos de la ejecución de las nueva estructuras de la aplicación IVUS sobre el cluster de producción, junto con los resultados que proporciona el simulador *p*MAP para los mismos casos de estudio. También se ha incluido el número de nodos de procesamiento utilizados en la ejecución para cada caso.

Se puede observar como la productividad que se obtiene de las nuevas estructuras de la aplicación IVUS aumenta siguiendo la misma tendencia que la productividad deseada, lo cual demuestra como estas estructuras son capaces de aumentar la concurrencia en el procesamiento del flujo de entrada, gracias a la creación de nuevas líneas de ejecución. El resultado obtenido de las simulaciones posee un comportamiento similar, aunque debido a que no es capaz de tener en cuenta todas las características presentes de la ejecución en el cluster, los valores son más cercanos al teórico deseado. Aún así éstos pueden ser tomados como referencia a la hora de definir la tendencia que sigue la productividad y nos sirven para validar el uso del simulador.

En los casos de estudio x16, x32, x64 y x128, los resultados mostrados en la Figura 4.22(b), son los obtenidos exclusivamente de la simulación realizada con el entorno *p*MAP. Para cada caso se ha incluido el número de nodos de procesamiento utilizados. Como se puede observar, a medida que se aumenta el requerimiento de productividad el resultado que se obtiene se distancia más del valor teórico deseado, debido principalmente al proceso de repartición de los bloques de imágenes y a la serialización de las comunicaciones, aún así los resultados muestran la capacidad que tienen las nuevas estructuras obtenidas y el mapping utilizado para escalar la productividad a medida que se crean nuevas líneas de ejecución.

4.4.3. BASIZ (*Bright and Satured Image Zones*)

La aplicación BASIZ (*Bright and Satured Image Zones*) está orientada al procesamiento de secuencias de vídeo, en las que se localiza y marca aquellas zonas que poseen un mayor brillo y saturación de color. Estas zonas representan las áreas que focalizan el interés primario del ojo humano. La aplicación está formada por 22 tareas, $\{T_0, .., T_{21}\}$, implementadas en lenguaje C junto con la librería de paso de mensajes PVM y organizadas en siete etapas funcionales diferenciadas tal y como se ilustra en

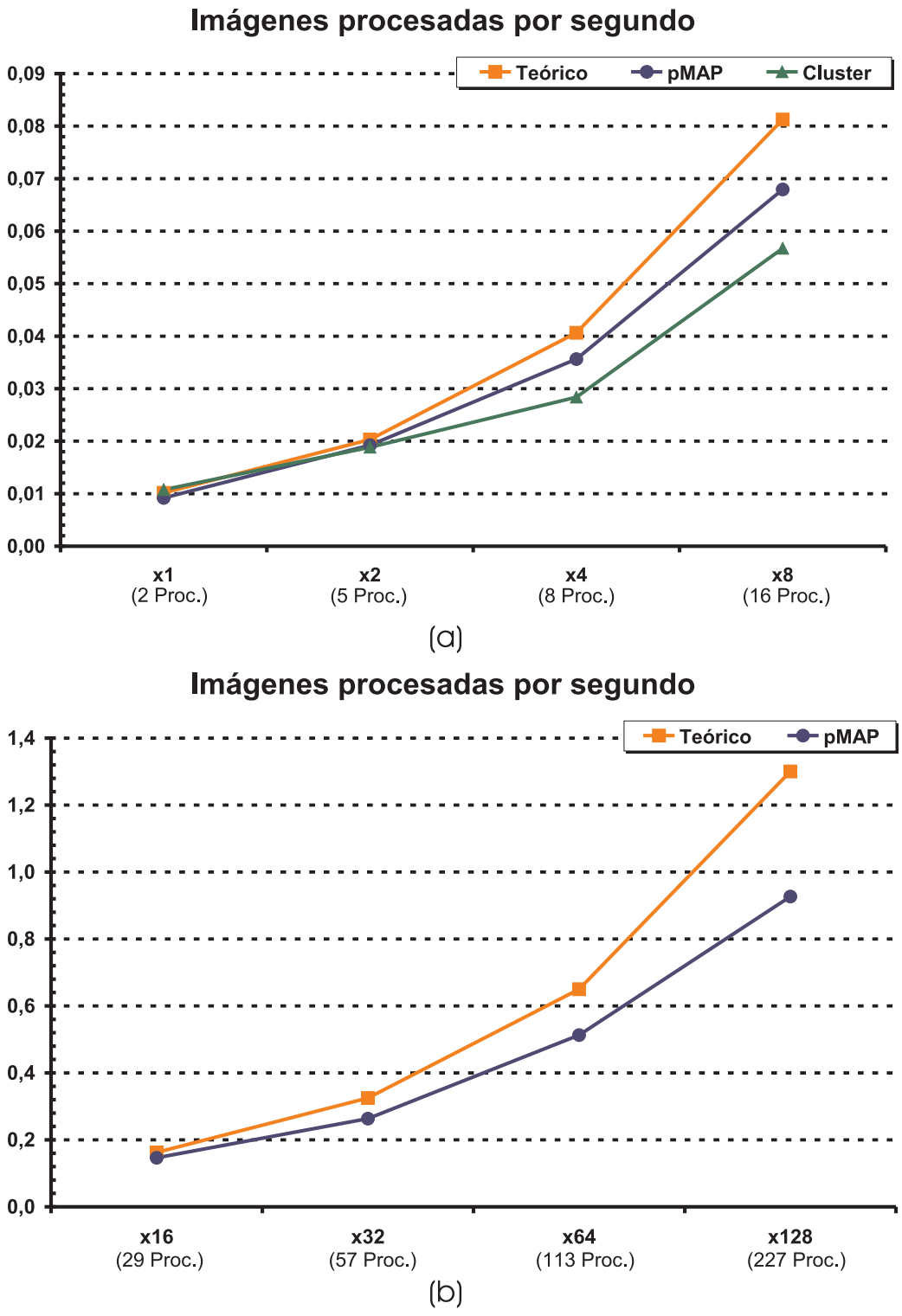


Figura 4.22: Productividad obtenida: (a) ejecución en el cluster de producción y utilizando el entorno de simulación *p*MAP, (b) utilizando el entorno de simulación *p*MAP.

la Figura 4.23.

Las funciones que se llevan a cabo en cada una de estas etapas son las siguientes:

- *Separación de las componentes de color.* Cada una de las imágenes se descompone en los tres colores primarios que serán procesados por separado en las siguientes etapas.
- *Difuminado.* A cada color se le aplica por separado tres filtros Gaussianos que realizan un difuminado que elimina las zonas de la imagen que no son relevantes.
- *Suma.* El resultado del difuminado, es unido mediante una función de suma en la que quedará exclusivamente las zonas más importantes de la imagen.
- *Mezcla.* Los resultados para cada color primario, son unidos formando una nueva imagen.
- *Conversión.* Para poder tratar las imágenes en las etapas posteriores, éstas son convertidas del formato RGB (*Red-Green-Blue*) al formato HSI (*Hue-Saturation-Intensity*).
- *Umbral.* A las imágenes se les aplica un umbral de intensidad y saturación, detectando las zonas que aún están presentes tras la etapa de difuminado y que son las más relevantes.
- *Marcado de las zonas.* Finalmente, las zonas que han quedado son marcadas sobre las imágenes originales.

Esta aplicación ha sido utilizada con las Técnicas de Paralelización y de Replicación conjuntamente para obtener una productividad muy superior a la que es posible alcanzar por la aplicación original. Tras la aplicación de estas técnicas se obtiene una nueva estructura del grafo de tareas para la aplicación BASIZ en la que aparecen nuevas tareas, en forma de subtareas para aquellas que han sido tratadas mediante la Técnica de Paralelización, o como copias de las ya existentes en el caso de ser tratadas por la Técnica de Replicación. Con la nueva estructura se ha procedido a aplicar la heurística de mapping MPASS dando lugar a una asignación que requiere un número de nodos de procesamiento superior a los disponibles en los clusters de computación utilizados para la experimentación previa. Por este motivo se ha utilizado el entorno de simulación *p*MAP [GRRL04a].

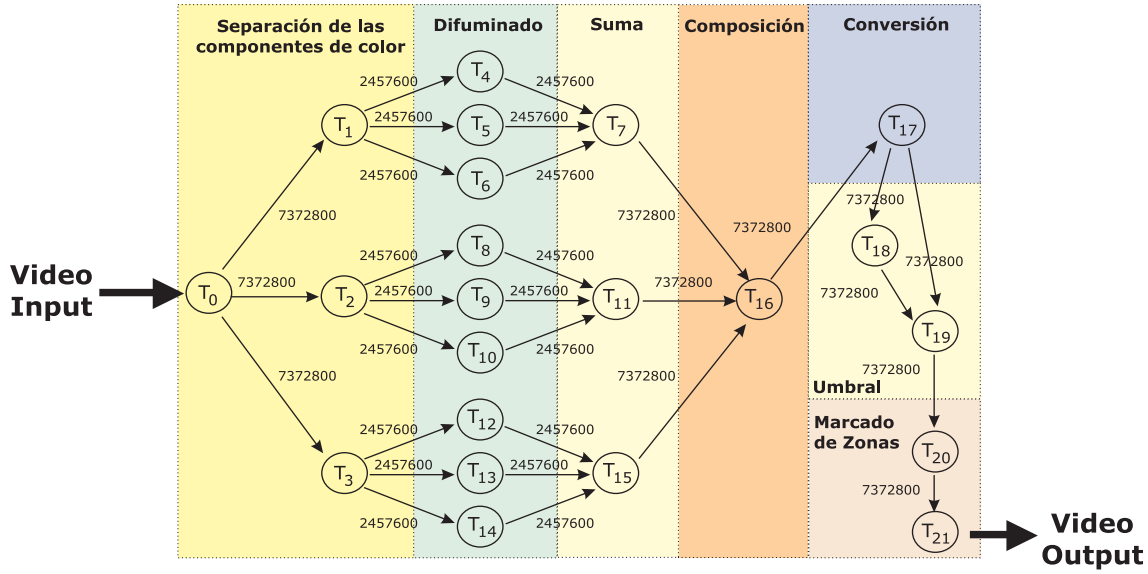


Figura 4.23: Grafo de tareas que representa la estructura de la aplicación BASIZ y etapas funcionales que la forman.

Para poder utilizar el entorno de simulación *p*MAP, previamente se ha de caracterizar el comportamiento de la aplicación, definiendo las dependencias entre sus tareas, los tiempos de cómputo de éstas y el volumen de datos que intercambian, así como las características de la arquitectura en la que se desea evaluar la ejecución de la aplicación.

En el caso de la aplicación BASIZ, su comportamiento se ha obtenido de la ejecución en el cluster de producción, y posteriormente mediante el análisis de las trazas obtenidas. El flujo de datos utilizado en el procesamiento corresponde a una secuencia de 200 imágenes de dimensión 1920×1280 píxeles y una profundidad de color de 24 bits. Cada una de las imágenes ocupa un total de 7372800 bytes, mostrándose en la Figura 4.23 el volumen de datos transferidos entre las tareas para cada imagen.

En la Figura 4.24 se muestra el tiempo de ejecución de cada una de las tareas en el procesamiento de una imagen, así como la función que tienen asignada.

A partir del conocimiento que se tiene de las funciones implementadas en las tareas, se ha identificado aquellas que por su funcionalidad pueden ser paralelizadas. Estas tareas son $\{T_4, T_5, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14}, T_{15}\}$, correspondientes a las tareas que forman parte de la fase de difuminado y a las tareas de la etapa de suma, encargadas de combinar el resultado obtenido de la etapa anterior. Estas tareas son tratadas mediante la Técnica de Paralelización presentada en la sección 3.1.1, determinando el número de subtareas que se debe crear para cada una de ellas y los subgrafos de los que forman parte.

Etapa	Tarea	Function	Tiempo (mseg.)
Separación de las componentes de color	T_0	Decoder	113.0
	T_1	RLESwap	78.0
	T_2		75.5
	T_3		70.0
Difuminado	T_4	Smush (Gaussian Filter level3)	677.5
	T_8		1311.5
	T_{16}		1966.5
	T_5	Smush (Gaussian Filter level4)	896.5
	T_9		1756.0
	T_{13}		2584.0
	T_6	Smush (Gaussian Filter level5)	1087.5
	T_{10}		2190.0
	T_{14}		3245.0
Suma	T_7	Add	113.0
	T_{11}		166.0
	T_{15}		227.5
Mezcla	T_{16}	MergeChannels	135.0
Conversión	T_{17}	RGB2HSV	353.0
Umbral	T_{18}	ThrDetect	85.5
	T_{19}	Threshold	93.5
Marcado de Zonas	T_{20}	Detect-Bobs	10.0
	T_{21}	Mark-Bobs	115.0

Figura 4.24: Funciones asociadas a las tareas y su tiempo de cómputo en milisegundos.

El resto de tareas de la aplicación no pueden ser paralelizadas por lo que son procesadas mediante la Técnica de Replicación presentada en la sección 3.1.2, que determinará el número de copias a obtener de ellas y su estructura en forma de subgrafos.

Resultados experimentales

La productividad que se ha tomado como objetivo a alcanzar toma como base el mayor tiempo de cómputo de las tareas de la aplicación, siendo este valor el de la tarea T_{14}. De esta forma el valor de IP objetivo utilizado en las Técnicas de Paralelización y de Replicación así como en la fase posterior de mapping corresponde a dividir este tiempo de cómputo por 2, 4, 8, 16, 32, 64, 128 y 256, y que se han representado en las figuras y cuadros de resultados como x2, x4, x8, x16, x32, x64, x128 y x256 respectivamente.

A continuación se exponen por separado los pasos seguidos al aplicar la Técnica de Paralelización y la Técnica de Replicación sobre la aplicación BASIZ.

Técnica de Paralelización.

Para poder aplicar correctamente la Técnica de Paralelización, se ha caracterizado el tiempo de ejecución de las tareas sobre las que esta técnica ha sido utilizada. Para ello se ha ejecutado cada tarea utilizando diferentes tamaños de imagen a tratar. Las

dimensiones utilizadas, además de la imagen original, representan a franjas de altura 2, 4, 8 y 16 veces inferior a la inicial, representando cada caso el tiempo de cómputo para procesarlas por parte de una subtarea en el que la tarea original se divide al aplicar el paralelismo de datos. En el Cuadro 4.12 se muestran los tiempos de ejecución obtenidos en segundos.

Dimensión	T_4	T_5	T_6	T_7	T_8	T_9
1920×1280	0,6775	0,8965	1,0875	0,1130	1,3115	1,7560
1920×640	0,1685	0,2180	0,2750	0,0292	0,3300	0,4350
1920×320	0,0430	0,0555	0,0700	0,0061	0,0850	0,1090
1920×160	0,0115	0,0140	0,0195	0,0031	0,0220	0,0305
1920×80	0,0015	0,0075	0,0060	0,0013	0,0060	0,0112

Dimensión	T_{10}	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}
1920×1280	2,1900	0,1660	1,9665	2,5840	3,2452	0,2275
1920×640	0,5440	0,0431	0,4925	0,6415	1,0614	0,0505
1920×320	0,1394	0,0114	0,1250	0,1665	0,3473	0,0125
1920×160	0,0374	0,0023	0,0330	0,0435	0,1142	0,0061
1920×80	0,0122	0,00045	0,0105	0,0120	0,0374	0,0021

Cuadro 4.12: Tiempos de cómputo, en segundos, de las tareas identificadas como paralelizables para diferentes dimensiones de imagen.

A partir de estos valores, se ha determinado las funciones que caracterizan el tiempo de cómputo de las subtareas, en función de las divisiones de la imagen. Debido a que las dimensiones utilizadas en la caracterización corresponden a potencias de 2, estas funciones se han dimensionado adecuadamente aplicando la función log_2 sobre el número de divisiones.

$$
\begin{aligned}
tiempo\_cómputo(T_4) &= 2,5655e^{-1,3736\times log_2(n\_subtareas)} \\
tiempo\_cómputo(T_5) &= 3,0516e^{-1,2994\times log_2(n\_subtareas)} \\
tiempo\_cómputo(T_6) &= 3,7863e^{-1,3053\times log_2(n\_subtareas)} \\
tiempo\_cómputo(T_7) &= 0,2547e^{-1,0978\times log_2(n\_subtareas)} \\
tiempo\_cómputo(T_8) &= 4,9588e^{-1,3497\times log_2(n\_subtareas)} \\
tiempo\_cómputo(T_9) &= 6,3519e^{-1,328\times log_2(n\_subtareas)} \\
tiempo\_cómputo(T_{10}) &= 7,5532e^{-1,3105\times log_2(n\_subtareas)} \\
tiempo\_cómputo(T_{11}) &= 0,5035e^{-1,2839\times log_2(n\_subtareas} \\
tiempo\_cómputo(T_{12}) &= 6,98e^{-1,3204\times log_2(n\_subtareas)} \\
tiempo\_cómputo(T_{13}) &= 9,1464e^{-1,3229\times log_2(n\_subtareas)} \\
tiempo\_cómputo(T_{14}) &= 9,9224e^{-1,1177\times log_2(n\_subtareas)} \\
tiempo\_cómputo(T_{15}) &= 0,611e^{-1,1871\times log_2(n\_subtareas)}
\end{aligned}
$$

Las funciones anteriores, permiten determinar para un valor prefijado de tiempo de cómputo, el número de subtareas en que cada tarea debe ser paralelizada. Este cálculo se ha realizado mediante el uso de un calculador simbólico obteniendo los resultados mostrados en el Cuadro 4.13, buscando el valor de $n\_subtareas$ que cumple la igualdad de la expresión:

$$IP = tiempo\_cómputo(tarea) \tag{4.4}$$

En la sección 3.1.1 se comentó como al aumentar el número de subtareas provoca un incremento en la sobrecarga producida por las comunicaciones. Por este motivo se ha evaluado la expresión (3.1) para determinar cual es valor máximo de subtareas a partir del cual las comunicaciones pueden provocar una disminución en el rendimiento. Para resolver la expresión (3.1), previamente se debe modelar el coste asociado a las comunicaciones en función del volumen de datos a transmitir. De entre todos los modelos analíticos, se ha escogido el modelo LoGPC [MF01], ya que ofrece una buena estimación del tiempo de comunicación, sin añadir una elevada complejidad al estudio realizado.

En este modelo, el coste temporal de la comunicación realizada por una tarea T_i

Tarea	×2	×4	×8	×16	×32	×64	×128	×256
T_4	-	-	2,54	3,60	5,11	7,24	10,28	14,57
T_5	-	2,03	2,93	4,25	6,15	8,90	12,88	18,64
T_6	-	2,27	3,27	4,73	6,84	9,88	14,27	20,63
T_7	-	-	-	-	1,79	2,77	4,29	6,65
T_8	-	2,53	3,62	5,16	7,37	10,52	14,93	21,44
T_9	2,04	2,94	4,23	6,08	8,74	12,57	18,07	25,99
T_{10}	2,26	3,25	4,70	6,78	9,78	14,10	20,35	29,36
T_{11}	-	-	-	-	2,38	3,45	5,02	7,30
T_{12}	2,15	3,09	4,45	6,41	9,22	13,27	19,08	27,47
T_{13}	2,47	3,56	5,12	7,36	8,23	15,21	21,87	31,45
T_{14}	3,04	4,72	7,26	11,21	17,15	26,35	40,50	62,30
T_{15}	-	-	-	1,90	2,85	3,99	6,41	9,61

Cuadro 4.13: Número de subtareas a obtener de cada tarea paralelizable atendiendo al tiempo de cómputo según la expresión (4.4).

puede ser calculado como:

$$com(T_i) = o_{sl} + L + (B-1) * G \tag{4.5}$$

donde,

o_{sl}, es el tiempo de inicio del sistema para realizar una comunicación.

$L(Latency)$, es el tiempo promedio para que la cabecera del mensaje llegue hasta el receptor de la comunicación.

G, es el ancho de banda de la red.

B, es la longitud del mensaje medido en bytes.

Los parámetros del modelo, dependen de la red de comunicaciones utilizada, por lo que para definir cada valor se ha llevado a cabo un estudio de la red de comunicaciones presente en la arquitectura tomada como base para el entorno de simulación pMAP, utilizando para ello los siguientes *benchmarks*:

- *lmbench* [MS96]. Esta suite posee múltiples microbenchmarks, desarrollados en ANSI/C, cuya finalidad principal es la de medir la latencia y el ancho de banda, tanto del subsistema de comunicaciones como del propio sistema operativo. Mediante su uso se ha caracterizado el valor del parámetro o_{sl}.

- *netperf* [Pac]. Este benchmark es utilizado para la medida del rendimiento de diferentes redes de interconexión. Provee un conjunto de pruebas orientadas tanto a la medida del rendimiento en la comunicación unidireccional entre tareas, como a la medida de la latencia punto a punto. Con él se ha determinado el ancho de banda, parámetro G, y el tiempo involucrado en el transporte de la cabecera de los mensajes, parámetro L.

En el Cuadro 4.14 se muestran los valores obtenidos de la caracterización de la red de comunicaciones del cluster de producción.

Parámetro	Valor (segundos)
o_{sl}	7,375E-07
L	4,787E-05
G	8,817E-09

Cuadro 4.14: Parámetros del modelo LoGPC para la red de comunicaciones.

Mediante estos valores la expresión (3.1) se define el valor máximo de subtareas para cada tarea T_i de forma que el coste de las comunicaciones coincide con el tiempo de cómputo de una de las subtareas. Esta igualdad se obtiene a partir de la expresión:

$$tiempo\_cómputo(n\_subtareas) - (n\_subtareas - 1) \times \left(o_{sl} + L + \frac{(B-1) * G}{n\_subtareas} \right) = 0 \tag{4.6}$$

Mediante el uso de un calculador simbólico se ha obtenido el número de subtareas que satisfacen la igualdad para cada una de las tareas T_i identificadas como paralelizables. Estos valores se muestran en el Cuadro 4.15, y determinan el número de subtareas a partir del cual no es aconsejable seguir aplicando el paralelismo de datos.

Tarea	n_subtareas	Tarea	n_subtareas
T_4	11,49	T_{10}	22,09
T_5	14,31	T_{11}	5,99
T_6	15,78	T_{12}	20,77
T_7	5,36	T_{13}	23,64
T_8	16,50	T_{14}	42,83
T_9	19,71	T_{15}	7,56

Cuadro 4.15: Número máximo de subtareas para cada tarea paralelizable atendiendo a las comunicaciones.

Finalmente, para cada tarea el número de subtareas a obtener en cada caso, corresponde al valor mínimo de entre el obtenido atendiendo al tiempo de cómputo (Cuadro 4.13), y teniendo presente las comunicaciones (Cuadro 4.15), redondeado por exceso. En el Cuadro 4.16 se muestra para cada tarea y cada caso de estudio el número de subtareas que corresponde.

Tarea	×2	×4	×8	×16	×32	×64	×128	×256
T_4	-	-	3	4	6	8	11	12
T_5	-	3	3	5	7	9	13	15
T_6	-	3	4	5	7	10	15	16
T_7	-	-	-	-	2	3	5	6
T_8	-	3	4	6	8	11	15	17
T_9	3	3	5	7	9	13	19	20
T_{10}	3	4	5	7	10	15	21	23
T_{11}	-	-	-	-	3	4	6	6
T_{12}	3	4	5	7	10	14	20	21
T_{13}	3	4	6	8	9	16	22	24
T_{14}	4	5	8	12	18	27	41	43
T_{15}	-	-	-	2	3	4	7	8

Cuadro 4.16: Número de subtareas adecuado para cada tarea paralelizable y cada requisito de productividad.

Una vez que las tareas paralelizables de la aplicación han sido caracterizadas, la Técnica de Paralelización identifica los subgrafos que engloban a estas tareas así como el número de veces en que éste debe aparecer. Además para cada tarea en un subgrafo determina el número de subtareas que le corresponde. Para cada caso, los subgrafos identificados son diferentes y varía en función del criterio de productividad a alcanzar. En el Cuadro 4.17 se muestra el resultado para cada caso.

Debido a que no todas las divisiones de las imágenes en franjas son válidas, el número de subtareas se debe de ajustar para poder aplicar de forma adecuada el paralelismo, esto hace que en algunas tareas se sobre-explote el paralelismo al utilizar un mayor número de subtareas de las que sería necesario. Un ejemplo de esta situación es el caso de estudio x256 para la tarea T_{14}. Para esta tarea, el número más adecuado de subtareas es de 43, como se puede observar en el Cuadro 4.15, siendo finalmente 64 las subtareas a crear, 8 de ellas en cada subgrafo, en un total de 8 subgrafos para poder crear la porción de imagen correcta. Este ajuste en el número de subtareas a utilizar provoca que la granularidad de algunas de ellas sea inferior a la que se desea obtener, factor que se corrige en la fase de mapping utilizando la heurística MPASS, en el que

algunas de estas tareas son agrupadas con sus adyacentes si se encuentran definidas en la misma etapa síncrona ajustando la granularidad de las estas agrupaciones al valor de IP a alcanzar.

Caso	$Subgrafo$ - SG	n_copias[SG]	$(n\_subtareas[T_i])$
x2	$\{T_9\}$,$\{T_{10}\}$,$\{T_{12}\}$,$\{T_{13}\}$,$\{T_{14}\}$	4,4,4,4,4	(1),(1),(1),(1),(1)
x4	$\{T_5\}$,$\{T_6\}$,$\{T_8\}$,$\{T_9\}$, $\{T_{10}\}$,$\{T_{12}\}$,$\{T_{13}\}$,$\{T_{14}\}$	4,4,4,4, 4,4,4,5	(1),(1),(1),(1) (1),(1),(1),(1)
x8	$\{T_4\}$,$\{T_5\}$,$\{T_6\}$,$\{T_8\}$,$\{T_9\}$, $\{T_{10}\}$,$\{T_{12}\}$,$\{T_{13}\}$,$\{T_{14}\}$	4,4,4,4,5, 5,5,8,8	(1),(1),(1),(1),(1) (1),(1),(1),(1)
x16	$\{T_4\}$,$\{T_5\}$,$\{T_6\}$,$\{T_8\}$,$\{T_9\}$, $\{T_{10}\}$,$\{T_{12}, T_{13}, T_{14}, T_{15}\}$	4,5,5,8,8, 8,2	(1),(1),(1),(1),(1) (1),(4,4,8,1)
x32	$\{T_4, T_5, T_6, T_7\}$,$\{T_8, T_9, T_{10}, T_{11}\}$, $\{T_{12}, T_{13}, T_{14}, T_{15}\}$	2,4, 4	(4,4,4,1),(2,4,4,1) (4,4,5,1)
x64	$\{T_4, T_5, T_6, T_7\}$,$\{T_8, T_9, T_{10}, T_{11}\}$, $\{T_{12}, T_{13}, T_{14}, T_{15}\}$	4,4, 5	(2,4,4,1),(4,4,4,1) (4,4,8,1)
x128	$\{T_4, T_5, T_6, T_7\}$,$\{T_8, T_9, T_{10}, T_{11}\}$, $\{T_{12}, T_{13}, T_{14}, T_{15}\}$	4,8, 8	(4,4,4,1),(2,4,4,1) (4,4,8,1)
x256	$\{T_4, T_5, T_6, T_7\}$,$\{T_8, T_9, T_{10}, T_{11}\}$, $\{T_{12}, T_{13}, T_{14}, T_{15}\}$	8,8, 8	(2,2,2,1),(4,4,4,1) (4,4,8,1)

Cuadro 4.17: Técnica de Paralelización. Subgrafos identificados, número de veces que aparecen, y número de subtareas de cada tarea.

Técnica de Replicación.

Como se ha comentado previamente, hay un conjunto de tareas sobre las que no es posible aplicar la Técnica de Paralelización. Éstas si son susceptibles de ser utilizadas en la Técnica de Replicación en la forma en que se presento en la sección 3.1.2, siendo estas tareas $\{T_0, T_1, T_2, T_3, T_{16}, T_{17}, T_{18}, T_{19}, T_{20}, T_{21}\}$.

Para aplicar la Técnica de Replicación se tiene en cuenta tanto el tiempo de ejecución de las tareas, como el tiempo necesario para la transmisión de los datos entre las tareas adyacentes. Debido a las características de la red de comunicaciones, el tiempo involucrado en las comunicaciones es muy inferior al tiempo de ejecución de las tareas, por lo que sólo éste último ha sido el factor que ha determinado el grado de replicación de cada una de ellas.

En el Cuadro 4.18 se muestra el resultado obtenido tras aplicar la Técnica de Replicación, identificando los subgrafos, las veces en que éstos aparecen, así como las tareas que forman parte de ellos y su número de replicaciones. En el Apéndice A.0.9

aparecen los grafos que representan a las nuevas estructuras obtenidas para la aplicación BASIZ tras aplicar la Técnica de Paralelización y la Técnica de Replicación, a excepción de los casos x64, x128 y x256 que debido a su tamaño hace que no se pueda distinguir con claridad la estructura del grafo de tareas.

Caso	$Subgrafo$ - SG	n_copias[SG]	$(n\_rep(T_i))$
x2	-	-	-
x4	-	-	-
x8	-	-	-
x16	$\{T_{17}\}$	2	(1)
x32	$\{T_0\}$,$\{T_{16},T_{17}\}$,$\{T_{21}\}$	2,2,2	(1),(1,2),(2)
x64	$\{T_0,T_1,T_2,T_3\}$, $\{T_{16},T_{17},T_{18},T_{19}\}$,$\{T_{21}\}$	2, 2,2	(2,1,1,1), (2,4,1,1),(2)
x128	$\{T_0,T_1,T_2,T_3\}$, $\{T_{16},T_{17},T_{18},T_{19}\}$,$\{T_{21}\}$	4, 4,5	(2,1,1,1), (2,4,1,1),(1)
x256	$\{T_0,T_1,T_2,T_3\}$, $\{T_{16},T_{17},T_{18},T_{19}\}$,$\{T_{21}\}$	8, 8,10	(2,1,1,1), (2,4,1,1),(1)

Cuadro 4.18: Técnica de Replicación. Subgrafos obtenidos, veces en que aparecen y tareas que forman parte de ellos.

Hay que destacar como en los casos x2, x4 y x8, no es necesario replicar ninguna tarea debido a que el valor de IP a alcanzar es superior al tiempo de cómputo de las tareas evaluadas. Al disminuir el valor de IP, las tareas comienzan a agruparse en subgrafos y a ser replicadas. En este caso, el número de veces en que las tareas deben ser replicadas no debe ajustarse a la dimensión de las imágenes, como era el caso de la Técnica de Paralelización, debido a que cada copia de las tareas replicadas procesa una imagen completa. Aún así debido a que el subgrafo forma una unidad, y bajo el criterio de simplificar el patrón de comunicaciones, el número de veces en que aparecen las tareas dentro del propio subgrafo sí que se redondea por exceso dando lugar a que se obtenga un mayor número de replicaciones del calculado inicialmente.

Con la información obtenida tras aplicar la Técnica de Paralelización y la Técnica de Replicación se obtiene una nueva estructura para el grafo de tareas de la aplicación BASIZ. Sobre ésta se ha aplicado la heurística de mapping MPASS marcando como objetivo la productividad prefijada para cada caso de estudio. Los resultados obtenidos se muestran en la Figura 4.25, en la que se ha incluido para cada caso el número de nodos de procesamiento utilizados, el número de tareas resultantes y el porcentaje de desviación con respecto al valor deseado.

Se puede observar como la productividad crece a medida que las Técnicas de Parale-

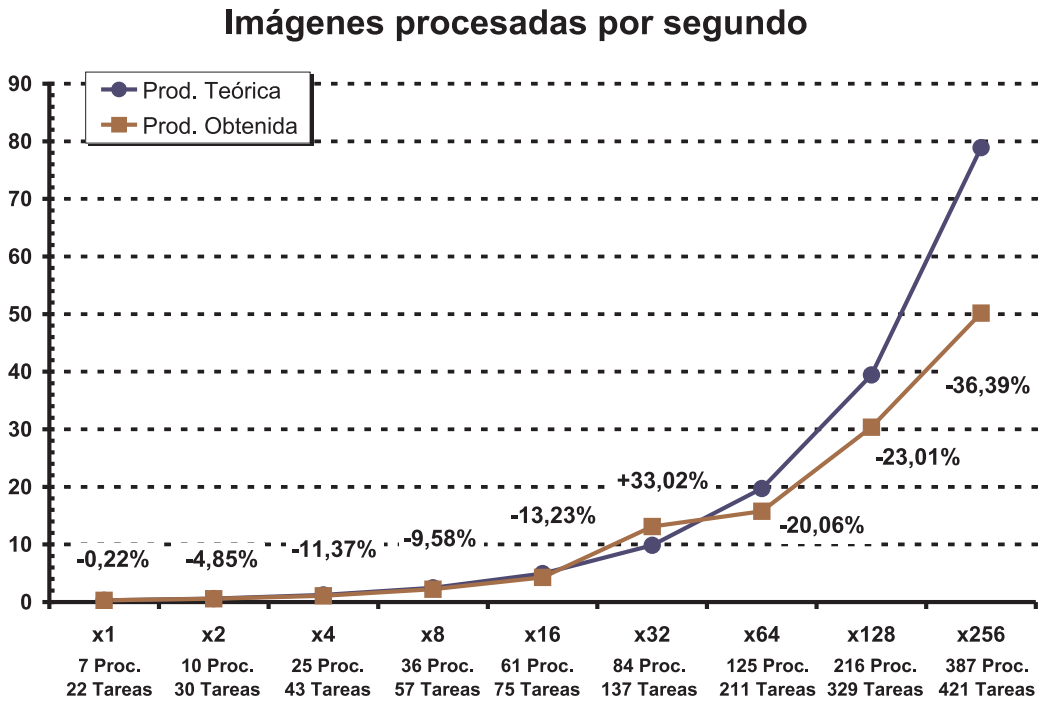


Figura 4.25: Productividad obtenida para la aplicación BASIZ tras redefinir su estructura y aplicar la heurística de mapping MPASS.

lización y de Replicación, de forma conjunta, disminuyen la granularidad de las tareas y aumentan el número de imágenes procesadas de forma concurrente. Hay que destacar el caso x32, en el que la productividad obtenida excede a la teórica, debido a dos factores. El primero corresponde a que se ha sobreexplotado la paralelización de las tareas dando lugar a una menor granularidad, pero sin exceder del valor máximo de subtareas que se ha definido anteriormente como límite en la paralelización (Cuadro 4.15). El segundo a que las tareas replicables entran en juego aportando un mayor número de imágenes a procesar concurrentemente dentro de la estructura pipeline de la aplicación.

Al aumentar la productividad objetivo a alcanzar, casos x64, x128 y x256, el valor que se obtiene empieza a distanciarse del valor teórico, aunque mantiene la misma tendencia creciente. El principal motivo de este distanciamiento es debido a la sobreexplotación del paralelismo en algunas tareas, dando lugar a una muy baja granularidad con respecto a las comunicaciones, lo cual provoca que la sobrecarga de las comunicaciones sea un factor significativo. Aún así la fase de mapping, en la mayoría de los casos, ha conseguido agruparlas en un único nodo de procesamiento, corrigiendo este exceso y adaptando la granularidad del cómputo en cada nodo de procesamiento.

Hay que destacar que la diferencia entre el número de tareas y de nodos de proce-

samiento utilizados es pequeña, ya que la mayoría de las tareas se han asignado a un único nodo de procesamiento, debido a que su tiempo de cómputo es cercano al valor de IP que se desea obtener, en el caso de que provengan de la Técnica de Paralelización, o porque al ser copias exactas de la tarea original exceden del valor de IP, como es el caso de las tareas obtenidas por la Técnica de Replicación.

Se puede concluir, que las Técnicas de Paralelización y de Replicación utilizadas de forma conjunta ofrecen una herramienta eficaz para la mejora del rendimiento de las aplicaciones pipeline, al redefinir la estructura del grafo de tareas, ofreciendo una forma alternativa y óptima de implementar la aplicación pipeline. El uso posterior de la heurística de mapping MPASS es capaz de ajustar la granularidad obtenida de la sobreexplotación del paralelismo por parte de la técnica de Paralelización, consiguiendo una asignación que cumple con los criterios de optimización y reduciendo el número de nodos de procesamiento necesarios.

Conclusiones y principales contribuciones

Las aplicaciones paralelas que procesan flujos continuos de datos de entrada, son de gran interés para la comunidad científica. Para estas aplicaciones se desea un rendimiento basado en dos criterios: la obtención de resultados en el mínimo tiempo posible, denominado como latencia o que estos resultados se obtengan con un ratio de datos procesados por unidad de tiempo prefijado, denominado como productividad.

La necesidad de procesar una secuencia continua de datos hace que estas aplicaciones añadan un factor de iteratividad en la ejecución de sus tareas, que supone un incremento de complejidad en su optimización respecto al que se realiza en la computación paralela clásica. Debido a la característica de iteratividad y a la disposición en etapas de las tareas, a estas aplicaciones se las denomina como aplicaciones pipeline.

En este sentido, el objetivo de este trabajo ha sido el de aportar una solución para la optimización de las aplicaciones pipeline. En el proceso de optimización presentado, se aborda tanto la definición del grafo de tareas que identifica la estructura de la aplicación pipeline para alcanzar el rendimiento deseado, como la forma en que las tareas de la aplicación se deben asignar a la arquitectura sobre la que se ejecutará.

A continuación se resume los puntos que se han tratado en este trabajo, añadiendo las aportaciones científicas a que las que han dado lugar.

- Se ha realizado un estudio del comportamiento de las aplicaciones pipeline, a partir del cual se ha obtenido el conocimiento necesario para determinar que factores actúan sobre los parámetros de rendimiento; latencia y productividad. El estudio preliminar de este tipo de aplicaciones aparece publicado en:

F. Guirado, A. Ripoll, C. Roig, E.Luque
Predicting the Best Mapping for Efficient Exploitation of Task and Data Parallelism.
Euro-Par 2003 Parallel Processing. LNCS Vol. 2790, pp. 218-223.

- Este conocimiento ha sido la base para desarrollar dos técnicas que permiten definir el grafo de tareas de la aplicación más adecuado para alcanzar la productividad deseada, que de otra forma y por las características de la aplicación original no sería posible alcanzar. En ambas técnicas se localizan las tareas que actúan como cuello de botella de la aplicación y en función de sus características y del tipo de flujo de datos a tratar aplica: (a) paralelismo de datos para disminuir su granularidad o (b) replicación de tareas para aumentar la capacidad de procesar, de forma concurrente, más datos del flujo de entrada. Las técnicas presentadas se han denominado como Técnica de Paralelización y Técnica de Replicación respectivamente.

  Las aportaciones que se han realizado en este apartado aparecen publicadas en:

  F. Guirado, A. Ripoll, C. Roig, E.Luque
  Optimizing Latency under Throughput Requirements for Streaming Applications on Cluster Execution.
  IEEE International Conference on Cluster Computing (Cluster 2005). Publicado en CD-ROM.

  F. Guirado, A. Ripoll, C. Roig, A. Hernàndez, E. Luque
  Exploiting Throughput for Pipeline Execution in Streaming Image Processing Applications.
  Euro-Par 2006 Parallel Processing. LNCS Vol. 4128, pp. 1095-1105.

- Con el objetivo de obtener un mapping, para las tareas de la aplicación pipeline, que optimice el rendimiento de éstas, se ha desarrollado dos heurísticas de mapping basadas en el concepto de etapa síncrona. Estas heurísticas se han denominado como MPASS (*Mapping of Pipeline Applications based on Synchronous Stages*) y MPART (*Mapping of Pipeline Applications based on Reduced Tree*).

  La utilización de las heurísticas requiere únicamente que las aplicaciones pipeline

puedan ser representadas mediante un grafo dirigido acíclico TPG, sin ningún tipo de restricción, y que la arquitectura sobre la que se realizará la asignación sea un cluster de computadores.

Las aportaciones en las que se tratan las heurísticas de mapping son:

F. Guirado, A. Ripoll, C. Roig, E. Luque
Exploitation of Parallelism for Applications with an Input Data Stream: Optimal Resource-Throughput Tradeoffs.
13th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'05), pp. 170-178.

F. Guirado, A. Ripoll, C. Roig, A. Hernàndez, E. Luque
A Pipeline-Based Approach for Mapping Message-Passing Applications with an Input Data Stream.
Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI-2004). LNCS Vol. 3241, pp. 224-233.

- En el apartado experimental se ha llevado a cabo la evaluación de las Técnicas de Paralelización y de Replicación, así como de las heurísticas de mapping MPASS y MPART.

  En el caso de las Técnicas de Paralelización y de Replicación, se ha comprobado sobre una aplicación pipeline sintética desarrollada en C+MPI, como ambas son capaces de proponer una nueva estructura para el grafo de tareas, que permite adoptar una implementación de la aplicación adecuada a los requisitos de rendimiento deseados. En el caso particular de la Técnica de Paralelización, se alcanza el rendimiento deseado añadiendo un número reducido de nuevas tareas al grafo de la aplicación. Por el contrario, las tareas no paralelizables de la aplicación, determinan la cota máxima en el rendimiento que es posible alcanzar mediante esta técnica.

  Para el caso de la Técnica de Replicación se genera un mayor número de tareas en el nuevo grafo creado. Aún así, esta técnica no posee limitación alguna por parte de la aplicación pipeline en la obtención de un rendimiento prefijado, si no que depende directamente del ratio del flujo de datos de entrada. Por el contrario su uso está limitado al procesamiento de flujos de datos de entrada no dependientes.

En la evaluación de las heurísticas de mapping MPASS y MPART, se ha ejecutado un conjunto de programas sintéticos desarrollados en C+MPI, y cuyos resultados se han comparado con dos heurísticas presentes en la literatura. Las heurísticas propuestas en este trabajo, han demostrado ser capaces de alcanzar los objetivos de rendimiento marcados utilizando el menor número de nodos de procesamiento, optimizando de esta forma los recursos.

La experimentación se ha completado con el estudio conjunto de las técnicas de Paralelización y de Replicación, y la heurística de mapping MPASS. Para ello se han utilizado tres aplicaciones pipeline: el compresor de vídeo MPEG2, la aplicación IVUS encargada del tratamiento de imágenes médicas y la aplicación BASIZ capaz de detectar regiones con mayor nivel de saturación de color y brillo. En todas ellas se ha comprobado que es posible alcanzar resultados próximos a los deseados mediante la aplicación de la nueva estructura para el grafo de tareas que se propone y utilizando el mapping obtenido. En el caso concreto de cada aplicación, hay que destacar:

- La implementación pipeline del compresor de vídeo MPEG2, sobre el que se ha utilizado la Técnica de Paralelización, se ha comparado con la implementación paralela clásica Master-Worker, comprobando como el procesamiento pipeline y la nueva estructura de tareas propuesta es capaz de alcanzar mayores cuotas de rendimiento, debido a la capacidad de procesar más de una imagen de forma simultanea dentro de la estructura pipeline.
- En la aplicación IVUS, sobre la que se ha aplicado la Técnica de Replicación, se puede aumentar la productividad tal y como se deseaba, al incrementar el número de datos procesados concurrentemente.
- En el caso de la aplicación BASIZ, se han utilizado tanto la Técnica de Paralelización como la Técnica de Replicación, siendo su resultado utilizado como entrada para la heurística de mapping MPASS. El entorno de estudio ha sido el simulador *p*MAP, en el que los resultados obtenidos demuestran que es posible aumentar la productividad de una forma que sería difícil de alcanzar utilizando las técnicas propuestas por separado.

El entorno de simulación *p*Map, utilizado en algunos puntos de la experimentación, aparece presentado en la publicación:

F. Guirado, A. Ripoll, C. Roig, E. Luque
Performance Prediction Using an Application-Oriented Mapping Tool.
12th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP'04), pp. 184-191.

Líneas abiertas

A medida que este trabajo se ha ido desarrollando, han ido surgiendo la siguientes nuevas líneas de interés que pueden contribuir a un mayor refinamiento en la optimización de las aplicaciones pipeline.

- Desarrollar una metodología que para un requisito de productividad demandado y partiendo del grafo de la aplicación pipeline, permita escoger de forma automática la Técnica de Paralelización o de Replicación, indicando si su uso debe realizarse de forma individual o conjunta. Esta metodología podría ser incluida en *frameworks* o compiladores de aplicaciones pipeline ya existentes.

- Desarrollar un modelo analítico que permita predecir cual es el rendimiento óptimo que se puede alcanzar para la aplicación pipeline, y la forma de obtenerlo a partir de la estructura para el grafo de tareas.

  Aportando este modelo al usuario, le permitiría no tener que evaluar de forma individual las diferentes opciones para la configuración del grafo de tareas de la aplicación pipeline, con el objetivo de determinar cual de ellas es la que mejor le conviene.

- Añadir a las heurísticas de mapping la capacidad de tener presente la heterogeneidad en las capacidades de cómputo, presentes en la actualidad en los nodos de procesamiento, en las plataformas multi-CPU y multi-Core.

- Incluir a las Técnicas de Paralelización y de Replicación así como a las heurísticas de mapping, la capacidad de limitar el número máximo de nodos de procesamiento disponible, y que en función de éste se determine la mejor estructura de dependencias de tareas para la aplicación, así como el mapping a realizar.