Esta tesis doctoral contiene un índice que enlaza a cada uno de los capítulos de la misma.
Existen asimismo botones de retorno al índice al principio y final de cada uno de los capítulos.

Ir directamente al índice

Para una correcta visualización del texto es necesaria la versión de Adobe Acrobat Reader 7.0 o posteriores

Aquesta tesi doctoral conté un índex que enllaça a cadascun dels capítols. Existeixen així mateix botons de retorn a l'index al principi i final de cadascun deis capítols .

Anar directament a l'index

Per a una correcta visualització del text és necessària la versió d' Adobe Acrobat Reader 7.0 o posteriors.

Universidad de Alicante

Departamento de Física, Ingeniería de Sistemas y Teoría de la Señal

Extensión de técnicas de planificación espacio-temporal a sistemas de visión por computador

TESIS DOCTORAL

Francisco Andrés Candelas Herías
Alicante, Octubre 2001

Universitat d'Alacant
Universidad de Alicante

Departament de Física, Enginyeria de Sistemes i Teoria de la Senyal
Departamento de Física, Ingeniería de Sistemas y Teoría de la Señal

FERNANDO TORRES MEDINA, Profesor Titular de Universidad de Ingeniería de Sistemas y Automática del Departamento de Física, Ingeniería de Sistemas y Teoría de la Señal de la Universidad de Alicante,

CERTIFICA: Que la presente Memoria titulada ***"Extensión de técnicas de planificación espacio-temporal a sistemas de visión por computador"*** ha sido realizada bajo su dirección por **D. Francisco Andrés Candelas Herías** en el Departamento de Física, Ingeniería de Sistemas y Teoría de la Señal de la Universidad de Alicante, y constituye su Tesis Doctoral para optar al Grado de Doctor.

Y para que conste, y en cumplimiento de la legislación vigente, firma el presente certificado en Alicante a diecinueve de octubre de dos mil uno.

DEPARTAMENTO DE FÍSICA
INGENIERÍA DE SISTEMAS Y
TEORÍA DE LA SEÑAL
UNIVERSIDAD DE ALICANTE

Fdo. Fernando Torres Medina

Tel. 965 90 3682- Fax 965 90 3682
Campus de Sant Vicent del Raspeig
Apt. 99. E-03080 Alacant
e-mail: dfits@ua.es
Web: http://www.ua.es

A mi madre.

Agradecimientos

Han sido muchas la personas que de una forma u otra me han ayudado en la realización de esta tesis. A continuación quiero listar a todas aquellas cuyo apoyo considero que me ha resultado imprescindible.

Debo destacar la ayuda prestada por los otros miembros del Grupo de Automática, Robótica y Visión Artificial al que pertenezco; Santiago, Javi, Pablo y Jorge, y muy especialmente la de Fernando, que también me ha alentado y orientado continuamente en la realización de esta tesis, aportándome además innumerables ideas.

No debo olvidar tampoco el apoyo y la amistad ofrecida por la innumerable lista que forman el resto compañeros del Dpto. De Física, Ingeniería de Sistemas y Teoría de la Señal de la Universidad de Alicante, donde comencé, y continúo con gran agrado, mis tareas docentes e investigadoras.

Y en el aspecto más personal, quiero agradecer a mis padres y a mi hermana sus ánimos y preocupación, no sólo durante del desarrollo de esta tesis, sino también a lo largo de mis estudios desde siempre.

Alicante, 19 de Octubre del 2001.

Índice

1. INTRODUCCIÓN 1

1.1. MOTIVACIÓN DE LA TESIS 2
1.2. ESTRUCTURA 3
1.3. APORTACIONES DE LA TESIS 4
1.4. TÉRMINOS UTILIZADOS 6

2. ASIGNACIÓN Y PLANIFICACIÓN DE TAREAS 7

2.1. INTRODUCCIÓN 8
2.1.1. Asignación espacial y planificación temporal de tareas 8
2.2. CONCEPTOS BÁSICOS 10
2.2.1. Las tareas 10
2.2.2. Interrumpibilidad y migración de tareas 12
2.2.3. Relaciones entre tareas 13
2.2.3.1. Relaciones de precedencia 13
2.2.3.2. Relaciones de prioridad 14
2.2.3.3. Relaciones de exclusión en el acceso a recursos 15
2.2.4. Inversión de prioridades 16
2.2.5. Arquitecturas hardware 16
2.2.6. Un ejemplo 17
2.3. PARTICULARIDADES EN APLICACIONES DE VISIÓN POR COMPUTADOR EN TIEMPO REAL 18
2.3.1. Tareas y procesadores en sistemas de visión por computador de tiempo real 19
2.3.2. Los tipos de planificación en sistemas de visión por computador 20
2.4. PLANIFICACIÓN DE TAREAS; ESTADO DEL ARTE 21
2.4.1. Esquemas básicos para planificación estática 22
2.4.2. Características de los esquemas estáticos existentes 23
2.5. RESUMEN 26

3. TÉCNICAS DE PLANIFICACIÓN TEMPORAL 27

3.1. INTRODUCCIÓN 28
3.2. DEFINICIONES 28
3.2.1. Grafo de tareas 28

3.2.2. Conectividad del grafo 29
3.2.3. Otra información del grafo 29
3.2.4. Asignación espacial de las tareas 30
3.2.5. Estado de los procesadores 30
3.3. PLANIFICACIÓN TEMPORAL SEGÚN CAMINO CRÍTICO 32
3.3.1. Inicializar. Definición de las variables utilizadas 33
3.3.2. TareasListas. Buscar las tareas que se pueden planificar 36
3.3.3. SeleccionaTareaLista. Selección de la tarea a planificar 37
3.3.4. ActualizaTiempo. Determinar el siguiente evento 38
3.3.5. Planificar. Planificación de la tarea seleccionada 39
3.3.6. RetrasarTareas. Retardar todas las tareas no planificadas 42
3.3.7. Bloquear. Interrumpir la tarea que hay en un procesador 44
3.3.8. Retardar. Posponer la ejecución de una tarea 44
3.3.9. CalcularMatrices. Actualizar los valores de tiempo y peso 44
3.3.10. Compactar. Eliminar cambios de tarea no necesarios 45
4. TÉCNICAS DE ASIGNACIÓN ESPACIAL 47
4.1. INTRODUCCIÓN 48
4.1.1. Gestión de los intervalos de tiempo 49
4.2. ASIGNACIÓN ESPACIAL 50
4.2.1. Inicializar. Definición de variables utilizadas 52
4.2.2. QuedanIntervalos. Comprobar si hay intervalos por analizar 54
4.2.3. ElegirProcesador. Elección de procesador para un intervalo 54
4.2.4. BuscaProcesador. Búsqueda del mejor procesador 57
4.2.5. RetardarSuc. Retardar tareas sucesoras 58
4.2.6. AsignarProcesadores. Asignación final a los procesadores 59
4.3. ASIGNACIÓN ESPACIAL SIN HUECOS 60
4.3.1. Inicializar. Definición de variables utilizadas 60
4.3.2. ElegirProcesador. Elección del procesador para un intervalo 61
4.3.3. BuscaProcesador. Búsqueda del mejor procesador 63
4.3.4. CrearHueco y ReducirHueco. Gestión de huecos de tiempo 66
4.4. ASIGNACIÓN ESPACIAL SIN HUECOS CON LÍMITE DE PROCESADORES 68

5. TÉCNICAS DE ASIGNACIÓN ESPACIAL Y PLANIFICACIÓN TEMPORAL SIMULTÁNEAS 71

5.1. INTRODUCCIÓN 72

5.2. TÉCNICAS DE ASIGNACIÓN ESPACIAL Y PLANIFICACIÓN TEMPORAL SIMULTÁNEAS QUE NO CONSIDERAN COSTES 73

5.2.1. Introducción 73

5.2.2. Asignación espacial y planificación temporal 73

5.2.2.1. Inicializar. Definición e inicialización de las variables utilizadas 74

5.2.2.2. ActualizaTiempo. Determinar el siguiente evento 75

5.2.2.3. Planificar. Planificación de la tarea seleccionada 76

5.2.2.4. RetrasarTareas. Retardar todas las tareas listas que no se han planificado 78

5.2.2.5. Bloquear. Interrumpir la tarea que se ejecuta en un procesador 78

5.2.2.6. SeleccionaCPU y SeleccionaTAPI. Seleccionar procesadores 79

5.2.2.7. AsignarResultado. Inicializar los objetos procesador 81

5.2.3. Asignación espacial y planificación temporal con límite de procesadores 82

5.2.3.1. Inicializar. Definición de nuevas variables 83

5.2.3.2. TareasListas2. Buscar todas las tareas que pueden planificarse 84

5.2.3.3. SeleccionaCPU y SeleccionaTAPI. Seleccionar procesadores 85

5.2.3.4. Planificar. Planificación temporal de una tarea según su tipo 87

5.2.3.5. PlanificarCPU, PlanificarTAPI y PlanificarCPUTAPI. Planificar una tarea .. 89

5.2.3.6. Bloquear. Interrumpir la tarea que se ejecuta en un procesador 90

5.2.3.7. BuscaTareaQueRetrasa. Determinar tarea a la que se debe esperar 91

5.2.3.8. AsignarResultado. Inicializar los objetos procesador 92

5.3. ASIGNACIÓN ESPACIAL Y PLANIFICACIÓN TEMPORAL CONSIDERANDO LOS COSTES DE INTERRUPCIÓN 92

5.3.1. Consideraciones generales 92

5.3.1.1. Costes de interrupción de las tareas 94

5.3.2. Asignación espacial y planificación temporal con costes de interrupción constantes 95

5.3.2.1. Inicializar. Definición de nuevas variables 95

5.3.2.2. Bloquear. Interrumpir la tarea que se ejecuta en un procesador 96

5.3.2.3. PlanificarCPU, PlanificarTAPI y PlanificarCPUTAPI. Planificar una tarea.100
5.3.3. Asignación espacial y planificación temporal con función de costes de interrupción *102*
5.3.3.1. Inicializar. Definición de nuevas variables 105
5.3.3.2. Bloquear. Interrumpir la tarea que se ejecuta en un procesador 105
5.3.3.3. ObtenerCostes. Aplicar la función de costes de interrupción 107
5.3.3.4. PlanificarCPU, PlanificarTAPI. Planificar una tarea 109
5.4. ASIGNACIÓN ESPACIAL Y PLANIFICACIÓN TEMPORAL CONSIDERANDO LOS COSTES DE COMUNICACIÓN 110
5.4.1. Consideraciones generales *110*
5.4.1.1. Tipos de tareas 111
5.4.1.2. Costes de comunicación 112
5.4.2. Asignación espacial y planificación temporal con costes de interrupción constantes *113*
5.4.2.1. Iniciar. Definición e iniciación de las de variables utilizadas 115
5.4.2.2. ActualizaTiempo. Determinar el siguiente evento 118
5.4.2.3. CalcularPesos. Calcula los pesos de cada tarea 120
5.4.2.4. SeleccionarTareaLista. Selección de la tarea a planificar 121
5.4.2.5. SeleccionarProcesadores. Selección de los procesadores 123
5.4.2.6. Planificar. Planificación de la tarea seleccionada 125
5.4.2.7. PlanificarCPU, Planificar TAPI. Planificar las tareas básicas 126
5.4.2.8. PlanificarCPUCPU, PlanificarTAPITAPI, PlanificarCPUTAPI. Planificar tareas de comunicación. 130
5.4.2.9. Bloquear. Interrumpir la tarea que se ejecuta en un procesador 131
5.4.2.10. CalcularMatrices. Actualizar los valores de tiempo y peso 132
5.4.2.11. AsignarResultado. Iniciar los objetos procesador 132
6. ENTORNO DE PRUEBAS 135
6.1. INTRODUCCIÓN 136
6.1.1. Motivación *136*
6.1.2. Orígenes *136*
6.1.3. Arquitectura del entorno de simulación y planificación *137*

6.2. EL ENTORNO VISUAL Y LA BASE DE DATOS 138
6.2.1. La interfaz con el usuario *138*
6.2.2. La base de datos *142*
6.2.3. El generador del grafo de tareas *145*
6.2.3.1. Inicialización 147
6.2.3.2. Generación de tareas 147
6.2.3.3. Finalización 150
6.2.4. La simulación dentro del entorno visual *152*
6.3. LA APLICACIÓN DEL PLANIFICADOR 153
6.3.1. Características generales *153*
6.3.2. Opciones de planificación e información obtenida *154*
6.3.3. Algoritmo de planificación temporal *156*
6.3.4. Algoritmos de asignación espacial *157*
6.3.5. Algoritmos de planificación temporal y asignación espacial *158*
6.4. OTRAS HERRAMIENTAS 158
6.4.1. El editor de grafos *159*
6.4.1.1. El visualizador remoto 160
6.4.2. Edición e inserción de tipos de OPIs *160*

7. APLICACIÓN A UN ALGORITMO DE CORRESPONDENCIA DE DOS IMÁGENES ESTEREOSCÓPICAS 163

7.1. INTRODUCCIÓN 164
7.2. ALGORITMO DE CORRESPONDENCIA DE DOS IMÁGENES ESTEREOSCÓPICAS 164
7.2.1. Los detectores de características *166*
7.2.1.1. Detector de esquinas 166
7.2.1.2. Detector de bordes 168
7.2.1.3. Detector de regiones 168
7.2.2. Tratamiento de las esquinas *169*
7.2.3. Tratamiento de los bordes *169*
7.2.4. Tratamiento de las regiones *170*
7.3. ESPECIFICACIÓN DE LA APLICACIÓN 172
7.3.1. Hardware disponible *172*

7.3.2. Características de las operaciones *172*
7.3.3. Grafo de tareas *175*
7.4. PRUEBAS DE LOS ALGORITMOS 177
7.4.1. Planificación temporal según camino crítico *177*
7.4.2. Asignación espacial *180*
7.4.3. Planificación espacio-temporal *182*
7.4.4. Planificación espacio-temporal con costes de interrupción *185*
7.4.5. Planificación espacio-temporal con costes de comunicación *187*

8. CONCLUSIONES 189

8.1. INTRODUCCIÓN 190
8.2. RESULTADOS OBTENIDOS 190
8.3. CONCLUSIONES DE LA TESIS 191

REFERENCIAS 193

Capítulo 1

Introducción

1.1. Motivación de la tesis

En el marco del proyecto CICYT coordinado "Construcción de un Entorno para la Investigación y Desarrollo en Visión Artificial" (Ref. TAP96-0629-C04), en el que ha participado el autor de esta tesis, se han desarrollado un conjunto de herramientas para trabajar con aplicaciones de visión por computador, destinadas fundamentalmente a la investigación y desarrollo de nuevos algoritmos en las etapas de diseño.

Más concretamente, dentro del proyecto anterior, el autor de esta tesis trabajó en el subproyecto "Driver para Tarjeta de Adquisición y Procesamiento de Imágenes y Desarrollo de una Interfaz de Usuario" (Ref. TAP96-0629-CO4-01), relativo al desarrollo de las aplicaciones de interfaz que permiten a los usuarios utilizar el entorno de forma amigable. De ese trabajó surgió un entorno de especificación gráfica o visual de alto nivel para algoritmos de visión por computador, en la línea de otros existentes (Khoros, WiT, Evision...), pero tratando de mejorar la interfaz de usuario, facilitando el aprendizaje y utilización, así como una gran flexibilidad a la hora de incluir nuevas operaciones.

Actualmente, las herramientas comerciales existentes para la especificación de algoritmos de visión por computador no van más allá de una especificación de alto nivel y de su simulación; no suministran información relativa a la planificación de tareas y tiempos de ejecución para del algoritmo diseñado sobre un determinado hardware. Este motivo ha llevado a incorporar al entorno gráfico desarrollado en el ámbito del proyecto CICYT comentado, la capacidad de manejar información sobre las tareas elementales que forman las operaciones, para permitir una posterior planificación de éstas con el objetivo de obtener el rendimiento deseado de la aplicación final.

Por otra parte, se han planteado muchas estrategias de planificación de tareas destinadas a diferentes tipos de sistemas, pudiéndose encontrar también algunas especificas para sistemas de visión por computador. Sin embargo, la gran mayoría de éstas están destinadas a optimizar operaciones concretas sobre determinadas arquitecturas hardware, y no tienen aspectos importantes como pueden ser la existencia de procesadores específicos para ciertas tareas, como por ejemplo las tarjetas de adquisición y procesamiento de imágenes en el caso de aplicaciones de visión por computador.

En esta tesis se presenta una serie de nuevos algoritmos de planificación estáticos destinados a aplicaciones de visión por computador. Los algoritmos consideran las características relevantes de esas aplicaciones, por lo que ofrecen resultados mucho más realistas y factibles de llevar a la práctica que otros algoritmos propuestos.

Además, no sólo se ha pretendido desarrollar nuevos algoritmos de planificación, sino que se han implementado e integrado con el entorno gráfico mencionado para que un usuario o desarrollador pueda utilizarlos directamente, y evaluar los resultados que ofrecen para los esquemas de alto nivel diseñados.

1.2. Estructura

Tras este capítulo de introducción, en el 2 se introducen los conceptos básicos relativos a las técnicas de asignación espacial y planificación temporal de tareas. Las características y relaciones existentes entre tareas se estudian con detalle. Además, se muestra la relación de todo ello con los sistemas de visión por computador, destacando las particularidades que éstos presentan. Centrando el estudio en la planificación estática, se describen los esquemas básicos utilizados, y se muestra un estado del arte de las distintas estrategias propuestas en los últimos años que más se pueden ajustar a las necesidades de los sistemas de visión por computador. Éstas se comparan con los algoritmos propuestos en la tesis.

Los capítulos 3, 4 y 5, parte central de esta tesis, describen detalladamente las nuevas técnicas de asignación y planificación estática diseñadas para sistemas de visión por computador. Se ha procurado utilizar un nivel alto de detalle que permita una implementación directa de los diferentes algoritmos, manteniendo también el formalismo necesario y unas descripciones claras y esquematizadas para su fácil compresión.

Así, el capítulo 3 se centra en el algoritmo de camino crítico, que realiza únicamente la planificación temporal, y está desarrollado a partir de otro existente que ha sido mejorado y adaptado. En el capítulo 4 se presenta un algoritmo dedicado a la asignación espacial, que puede ser utilizado conjuntamente con los del capítulo anterior para lograr mejores resultados. Posteriormente, el capítulo 5 detalla una serie de algoritmos basados en el de camino crítico que combinan la asignación espacial y la planificación temporal, y consideran además los

costes que implica la interrupción de tareas o la comunicación entre procesadores. Estos últimos son los más completos, sofisticados y novedosos, representando una de las principales aportaciones de la tesis.

El capítulo 6 está destinado a presentar el entorno de aplicaciones que permite a un desarrollador o investigador de aplicaciones de visión por computador hacer uso de los algoritmos presentados en capítulos anteriores mediante una interfaz de usuario muy amigable. Estas aplicaciones permiten, además de la especificación gráfica de aplicaciones de visión artificial a alto nivel, la aplicación de los diferentes algoritmos presentados en los capítulos 3, 4 y 5 para determinar la distribución optima de las tareas o el hardware necesario para que la aplicación final se ejecute dentro de los tiempos establecidos.

Para la pruebas y evaluación de los algoritmos diseñados, se ha considerado una aplicación de visión por computador concreta, un algoritmo de correspondencia de imágenes estereoscópicas tratado en el capítulo 7. Tras describir esta aplicación y mostrar su especificación en el entorno desarrollado, se estudian y comparan los resultados obtenidos al aplicar los diferentes algoritmos presentados en esta tesis. De todo ello se obtienen una serie de conclusiones, detalladas en el capítulo 8.

Finalmente, la tesis concluye con la lista detallada de las referencias, principalmente bibliográficas, utilizadas a lo largo de la tesis.

1.3. Aportaciones de la tesis

Una de las principales aportaciones de esta tesis son nuevos algoritmos de planificación espacio-temporal estática y su extensión a sistemas de visión por computador, ya que consideran sus características particulares, como son la relaciones de precedencia y exclusión entre tareas, y los diferentes tipos de procesadores y tareas. Los algoritmos son adecuados principalmente para la investigación y desarrollo de aplicaciones visión sobre los que se posee un conocimiento previo de las tareas, y permiten evaluar los tiempos de ejecución global sobre sistemas multiprocesador, así como el hardware necesario.

Otra aportación principal se centra en las versiones obtenidas de los algoritmos que consideran costes de interrupción o de comunicación, fijos o dados por una función,

características novedosas teniendo en cuenta el estado del arte sobre algoritmos de planificación para sistemas de visión por computador.

También cabe destacar como aportaciones, aunque a un nivel inferior, las descritas a continuación. En primer lugar, la creación de un entorno gráfico para la especificación de aplicaciones de visión por computador mediante esquemas de alto nivel y la simulación de estos. El entorno también dispone de herramientas que no sólo permiten definir y gestionar de forma flexible las operaciones básicas con las que construir los esquemas, sino también tareas elementales de que se componen y sus características, de cara a su implementación en un sistema paralelo.

En segundo lugar, la herramienta más destacable del entorno es una que permite al diseñador de aplicaciones de visión por computador evaluar con facilidad los resultados que ofrecen los diferentes algoritmos de planificación presentados sobre los esquemas de alto nivel especificados en el entorno gráfico. Esta herramienta permite comprobar si la ejecución se lleva a cabo en los tiempos previstos dado un hardware disponible, u obtener el hardware que hace falta para ello.

Los algoritmos de planificación y las herramientas desarrolladas en esta tesis han dado lugar a las siguientes publicaciones:

- *Publicaciones impactadas:*
  - *"Simulation and Scheduling of Real-Time Computer Vision Algorithms"*. F. Torres, F. A. Candelas, S. T. Puente, L. M. Jiménez, C. Fernández, R. J. Agulló. Lecture Notes In Computer Science. Springer, vol. 1542, pp. 98-114, 1999.
  - *"Graph Models Applied to Specification, Simulation, Allocation and Scheduling of Real-Time Computer Vision Applications"*. F. Torres, F. A. Candelas, S. T. Puente, F. G. Ortiz. International Journal of Imaging Systems & Technology, vol. 11 (5), pp. 287-291, 2000.
- *Otras publicaciones:*
  - *"Simulación y Planificación en Entorno Distribuido Para Algoritmos en Tiempo Real de Visión Artificial"*. Francisco A. Candelas. Memoria de Investigación, Universidad de Alicante, 1998.

Volver al índice/Tornar a l'índex

- *"Metodología y Herramientas de Planificación Estática para Aplicaciones de Visión Artificial"*. Francisco A. Candelas, S.T. Puente, F. Torres. Proceedings XX Jornadas de Automática (Salamanca), pp. 19-23, 1999.

1.4. Términos utilizados

Se ha querido expresar correctamente en castellano los diferentes términos técnicos pertenecientes al ámbito de la tesis que son habitualmente referidos con las expresiones o palabras que aparecen en la bibliografía inglesa. A continuación se listan las traducciones menos literales que se han realizado en esta tesis.

Context switching	Conmutación de tareas
Deadline	Tiempo límite
Feasible scheduling	Planificación factible
Idle-time	Tiempo de inactividad
Laxity	Laxitud
Load balancing	Equilibrado de carga
On-line	En línea
Overhead	Sobrecoste
Parallelizable	Paralelizable, que puede ejecutarse en paralelo
Preemptive algorithm	Algoritmo con posibilidad de interrupción de tareas
Priority ceiling	Tope de prioridad
Run-time	En tiempo de ejecución

Capítulo 2

Asignación y planificación de tareas

2.1. Introducción

Se puede describir un *sistema* como una interconexión de componentes que en conjunto dispone de una serie de entradas y salidas, y que realiza un proceso que proporciona una respuesta especifica en las salidas según los valores de entrada. Cualquier fenómeno de la naturaleza se puede modelar como un sistema, aunque son los artificiales como robots, sistemas de visión por computador, sistemas informáticos, etc. los que interesan en este punto.

El tiempo que transcurre desde que un sistema adquiere sus entradas hasta que proporciona sus salidas se denomina *tiempo de respuesta*. Cuando la salida de un sistema depende, no solo del proceso llevado a cabo, sino también del tiempo de respuesta, se puede hablar de *sistema de tiempo real*. Qué un sistema sea de tiempo real no implica necesariamente que el proceso llevado a cabo por él deba realizarse muy rápidamente. Sin embargo si involucra que para que la salida sea válida, ésta debe proporcionarse dentro de unos límites de tiempo determinados, que dependen del propio sistema.

Según el carácter de los procesos realizados, los sistemas de tiempo real pueden clasificarse como *críticos* o *no críticos*. En los primeros, proporcionar una salida fuera de un tiempo de respuesta preestablecido resulta catastrófico. Sin embargo, en los segundos, aunque es importante generar la salida dentro del tiempo de respuesta establecido, existe una tolerancia al incumplimiento ocasional de ese margen de tiempo.

2.1.1. Asignación espacial y planificación temporal de tareas

Las transformaciones elementales que se realizan dentro de un sistema se nombran de forma diferente en la bibliografía especializada dependiendo del contexto en el que aquel se enmarca. Son frecuentes términos como *trabajo* (job), *proceso* (process) o *tarea* (task). El último término será el empleado en esta tesis.

Uno de los grandes objetivos en sistemas paralelos es desarrollar técnicas efectivas para distribuir espacial y temporalmente las tareas a ejecutar sobre las unidades de proceso disponibles, de forma que se logren ciertos objetivos de rendimiento a la vez que se minimizan el tiempo de ejecución, los retardos de comunicación entre tareas y se maximiza la

utilización de los recursos del sistema, teniendo en cuenta el cumplimiento de las restricciones asociadas a las tareas.

Por ejemplo, cuando se trabaja con un sistema informático, estas técnicas forman parte del sistema operativo, y los procesos son los programas que pueden ejecutarse concurrentemente o de forma paralela.

Se puede hablar de dos tipos de técnicas de planificación. La *planificación local* o *planificación en sí* (o *scheduling*) trata el problema de repartir el tiempo de ejecución de un procesador del sistema entre las tareas que deben ejecutarse en ese procesador. La *planificación global* o *asignación de tareas* (o *task allocation*) tiene además como objetivo decidir en que procesador de un sistema multiprocesador se debe ejecutar cada tarea. Así, mientras que la planificación trata el orden temporal de las tareas, la asignación gestiona el ordenamiento espacial [1][2].

Por otra parte, atendiendo al instante de tiempo en que se ejecuta el planificador, las técnicas de planificación se dividen en dos grupos: *estáticas* y *dinámicas* [1][3]. También es posible la combinación de ambas.

Con una planificación estática las tareas se asignan a los procesadores antes de que el proceso o programa empiece a ejecutarse. En este caso se requiere conocer a priori o estimar los tiempos de ejecución y los recursos de procesamiento en el momento de crear el programa. Las planificaciones generadas por técnicas estáticas y las posibles prioridades asociadas a las tareas son fijas durante la ejecución. De esta manera, no son flexibles a cambios en la información de entrada al planificador, y para considerar nuevas tareas se requiere una nueva planificación. Sin embargo este tipo de planificación tiene dos grandes ventajas. En primer lugar, el mínimo coste que supone la planificación en la ejecución, al haberse realizado en tiempo de compilación, resultando una ejecución muy eficiente. En segundo lugar una ejecución muy predecible, algo muy importante en aplicaciones donde la seguridad es crítica.

En contraste con las técnicas estáticas, las dinámicas están diseñadas para trabajar con tareas de las que no se conocen todos sus parámetros a priori, y tienen tiempos de llegada o ejecución no predecibles. Son técnicas que se adaptan a cambios en el sistema con flexibilidad. Para ello realizan la distribución de las tareas durante la ejecución. Como desventaja estás técnicas añaden un sobrecoste adicional al de ejecución de las tareas debido a la planificación en línea.

Obtener una planificación óptima en general es un problema NP-completo, incluso con técnicas estáticas que trabajan con conocimiento previo. Dicho de otro modo, sólo son posibles soluciones óptimas en casos restringidos. Por eso también se estudia la planificación para soluciones sub-óptimas, basadas en heurísticas o aproximaciones que buscan respectivamente la mejor solución o una aceptable.

2.2. Conceptos básicos

Este apartado se dedica principalmente a describir formalmente el concepto de tarea, los tipos de tareas existentes, sus características y sus relaciones. Además se presentan también los distintos tipos de arquitecturas sobre las que se pueden planificar y ejecutar las tareas.

2.2.1. Las tareas

Las tareas representan las transformaciones elementales que se realizan dentro de un sistema, que puede ser desde un sistema físico a un programa software.

Una tarea evoluciona por tres estados básicos; *creación*, *ejecución* y *conclusión*, en este orden. Dentro del estado de ejecución la tarea se puede encontrar en distintas etapas. Si puede ser ejecutada, pero el sistema no está disponible para ello, se dice que la tarea está *preparada* o *lista*. Si el sistema está disponible y puede ejecutar la tarea, ésta pasa a la *ejecución*. Mientras que una tarea está en la etapa de ejecución el sistema puede decidir ejecutar otra, y si no es posible una ejecución paralela la primera pasará a una etapa de *bloqueo*. Antes de que una tarea bloqueada pueda continuar su ejecución en el sistema debe ser nuevamente preparada.

Las tareas se pueden clasificar según la regularidad con la que se crean. Se llaman tareas *periódicas* a las que se generan regularmente cada cierto intervalo de tiempo llamado periodo. En contraste, son tareas *esporádicas* las que se crean en instantes arbitrarios de tiempo, sin un periodo regular. Las primeras ocurren típicamente en sistemas de monitorización de tiempo real, mientras que las segundas son más características en control en tiempo real [3].

Una tarea esporádica τ_i se identifica de forma general con el siguiente vector:

$$\tau_i = (a_i, r_i, c_i, d_i)$$

Donde:

- a_i *es el instante absoluto de tiempo en que la tarea se crea o llega al sistema, medido en relación al instante inicial de tiempo* $t = 0$.
- r_i *representa el valor de tiempo más temprano en que puede comenzar la ejecución de la tarea medido desde su instante de llegada* a_i. *Muchos autores consideran como simplificación que este valor es nulo, y la tarea se puede expresar entonces como* $\tau_i = (a_i, c_i, d_i)$.
- c_i *es el tiempo de computación requerido por la tarea. Se suele trabajar con el tiempo de computación medio o con el de peor caso.*
- d_i *representa el instante de tiempo límite para completar la ejecución de la tarea, medido desde el instante en que llega la tarea.*

En las tareas periódicas también se indica su periodo de ejecución p_i. Además, para estas tareas d_i se define desde el inicio del periodo, con la condición de $d_i \leq p_i$:

$$\tau_i = (a_i, r_i, c_i, d_i, p_i)$$

La instancia o activación $k+1$ (con $k \in Z$) de una tarea periódica τ_i puede interpretarse como una tarea esporádica $\tau_{i,k+1}$ con los siguientes parámetros:

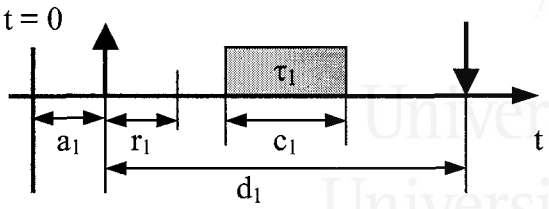
$$\tau_{i,k+1} = (a_i + k \cdot p_i, r_i, c_i, d_i, p_i)$$

La Figura 2-1 muestra ejemplos de una tarea esporádica y de una periódica, así como el significado de sus parámetros.

Algunos autores también expresan los valores de r_i y d_i de una tarea esporádica o una instancia de una tarea periódica de forma absoluta, en relación al instante de tiempo $t = 0$.

Bajo ciertas condiciones, también resulta posible convertir una tarea esporádica a una periódica equivalente, lo que puede ser interesante al permitir aplicar técnicas de planificación de tareas periódicas a tareas esporádicas [3].

a) Ejemplo de tarea esporádica.



b) Ejemplo de tarea periódica.

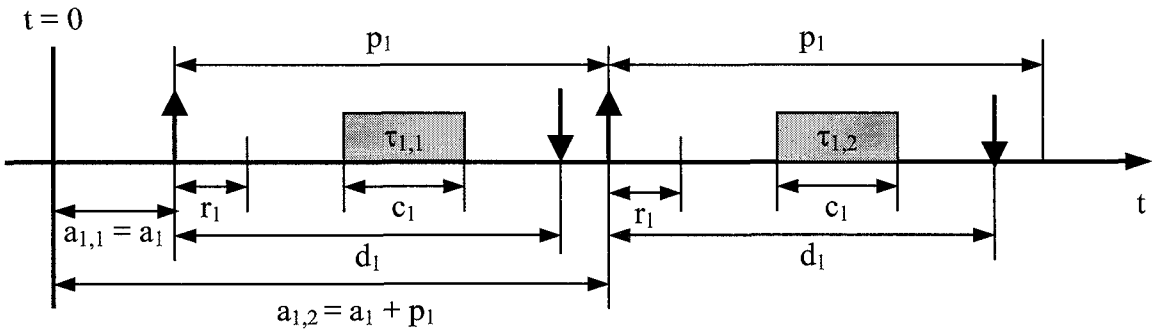


Figura 2-1. Ejemplos de tareas.

Otro parámetro muy utilizado es el concepto de laxitud, que expresa como de urgente es la ejecución de una tarea y cuanta libertad hay para posponer su ejecución en un instante de tiempo dado. Siendo $c_i(t)$ el tiempo de computo que queda por ejecutar de la tarea τ_i en el instante t, se puede obtener como:

$$l_i(t) = d_i - [c_i(t) + t]$$

Considerando una tarea τ_i en el instante t, cuando hay suficiente tiempo para ejecutarla y completarla antes de su tiempo límite se tiene $l(t)_i > 0$. Si va a ser ejecutada inmediatamente para completarse sin interrupción justo antes de su tiempo límite se tiene $l(t)_i = 0$. Finalmente se tiene $l(t)_i < 0$ cuando se ha incumplido el tiempo límite de la tarea.

2.2.2. Interrumpibilidad y migración de tareas

Las técnicas de planificación también se pueden clasificar en función de si consideran la posibilidad de interrumpir o no la ejecución de unas tareas para ejecutar otras. De este modo se puede hablar de una *planificación con* o *sin opción de interrupción* de tareas [1][4].

Cuando una tarea puede ser interrumpida y después reanudada sin perjudicar sus cálculos se dice que es *interrumpible*. La existencia de tareas interrumpibles depende de la naturaleza del sistema tratado. Con este tipo de tareas se puede plantear una planificación con

interrupción que proporcione mejores resultados [3]. Pero también hay que considerar que la interrupción de tareas puede causar sobrecostes adicionales en la ejecución, debido a la necesidad de cambio del contexto del procesador, lo que resulta más crítico en una planificación dinámica.

Al considerar tareas interrumpibles tiene sentido asociar prioridades a éstas, de forma que se establezca un orden. Una tarea tendrá más prioridad que otra cuando el planificador puede interrumpir la ejecución de la segunda en un determinado procesador para pasar a ejecutar la primera. Las prioridades que se asocian a las tareas dependen de las particularidades de la aplicación, y de la importancia de las tareas dentro de ésta.

Además se pueden establecer *prioridades fijas* o *dinámicas*. En el primer caso, las prioridades asociadas a las tareas permanecen constantes durante la planificación, y se habla de una planificación de prioridad fija. Ésta es típica en la planificación estática. En contraste, una planificación dinámica suele trabajar con prioridades dinámicas, esto es, las prioridades de las tareas pueden cambiar según el instante de tiempo en que se evalúan.

2.2.3. Relaciones entre tareas

Debido a varios motivos se pueden establecer diferentes tipos de relaciones entre las tareas de un sistema, entre las que destacan las siguientes [1][3].

2.2.3.1. Relaciones de precedencia

Son debidas al propio orden de procesamiento que impone una aplicación sobre las tareas y a las necesidades de intercambio de datos entre éstas.

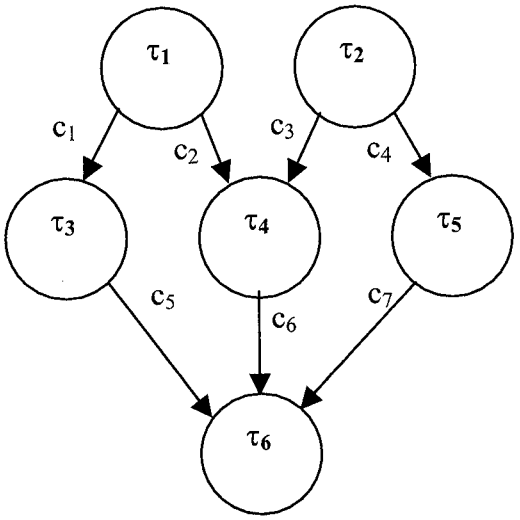
Formalmente, un conjunto de tareas T puede ser parcialmente ordenado por una relación de precedencia R_{prec} irreflexiva, antisimétrica y transitiva de la siguiente manera:

$$T = \{\tau_1, \tau_2, \cdots \tau_n\}\ R_{prec}: T \leftrightarrow T$$

$$\tau_1\ R_{prec}\ \tau_2 \Leftrightarrow \textit{La ejecución de } \tau_1 \textit{ debe ser completada antes de que comience la ejecución de } \tau_2$$

Dicho de otro modo, τ_2 no puede empezar a ejecutarse hasta que haya acabado τ_1, debido a que τ_2 requiere información que genera τ_1.

Las relaciones de precedencia implican además operaciones de comunicación entre las tareas que se ejecutan en diferentes procesadores, las cuales conllevan unos costes temporales (costes de comunicación). Estos deberían ser tenidos en cuenta al planificar las tareas, aunque muchos algoritmos los consideran nulos.



$$T = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6\}$$

$$E = \{(\tau_1, \tau_3, c_1), (\tau_1, \tau_4, c_2), (\tau_2, \tau_4, c_3), (\tau_2, \tau_5, c_4), (\tau_3, \tau_6, c_5), (\tau_4, \tau_6, c_6), (\tau_5, \tau_6, c_7)\}$$

$$G = (T, E)$$

Figura 2-2. Ejemplo de DAG.

El *Directed Acyclic Graph* (Grafo Dirigido Acíclico) o DAG, es el modelo más popular para simbolizar un programa paralelo. En este grafo los nodos representan las tareas y sus parámetros y los arcos dirigidos representan las restricciones de precedencia y los costes de comunicación entre tareas (Figura 2-2). Un grafo de este tipo puede definirse como:

$$G = (T, E)$$

Conjunto de tareas: $T = \{\tau_1, \tau_2, \cdots \tau_n\}$

Conjunto de arcos: $E = \{(\tau_i, \tau_j, c_{i,j}) / \tau_i R_{prec} \tau_j$ *con coste de comunicación* $c_{i,j}\}$

2.2.3.2. Relaciones de prioridad

Cuando se utiliza un esquema de planificación con opción de interrupción (apartado 2.2.2), se pueden considerar dos tipos de tareas en el sistema; las *interrumpibles* y las *no interrumpibles*. El planificador puede decidir interrumpir una tarea del primer tipo durante su ejecución y llevarla a situación de bloqueo para dar paso a la ejecución de otra tarea preparada. En cambio, una tarea del segundo tipo no puede ser interrumpida ni bloqueada durante su ejecución.

Dado un conjunto T de tareas a planificar, éstas pueden ordenarse conforme a una prioridad según una relación $>_{pr}$ irreflexiva, antisimétrica y transitiva de la siguiente manera:

$$T = \{\tau_1, \tau_2, \cdots \tau_n\} >_{pr}: T \leftrightarrow T$$

$$\tau_1 >_{pr} \tau_2 \Leftrightarrow \tau_1 \textit{ Tiene mayor prioridad que } \tau_2$$

Considerando dos tareas τ_1 y τ_2 que deben ejecutarse en el mismo procesador, τ_1 tiene mayor prioridad que τ_2 cuando τ_2, estando en ejecución, puede ser interrumpida para ejecutar τ_1. Para más de dos tareas a ejecutar en un mismo procesador, la tarea de mayor prioridad será la que se ejecute en un instante dado. Un caso de empate de prioridades se resuelve arbitrariamente.

También se puede considerar que las tareas no interrumpibles son las que tienen el valor de prioridad más alto de entre las de *T*.

Al aplicar la relación $>_{pr}$ en una técnica de planificación se habla entonces de una planificación basada en prioridades.

Cuando se trabaja con prioridades fijas, la relación $>_{pr}$ es constante a lo largo del tiempo. Sin embargo, en una planificación con prioridad dinámica, $>_{pr}$ es una función del tiempo.

2.2.3.3. Relaciones de exclusión en el acceso a recursos

Están derivadas del uso de recursos como procesadores, memoria, archivos, etc. que deben ser utilizados de forma exclusiva por una sola tarea en modo no interrumpible.

Una tarea τ_i excluye a otra τ_j si en el instante en que τ_j tiene que ser ejecutada, un recurso requerido por τ_j está siendo utilizado por τ_i.

Cuando el recurso requerido por más de una tarea se trata de un mismo procesador, la relación entre las tareas es la comentada en el punto anterior, y el planificador deberá asignar el procesador teniendo en cuenta si las tareas son interrumpibles o no, así como sus prioridades en el primer caso.

Para otro tipo de recursos, las exclusiones son tratadas normalmente mediante restricciones de exclusión mutua entre tareas que garanticen un acceso secuencial a los recursos.

2.2.4. Inversión de prioridades

En una planificación con opción de interrupción y gestión de prioridades y donde además existen relaciones de exclusión en el acceso a recursos entre tareas, la solución del acceso secuencial puede causar problemas. Así, puede ocurrir que una tarea de prioridad alta quede bloqueada y en espera de que acabe otra de menor prioridad que ha conseguido antes el acceso a un recurso. Esto se conoce como inversión de prioridad, y provoca que el comportamiento del planificador no sea el esperado. Esta situación puede agravarse si la tarea de menor prioridad es interrumpida por otras, ya que el tiempo que dura el bloqueo de la tarea de mayor prioridad se hace entonces impredecible.

La solución habitual es recurrir a algoritmos como el *Priority Ceilling Protocol,* que previene que una tarea de mayor prioridad quede bloqueada por más de una tarea de menor prioridad en cualquier instante de tiempo [5][6][7].

2.2.5. Arquitecturas hardware

Se pueden distinguir dos tipos de arquitecturas hardware para un sistema de tiempo real conforme al número de procesadores dedicados a ejecutar tareas: los *monoprocesador* y los *multiprocesador.*

En una arquitectura monoprocesador, compuesta por una sola unidad de proceso, no se requiere realizar una asignación de tareas, y únicamente es necesaria una planificación temporal. Si la planificación se realiza dinámicamente, el tiempo de proceso debe repartirse entre el planificador y las tareas.

Más complejas son las arquitecturas con más de un procesador o multiprocesador. En ellas, además de la planificación temporal se requiere una asignación espacial previa de las tareas que determine en que procesador debe ejecutarse cada tarea. Con estas técnicas el planificador se puede ejecutar en un procesador determinado (planificador centralizado) o de forma distribuida entre los procesadores del sistema.

Dentro de las arquitecturas multiprocesador se puede distinguir entre las que emplean un modelo de memoria compartida y las que tienen una memoria local para cada procesador, también denominadas distribuidas. En el primer caso, aunque existe más flexibilidad en la planificación, las relaciones entre tareas toman mayor importancia.

Además, se puede diferenciar entre las arquitecturas multiprocesador cuyos elementos de proceso tienen iguales características y capacidades, siendo estos de propósito general (sistemas homogéneos), y las que tienen distintos tipos de procesadores especializados en ejecutar determinados tipos de tareas u operaciones concretas. Por ejemplo, este es el caso de un sistema de adquisición y procesamiento de imágenes de tiempo real, en donde, además de procesadores genéricos que ejecutan tareas software, hay procesadores dedicados a la adquisición y preprocesamiento de imágenes que realizan funciones muy especificas.

En el caso de procesadores especializados, estos tienen normalmente diferentes capacidades de proceso, y durante su funcionamiento pueden existir diferencias en el nivel de carga que ejecutan dependiendo del tipo de tareas que lleguen al sistema. Por otra parte, las arquitecturas homogéneas pueden emplear un modelo de memoria local o de memoria compartida, pero los no homogéneos suelen utilizar memoria local.

Cuando se diferencian tipos de procesadores, también se puede hacer una distinción de las tareas que puede ejecutar cada uno de ellos según clases o tipos.

2.2.6. Un ejemplo

Considérese que las tareas del DAG de la Figura 2-2 tienen los siguientes parámetros (a_i, c_i, d_i):

$\tau_1 = (0, 5, 10)$	$\tau_4 = (5, 5, 15)$	$\tau_5 = (5, 10, 20)$
$\tau_2 = (0, 10, 15)$	$\tau_3 = (0, 10, 15)$	$\tau_6 = (10, 5, 10)$

Se puede establecer una asignación y planificación sobre dos procesadores, considerando las relaciones de precedencia y sin opción de interrupción, como la mostrada en la Figura 2-3. En ella las tareas τ_1, τ_3, τ_4 y τ_6 se han asignado al procesador 1, y las tareas τ_2 y τ_5 al procesador 2. La planificación es válida ya que todas las tareas se ejecutan dentro de su tiempo límite.

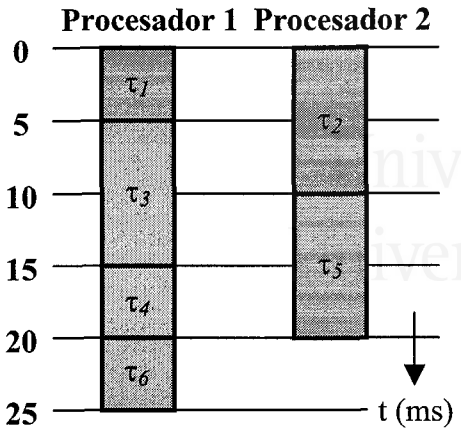


Figura 2-3. Asignación y planificación para el DAG de la Figura 2-2.

2.3. Particularidades en aplicaciones de visión por computador en tiempo real

Un equipo de visión por computador es un sistema cuyas señales de entrada son de naturaleza lumínica, y que adquiridas con ayuda de una cámara y un hardware digitalizador proporcionan una matriz de valores numéricos que representan la imagen de forma discreta. Para ello se emplea un proceso de muestreo, por lo que los sistemas de visión por computador son sistemas discretos. Además, el sistema debe realizar el procesamiento de imágenes discretas mediante las operaciones necesarias para una determinada aplicación, como puede ser la inspección visual, el guiado de vehículos móviles, el reconocimiento de escenas, el control visual de un robot, etc. Según la aplicación, el sistema generará como respuesta unas salidas en forma de señales de alarma, valores de velocidad o dirección, características de un patrón reconocido, coordenadas de posición, etc.

Cuando se dispone de una aplicación de visión por computador que tiene que ofrecer una respuesta en un determinado margen de tiempo se puede considerar que es un sistema de tiempo real. Pero un sistema de visión por computador no es de tiempo real solo por requerir un procesamiento de imágenes que tiene en cuenta el tiempo de respuesta. Hay que considerar otros problemas como el rendimiento frente a la resolución y tamaño de las imágenes, así como frente al ancho de banda de los elementos de entrada, de salida y de almacenamiento; el conjunto de tareas a realizar de forma sincronizada; o la necesidad de algoritmos de

procesamiento de forma paralela. Todo ello lleva a la necesidad de un sistema multiprocesador y un planificador de tareas.

2.3.1. Tareas y procesadores en sistemas de visión por computador de tiempo real

Las posibles relaciones entre tareas vistas en el punto 2.2.3 de este capítulo son un punto determinante en visión por computador. Así, la relación de precedencia es innata a cualquier sistema de visión por computador. Por ejemplo, una imagen no puede analizarse sin realizar una tarea previa de captación. Tampoco se puede realizar una tarea de clasificación sin una tarea anterior de segmentación.

Teniendo en cuenta las relaciones de precedencia, el tiempo más temprano de ejecución de una tarea está en función de lo que tarden en ejecutarse sus predecesoras, aunque llegue antes al sistema. Así, de cara a la planificación se puede considerar que los tiempos de llegada de las tareas están determinados por la precedencia en vez de especificarlos explícitamente.

Por otra parte, las relaciones de prioridad y exclusión constituyen una diferencia entre los sistemas de visión por computador y otros sistemas de tiempo real. En los sistemas de visión por computador hay tareas que pueden ser interrumpidas y otras que no. Generalmente son las tareas de bajo nivel de procesamiento que son ejecutadas en hardware o tarjetas de adquisición y procesamiento las que no son interrumpibles. Por ejemplo, la captación y digitalización de una imagen no se puede interrumpir. Las tareas desarrolladas a un nivel más alto, normalmente en procesadores genéricos, son las que permiten una ejecución con posibilidad de interrupción [8].

Lo anterior permite diferenciar dos tipos de unidades de ejecución para las tareas de sistemas de visión por computador: las tarjetas de adquisición y procesamiento o *TAPIs* y los procesadores genéricos o *CPUs*. Según esos dos tipos de unidades se puede establecer una clasificación de los tipos de tareas de un sistema de visión por computador [8][9]:

- Tareas tipo **tapi**. Son las que se ejecutan en los niveles más bajos de procesamiento, y que son efectuadas generalmente en una TAPI de forma no interrumpible. Entre este tipo de tareas existen relaciones de exclusión. Para iniciar

una tarea de este tipo, antes es necesario realizar una tarea del tipo cpu/tapi, que se expone a continuación.

- Tareas tipo **cpu**. Son ejecutadas en una CPU del sistema, y están formadas normalmente por operaciones de alto nivel en el procesamiento, aunque también pueden incluirse en este tipo tareas de niveles inferiores que requieren alguna mejora de velocidad, precisión, etc. que puede ser aportada una CPU. En cualquier caso, se trata de tareas interrumpibles, y sin relaciones de exclusión.
- Tareas tipo **cpu/tapi**. Son tareas que emplean a la vez los dos tipos de recursos de procesamiento: TAPIs y CPUs. No pueden ser interrumpidas, y gestionan el intercambio de información entre una CPU y una TAPI, en ambos sentidos. Por ejemplo, la lectura de datos de una imagen desde la memoria de una TAPI hacia una CPU representa una tarea de este tipo.

2.3.2. Los tipos de planificación en sistemas de visión por computador

Según lo visto en el punto 2.1.1, la planificación estática requiere un conocimiento previo de los parámetros de las tareas, y por ello presenta el inconveniente de que genera una distribución de las tareas demasiado rígida, en la que no es posible introducir modificaciones durante el procesamiento. Aun así, en muchos sistemas de visión por computador, como pueden ser los de inspección industrial, se dispone de ese conocimiento, y en ellos una planificación estática es efectiva y menos costosa [8].

En cambio, la planificación dinámica tiene en cuenta las posibles alteraciones sufridas en la ejecución de un procesamiento, y puede introducir modificaciones en la planificación. Así, se puede considerar que el grafo con las relaciones de precedencia entre tareas contenga varios caminos alternativos para llegar al nodo final que genera la salida, y la elección del camino a seguir en el procesamiento venga impuesta en función del resultado de un procesamiento anterior. Un ejemplo de esto puede ser la necesidad de efectuar la reconstrucción de una escena por segunda vez por que en un primer tratamiento no se obtuvieron resultados satisfactorios. Esta necesidad no viene impuesta a priori en el planificador, sino que se determina durante el procesamiento de la imagen.

Pero la planificación dinámica puede presentar dos grandes inconvenientes en visión por computador. El primero se debe a la carga computacional adicional en los procesadores que realizan la planificación cuando estos son también CPUs del sistema dedicadas al procesamiento de tareas de visión de alto nivel, lo que puede generar una demora importante en el tiempo de respuesta del sistema en conjunto. El segundo problema surge cuando ese retraso en el tiempo de respuesta del sistema llega a valores no permisibles.

Según como se presenten estas particularidades en un sistema de visión por computador, y esto depende de la aplicación concreta a la que se dedica el sistema, se deberá elegir entre un planificador estático o uno dinámico.

2.4. Planificación de tareas; estado del arte

Un algoritmo estático planifica las tareas de un sistema previamente a su ejecución, y para ello es necesario un conocimiento previo de los parámetros de todas las tareas. Básicamente sus tiempos de llegada, computo y tiempos límite. El resultado de un planificador de este tipo es entonces la asignación espacial y ordenación temporal de las tareas que deben considerarse al ejecutar el proceso que lleva a cabo el sistema.

Durante la ejecución no interviene ya el planificador, y de este motivo se deriva una gran ventaja de la planificación estática: el planificador no requiere tiempo de computo en el sistema y no supone sobrecostes adicionales.

Otra característica de este tipo de planificación es la predicibilidad existente sobre la ejecución, ya que las tareas y su planificación se conocen previamente. También se puede determinar de forma previa a la ejecución si un conjunto de tareas es planificable por el algoritmo escogido y dadas las características del sistema, y de no ser así cambiar de estrategia o adaptar el sistema para lanzar la ejecución con la seguridad de que las tareas se ejecutarán en sus plazos de tiempo [3].

De hecho, en muchos casos una planificación estática es la única forma de garantizar la predicibilidad que conlleve una planificación resultante donde se cumplen las exigencias temporales de las tareas. Incluso en el caso de no conocerse con exactitud los características de las tareas, se pueden estimar los valores en un peor caso para garantizar que una planificación estática ofrece un resultado válido [3].

Pero también debe considerarse que en muchos sistemas resulta difícil conocer con exactitud o estimar los parámetros de las tareas. Además, un conocimiento de las tareas y una planificación previos a la ejecución implican la desventaja de que el sistema no resulta flexible o capaz de adaptarse durante la ejecución a cambios del entorno.

2.4.1. Esquemas básicos para planificación estática

Partiendo de que se dispone de un conocimiento previo de las tareas, es habitual usar esquemas de planificación de prioridad fija, destacando los basados en el clásico algoritmo **RM** (*Rate Monotonic*).

La base de una planificación con el RM consiste en asignar una prioridad mayor a la tarea periódica con frecuencia de llegada mayor (o con menor periodo entre llegadas). Considerando que el sistema trabaja con tareas periódicas independientes cuyos tiempos límites sean iguales a sus periodos, este algoritmo de planificación resulta óptimo en un sistema monoprocesador en el sentido de que es capaz de planificar cualquier conjunto de tareas que sea planificable por cualquier otro algoritmo de prioridades fijas [1][5][6].

El RM fue ideado inicialmente para tareas periódicas interrumpibles en arquitecturas monoprocesador [10], pero ha sido ampliamente estudiado y se ha extendido para tratar también tareas esporádicas (por ejemplo, el algoritmo **DS** o *Deferred Server*), y ser aplicado a arquitecturas multiprocesador. Además, se han definido extensiones para trabajar con recursos compartidos mediante técnicas como el *Priority Ceiling Protocol* [1][5][6][7].

También se han utilizado esquemas de prioridad dinámica, que consideran que las prioridades asignadas a las tareas no permanecen constantes al valor inicialmente asignado y son dependientes del instante en que se evalúan durante la planificación. Dentro de estos esquemas destacan los de tipo **ED** (*Earliest Deadline*), también conocidos como **DM** (*Deadline Monotonic*), basados en planificar las tareas que llegan al sistema en función de su tiempo límite de ejecución. Un esquema clásico de este tipo es el **EDF** (*Earliest Deadline First*) [10]. Éste se caracteriza por dar mayor prioridad de ejecución a la tarea que tenga el tiempo límite más cercano al instante actual de entre las tareas que ya hayan llegado al sistema. En caso de existir varias tareas en esa situación, se hace una elección no determinista entre ellas [1][6].

El EDF resulta óptimo bajo diferentes situaciones y ha sido extendido a sistemas de diversas características, aunque los resultados originales obtenidos para el EDF no aceptaban relaciones de precedencia o recursos compartidos [5]. Así, por ejemplo, el EDF es óptimo en arquitecturas monoprocesador con tareas independientes y que pueden ser interrumpidas [6].

Para arquitecturas multiprocesador, la planificación es un problema NP-duro en general, y debe abordarse con heurísticas como las estudiadas en [11] y [12], o bajo ciertas restricciones [5].

2.4.2. Características de los esquemas estáticos existentes

A continuación se comentan las técnicas más recientes de planificación estática. En primer lugar, están los algoritmos que consideran los tiempos límite de las tareas, habitualmente basados en estrategias ED, como son los descritos por Manabe y Aoyagi [13], Audsley y Burns [14], y el algoritmo *PREC1* presentado en [6]. Los dos últimos presentan como principal inconveniente la falta de estrategias de asignación ya que están pensados para arquitecturas monoprocesador.

Otros algoritmos, como los descritos en [15] o [16] si consideran relaciones de precedencia, y alguno de ellos tienen en cuenta además los costes de comunicación, pero tampoco realizan asignación de tareas.

Por otra parte, hay algoritmos que tratan de reducir los costes de comunicación mediante un equilibrado de carga entre los diferentes procesadores de un sistema multiprocesador, como los descritos por Dong-lk y T.P Baker [17], por Bampis, Delorme y König [18], o el *U.B.A.* [6]. En cambio, estos no consideran relaciones de precedencia entre tareas.

Gran parte de los algoritmos consideran solo tareas periódicas. Tal es el caso del *Slack Method* [12], del *Empty-Slots Method* con RM [11], y de los algoritmos propuestos por Dong-Ik y Baker [17], y por Manabe y Aoyagi [13].

Se han encontrado algoritmos bastante completos que realizan asignación y planificación y consideran los costes de comunicación entre procesadores, entre los que destacan el *Slack Method* [12], basado en un esquema ED, el *Assignment with Precendence Relations Algorithm* [6], y el *Empty-Slots Method* [11], que hace uso de una extensión para

multiprocesador del RM y considera que existen procesadores especializados. Además, el *Empty-Slots Method*, considera la posibilidad de diferentes tipos de procesadores. Sin embargo, el *Slack Method* no tiene en cuenta costes de comunicación. Y aunque estos si son considerados en los otros dos algoritmos, el tratamiento que hacen de los costes de comunicación está enfocado a redes de computadores, y no son aplicables a entornos donde la comunicación entre dos procesadores requiere que ambos estén disponibles en el mismo instante. Por ejemplo, cuando una tarjeta de adquisición y procesamiento de imágenes debe intercambiar datos con un procesador del equipo donde está ubicada.

Excepto el *Empty-Slots Method*, ningún otro algoritmo de los encontrados consideran la posibilidad de diferentes tipos de procesadores. Éste tiene en cuenta la existencia de recursos de proceso especiales (como coprocesadores) que son la única opción donde procesar ciertas tareas. Esas tareas son preasignadas sobre esos recursos de forma previa a la asignación global. Pero como ya se ha comentado, el esquema original está pensado para redes de ordenadores.

Con respecto a algoritmos específicos para visión por computador, la gran mayoría de los existentes tienen como objetivo aumentar el rendimiento de determinadas secuencias de operaciones, muchas veces sobre arquitecturas hardware especificas. Por ejemplo, así ocurre con la técnica presentada en [19], que además no tienen en cuentas los costes de comunicación.

Más generales son las técnicas presentadas por C. Lee, Y. F. Wang y T. Yang [20][21]. Éstas consideran la partición de tareas, relaciones de precedencia, la distribución espacial y la planificación temporal de tareas de aplicaciones de visión por computador en arquitecturas monoprocesador o multiprocesador basadas en matrices 2D de procesadores. Además, tienen en cuenta el tamaño de las imágenes procesadas. Sin embargo, no considera diferentes tipos de procesadores, o procesadores específicos para visión por computador (TAPI).

Otra orientación es la seguida para *Cloner* [22][23], un entorno de especificación de algoritmos de visión por computador que explota el paralelismo de estos. Realmente no hace uso de técnica de planificación, sino que se basa en una extensa librería de operaciones codificadas para ofrecer el mayor rendimiento para determinadas arquitecturas hardware y en función de los tipos de datos empleados.

El algoritmo estático aportado en esta tesis, además de realizar una asignación espacial y planificación temporal de tareas esporádicas para una arquitectura multiprocesador que tienen en cuenta costes de comunicación, las relaciones de precedencia e interrupción de tareas, considera también la existencia de procesadores de distintos tipos. Así la asignación se efectúa en base al tipo de procesador que demanda cada tarea. También tiene en cuenta que los procesadores que deben intercambiar información deben estar disponibles simultáneamente durante la comunicación. Ningún algoritmo de los anteriormente comentados presenta conjuntamente todas estas características.

Las características de los algoritmos comentados se comparan en la siguiente tabla.

Técnica	Base	Proc.[1]	Tipo de tareas	Inter.[2]	Prec.[3]	Tipos[4]	Com[5].	Otros
Slack Method [12]	ED	M	Periódicas	Si	Si	No	No	Busca solución válida antes que óptima
Empty-Slots Method + RM [11]	RM	M	Periódicas	Si	Si	Si	Si	Procesadores no homogéneos. Orientado para RT-LANs.
PREC1 [6]	EDL	1	Esporádicas	Si	Si	No	No	
U.B.A [6].	-	M	Periódicas y esporádicas	No	No	No	Si	Equilibrado de carga y tolerante a fallos
Assignment with Precendence Relations Algorithm [6]	ED	M	Esporádicas		Si	No	Si	
Dong-Ik – Baker [17]	RM	M	Periódicas	Si	No	No	Si	
Manabe- Aoyagi [13]	RM / EDL	M	Periódicas	Si	No	No		Sin Asignación
Lee-Wang-Yang [20][21]	Asigna-ción de pesos	1/M	Esporádicas	No	Si	No	Si	Especifico para Visión por Comp.

[1] Número de procesadores: multiprocesador (M) o monoprocesador (1).
[2] Posibilidad de interrupción de tareas.
[3] Relación de precedencia entre tareas.
[4] Diferentes tipos de procesadores y tareas.
[5] Considera costes de comunicación.

Volver al índice/Tornar a l'índex

Técnica	Base	Proc.[1]	Tipo de tareas	Inter.[2]	Prec.[3]	Tipos[4]	Com[5].	Otros
Algoritmo propuesto	Camino crítico	M	Esporádicas	Si	Si	Si	Si	Considera que dos procesadores deben estar disponibles a la vez para comunicarse

Tabla 2-1. Características de distintas técnicas de planificación estática.

2.5. Resumen

En este capítulo se ha presentado la necesidad que existe en los sistemas de tiempo real de una planificación de tareas, tanto temporal como espacialmente, profundizando en los aspectos relevantes de la planificación estática. También se han descrito formalmente el concepto de tarea y sus características. Posteriormente se ha mostrado que relación guardan estos temas con los sistemas de visión por computador.

Finalmente se ha descrito el estado del arte de la planificación estática, mostrando como los algoritmos existentes no cubren simultáneamente todas las necesidades para las aplicaciones planteadas en esta tesis. Por ello se han planteado nuevos esquemas de planificación que si poseen las características necesarias, los cuales serán tratados en posteriores capítulos.

Capítulo 3

Técnicas de planificación temporal

3.1. Introducción

El algoritmo de camino crítico descrito en este capítulo, al igual que los estudiados en capítulos siguientes, está principalmente destinado a sistemas de visión por computador, ya que tiene en cuenta características importantes de estos, como son las relaciones de precedencia y exclusión existentes entre tareas y la existencia de unidades de proceso específicas (Tarjetas de Adquisición y Procesamiento de Imágenes o TAPIs) además de las CPU genéricas.

Se trata además de un algoritmo que realiza la planificación temporal de las tareas de un DAG considerando que ya existe una asignación espacial previa que indica en que procesador se ejecuta cada tarea. Esto es, no realiza asignación de tareas, aunque considera un sistema multiprocesador. En el capítulo 5 se describirán nuevas versiones de este algoritmo que si realizan la asignación espacial a la vez que la planificación temporal.

Pero antes de comenzar el estudio del algoritmo en cuestión en el apartado 3.3, en el siguiente se expondrán algunas definiciones generales que también son aplicables a los algoritmos descritos en siguientes capítulos de esta tesis.

3.2. Definiciones

3.2.1. Grafo de tareas

La entrada para los diferentes algoritmos es un grafo G dirigido y sin ciclos definido como un conjuntos T de tareas y otro E de relaciones de precedencia:

$$G = (T, E),\ T = \{\tau_1, \tau_2, \tau_3, \cdots \tau_N\},\ \tau_i = (a_i, c_i, d_i)$$

$$E = \{(\tau_i, \tau_j, c_{i,j}) / \tau_i\ R_{prec}\ \tau_j \text{ con coste de comunicación } c_{i,j}\}$$

El grafo tiene N tareas, con $N:=cardinal(T)$, numeradas en el rango $[1, 2... N]$. Para cada tarea i los valores a_i, c_i y d_i representan su tiempo de llegada, su tiempo de computo y su tiempo límite respecto al instante de llegada. a_i, c_i y d_i son valores enteros (a_i, c_i y $d_i \in Z\ \forall i$).

Las relaciones de precedencia entre tareas tienen asociadas un valor que representa el coste que implica la comunicación correspondiente.

Por otra parte, se considera que G no tiene ciclos y que los índices o identificadores de sus tareas están ordenados de la siguiente forma:

$$SI\ \tau_i\ R_{prec}\ \tau_j \rightarrow i < j$$

3.2.2. Conectividad del grafo

Para representar las relaciones de precedencia del grafo de tareas G en los algoritmos, en todos ellos se crean inicialmente las dos siguientes matrices:

- $C_{(N\ x\ N)}$. Matriz de conectividad, que representa las relaciones de precedencia inmediata entre tareas (padres de una tarea). Considerando la ordenación de los índices de las tareas de G descrita antes, esta matriz sólo tiene valores útiles en su parte triangular superior.

$$C_{i,j} := \begin{cases} 1 & \text{SI } \tau_i\ R_{prec}\ \tau_j \\ 0 & \text{En otro caso} \end{cases} \qquad i,j := 1, 2, \ldots N$$

- $T_{(N\ x\ N)}$. Matriz de conectividad total, que representa las relaciones de precedencia (inmediata o no) entre tareas, es decir, todas las tareas que son predecesoras de una dada. Así, $T_{i,j}=1$ significa que la tarea τ_i debe ejecutarse antes que la τ_j. A igual que C, esta matriz sólo tiene valores útiles en su parte triangular superior.

$$T_{i,j} := \begin{cases} 1 & \text{SI } C_{i,,j} = 1 \wedge \exists k / (C_{k,j} = 1 \vee T_{i,k} = 1) \\ 0 & \text{En otro caso} \end{cases} \qquad i, j, k := 1, 2, \ldots N$$

3.2.3. Otra información del grafo

Las tareas del grafo, además de sus tiempos, tienen asociadas otra información. En concreto, para cada tarea, se guarda su tipo y la posibilidad de interrumpirla o no. Para conocer esta información, en algoritmos expuestos más adelante se utilizarán estas funciones:

- *TipoTarea(i)*. Devuelve una constante que indica el tipo de τ_i, y puede ser uno de estas constantes: TCPU, TTAPI y TCPUTAPI, que representan tareas cpu, tapi o cpu/tapi respectivamente.
- *EsInterrumpible(i)*. Devuelve un valor lógico; CIERTO si la tarea τ_i puede ser interrumpida, o FALSO en caso contrario. Las tareas cpu/tapi son siempre no interrumpibles.

3.2.4. Asignación espacial de las tareas

La asignación espacial de las tareas se representa con la matriz *Tabla* de dimensiones (*N x 2*). Esta matriz indica los procesadores asignados a una tarea, y cuales son estos. Tiene dos columnas, la primera para indicar la CPU y la segunda la TAPI. Los procesadores se identifican con números enteros desde 1. *Tabla* se inicializa a partir del tipo de las tareas del grafo y de una asignación espacial previa.

$$Tabla_{i,1} := \begin{cases} -1 & \text{SI } \tau_i \text{ no es tarea } cpu \text{ ni } cpu/tapi \\ 0 & \text{SI } \tau_i \text{ es tarea } cpu \text{ o } cpu/tapi \text{ sin procesador asignado} \\ n > 0 & \text{SI } \tau_i \text{ es tarea } cpu \text{ o } cpu/tapi \text{ asignada a la CPU n} \end{cases}$$

$$Tabla_{i,2} := \begin{cases} -1 & \text{SI } \tau_i \text{ no es tarea } tapi \text{ ni } cpu/tapi \\ 0 & \text{SI } \tau_i \text{ es tarea } tapi \text{ o } cpu/tapi \text{ sin procesador asignado} \\ n > 0 & \text{SI } \tau_i \text{ es tarea } tapi \text{ o } cpu/tapi \text{ asignada a la TAPI n} \end{cases}$$

La matriz *Tabla* también sirve para determinar el tipo de una tarea, facilitando la detección de tareas que usan una CPU (tareas cpu y cpu/tapi) y tareas que usan una TAPI (tareas tapi y cpu/tapi).

3.2.5. Estado de los procesadores

Los procesadores se modelan como objetos capaces de mantener la información relativa a todas las tareas que ejecutan a lo largo del tiempo, y el estado de ejecución de estas. Para conocer y modificar el estado de los procesadores se dispone las siguientes funciones:

- *IniciarTarea(tipo, p, i, t, estado, tipoTarea)*. Inicia la ejecución de la tarea τ_i en el procesador *p* en el instante de tiempo *t*. El tipo de procesador de *p*, CPU o TAPI, está indicado por *tipo*. Los valores *estado* y *tipoTarea* indican respectivamente si la

tarea es interrumpible o no (INT, NOINT) y el tipo de tarea (TCPU, TTAPI y TCPUTAPI).

- *Interrumpible(tipo, p)*. Devuelve CIERTO si la tarea que se ejecuta en el procesador *p* (CPU o TAPI según *tipo*) puede ser interrumpida, o FALSO en caso contrario.
- *ComienzoTarea(tipo, p)*. Devuelve el instante de tiempo absoluto en que comenzó la tarea que actualmente se ejecuta en la *p*. Según *tipo*, *p* refiere a una CPU o TAPI.
- *FinalTarea(tipo, p, t)*. Finaliza la ejecución de la tarea en el procesador *p* en el instante de tiempo *t*, devolviendo el tiempo de ejecución que se ha dedicado a la tarea. Según *tipo*, *p* refiere a una CPU o TAPI.
- *TareaEn(tipo, p)*. Devuelve la tarea actual en el procesador *p*, o un identificador no válido (-1) si la CPU está libre. Según *tipo*, *p* refiere a una CPU o TAPI.
- *EsTarea(i)*. Devuelve CIERTO si τ_i es una tarea válida, esto es, si el valor devuelto por una de las dos funciones anteriores indica que se está ejecutando una tarea en el procesador correspondiente. Devuelve FALSO en otro caso.

Todos los parámetros de tiempo, índices de procesadores e índices de tareas referenciados son valores enteros: $t \in \{0, 1, 2, 3...\}$, $p, i \in \{1, 2, 3...\}$. Las unidades de tiempo pueden referenciar segundos, milisegundos, microsegundos, etc. según la implementación y la aplicación de los algoritmos que se proponen.

El número de procesadores disponible de cada tipo viene indicado por estas dos variables:

- *numeroCPUs* $\in Z$. Número de procesadores tipo CPU disponibles. Así, las CPU se numeran en el rango rango [1, 2, ... *numeroCPUs*].
- *numeroTAPISs* $\in Z$. Número de procesadores tipo TAPIs disponibles. Así, las CPU se numeran en el rango rango [1, 2, ... *numeroTAPIs*].

3.3. Planificación temporal según camino crítico

Este algoritmo es una mejora del desarrollado en [8], estudiada también en [24]. Realiza la planificación temporal de las tareas del DAG de entrada sobre varios procesadores de tipo CPU o TAPI, teniendo en cuanta una asignación previa que indica en que procesador se ejecuta cada tarea. Además es capaz de interrumpir las tareas para conseguir un mayor paralelismo en la ejecución, aunque sin considerar los costes que eso supone. Por otra parte, el algoritmo no tiene en cuenta los costes de comunicación que representan el intercambio de información entre tareas ejecutadas en distintos procesadores.

El tiempo de llegada a_i de las tareas se considera nulo, y el algoritmo puede iniciar la ejecución de una tarea a partir del momento en que todas sus predecesoras han concluido. Dicho de otro modo, el tiempo de llegada está en función de las relaciones de precedencia, Además, no se considera un tiempo límite d_i, y el objetivo es concluir las tareas lo antes posible mediante una ejecución con el mayor grado posible de paralelismo.

A continuación se expone el código del bucle principal del algoritmo. Básicamente, en cada paso del bucle se escoge una tarea del conjunto de tareas listas (tareas con predecesoras ejecutadas que pueden comenzar antes) que además es antecesora de una tarea crítica (tarea sin sucesoras con mayor tiempo de finalización). La tarea escogida será la siguiente que hay que planificar. Con ello se pretende planificar lo antes posible las tareas que más pueden retardar el final de la ejecución.

```
PROC CamCrítico
   Inicializar()
   numTareas:=TareasListas()
   MIENTRAS numTareas > 0 HACER
      MIENTRAS numTareas > 0 HACER
         tareaLista:=SeleccionaTareaLista()
         numTareas:=numTareas - 1
         ActualizaTiempo(Creacion_tareaLista)
         numTareas:=Planificar(tareaLista, numTareas)
      FMIENTRAS
      RetrasarTareas()
      CalculaMatrices()
      numTareas:=TareasListas()
   FMIENTRAS
   ActualizaTiempo(-1)    # Finalizar tareas en ejecución
   Compactar()
FIN
```

El funcionamiento del algoritmo se describe con detalle en los apartados sucesivos, en donde se explica como operan cada una de las funciones que intervienen en él.

3.3.1. Inicializar. Definición de las variables utilizadas

En primer lugar se inician las matrices y variables utilizadas por el algoritmo de la siguiente forma:

- *tiempo*. Contador que indica el instante de tiempo absoluto que se está procesado.
- $Cp_{(N)}$. Vector que indica el tiempo de computo que falta a cada tarea para acabar. Se inicia del siguiente modo:

$$Cp_i := c_i \quad i := 1, 2, \ldots N$$

- $C_{(N \times N)}$, $T_{(N \times N)}$. Matrices de conectividad y conectividad total, tal como se definen en el punto 3.2.2.
- $O_{(N)}$, $D_{(N)}$. Vectores de origen y destino que indican, respectivamente, el número de predecesoras y sucesoras (inmediatas o no) que tiene cada tarea. Se pueden definir del siguiente modo:

$$O := (1, \ldots 1)_{(N \times 1)} \cdot C$$

$$D := [C \cdot (1, \ldots 1)^T_{(N \times 1)}]^T$$

- $Ret_{(N)}$. Este vector de enteros indica el tiempo de retraso acumulado para el comienzo de ejecución de una tarea desde el instante actual. Todas sus componentes valen inicialmente cero (la tarea puede ejecutarse en cuanto sus predecesoras lo hayan hecho), pero se incrementarán durante la ejecución del algoritmo debido a las interrupciones y posposiciones de ejecución de las tareas.
- $F_{(N)}$. Vector de enteros que indica que tareas quedan por acabar de ejecutar. La componente F_i presenta el valor 0 si y sólo si τ_i ha acabado su ejecución. Todas las componentes se inician a *1*, lo que indica que ninguna tarea ha finalizado.
- $L_{(N)}$. Vector de enteros que indica que tareas están listas para ser planificadas. Éstas son las que tienen todas sus antecesoras ejecutadas. La componente L_i da el valor *1* si y sólo si τ_i está lista para ser planificada. Todas las componentes se inician a *0*.
- $Fin_{(N)}$. Vector de enteros con los tiempos absolutos en que las tareas pueden finalizar en el mejor de los casos. Cada valor del vector se inicia con la suma del de tiempo de computo de la tarea relacionada más el máximo tiempo de finalización de entre todas las tareas predecesoras. Así, el valor de tiempo obtenido es el

necesario para completar la tarea si todas sus predecesoras se ejecutan con el máximo paralelismo posible. A ese valor se le añade además el tiempo de retraso de la propia tarea, para reflejar la falta de paralelismo. Este valor que inicialmente es nulo, puede incrementarse durante la ejecución del algoritmo debido a las interrupciones y posposiciones de ejecución, por lo es necesario recalcularlo. El vector se inicia de este modo:

$$Fin_i := Cp_i + Ret_i + \underset{j:=1...i-1}{MAX}\left(T_{j,i} \cdot Fin_j\right) \quad i := 1, 2, \ldots N$$

- *Creación*$_{(N)}$. Este vector de enteros representa los mayores tiempos absolutos de creación de las tareas y se calcula a partir de los tiempos de finalización, en cuyo cálculo se tuvo en cuenta las relaciones de precedencia.

$$Creación_i := Fin_i - Cp_i \quad i := 1, 2, \ldots N$$

- *PesoAcumulado*$_{(N \times 2)}$. Esta matriz de enteros representa, para cada tarea, el mínimo tiempo de ejecución necesario en procesadores CPU (fila 1) y en procesadores TAPI (fila 2) para ejecutar completamente todas las tareas sucesoras suponiendo el máximo paralelismo posible en la ejecución de todas las tareas.

- *Peso*$_{(N \times 2)}$. Esta matriz de enteros representa, para cada tarea, el mínimo tiempo de ejecución necesario en procesadores CPU (fila 1) y en procesadores TAPI (fila 2) para acabar de ejecutar la tarea y todas las tareas sucesoras suponiendo el máximo paralelismo posible en la ejecución de todas las tareas. Sus valores son utilizados para comparar las tareas en el momento de seleccionar la tarea a planificar. Durante la ejecución del algoritmo la matriz *Peso* se calcula a partir de *PesoAcumulado* de este modo:

$$Peso_{i,1} := \begin{cases} PesoAcum_{i,1} + Cp_i & \text{SI SI } TipoTarea(i) \in \{\text{TCPU}, \text{TCPUTAPI}\} \\ PesoAcum_{i,1} & \text{En otro caso} \end{cases}$$

$$Peso_{i,2} := \begin{cases} PesoAcum_{i,2} + Cp_i & \text{SI SI } TipoTarea(i) \in \{\text{TTAPI}, \text{TCPUTAPI}\} \\ PesoAcum_{i,2} & \text{En otro caso} \end{cases}$$

$$i := N, N\text{-}1, \ldots 1$$

Las matrices *PesoAcumulado* y *Peso* se inician con estos pasos aplicados para *i:=N, N-1, ...0*:

$$PesoAcum_{i,1} := \begin{cases} \underset{j:=i+1...N}{MAX}\left(C_{i,j} \cdot Peso_{j,1}\right) & i < N \\ 0 & i = N \end{cases}$$

$$PesoAcum_{i,2} := \begin{cases} \underset{j:=i+1...N}{MAX}\left(C_{i,j} \cdot Peso_{j,2}\right) & i < N \\ 0 & i = N \end{cases}$$

$$Peso_{i,1} := \begin{cases} PesoAcum_{i,1} + Cp_i & \text{SI } TipoTarea(i) \in \{\text{TCPU, TCPUTAPI}\} \\ PesoAcum_{i,1} & \text{En otro caso} \end{cases}$$

$$Peso_{i,2} := \begin{cases} PesoAcum_{i,2} + Cp_i & \text{SI } TipoTarea(i) \in \{\text{TTAPI, TCPUTAPI}\} \\ PesoAcum_{i,2} & \text{En otro caso} \end{cases}$$

Como ejemplo, para el grafo G de la Figura 3-1, se tendrían los valores iniciales mostrados a continuación.

$$C = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad T = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad Tabla = \begin{pmatrix} -1 & 1 & 1 & 1 & 1 & 1 & -1 \\ 1 & 1 & -1 & -1 & -1 & 1 & 1 \end{pmatrix}$$

$$O = (0 \;\; 1 \;\; 1 \;\; 1 \;\; 1 \;\; 2 \;\; 2)$$
$$D = (1 \;\; 3 \;\; 2 \;\; 1 \;\; 0 \;\; 1 \;\; 0)$$

$$Cp = (2 \;\; 1 \;\; 3 \;\; 2 \;\; 2 \;\; 1 \;\; 2) \quad Fin = (2 \;\; 3 \;\; 6 \;\; 5 \;\; 8 \;\; 7 \;\; 9) \quad Creacion = (0 \;\; 2 \;\; 3 \;\; 3 \;\; 6 \;\; 6 \;\; 7)$$

$$PesoAcumulado = \begin{pmatrix} 5 & 4 & 1 & 1 & 0 & 0 & 0 \\ 4 & 3 & 3 & 3 & 0 & 2 & 0 \end{pmatrix} \quad Peso = \begin{pmatrix} 5 & 5 & 4 & 3 & 2 & 1 & 0 \\ 6 & 4 & 3 & 3 & 0 & 3 & 2 \end{pmatrix}$$

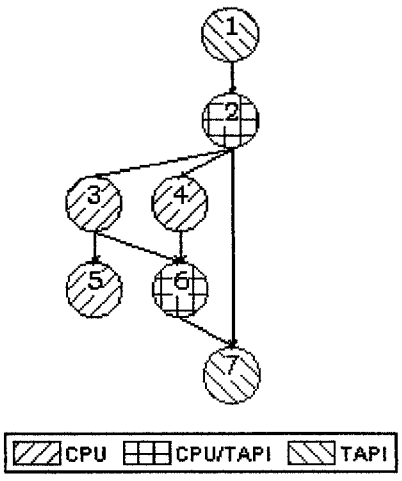


Figura 3-1. Ejemplo de grafo.

3.3.2. TareasListas. Buscar las tareas que se pueden planificar

En este punto se detalla el calculo del vector L que indica las tareas que se pueden planificar en el instante actual. L contiene un 1 para las tareas con un tiempo de creación mínimo de entre las tareas no ejecutadas que tienen todas sus predecesoras ejecutadas o no tienen predecesoras.

$$creaciónMin := \underset{i:=1...N}{MIN}\, Creación_i$$

$$L_i := \begin{cases} 1 & \text{SI } F_i = 1 \wedge \left(\forall j, j := 1...N / C_{j,i} = 1 \rightarrow F_j = 0\right) \wedge Creación_i \leq creacionMin \\ 0 & \text{En otro caso} \end{cases} \qquad i := 1, 2, \ldots N$$

$$numTareas := \sum_{i:=1...N} Li$$

La forma de calcular L en la práctica se describe a continuación. Para cada tarea τ_i no ejecutada, se comprueba si posee alguna predecesora que todavía no se ha ejecutado. De ser así, la tarea no está lista y se deja su componente de L_i con un 0. En caso contrarío se marca la componente de L_i con un 1, y se incrementa el contador de tareas listas. A la vez se calcula el tiempo de creación menor de todas las tareas listas. En segundo lugar se comprueba para cada tarea lista si su tiempo de creación es mayor que el mínimo calculado antes. Las tareas que cumplen esto, es decir, comienzan después que otra tarea lista, se marcan como que no están listas.

```
FUNCION TareasListas()
   tiempo:=INFINITO
   numTareas:=0
   PARA i:=1 HASTA N HACER
      Li:=0
      cont:=0
      SI Fi=1
         PARA j:=1 HASTA N HACER
            SI Fj=1
               SI Cj,i=1 ENTONCES cont:=cont + 1
            FSI
         FPARA
         SI cont=0 ENTONCES
            numTareas:=numTareas + 1
            Li:=1
            SI Creacióni ≤ tiempo ENTONCES
               tiempo:=Creacioni
            FSI
         FSI
      FSI
   FPARA
   PARA i:=1 HASTA N HACER
      SI Li=1 Y Creacioni>tiempo
         Li:=0
         numTareas:=numTareas - 1
      FSI
   FPARA
   DEVOLVER numTareas
FIN
```

3.3.3. SeleccionaTareaLista. Selección de la tarea a planificar

En primer lugar se crea un vector que tiene un 1 en las componentes correspondientes a tareas finales, esto es, tareas que no tienen otras sucesoras.

$$Final_i := \begin{cases} 1 & \text{SI } D_i = 0 \\ 0 & \text{En otro caso} \end{cases} \qquad i := 1, 2, \ldots N$$

$$numFin := \sum_{i:=1\ldots N} Final_i$$

Seguidamente se busca la tarea final que tiene un mayor valor de tiempo de finalización, que se llamará tarea crítica.

$$t_c \text{ es tarea crítica} \Leftrightarrow Fin_{tc} = \underset{i=1\ldots N}{MAX}\left(Final_i \cdot Fin_i\right)$$

Después, se busca la tarea lista ($L_i=1$) que es antecesora de la tarea crítica y posee un mayor peso, comparando indistintamente este valor para CPU o TAPI. En caso de que varias tareas tengan el mismo peso para un tipo de procesador (por ejemplo CPU), se escogerá la que tenga además menor peso para el otro tipo (TAPI), ya que ésta liberará antes los recursos. La tarea crítica se elimina del vector *Final* poniendo su componente a 0. Este proceso se repite hasta que se consigue escoger una tarea lista, que será la tarea a planificar, o hasta que no queden más tareas finales.

Si con el proceso anterior no se ha escogido una tarea a planificar, entonces se busca entre todas las tareas finales que están listas la que tiene un mayor peso. Para ello se vuelve a tener en cuenta todas las tareas finales y el peso máximo establecido anteriormente.

El algoritmo es el siguiente:

```
FUNCION SeleccionaTareaLista()
   seleccion:=-1
   pesoMax:=0

   # Determinar cuales son las tareas finales (sin sucesoras)
   (numFin,Final):=Crear_Vector_Final()

   HACER
      # Buscar tarea final con mayor tiempo de finalización
      tc:=Buscar_Tarea_Crictica(Final)

      # Una tarea final menos tras seleccionar
      numFin:=numFin-1
      Final_tc:=0

      # Para cada tarea lista que es antecesora de la tarea crítica
      PARA i:=1 HASTA N HACER
         SI L_i=1 Y T_i,tc=1 ENTONCES
            # Comprueba si la tarea lista i tiene mayor peso CPU y TAPI
```

```
            # y actualiza máximos para CPU, TAPI y general
            SI Peso_{i,1}>pesoMax O Peso_{i,2}>pesoMax ENTONCES
               pesoMax:=MAXIMO(Peso_{i,1}, Peso_{i,2})
               pesoMaxC:=Peso_{i,1}
               pesoMaxT:= Peso_{i,2}
               seleccion:=i
            SINO SI Peso_{i,1}=pesoMax Y Peso_{i,2}<pesoMaxT
               pesoMax:=MAXIMO(Peso_{i,1}, Peso_{i,2})
               pesoMaxC:=Peso_{i,1}
               pesoMaxT:= Peso_{i,2}
               seleccion:=i
            SINO SI Peso_{i,2}=pesoMax Y Peso_{i,1}<pesoMaxC
               pesoMax:=MAXIMO(Peso_{i,1}, Peso_{i,2})
               pesoMaxC:=Peso_{i,1}
               pesoMaxT:= Peso_{i,2}
               seleccion:=i
            FSI
         FSI
      FPARA
   MIENTRAS numFin>0 Y selección=-1

   # Si no encontró tarea busca tarea final lista con mayor peso
   SI seleccion=-1
      PARA i:=1 HASTA N HACER
         SI D_i=0 Y L_i=1
            # Comprueba si la tarea i tiene mayor peso
            SI Peso_{i,1}≥pesoMax O Peso_{i,2}≥pesoMax
               pesoMax:=MAXIMO(Peso_{i,1}, Peso_{i,2})
               seleccion:=i
            FSI
         FSI
      FPARA
   FSI

   DEVOLVER seleccion
FIN
```

3.3.4. ActualizaTiempo. Determinar el siguiente evento

Tras seleccionar una tarea lista en el bucle principal, es necesario actualizar el contador de tiempo al instante del siguiente evento que puede ocurrir. El evento puede ser la finalización de una tarea en ejecución, o la creación de la nueva tarea lista. Para ello hay que buscar entre todas las tareas en ejecución las que tienen un tiempo de finalización menor que el de creación de la tarea lista (dado en el parámetro *inicio*), y acabar la ejecución de estas.

```
PROC ActualizaTiempo(inicio)
   HACER
      SI inicio<0 ENTONCES
         inicio:=INFINITO
      FSI

      # Buscar tiempo final más temprano en las CPUs y TAPIs
      fin:=INFINITO
      PARA p:=1 HASTA numeroCPUs HACER
         i:=TareaEn(CPU, p)
         SI EsTarea(i) Y fin>Fin_tarea ENTONCES fin:=Fin_tarea
      FPARA
      PARA p:=0 HASTA numeroTAPIs HACER
         i:=TareaEn(TAPI, p)
         SI EsTarea(i) Y fin>Fin_tarea ENTONCES fin:=Fin_tarea
      FPARA

      porAcabar:=0
```

```
        # Si la tarea lista empieza antes de que otras acaben
        SI inicio<fin ENTONCES tiempo:=inicio

        # Si hay tareas que acaban antes de que empiece la lista
        SINO
           tiempo:=fin
           PARA p:=1 HASTA numeroCPUs HACER
              i:=TareaEn(CPU, p)
              SI EsTarea(i) ENTONCES
                 porAcabar:=porAcabar + 1
                 SI fin=Fin_tarea ENTONCES
                    AcabarTarea(CPU, p, tiempo)
                    porAcabar:=porAcabar - 1
                 FSI
              FSI
           FPARA
           PARA p:=0 HASTA numeroTAPIs HACER
              i:=TareaEn(TAPI, p)
              SI EsTarea(i) ENTONCES
                 porAcabar:=porAcabar + 1
                 SI fin=Fin_tarea ENTONCES
                    AcabarTarea(TAPI, p, tiempo)
                    porAcabar:=porAcabar -1
                 FSI
              FSI
           FPARA
        FSI
     MIENTRAS porAcabar>0 Y (inicio:=INFINITO O fin<inicio))
  FIN
```

3.3.5. Planificar. Planificación de la tarea seleccionada

Ya seleccionada una tarea lista para planificar, se determina su tipo y se buscan el procesador (tarea cpu o tapi) o los procesadores (si es una tarea cpu/tapi) correspondientes donde se debe ejecutar. Dependiendo del tipo de la tarea, y de si los procesadores requeridos están libre o ocupados, se pueden dar distintas situaciones, que están reflejadas en la Figura 3-2 y se describen a continuación:

- Si el procesador (una CPU o una TAPI) o procesadores (una CPU y una TAPI) requeridos por la tarea están libres, se inicia en ellos el proceso asociado a la tarea en el instante de tiempo actual y se marca la componente correspondiente a aquella en los vectores F y L a 0, para indicar que ha comenzado su ejecución (y en principio la va a acabar) y que ya no es una tarea lista.
- Si la tarea es cpu/tapi y la TAPI correspondiente está libre pero la CPU está ocupada, se comprueba si la tarea presente en la última es interrumpible. De ser así, se intenta bloquear esa tarea. Si resulta posible el bloqueo, entonces se dispone de dos tareas listas para planificar; la seleccionada para esta iteración y la que se estaba ejecutando en el procesador. Para conseguir planificar la tarea adecuada, se incrementa en una unidad el contador de tareas listas (lo que indica que la tarea lista no se ha planificado) y se recalculan las matrices de tiempos y pesos. Así, en el

bucle principal se asegura otra futura iteración en la que se escogerá la tarea adecuada de entre todas las nuevas tareas listas.

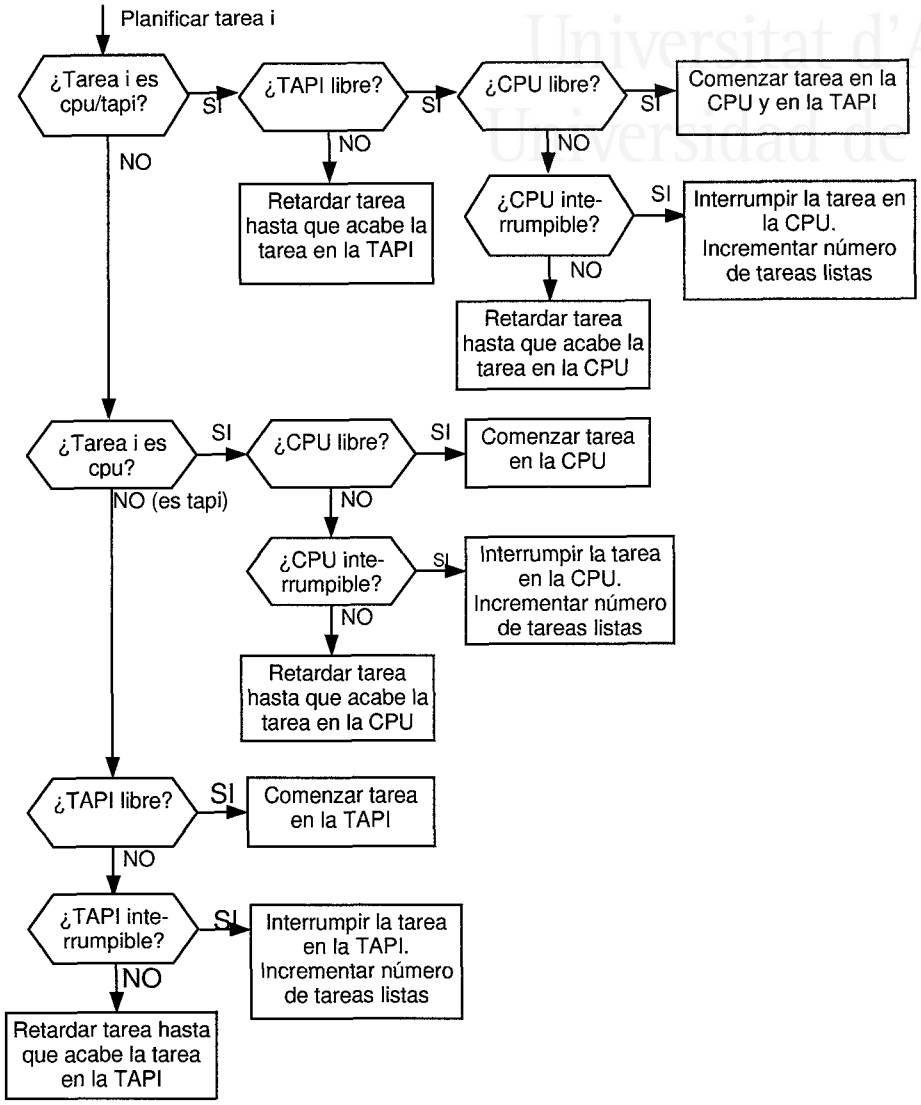
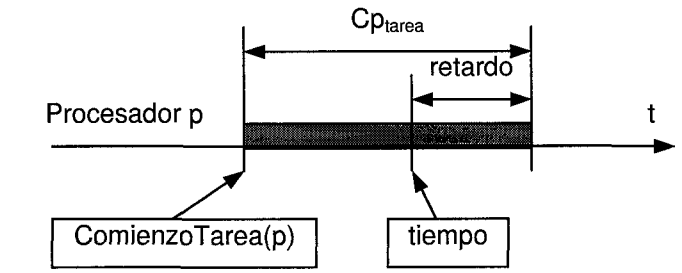


Figura 3-2. Pasos seguidos al planificar una tarea lista.

- Si la CPU no se puede interrumpir, se retarda la tarea lista a planificar hasta que la tarea que hay en el procesador acabe su ejecución. Si, por el contrario, la TAPI estuviese ocupada, directamente se retrasaría la tarea planificar.
- Si la tarea es cpu o tapi, y el procesador requerido está ocupado, se comprueba si la tarea que se ejecuta en éste es interrumpible. De ser así, se intenta bloquear esa tarea, y si resulta posible se incrementa el contador de tareas listas, se recalculan las matrices de tiempos y pesos y se reinicia el proceso. Si el procesador no se puede

interrumpir, se retarda la tarea a planificar hasta que la tarea que hay en el procesador acabe su ejecución.

Cuando hay que retardar una tarea por que el procesador *p* está ocupado, el tiempo de retardo que debe posponerse esa tarea se calcula según se representa en la Figura 3-3. La función *ComienzoTarea(p)* devuelve el instante de tiempo absoluto en que comenzó la tarea que actualmente se ejecuta en *p*, y *TareaEn(p)* el identificador de esa tarea. Finalmente, *tiempo* representa el instante de tiempo actual en que se está planificando.



tarea:=TareaEn(p)
retardo:=Cp_{tarea} -(tiempo-ComienzoTarea(p))

Figura 3-3. Cálculo del tiempo que hay que retardar una tarea.

El algoritmo en pseudcódigo que realiza todo el proceso se expone a continuación.

```
FUNC Planificar(i, numTareas)
   # Números de CPU y TAPI para la tarea i, y estado
   numCPU:=Tabla_{i,1}
   numTAPI:=Tabla_{i,2}
   estado:=EsInterrumpible(i)

   # La tarea es cpu/tapi
   SI numCPU≥0 Y numTAPI≥0
      # Los procesadores están disponibles
      SI TareaEn(CPU,numCPU)=-1 Y TareaEn(TAPI,numTAPI)=-1 ENTONCES
         IniciarTarea(CPU, numCPU, i, tiempo, estado, TCPUTAPI)
         IniciarTarea(TAPI, numTAPI, i, tiempo, estado, TCPUTAPI)
         F_i:=0
         L_i:=0

      #Sólo la TAPI esta libre
      SINO SI TareaEn(TAPI, numTAPI)=-1 ENTONCES
         # Si la CPU se puede interrumpir
         SI Interrumpible(CPU, numCPU) ENTONCES
            SI Bloquear(CPU, i, numCPU)
               numTareas:=numTareas+1
               CalculaNuevasMatrices()
            FSI

         # La CPU no es interrumpible
         SINO
            tarea:=TareaEn(CPU, numCPU)
            Retardar(i, Cp_tarea-(tiempo-ComienzoTarea(CPU, numCPU))
         FSI

      # Tanto la TAPI como la CPU están ocupadas
      SINO
         tarea:=TareaEn(TAPI, numTAPI)
         Retardar(i, Cp_tarea-(tiempo-ComienzoTarea(TAPI, numTAPI))
```

```
      FSI

   # La tarea es tipo cpu
   SINO SI numCPU≥0 ENTONCES
      # La CPU está disponible
      SI TareaEn(CPU, numCPU)=-1 ENTONCES
         IniciarTarea(CPU, numCPU, i, tiempo, estado, TCPU)
         Fi:=0
         Li:=0

      # La CPU no está libre, pero es interrumpible
      SINO SI Interrumpible(CPU, numCPU) ENTONCES
         SI Bloquear(CPU, i, numCPU) ENTONCES
            numTareas:=numTareas+1
            CalculaNuevasMatrices()
         FSI

      # La CPU no es interrumpible
      SINO
         tarea:=TareaEn(CPU, numCPU)
         Retardar(i, Cptarea-(tiempo-ComienzoTarea(CPU, numCPU))
      FSI

   # La tarea es tipo tapi
   SINO
      # La TAPI está disponible
      SI TareaEn(TAPI, numTAPI) ENTONCES
         IniciarTarea(TAPI, numTAPI, i, tiempo, estado, TTAPI)
         Fi:=0
         Li:=0

      # La TAPI no está libre, pero es interrumpible
      SI Interrumpible(TAPI, numTAPI)
         SI Bloquear(TAPI, i, numTAPI) ENTONCES
            numTareas:=numTareas+1
            CalculaNuevasMatrices()
         FSI

      # La TAPI no es interrumpible
      SINO
         tarea:=TareaEn(TAPI, numTAPI)
         Retardar(i, Cptarea-(tiempo-ComienzoTarea(TAPI, numTAPI))
      FSI
   FSI

   DEVOLVER numTareas
FIN
```

3.3.6. RetrasarTareas. Retardar todas las tareas no planificadas

Hay que tener en cuenta que debido a posibles bloqueos de los procesadores en la función *Planificar()* se han podido crear nuevas tareas listas, con lo que se tienen más que las generadas inicialmente por *TareasListas()*. Esto da lugar a que puedan existir tareas listas no planificadas y cuya ejecución deba retrasarse.

Así, este procedimiento se encarga de que todas las tareas listas que no se han podido planificar se retrasan hasta que acaben las que están ocupando los procesadores requeridos según el tipo de tarea, conforme el esquema de la Figura 3-4.

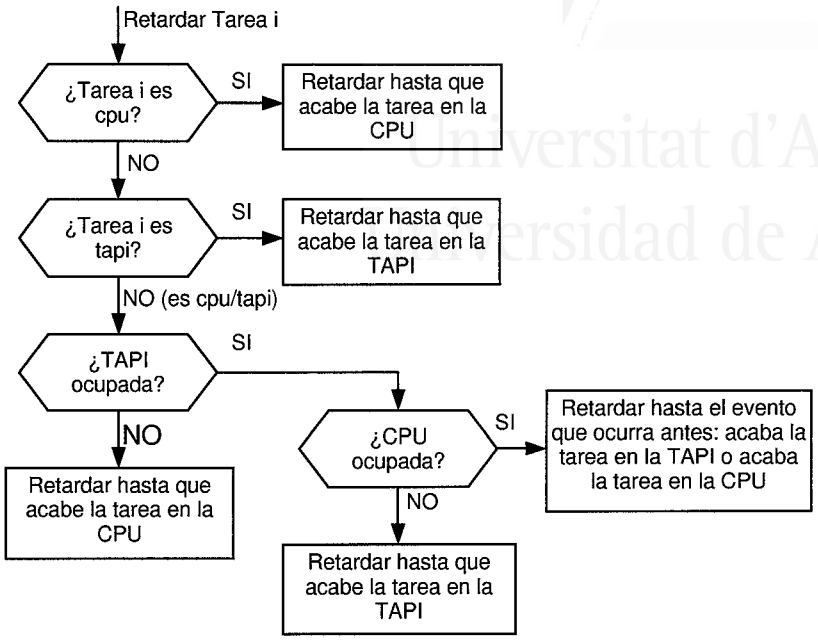


Figura 3-4. Pasos seguidos para retardar una tarea lista no planificada.

El código del procedimiento es éste:

```
PROC RetrasarTareas()
   PARA i:=1 HASTA N HACER
      SI Li=1
         # Obtiene procesadores para la tarea i
         numCPU:=Tabla i,1
         numTAPI:=Tabla i,2

         # Si es tarea cpu y la CPU está ocupada
         SI numTAPI<0 ENTONCES
            tarea:=TareaEn(CPU, numCPU)
            Retardar(i, cptarea-(tiempo-ComienzoTarea(CPU, numCPU)))

         # Si es tarea tapi y la TAPI está ocupada
         SINO SI numCPU<0 ENTONCES
            tarea:=TareaEn(TAPI, numTAPI)
            Retardar(i, cptarea-(tiempo-ComienzoTarea(TAPI, numCPU)))

         # Si es tarea cpu/tapi
         SINO
            tareaT:=TareaEn(TAPI, numTAPI)
            tareaC:=TareaEn(CPU, numCPU)

            # La TAPI está ocupada
            SI EsTarea(tareaT) ENTONCES
               teT:= cptareaT-(tiempo-ComienzoTarea(TAPI, numCPU))

               # LA CPU (además de la TAPI) está ocupada
               SI EsTarea(tareaC) ENTONCES
                  teC:= cptareaC-(tiempo-ComienzoTarea(CPU, numCPU))
                  Retardar(i, MINIMO(teT,teC))

               # Sólo la TAPI está ocupada
               SINO
                  Retardar(i, teT)
               FSI

            # La CPU está ocupada
            SINO
               teC:= cptareaC-(tiempo-ComienzoTarea(CPU, numCPU))
```

```
                Retardar(i, teC)
            FSI
        FSI
    FSI
  FPARA
FIN
```

3.3.7. Bloquear. Interrumpir la tarea que hay en un procesador

Primero se comprueba si la tarea τ_i que hay en el procesador dado por el parámetro p acaba de empezar. De ser así, se da más prioridad a τ_i, a la cual ya fue asignado el procesador, y se retrasa la ejecución de la tarea lista a planificar hasta el final de τ_i. Si, por el contrario, τ_i comenzó con anterioridad al instante actual, se bloquea actualizando sus valores de retardo (Ret_i) y tiempo de proceso (Cp_i) con la duración del intervalo ejecutado, y se marca como tarea lista ($L_i=1$) que no ha acabado ($F_i=1$). El procesador queda libre.

```
FUNC Bloquear(tipoProc, tareaLista, p)
   i:=TareaEn(tipoProc, p)

   # La tarea que está en el procesador acaba de empezar
   SI ComienzoTarea(tipoProc, p)=tiempo ENTONCES
      Retardar(tareaLista, Cpi)
      resultado:=FALSO

   SINO
      duracion:=FinalTarea(tipoProc, p, tiempo)
      Fi:=1                  # No ha acabado
      Li:=1                  # Está lista
      Reti:=Reti+duracion    # Actualiza tiempo de retardo
      Cpi:=Cpi-duracion      # Actualiza Tiempo de computo
      resultado:=CIERTO
   FSI
DEVOLVER resultado
```

3.3.8. Retardar. Posponer la ejecución de una tarea

Incrementa el tiempo de retardo de la tarea indicada según el valor de tiempo dado, y marca esa tarea como que no está lista para planificar:

```
PROC Retardar(i, retardo)
   Reti:=Reti + retardo
   Li:=0
FIN
```

3.3.9. CalcularMatrices. Actualizar los valores de tiempo y peso

Es necesario actualizar los vectores con los tiempos de finalización, y creación, así como la matriz de pesos, para considerar los nuevos tiempos de computo, retardo e

interrupción de las tareas planificadas. Para ello se desarrollan estos cálculos en el procedimiento *CalcularMatrices*:

$$Fin_i := Cp_i + Ret_i + \underset{j:=1...i-1}{MAX}\left(T_{j,i} \cdot Fin_j\right) \quad i := 1, 2, \ldots N$$

$$Creación_i := Fin_i - Cp_i \quad i := 1, 2, \ldots N$$

$$Peso_{i,1} := \begin{cases} PesoAcum_{i,1} + Cp_i & \text{SI } Tabla_{i,1} \geq 0 \\ PesoAcum_{i,1} & \text{En otro caso} \end{cases}$$

$$Peso_{i,2} := \begin{cases} PesoAcum_{i,2} + Cp_i & \text{SI } Tabla_{i,2} \geq 0 \\ PesoAcum_{i,2} & \text{En otro caso} \end{cases} \qquad i := N, N\text{-}1, \ldots 1$$

3.3.10. Compactar. Eliminar cambios de tarea no necesarios

En este algoritmo la compactación consiste en eliminar los cambios innecesarios de tareas en los procesadores utilizados. Estos ocurren cuando la planificación genera diferentes intervalos de tiempo consecutivos para una misma tarea en el mismo procesador debido a la posibilidad de interrupción de tareas. La Figura 3-5 muestra un ejemplo de planificación con cambios innecesarios, y su versión compactada.

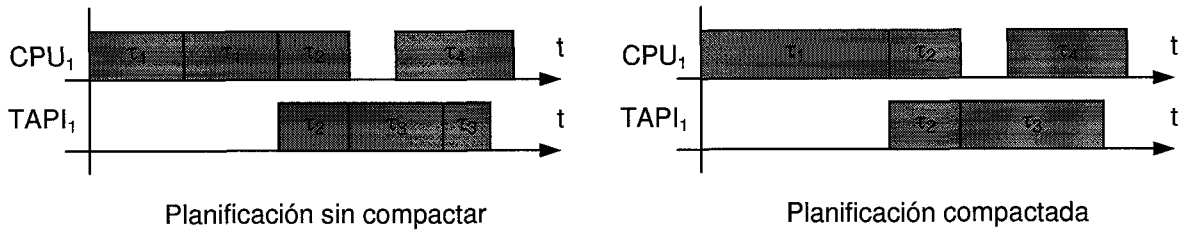


Figura 3-5. Ejemplo de compactación de una planificación.

Capítulo 4

Técnicas de asignación espacial

4.1. Introducción

El algoritmo de camino crítico descrito en el capítulo anterior realiza una planificación temporal de las tareas sobre un sistema multiprocesador, teniendo en cuenta una asignación manual previa de las tareas a determinados procesadores. Pero normalmente dada una aplicación que se ejecuta sobre una arquitectura multiprocesador, existirá la opción de ejecutar una tarea dada en diferentes procesadores, siempre que sean del tipo adecuado, y resulta más interesante y útil que la asignación de las tareas la realice automáticamente un algoritmo que trate de encontrar la mejor opción.

En este capítulo se describe un algoritmo original capaz de realizar la asignación de tareas, teniendo en cuenta la existencia de las relaciones de precedencia entre ellas, así como la existencia de diferentes tipos de procesadores (CPU y TAPI) y de tareas (cpu, tapi y cpu/tapi). Por ello es adecuado para aplicaciones de visión por computador.

El algoritmo, se describe detalladamente en el apartado 4.2, considera como entrada una planificación temporal de las tareas de un DAG G sobre dos procesadores, uno de tipo CPU y otro TAPI, para dar como resultado una asignación espacio-temporal de las tareas sobre un cierto número de procesadores CPU y TAPI. Esto es, realiza la distribución espacial de las tareas entre los procesadores manteniendo sus características temporales y teniendo en cuenta sus relaciones de precedencia.

El grafo de entrada G es como el descrito en el apartado 3.2.1. Para trabajar con las tareas, también se consideran las matrices de conectividad C y T descritas en el apartado 3.2.3. La planificación temporal sobre un procesador de cada tipo requerida se puede obtener con el algoritmo de camino crítico descrito en el Capítulo 3. Así, empleando primero el algoritmo de camino crítico y posteriormente el de asignación espacial se consigue una planificación temporal y una asignación espacial de las tareas.

Con el objetivo de obtener resultados mejores y más realistas se ha creado una segunda versión que aprovecha mejor los huecos de tiempo en los procesadores, y una tercera que además permite limitar el número de procesadores. Ambas se describen en este capítulo, en los apartados 4.3 y 4.4 respectivamente.

Pero antes de empezar a describir los algoritmos de asignación espacial, a continuación se definen algunos conceptos y operaciones necesarias para los mismos.

4.1.1. Gestión de los intervalos de tiempo

Los algoritmos presentados en este capítulo trabajan con los intervalos de tiempo ejecución de las tareas. Estos intervalos son los espacios de tiempo en que se divide la ejecución de una tarea en uno o varios procesadores, a causa de la posibilidad de interrupción. En el caso más simple una tarea se ejecuta en un solo intervalo en un único procesador. Este es también el caso de las tareas no interrumpibles.

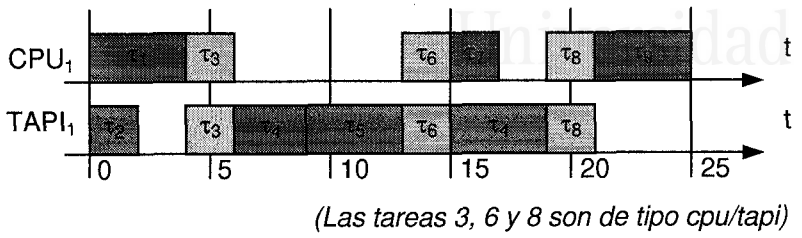
Para el acceso a los objetos que representan el estado de los procesadores, además de utilizarse las funciones descritas en 3.2.3, se definen las siguientes para manejar los intervalos de tiempo asignados a las tareas en una planificación temporal:

- *NúmeroIntervalos(tipo, p)*. Devuelve el número de intervalos de tiempo asignados a las tareas de la planificación existente sobre el procesador número *p* del tipo indicado por *tipo* (CPU o TAPI).
- *Reordenar(tipo, p)*. Asegura que los intervalos de tiempo de las tareas de la planificación existente sobre el procesador número *p* del tipo indicado por *tipo* están ordenados según sus tiempos. Esto es, el primer intervalo, indexado como 1, será el que se inicia y acaba antes, y el último, el dado por *NúmeroIntervalos()*, será el que tenga valores de tiempo mayores.
- *InterTarea(i, tipo, p)*. Devuelve el índice de la tarea que se ejecuta en el intervalo de tiempo *i*, para el procesador *p* de tipo dado por *tipo*.
- *InterComienzo(i, tipo, p), InterFinal(i, tipo, p)*. Devuelven los instantes de comienzo y finalización del intervalo de tiempo *i*, para el procesador *p* de tipo *tipo*.

Todos los parámetros y valores devueltos son valores enteros.

Como ejemplo, para la planificación temporal sobre una CPU y una TAPI que muestra la Figura 4-1, se tendrán los siguientes valores para los intervalos:

	CPU (6 intervalos)						**TAPI** (7 intervalos)						
Intervalos	1	2	3	4	5	6	1	2	3	4	5	6	7
Tareas	1	3	6	7	8	9	2	3	4	5	6	4	8
Comienzo	0	4	13	15	19	21	0	4	6	9	13	15	19
Final	4	6	15	17	21	25	2	6	9	13	15	19	21



(Las tareas 3, 6 y 8 son de tipo cpu/tapi)

Figura 4-1. Ejemplo de los intervalos de ejecución de una planificación.

4.2. Asignación espacial

Dada la planificación temporal sobre un procesador de tipo CPU y otro TAPI y el grafo *G* de entrada, este algoritmo genera una distribución espacial de las tareas planificadas sobre un sistema con varios procesadores de ambos tipos. El algoritmo toma tantos procesadores como hagan falta para minimizar el tiempo global de ejecución, dando como resultado, además de la planificación, los números máximos de procesadores útiles de cada tipo, a partir de los cuales el tiempo total no se puede disminuir.

El algoritmo trabaja con los intervalos de tiempo de ejecución de la planificación temporal de entrada, y no con las tareas a que pertenecen. De hecho, en general considera cada intervalo como independiente. Sólo se tiene en cuenta el hecho de que dos intervalos de ejecución sean de la misma tarea para hacer que se ejecuten en el mismo procesador de la planificación resultante. De este modo, la planificación temporal de entrada puede contener interrupciones de tareas, y este algoritmo de asignación puede decidir mantenerlas o evitar la interrupción de ciertas tareas si con ello se mejora el tiempo de ejecución global.

De forma general, los pasos que realiza el algoritmo son estos:

1. *Ordenación de los intervalos e iniciación.* Asegura que los intervalos de la planificación de entrada están ordenados según su tiempo de ejecución, para después procesarlos en orden. Además se crean las variables y matrices necesarias para el algoritmo.

2. *Seleccionar un intervalo*. Se selecciona el siguiente intervalo según orden de ejecución de la CPU y la TAPI con la planificación de entrada. Puede ser un intervalo de tarea cpu, tapi o cpu/tapi. Los intervalos se identifican según dos índices, uno para los que se ejecutan en la CPU y otros para los que lo hacen en la TAPI. La selección consiste en considerar los intervalos que marcan los índices.

3. *Elegir procesador o procesadores*. Se comprueba si el intervalo se puede ejecutar en uno de los procesadores existentes de la planificación resultado. Para esto se considera el tiempo de espera asociado al intervalo (instante más temprano en que puede comenzar el intervalo), y en que procesadores se ejecutaron posibles intervalos de la misma tarea o de tareas predecesoras a la correspondiente en el intervalo actual. De no poder planificar sobre un procesador existente se incrementa el contador de procesadores necesarios y se asigna al nuevo procesador según su tiempo de espera.

4. *Retardar sucesoras*. Cada uno de los intervalos correspondientes a la misma tarea que el intervalo planificado, o a alguna sucesora suya, se retrasa hasta que finalice el intervalo planificado. Esto es, en caso de que los tiempos de espera de esos intervalos sean anteriores al final del planificado, se igualan a dicho valor.

5. *Repetir desde el paso 2 mientras queden intervalos por procesar.*

El siguiente fragmento de código muestra el procedimiento principal del algoritmo, y los siguientes apartados describen las funciones de que consta. Se considera que la planificación temporal de entrada está almacenada en la CPU 1 y en la TAPI 1. Como salida se dispondrá de una planificación sobre los procesadores CPU y TAPI necesarios.

```
PROC AsignaciónEspacial
   Reordenar(CPU,1)
   Reordenar(TAPI,1)
   Inicializar()

   MIENTRAS QuedanIntervalos() HACER
      (tarea,fin):=ElegirProcesador()
      RetardarSucesoras(tarea, fin)
   FMIENTRAS
   AsignarProcesadores()
FIN
```

4.2.1. Inicializar. Definición de variables utilizadas

Dentro de este procedimiento se crean e inicializan las siguientes variables:

- *nInterCPU, nInterTAPI.* Estas variables representan el número de intervalos en la CPU y en la TAPI con la planificación temporal de entrada (las primeras). Se inician de este modo.

$$nInterCPU:=NumeroIntervalos(CPU,1)$$

$$nInterInter:=NumeroIntervalos(TAPI,1)$$

- *interCPU, interTAPI.* Estas variables representan el número de intervalo en la CPU y en la TAPI con la planificación temporal de entrada que se están analizando en cada paso del bucle principal. Se inician al primer intervalo de cada tipo:

$$interCPU:=1$$

$$interTAPI:=1$$

- $TareasCPU_{(nInterCPU)}$, $TareasTAPI_{(nInterTAPI)}$. Vectores que contienen los índices de las tareas que ejecutan en cada uno de los intervalos de tiempo de la planificación de entrada. Se inician a partir de la función *InterTarea()* relativa a los objetos que representan los procesadores descrita en el punto 4.1.1.

$$TareasCPU_i := InterTarea(i, CPU, 0) \quad i := 1, 2, \ldots nInterCPU$$

$$TareasTAPI_i := InterTarea(i, TAPI, 0) \quad i := 1, 2, \ldots nInterTAPI$$

- $ComCPU_{(nInterCPU)}$, $ComTAPI_{(nInterTAPI)}$. Vectores con los tiempos de comienzo de cada uno de los intervalos de tiempo de la planificación de entrada. Se inician a partir de la función *InterComienzo()* relativa a los objetos que representan los procesadores descrita en el punto 4.1.1.

$$ComCPU_i := InterComienzo\ (i, CPU, 0) \quad i := 1, 2, \ldots nInterCPU$$

$$ComTAPI_i := InterComienzo(i, TAPI, 0) \quad i := 1, 2, \ldots nInterTAPI$$

- $EjeCPU_{(nInterCPU)}$, $EjeTAPI_{(nInterTAPI)}$. Vectores con los tiempos de ejecución que duran de cada uno de los intervalos de la planificación de entrada. Se inician a partir de la función *InterFinal()* relativa a los objetos que representan los procesadores descrita en el punto 4.1.1.

$$EjeCPU_i := InterFinal\ (i, CPU, 0) - ComCPU_i \quad i := 1, 2, \ldots nInterCPU$$

$$EjeTAPI_i := InterFinal(i, TAPI, 0) - ComTAPI_i \quad i := 1, 2, \ldots nInterTAPI$$

- $EspCPU_{(nInterCPU)}$, $EspTAPI_{(nInterTAPI)}$. Vectores con los tiempos de espera correspondientes a cada uno de los intervalos de tiempo de la planificación de entrada. Todas sus componentes se inician a ceros.

- $TipoCPU_{(nInterCPU)}$, $TipoTAPI_{(nInterTAPI)}$. Vectores con los tipos de las tareas correspondientes a los intervalos de la planificación de entrada. No requieren una iniciación previa y sus valores se asignan durante la ejecución del algoritmo.

- $AsigCPU_{(nInterCPU\ x\ nInterCPU)}$, $AsigTAPI_{(nInterTAPI\ x\ nInterTAPI)}$. Matrices que representan la asignación espacial que el algoritmo genera. Contienen el instante en que se inicia la ejecución de cada intervalo (filas) en cada procesador (columnas). Por ejemplo, el valor de $AsigCPU_{2,3}$ es el instante de comienzo del intervalo *2* en el procesador CPU 3. Si un intervalo *i* no está asignado al procesador *p*, el valor de la componente *(i,p)* es –1. Las matrices tienen tantas columnas como intervalos de tiempo hay en la planificación de entrada, ya que en el caso más extremo se puede generar una planificación espacio-temporal de salida con un procesador para cada intervalo. Todas las componentes se de las matrices se inician con el valor –1 para indicar que inicialmente no hay asignación.

$$AsigCPU_{i,p} := -1 \quad i, p := 1, 2, \ldots nInterCPU$$

$$AsigTAPI_{i,p} := -1 \quad i, p := 1, 2, \ldots nInterTAPI$$

- $TCPU_{(nInterCPU)}$, $TTAPI_{(nInterTAPI)}$. Estos vectores mantienen los instantes de tiempo que se están procesando para cada procesador de la asignación resultante. Todas sus componentes se inician a ceros.

- *numCPUs, numTAPIs*. Números de procesadores tipo CPU y TAPI que se requieren para la planificación espacio-temporal resultante. Se inician a cero y se incrementan en el procedimiento *ElegirProcesador()* cuando se requiere.

Todos los valores pertenecen al conjunto de número enteros.

4.2.2. QuedanIntervalos. Comprobar si hay intervalos por analizar

Esta función devuelve CIERTO si alguno de los dos índices para referenciar los intervalos cpu y tapi (*interCPU* y *interTAPI*) no han superado su valor máximo, lo que indica que quedan intervalos de la planificación temporal de entrada por procesar.

```
FUNC QuedanIntervalos()
   SI interCPU≤nInterCPU O interTAPI≤nInterTAPI ENTONCES
      DEVOLVER CIERTO
   SINO
      DEVOLVER FALSO
   FSI
FIN
```

4.2.3. ElegirProcesador. Elección de procesador para un intervalo

Esta función considera los intervalos actualmente seleccionados, dado por las variables *interCPU* y *interTAPI,* para buscarles un procesador o procesadores en la planificación resultado. Cada una de las dos variables hace referencia a los intervalos en uno de los dos procesadores de entrada, uno CPU y otro TAPI, y pueden indicar un intervalo correspondiente a una tarea cpu (se ejecuta sólo en la CPU), tapi (sólo en la TAPI) o cpu/tapi (se ejecuta simultáneamente en la CPU y en la TAPI).

En primer lugar (ver Figura 4-2) la función comprueba si el intervalo cpu seleccionado comienza antes que el intervalo tapi o ya no quedan más intervalos tapi. Si es así se pasa a procesar el intervalo cpu seleccionado, que puede corresponder a una tarea cpu o cpu/tapi. En cualquier caso se busca un procesador para la planificación resultado en donde alojar el intervalo con la función *BuscaProcesadorCPU()*.

Si la tarea correspondiente al intervalo cpu es de tipo cpu (lo cual se determina mediante la función *TipoTarea()* descrita en el punto 3.2.3) se guardan las características del intervalo, como el tipo de la tarea o el comienzo del mismo para su posterior asignación al procesador dado por *BuscaProcesadorCPU()*. Si está función indicaba además que el procesador CPU debe ser uno nuevo, se incrementa el número de procesadores tipo CPU para la planificación resultado. También se incrementa el valor de *interCPU*, para dar paso al siguiente intervalo en la próxima ejecución de la función.

Por el contrario, si la tarea del intervalo es cpu/tapi, se busca también un procesador TAPI para el intervalo tapi correspondiente con la función *BuscaProcesadorTAPI()*. Además resulta necesario que el hueco encontrado para el intervalo tapi corresponda temporalmente

con el encontrado para el intervalo cpu. Para ello puede ser necesario ejecutar varias veces las funciones de búsqueda de procesador y atrasar la ejecución de los intervalos hasta encontrar procesadores válidos. Ya encontrados los procesadores, se guardan las características del intervalo tapi (tipo de la tarea, comienzo del intervalo ...) para su posterior asignación al procesador dado por *BuscaProcesadorTAPI()*. Si está función indicaba además que el procesador TAPI debe ser uno nuevo, se incrementa el número de procesadores tipo TAPI. También se incrementa el valor de *interTAPI* para pasar al siguiente intervalo de ese tipo. Tras procesar el intervalo tapi, se procede con el cpu asociado tal y como se explica en el párrafo anterior.

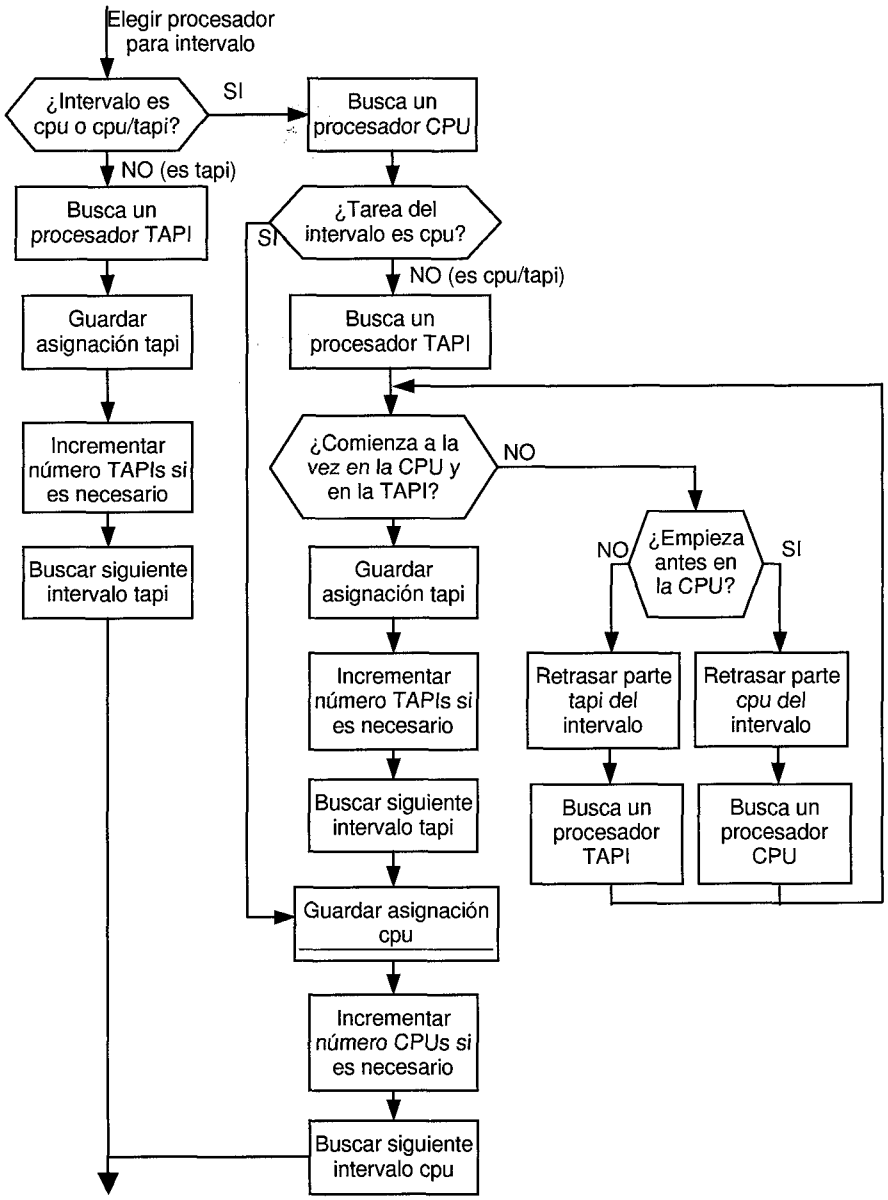


Figura 4-2. Proceso de elección del procesador o procesadores.

Si en vez de un intervalo cpu o cpu/tapi, en la función se decide procesar un intervalo de una tarea tapi por que éste comienza antes, se procede como en el caso de un intervalo de una tarea sólo cpu ya explicado antes, pero utilizando las variables y funciones relativas a los intervalos tapi.

La función acaba devolviendo el índice a la tarea correspondiente a el intervalo planificado (o intervalos en caso de una tarea cpu/tapi) y el tiempo de finalización de éste.

Los pasos que realiza la función se resumen en el diagrama de la Figura 4-2, y el código correspondiente completo se expone a continuación.

```
FUNC ElegirProcesador()
   # Se toma un intervalo cpu o cpu/tapi
   SI interCPU≤nInterCPU Y (interTAPI>nInterTAPI O
                              ComCPU_interCPU≤ComTAPI_interTAPI) ENTONCES
      # Busca una CPU
      (nCPU,comienzoCPU):=BuscaProcesadorCPU()

      # Si la tarea del intervalo es sólo cpu
      tarea:= TareasCPU_interCPU
      SI TipoTarea(tarea)=TCPU ENTONCES
         TipoCPU_interCPU:=TCPU

      # Si la tarea del intervalo es cpu/tapi
      SINO
         # Busca una asignación valida para cpu y tapi
         (nTAPI,comienzoTAPI):=BuscaProcesadorTAPI()
         MIENTRAS comienzoCPU≠comienzoTAPI HACER
            SI comienzoCPU<comienzoTAPI ENTONCES
               # Retrasa el intervalo cpu y busca nueva CPU
               EspCPU_interCPU:=comienzoTAPI
               (nCPU,comienzoCPU):=BuscaProcesadorCPU()
            SINO
               # Retrasa el intervalo tapi y busca nueva TAPI
               EspTAPI_interTAPI:=comienzoCPU
               (nTAPI,comienzoTAPI):=BuscaProcesadorTAPI()
            FSI
         FMIENTRAS

         # Guarda asignación para la TAPI
         AsigTAPI_interTAPI,nTAPI:=comienzoTAPI
         fin:=comienzoTAPI+EjeTAPI_interTAPI
         TTAPI_nTAPI:=fin
         TipoCPU_interCPU:=TCPUTAPI
         TipoTAPI_interTAPI:=TCPUTAPI

         # Comprueba si hay que incrementar número de TAPIs
         SI nTAPI≥numTAPIs ENTONCES
            numTAPIs:=numTAPIs+1
         FSI

         #Siguiente intervalo tapi
         interTAPI:=interTAPI+1
      FSI

      # Guarda asignación para la CPU
      AsigCPU_interCPU,nCPU:=comienzoCPU
      fin:=comienzoCPU+EjeCPU_interCPU
      TCPU_nCPU:=fin
      tarea:= TareasCPU_interCPU

      # Comprueba si hay que incrementar número de CPUs
      SI nCPU≥numCPUs ENTONCES
         numCPUs:=numCPUs+1
      FSI
```

```
      #Siguiente intervalo cpu
      interCPU:=interCPU+1

   # Se toma un intervalo tapi
   ELSE
      # Busca una TAPI
      (nTAPI,comienzoTAPI):=BuscaProcesadorTAPI()

      # Guarda asignación para la TAPI
      AsigTAPI_interTAPI,nTAPI:=comienzoTAPI
      fin:=comienzoTAPI+EjeTAPI_interTAPI
      TTAPI_nTAPI:=fin
      TipoTAPI_interTAPI:=TTAPI
      tarea:= TareasTAPI_interTAPI

      # Comprueba si hay que incrementar número de TAPIs
      SI nTAPI≥numTAPIs ENTONCES
         numTAPIs:=numTAPIs+1
      FSI

      #Siguiente intervalo tapi
      interTAPI:=interTAPI+1
   FSI

   DEVOLVER(tarea,fin)
FIN
```

4.2.4. BuscaProcesador. Búsqueda del mejor procesador

Aunque en la función de elección del procesador se utilizan dos funciones para buscar el procesador al cual asignar un intervalo dado según sea cpu y/o tapi, ambas realizan el mismo trabajo llamando a la función *BuscaProcesador()* con los parámetros adecuados tal y como se indica a continuación:

```
FUNC BuscaProcesadorCPU()
   DEVOLVER BuscaProcesador(interCPU, numCPUs, EspCPU, TCPU,
                            TareasCPU, AsigCPU)
FIN

FUNC BuscaProcesadorTAPI()
   DEVOLVER BuscaProcesador(interTAPI, numTAPIs, EspTAPI, TTAPI,
                            TareasTAPI, AsigTAPI)
FIN
```

El objetivo *BuscaProcesador()* es determinar el número de procesador al que debe asignarse a un intervalo determinado dado por el parámetro *interProc*. El resto de parámetros de esta función indican el número de procesadores y las matrices y vectores que deben utilizarse según el tipo de intervalo (cpu o tapi). La función devuelve el número de procesador asignado y el instante en que comienza el intervalo planificado. Para ello se repite las siguientes acciones para cada procesador existente del tipo adecuado.

Primero se comprueba si el tiempo de espera asociado al intervalo de entrada es igual o mayor que el tiempo de ejecución correspondiente al procesador actual. De ser así, el intervalo es asignado a ese procesador de forma que comience según su tiempo de espera. Si

no se cumple la condición, se pasa a comprobar si algún intervalo ya planificado en el procesador pertenece a la misma tarea que el intervalo de entrada, o a una tarea predecesora. En tal caso, el intervalo de entrada es asignado al procesador actual, y el instante de comienzo es el correspondiente al tiempo de ejecución actual del procesador.

Si, tras recorrer todos los procesadores, no encuentra un procesador para el intervalo, se incrementa el número de procesadores para la planificación resultante, y se asigna el intervalo al nuevo procesador. El intervalo comenzará a ejecutarse según su tiempo de espera.

A continuación se expone el código correspondiente a la función:

```
FUNC BuscaProcesador(interProc, numProc, EspProc, TProc,
                     TareasProc, AsigProc)
   n:=1 # Número de procesador elegido

   # Mientras no se encuentre un procesador entre los actuales
   seguirBuscando:=TRUE
   MIENTRAS seguirBuscando Y n≤numProc HACER
      # Si el tiempo de espera del intervalo es mayor al actual
      SI EspProc_interProc ≥ TProc_n
         comienzo:=EspProc_interProc
         seguirBuscando:=FALSO
      ELSE
         # Busca último intervalo planificado en procesador actual
         i:=interProc
         MIENTRAS i≥1 Y AsigProc_i,n=-1 HACER
            i:=i-1
         FPARA

         # Si encontró intervalo y pertenece a la misma tarea o
         # a una predecesora
         tareaAct=TareasProc_i
         tareaInt=TareasProc_interProc
         SI i≥1 Y (tareaAct=tareaInt O C_tareaInt,tareaAct=1) ENTONCES
            comienzo:=Tiempo_n
            seguirBuscando:=FALSO
         ELSE
            # Sigue buscando en otros procesadores
            n:=n+1
         FSI
      FSI
   FMIENTRAS

   # Si no se encontró procesador hay que añadir uno nuevo
   SI seguirBuscando ENTONCES
      comienzo:=EspProc_interProc
      n:=numProc+1
   FSI

   DEVOLVER (n,comienzo)
FIN
```

4.2.5. RetardarSuc. Retardar tareas sucesoras

Este procedimiento asegura que todos los intervalos correspondientes a tareas sucesoras de la tarea (dada por el parámetro *tarea*) cuyo intervalo acaba de ser planificado son retrasados hasta que acabe este intervalo (valor de tiempo dado por el parámetro *fin*), ya que

no podrán comenzar su ejecución antes. También se retrasan otros posibles intervalos correspondientes a la tarea planificada. Con todo ello se considera la posible interrupción de la tarea planificada.

El procedimiento se compone de dos bucles, uno para las tareas que requieren una CPU y otro para las que requieren una TAPI. Los bucles de búsqueda para tareas sucesoras pueden comenzar desde el número intervalo actual (*interCPU* o *interTAPI*), puesto que los intervalos están ordenados y los anteriores ya han sido planificados.

```
PROC RetardarSuc(tarea, fin)
   # para cada intervalo en una CPU
   PARA i:=interCPU HASTA nInterCPU HACER
      actual:=TareasCPUi
      SI Ttarea,actual=1 O tarea=actual ENTONCES
         EspCPUi:=MAX(EspCPUi, fin)
      FSI
   FPARA

   # para cada intervalo en una TAPI
   PARA i:=interTAPI HASTA nInterTAPI HACER
      actual:=TareasTAPIi
      SI Ttarea,actual=1 O tarea=actual ENTONCES
         EspTAPIi:=MAX(EspTAPIi, fin)
      FSI
   FPARA
FIN
```

4.2.6. AsignarProcesadores. Asignación final a los procesadores

Ya concluido el proceso de elección de procesadores para cada intervalo sólo queda la asignación real de éstos a los objetos que representan los procesadores. En primer lugar, para cada CPU de la planificación de salida se comprueba que intervalos están asignados a ella. Para los intervalos asignados se inicia el objeto CPU correspondiente con los datos de la tarea del intervalo, y los tiempos de inicio y final de ejecución de éste. En segundo lugar se repite el mismo proceso para todos los procesadores tipo TAPI. En ambos casos se considera el número de procesadores a que ha dado lugar el proceso de elección en el bucle principal del algoritmo.

```
PROC AsignarProcesadores()
   # Asigna intervalos a las CPU
   PARA i:=1 HASTA numCPUs HACER
      PARA j=:1 HASTA nInterCPU HACER
         tiempo:=AsigCPUj,i
         SI tiempo≠-1 ENTONCES
            IniciaTarea(CPU,i,TareasCPUj,tiempo,NOINT,TipoCPUj)
            final:=tiempo+EjeCPUj
            FinalTarea(CPU,i,final)
         FSI
      FPARA
   FPARA

   # Asigna intervalos a las TAPI
   PARA i:=1 HASTA numTAPISs HACER
```

```
        PARA j=:1 HASTA nInterTAPI HACER
            tiempo:=AsigTAPIj,i
            SI tiempo≠-1 ENTONCES
                IniciaTarea(TAPIU,i,TareasTAPIj,tiempo,NOINT,TipoTAPIj)
                final:=tiempo+EjeTAPIj
                FinalTarea(TAPI,i,final)
            FSI
        FPARA
    FPARA
FIN
```


4.3. Asignación espacial sin huecos

El algoritmo presentado a continuación es una versión del anterior capaz de realizar una mejor asignación espacial. Para ello trata de evitar que queden huecos de tiempo sin ejecución en los procesadores entre dos intervalos. Estos huecos son asignados a los intervalos para los que no se encuentra un intervalo perteneciente a la misma tarea en un procesador dado dentro de la función *BuscaProcesador()*. Con ello se consigue mejorar el tiempo de ocupación de los procesadores.

Todas las definiciones y código del algoritmo explicado en el punto anterior son aplicables a éste, con las mejoras que a continuación se describen para las funciones de elección y búsqueda de procesador. Además se incluyen nuevas variables y funciones para gestionar los huecos de tiempo.

4.3.1. Inicializar. Definición de variables utilizadas

Como añadido a lo expuesto en el punto 4.2.1, se definen los siguientes vectores y matrices para gestionar los huecos de tiempo en los procesadores de la planificación resultado:

- $ComHuecosCPU_{(nInterCPU\ x\ nInterCPU)}$, $HuecosTAPI_{(nInterTAPI\ x\ nInterTAPI)}$. Matrices que indican los instantes de comienzo de cada posible hueco (filas) en cada uno de los procesadores de la planificación resultado (columna). Todas sus componentes se inician a cero.
- $DurHuecosCPU_{(nInterCPU\ x\ nInterCPU)}$, $DurHuecosTAPI_{(nInterTAPI\ x\ nInterTAPI)}$. Matrices que indican las duraciones de cada posible hueco (filas) en cada uno de los procesadores de la planificación resultado (columna). Todas sus componentes se inician a cero.

- *NumHuecosCPU*$_{(nInterCPU)}$, *NumHuecosTAPI*$_{(nInterTAPI)}$. Vectores que indican el número de huecos existentes en cada uno de los procesadores de la planificación resultado. Se inician a cero, para indicar que al comienzo del algoritmo no existen huecos.

4.3.2. ElegirProcesador. Elección del procesador para un intervalo

Esta función es prácticamente igual a la del algoritmo anterior descrita en el punto 4.2.3. Las modificaciones hacen referencia a la creación y eliminación de huecos con las funciones *CrearHueco()* y *ReducirHueco()*, descritas posteriormente en el apartado 4.3.4.

Ahora, tras encontrar los procesadores a asignar a los intervalos con las funciones *BuscaProcesadorCPU()* y *BuscaProcesadorTAPI()*, se considera la creación de un hueco de tiempo en el procesador correspondiente si la ejecución del intervalo que se está planificando comienza después del instante de tiempo actual, que abarque ese espacio de tiempo.

Por otra parte, también se deben eliminar los huecos de tiempo que han sido asignados a intervalos de tiempo en las funciones *BuscaProcesadorCPU() y BuscaProcesadorTAPI()*. Hay que considerar que estas funciones devuelven ahora el índice del hueco de tiempo asignado a un intervalo, en caso de que se haya efectuado dicha asignación.

A continuación se expone el código equivalente, destacándose con comentarios en negrita las modificaciones con respecto al algoritmo anterior.

```
FUNC ElegirProcesador()

   # Se toma un intervalo cpu o cpu/tapi
   SI interCPU≤nInterCPU Y (interTAPI>nInterTAPI O
                            ComCPU_interCPU≤ComTAPI_interTAPI) ENTONCES
      # Busca una CPU
      (nCPU,comienzoCPU,huecoCPU):=BuscaProcesadorCPU()

      # Si la tarea del intervalo es sólo cpu
      tarea:= TareasCPU_interCPU
      SI TipoTarea(tarea)=TCPU ENTONCES
         TipoCPU_interCPU:=TCPU

      # Si la tarea del intervalo es cpu/tapi
      SINO
         # Busca una asignación valida para cpu y tapi
         (nTAPI,comienzoTAPI):=BuscaProcesadorTAPI()
         MIENTRAS comienzoCPU≠comienzoTAPI HACER
            SI comienzoCPU<comienzoTAPI ENTONCES
               # Retrasa el intervalo cpu y busca nueva CPU
               EspCPU_interCPU:=comienzoTAPI
               (nCPU,comienzoCPU,huecoCPU):=BuscaProcesadorCPU()
            SINO
               # Retrasa el intervalo tapi y busca nueva TAPI
               EspTAPI_interTAPI:=comienzoCPU
               (nTAPI,comienzoTAPI,huecoTAPI):=BuscaProcesadorTAPI()
```

```
            FSI
         FMIENTRAS

         # ---- Crear hueco de tiempo en la TAPI si queda hueco
         SI TTAPI_nTAPI<comienzoTAPI ENTONCES
            CrearHueco(nTAPI, TTAPI_nTAPI, comienzoTAPI,
                  NumHuecosTAPI, ComHuecosTAPI, DurHuecosTAPI)
         FSI

         # ---- Guarda asignación para la TAPI
         AsigTAPI_interTAPI,nTAPI:=comienzoTAPI
         fin:=comienzoTAPI+EjeTAPI_interTAPI
         SI fin>TTAPI_nTAPI ENTONCES
            TTAPI_nTAPI:=comienzoTAPI+EjeTAPI_interTAPI
         FSI
         TipoCPU_interCPU:=TCPUTAPI
         TipoTAPI_interTAPI:=TCPUTAPI

         # ---- Eliminar hueco si se ha utilizado
         SI huecoTAPI>0 ENTONCES
            ReducirHueco(huecoTAPI,nTAPI,comienzoTAPI,EjeTAPI_interTAPI,
                  NumHuecosTAPI, ComHuecosTAPI, DurHuecosTAPI)
         FSI

         # Comprueba si hay que incrementar número de TAPIs
         SI nTAPI≥numTAPIs ENTONCES
            numTAPIs:=numTAPIs+1
         FSI

         #Siguiente intervalo tapi
         interTAPI:=interTAPI+1
      FSI

      # ---- Crear hueco de tiempo en la CPU si queda hueco
      SI TCPU_nCPU<comienzoCPU ENTONCES
         CrearHueco(nCPU, TCPU_nTAPI, comienzoCPU,
               NumHuecosCPU, ComHuecosCPU, DurHuecosCPU)
      FSI

      # --- Guarda asignación para la CPU
      AsigCPU_interCPU,nCPU:=comienzoCPU
      fin:=comienzoCPU+EjeCPU_interCPU
      SI fin >TCPU_nCPU ENTONCES
         TCPU_nCPU:=comienzoCPU+EjeCPU_interCPU
      FSI
      tarea:= TareasCPU_interCPU

      # ---- Eliminar hueco si se ha utilizado
      SI huecoCPU>0 ENTONCES
         ReducirHueco(huecoCPU,nCPU,comienzoCPU,EjeCPU_interCPU,
               NumHuecosCPU, ComHuecosCPU, DurHuecosCPU)
      FSI

      # Comprueba si hay que incrementar número de CPUs
      SI nCPU≥numCPUs ENTONCES
         numCPUs:=numCPUs+1
      FSI

      #Siguiente intervalo cpu
      interCPU:=interCPU+1

   # Se toma un intervalo tapi
   ELSE
      # Busca una TAPI
      (nTAPI,comienzoTAPI,huecoCTAPI):=BuscaProcesadorTAPI()

      # ---- Crear hueco de tiempo en la TAPI si queda hueco
      SI tTAPI_nTAPI<comienzoTAPI ENTONCES
         CrearHueco(nTAPI, TTAPI_nTAPI, comienzoTAPI,
               NumHuecosTAPI, ComHuecosTAPI, DurHuecosTAPI)
      FSI

      # ---- Guarda asignación para la TAPI
      AsigTAPI_interTAPI,nTAPI:=comienzoTAPI
      fin:=comienzoTAPI+EjeTAPI_interTAPI
```

```
      SI fin>TTAPI_nTAPI ENTONCES
         TTAPI_nTAPI:=fin
      FSI
      TipoTAPI_interTAPI:=TTAPI
      tarea:= TareasTAPI_interTAPI

      # ---- Eliminar hueco si se ha utilizado
      SI huecoTAPI>0 ENTONCES
         ReducirHueco(huecoTAPI,nTAPI,comienzoTAPI,EjeTAPI_interTAPI,
                  NumHuecosTAPI, ComHuecosTAPI, DurHuecosTAPI)
      FSI

      # Comprueba si hay que incrementar número de TAPIs
      SI nTAPI≥numTAPIs ENTONCES
         numTAPIs:=numTAPIs+1
      FSI

      #Siguiente intervalo tapi
      interTAPI:=interTAPI+1
   FSI

   DEVOLVER(tarea,fin)
FIN
```

4.3.3. BuscaProcesador. Búsqueda del mejor procesador

Se siguen utilizando las funciones *BuscaProcesadorCPU()*, *BuscaProcesadorTAPI()* y *BuscaProcesador*() como en el algoritmo anterior (ver punto 4.2.4), aunque ahora resulta necesario incluir nuevos parámetros para gestionar los huecos de tiempo:

```
FUNC BuscaProcesadorCPU()

   DEVOLVER BuscaProcesador(interCPU, numCPUs, EspCPU, TCPU,
                  TareasCPU, AsigCPU,
                  EjeCPU, NumHuecosCPU, ComHuecosCPU, DurHuecosCPU)
FIN

FUNC BuscaProcesadorTAPI()

   DEVOLVER BuscaProcesador(interTAPI, numTAPIs, EspTAPI, TTAPI,
                  TareasTAPI, AsigTAPI,
                  EjeCPU, NumHuecosCPU, ComHuecosCPU, DurHuecosCPU)
FIN
```

La principal diferencia de funcionamiento de este algoritmo con respecto al anterior está en la elección del procesador para un intervalo de tiempo dentro de *BuscaProcesador()*. Con el anterior, para cada procesador se comprobaba si existía en él un intervalo perteneciente a la misma tarea que el de entrada, o a una tarea predecesora. Si se encontraba se asignaba el procesador correspondiente, y en caso contrario, se pasaba a examinar el siguiente procesador.

La mejora de este algoritmo es la siguiente. En caso de que para un procesador no se encuentre un intervalo que cumpla las condiciones anteriores, se pasa a buscar un hueco de tiempo en ese procesador donde se pueda ejecutar el intervalo de entrada. Debe tratarse de un hueco lo suficientemente grande que permita la ejecución del intervalo respetando su tiempo

de espera. Si se encuentra el hueco, se asigna el intervalo al procesador, y se redimensiona el espacio de tiempo del hueco para que pueda ser aprovechado para otros intervalos. Si no se encuentra un hueco, entonces se pasa a examinar el siguiente procesador.

Como valores de retorno, ahora la función devuelve el número de hueco asignado para el procesador escogido, además del número de procesador y el tiempo de comienzo para el intervalo como ya hacía en el algoritmo anterior.

A continuación se expone el código correspondiente a la función, en donde se ha destaca el código añadido frente al algoritmo anterior con los comentarios en negrita:

```
FUNC BuscaProcesador(interProc, numProc, EspProc, TProc,
           TareasProc, AsigProc,
           EjeProc, NumHuecosProc, ComHuecosProc, DurHuecosProc)


   n:=1 # Número de procesador elegido
   hueco:=-1 # Numero de hueco escogido (-1 si no se escoge)

   # Mientras no se encuentre un procesador entre los actuales
   seguirBuscando:=TRUE
   MIENTRAS seguirBuscando Y n≤numProc HACER
      # Si el tiempo de espera del intervalo es mayor al actual
      SI EspProc_interProc ≥ TProc_n
         comienzo:=EspProc_interProc
         seguirBuscando:=FALSO

      ELSE
         # Busca último intervalo planificado en procesador actual
         i:=interProc
         MIENTRAS i≥1 Y AsigProc_i,n=-1 HACER
            i:=i-1
         FPARA

         # Si encontró intervalo y pertenece a la misma tarea o
         # a una predecesora
         tareaAct=TareasProc_i
         tareaInt=TareasProc_interProc
         SI i≥1 Y (tareaAct=tareaInt O C_tareaInt,tareaAct=1) ENTONCES
            comienzo:=Tiempo_n
            seguirBuscando:=FALSO
         ELSE

         # --- Búsqueda de un hueco de tiempo
            # Busca hueco desde el más reciente
            huecoEncontrado:=FALSO
            i:=NumHuecosProc_n
            MIENTRAS NO huecoEncontrado Y i>0 HACER
               # Si el hueco es bastante grande para el intervalo
               SI DurHuecosProc_i,n≥EjeProc_interProc ENTONCES
                  # Hueco empieza a la vez o después que intervalo
                  SI ComHuecosProc_i,n≥EspProc_interProc ENTONCES
                     # Se mete intervalo en el hueco
                     comienzo=ComHuecosProc_i,n
                     seguirBuscando:=FALSO
                     huecoEncontrado:=CIERTO
                  # Hueco empieza antes que intervalo
                  SINO
                     # Si el intervalo cabe en el hueco
                     largo:=EspProc_interPro-ComHuecosProc_i,n+
                                 +EjeProc_interProc
                     SI DurHuecosProc_i,n≥largo ENTONCES
                        comienzo:=EspProc_interPro
                        seguirBuscando:=FALSO
```

```
                        huecoEncontrado:=CIERTO
                     FSI
                  FSI

                  # Reduce tamaño del hueco utilizado
                  SI huecoEncontrado ENTONCES
                     ReducirHueco(i, n, EspProc_interPro, EjeProc_interPro,
                           NumHuecosProc, ComHuecosProc, DurHuecosProc)
                     hueco:=i
                  FSI
               FSI
               i:=i-1
            FPARA

            SI NO huecoEncontrado ENTONCES
               # Sigue buscando en otros procesadores
               n:=n+1
            FSI
         # --- Fin búsqueda del hueco

         FSI
      FSI
   FMIENTRAS

   # Si no se encontró procesador hay que añadir uno nuevo
   SI seguirBuscando ENTONCES
      comienzo:=EspProc_interProc
      n:=numProc+1
   FSI

   DEVOLVER (n, comienzo, hueco)
FIN
```

En el código se puede apreciar como, tras comprobar si el hueco es lo bastante grande para el intervalo, se pueden dar tres situaciones: el intervalo comienza antes o a la vez que el hueco, el intervalo comienza después que el hueco y acaba dentro de él, y el intervalo comienza después que el hueco pero no acaba dentro de él.

En el primer caso, representado en la Figura 4-3-a, se considera que el intervalo comienza al inicio del hueco, y después se redimensiona el hueco para aprovechar el posible espacio de tiempo que quede al final de éste. En el segundo caso, representado en la Figura 4-3-b, el intervalo comienza según su tiempo de espera, y se redimensiona el hueco para aprovechar el tiempo restante a su comienzo, así como el que puede quedar a su final para crear un nuevo hueco.

En el tercer caso, simplemente se ignora este hueco y se pasa a explorar el anterior. Esta situación sería igual que el de la mostrada en la Figura 4-3-b, pero con $DurHuecosProc_{i,n}$ menor que el valor de *largo*.

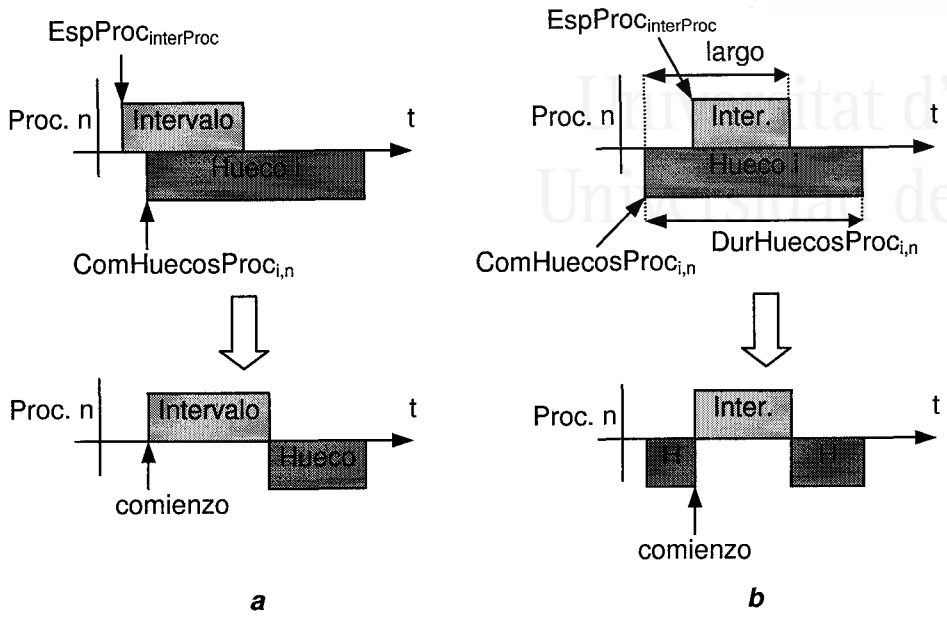


Figura 4-3. Asignación de un intervalo a un hueco.

4.3.4. CrearHueco y ReducirHueco. Gestión de huecos de tiempo

Para gestionar la creación y liberación de huecos de tiempo actualizando los valores de los vectores y matrices correspondientes, se definen estos dos procedimientos:

- *CrearHueco(proc, com, fin, NumHuecos, ComHuecos, DurHuecos)*. Crea un nuevo hueco de tiempo en el procesador *proc*, asignándole como índice el mayor valor de todos. El hueco comienza en el instante dado por *com* y acaba en el instante *fin*. *NumHuecos, ComHuecos, DurHuecos* refieren a los vectores y matrices que deben utilizarse dependiendo del tipo de procesador, CPU o TAPI.

- *ReducirHueco(h, proc, com, dur, NumHuecos, ComHuecos, DurHuecos)*. Reduce la duración de hueco de tiempo existente dado por *h* en el procesador *proc*, tras ser asignado a un intervalo de tiempo. El intervalo comienza en el instante *com* y dura *dur* unidades de tiempo. Si el intervalo ocupa todo el hueco, este último quedará vacío. En cambio, si el intervalo no ocupa todo el hueco, puede seguir quedando espacio al comienzo del hueco, e incluso se puede crear un nuevo hueco con el tiempo restante al final del mismo, tal y como muestra la *Figura 4-4*. *NumHuecos*,

ComHuecos, *DurHuecos* refieren a los vectores y matrices que deben utilizarse dependiendo del tipo de procesador, CPU o TAPI.

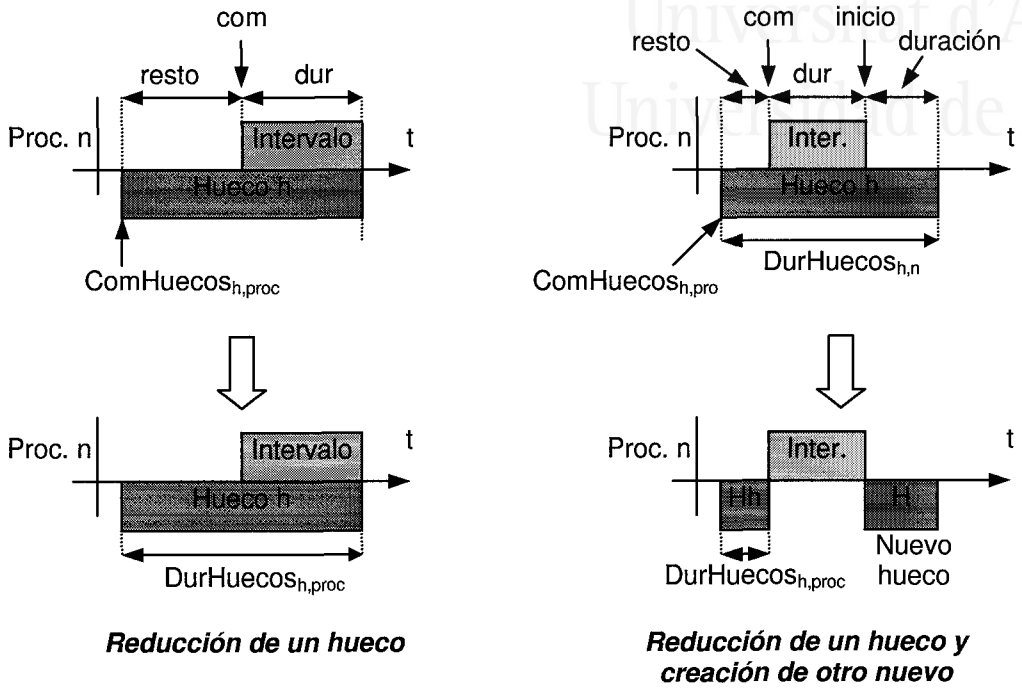


Figura 4-4. Reducción de un hueco de tiempo.

A continuación se detalla el código correspondiente a los dos procedimientos:

```
PROC CrearHueco(proc, com, fin, NumHuecos, ComHuecos, DurHuecos)

   nuevo:=NumHuecos_proc                  # índice del nuevo hueco
   ComHuecos_nuevo,proc:=com             # comienzo del nuevo hueco
   DurHuecos_nuevo,proc:=fin-com         # duración del nuevo hueco
   NumHuecos_proc:= NumHuecos_proc+1     # incrementa número de huecos
FIN

PROC ReducirHueco(h, proc, com, dur,
              NumHuecos, ComHuecos, DurHuecos)

   # Calcula tiempo restante al inicio del hueco
   resto:=com-ComHuecos_h,proc

   # Si queda tiempo al final del hueco se crea un nuevo hueco
   SI resto+dur<DurHuecos_i,proc ENTONCES
      inicio:=com+dur
      duracion:=DurHuecos_i,proc-resto-dur
      CrearHueco(proc, inicio, inicio+duracion,
                 NumHuecos, ComHuecos, DurHuecos)
   FSI

   # Asigna tiempo restante al hueco antiguo
   DurHuecos_h,proc:=resto
FIN
```

4.4. Asignación espacial sin huecos con límite de procesadores

A continuación se expone una segunda mejora del algoritmo de asignación espacial en la que se tiene en cuenta un limite para el número de procesadores que se pueden utilizar. El algoritmo aquí descrito se basa en el de asignación espacial sin huecos descrito en el punto anterior (4.4). Ambos son prácticamente iguales, y las modificaciones afectan sólo a la función *ElegirProcesador()*, donde ahora se comprueba si el procesador escogido por las funciones *BuscaProcesadorCPU()* o *BuscaProcesadorTAPI()* supera el límite de procesadores del tipo correspondiente. Para ello se ha creado la función *AjustaNumProc()*, que debe ser llamada después de obtener el procesador o procesadores para un intervalo de tiempo con las dos funciones anteriores.

AjustaNumProc() comprueba si el número de procesador escogido para un intervalo, bien CPU o bien TAPI, es mayor al límite para ese tipo, y en tal caso busca un nuevo procesador entre los ya existentes. El criterio de elección del procesador es simple: se escoge el que antes quede libre. Esto es lógico, ya que si la tarea fue asignada a un nuevo procesador en su momento, es debido a que no se podía planificar en ninguno de los existentes.

A continuación se detalla el código de esta función. La función recibe como parámetros el número de procesador escogido para un intervalo, el instante de comienzo de éste, la matriz con los tiempos actuales de los procesadores del tipo correspondiente al intervalo y el número máximo de procesadores de ese tipo. Como resultado la función devuelve un número de procesador, menor o igual que el máximo, y el nuevo instante de comienzo del intervalo.

```
FUNC AjustaNumProc(n, comienzo, TProc, max)

   # Si el procesador escogido es mayor al límite
   SI n>max ENTONCES
      comienzo:=INFINITO
      PARA i:=1 HASTA max HACER
         SI TProc_i<comienzo ENTONCES
            comienzo:=Tproc_i
            n:=i
         FSI
      FPARA
   FSI

   DEVOLVER (n,comienzo)
FIN
```

Por ejemplo, al escoger el procesador CPU para un intervalo en *ElegirProcesador()*, la función sería llamada de la siguiente manera:

```
...
# Busca una CPU
(nCPU,comienzoCPU,huecoCPU):=BuscaProcesadorCPU()
(nCPU,comienzoCPU):= AjustaNumProc(nCPU,comienzo,TCPU,numeroCPUs)
...
```

Capítulo 5

Técnicas de asignación espacial y planificación temporal simultáneas

---

5.1. Introducción

En los capítulos 3 y 4 se han presentado respectivamente algoritmos que realizan la planificación temporal y la asignación espacial de tareas de forma independiente. Los algoritmos presentados a continuación son más sofisticados e interesantes, ya que realizan tanto la planificación temporal como la asignación espacial de forma conjunta.

Así, en los sucesivos apartados de este capítulo se describen nuevos algoritmos de planificación espacio-temporal que tienen como base el de camino crítico descrito en el capítulo 3, manteniendo sus características, y que representan sucesivas mejoras de éste. Las versiones más complejas y que aportan características más originales consideran costes de interrupción o de comunicación entre tareas. En resumen, en este capítulo se presentan los siguientes algoritmos:

- Asignación espacial y planificación temporal sin considerar costes de interrupción ni comunicación. En el apartado 5.2 se describen dos variantes de este algoritmo; la primera considera un número ilimitado de procesadores (detallada en el punto 5.2.2) y la segunda un número máximo de procesadores de cada tipo (punto 5.2.3).
- Asignación espacial y planificación temporal considerando los costes de interrupción. Descrito a lo largo del apartado 5.3. También para este caso se detallan dos versiones; considerando los costes de interrupción constantes (punto 5.3.2) y con costes en función del tiempo (punto 5.3.3).
- Asignación espacial y planificación temporal considerando costes de comunicación entre tareas. Descrito en el apartado 5.4.

5.2. Técnicas de asignación espacial y planificación temporal simultáneas que no consideran costes

5.2.1. Introducción

En los dos capítulos anteriores se han expuesto por una parte algoritmos que realizan la planificación temporal y por otra la asignación espacial de tareas. A continuación, dentro de este apartado, se describen dos nuevos algoritmos originales que realizan los dos tipos de planificación simultáneamente. Ambos se basan en la planificación temporal de camino crítico descrito en el apartado 3.3, a la que se añade la funcionalidad de la asignación espacial:

- Asignación espacial y planificación temporal, sin limite de procesadores y sin tener en cuenta costes de interrupción y comunicación.
- Asignación espacial y planificación temporal con límite en el número de procesadores. Tampoco tiene en cuenta costes de interrupción y comunicación

Los dos algoritmos consideran como entrada un grafo de tareas G y utilizan las matrices de conectividad (C y T) y de asignación de tareas (*Tabla*) descritas en el apartado 3.2.2. También se utilizan las funciones de acceso a los objetos que representan el estado de los procesadores descritas en ese mismo apartado.

5.2.2. Asignación espacial y planificación temporal

Se trata de una nueva versión del algoritmo de camino crítico explicado en el apartado 3.3 capaz de efectuar la asignación de las tareas a los procesadores mientras realiza su planificación temporal. Como se muestra a continuación, el procedimiento principal del nuevo algoritmo se diferencia básicamente del anterior en que incluye las funciones *SeleccionarCPU()* y *SeleccionarTAPI()*, encargadas de buscar los procesadores en donde debe ejecutarse la tarea lista antes de ser planificada.

```
PROC Espacio_Temporal
   Inicializar()
   numTareas:=TareasListas()
   MIENTRAS numTareas > 0 HACER
      MIENTRAS numTareas > 0 HACER
         tareaLista:=SeleccionaTareaLista()
         NumTareas:=numTareas - 1
```

```
            ActualizaTiempo(Creacion_tareaLista)

            # Asignar procesadores a la tarea
            numCPU:=SeleccionarCPU(tareaLista)
            numTAPI:=SeleccionarTAPI(tareaLista)

            numTareas:=Planificar(tareaLista,numTareas,numCPU,numTAPI)
         FMIENTRAS
         RetrasarTareas()
         CalculaMatrices()
         numTareas:=TareasListas()
      FMIENTRAS
      ActualizaTiempo(-1)    # Finalizar tareas en ejecución
      AsignarResultado()
      Compactar()
   FIN
```

Así, este algoritmo no requiere una asignación espacial de entrada, especificada en la matriz *Tabla* (ver punto 3.2.3), cuyos valores se asignan durante la planificación.

Por otra parte, ya no se realiza la planificación directamente sobre los objetos que modelan los procesadores, como hace el algoritmo de camino critico, sino que la información sobre planificación se va guardando en unas matrices, y al acabar es cuando se hace la asignación a los objetos procesador. De esto último se encarga el procedimiento *AsignarResultado()*. El motivo principal para realizar esta modificación es que acelera bastante el cómputo del algoritmo, aunque también facilita considerablemente incluir las nuevas características de éste y de otras versiones descritas en apartados posteriores.

El nuevo algoritmo no repara en el número de procesadores CPU y TAPI necesarios, y así la planificación resultante estará hecha sobre el número de procesadores que permite reducir el tiempo de computo global. Es decir, un número mayor de procesadores no mejoraría el resultado.

Como derivado del algoritmo del camino crítico, considera la interrupción de tareas, aunque suponiendo que ésta tiene siempre coste nulo. Los costes de comunicación también se consideran nulos.

En los siguientes apartados se describen las funciones modificadas con respecto al algoritmo de camino crítico, así como las funciones nuevas incorporadas.

5.2.2.1. Inicializar. Definición e inicialización de las variables utilizadas

Además de la variables definidas en el punto 3.3.1, se definen las siguientes:

- $EjecNumInt_{(N)}$. Vector con el número de intervalos de tiempo en que se divide la ejecución de cada tarea debido a las interrupciones. Se inicia todo a unos.

- $EjecCom_{(N \times MAX\_INTER)}$ y $EjecFin_{(N \times MAX\_INTER)}$. Estas matrices contienen los instantes de comienzo y de final de cada intervalo (columna) para cada tarea (fila). Todas las componentes se inician a –1 para representar que no hay intervalos.
- $ActualCPU_{(MAX\_CPU)}$ y $ActualTAPI_{(MAX\_TAPI)}$. Vectores que almacenan las tareas que hay actualmente en cada procesador CPU o TAPI. Todas las componentes de los vectores se inician con el valor –1 para representar que no hay tareas en los procesadores.
- $EstadoCPU_{(MAX\_CPU)}$ y $EstadoTAPI_{(MAX\_TAPI)}$. Estos vectores indican el estado actual de cada procesador. El valor de cada componente puede ser *LIBRE*, *INT* (ejecución interrumpible), o *NOINT* (ejecución no interrumpible). Se inician todas sus componentes con el valor *LIBRE*.

El valor de MAX_INTER depende del número máximo de intervalos que puede tener una tarea. MAX_CPU y MAX_TAPI son el número máximo de procesadores de cada tipo.

5.2.2.2. ActualizaTiempo. Determinar el siguiente evento

Este es un procedimiento muy similar al descrito en el punto para el algoritmo de camino crítico, encargado de actualizar el contador de tiempo al instante del siguiente evento que puede ocurrir; finalización de una tarea en ejecución o la creación de la nueva tarea lista. La diferencia es que en vez de acceder a los objetos que modelan los procesadores se accede a las matrices que representan el estado de estos y los intervalos de ejecución de las tareas.

```
PROC ActualizaTiempo(inicio)
   HACER
      SI inicio<0 ENTONCES
         inicio:=INFINITO
      FSI

      # Buscar tiempo final más temprano en las CPUs y TAPIs
      fin:=INFINITO
      PARA p:=1 HASTA numeroCPUs HACER
         i:=ActualCPU_p
         SI i>0 Y fin>Fin_i ENTONCES fin:=Fin_i
      FPARA
      PARA p:=0 HASTA numeroTAPIs HACER
         i:=ActualTAPI_p
         SI i>0 Y fin>Fin_i ENTONCES fin:=Fin_i
      FPARA

      porAcabar:=0

      # Si la tarea lista empieza antes de que otras acaben
      SI inicio<fin ENTONCES tiempo:=inicio

      # Si hay tareas que acaban antes de que empiece la lista
      SINO
         tiempo:=fin
         PARA p:=1 HASTA numeroCPUs HACER
```

```
                i:=ActualCPUp
                SI i>0 ENTONCES
                   porAcabar:=porAcabar + 1
                   SI fin=Fini ENTONCES
                      j:=EjecNumInti
                      EjeFini,j:=tiempo
                      ActualCPUp:=-1
                      EstadoCPUp:=LIBRE
                      porAcabar:=porAcabar - 1

                      # Si la tarea es cpu cambia al siguiente intervalo
                      SI TipoTarea(i)=TCPU ENTONCES
                         # Si es cpu/tapi debe esperar a analizar la TAPI
                         EjecNumInti:=EjecNumInti+1
                      FSI
                   FSI
                FSI
             FPARA
             PARA p:=0 HASTA numeroTAPIs HACER
                i:=ActualTAPIp
                SI i>0 ENTONCES
                   porAcabar:=porAcabar + 1
                   SI fin=Fini ENTONCES
                      j:=EjecNumInti
                      EjeFini,j:=tiempo
                      ActualTAPIp:=-1
                      EstadoTAPIp:=LIBRE
                      porAcabar:=porAcabar - 1

                      #Siguiente intervalo de una tarea cpu o cpu/tapi
                      EjecNumInti:=EjecNumInti+1
                   FSI
                FSI
             FPARA
          FSI
       MIENTRAS porAcabar>0 Y (inicio:=INFINITO O fin<inicio))
    FIN
```

5.2.2.3. Planificar. Planificación de la tarea seleccionada

Como en el algoritmo del camino crítico, esta función se encarga de planificar en el tiempo la tarea lista de entrada en el procesador o procesadores que tiene asignados, siguiendo el esquema de la figura Figura 3-2 descrito en el punto 3.3.5.

Sin embargo existen dos diferencias. Las primera es que ahora los identificadores de los procesadores (*numCPU* y *numTAPI*) se reciben como parámetros de entrada, en vez de obtenerse de la matriz *Tabla*. La segunda es que no asigna directamente la tarea a los objetos que modelan los procesadores, y en vez de eso trabaja sobre las variables descritas anteriormente en el punto 5.2.2.1, las cuales representan el estado actual de los procesadores y los tiempos de comienzo y fin de los intervalos de ejecución de las tareas. Los motivos de la segunda modificación fueron descritos en el punto 5.2.2.

Por otra parte, se incluye la novedad de que cuando una tarea debe retardarse por que un procesador no está libre, también se desecha la asignación para esa tarea si es que se está planificando por primera vez, esto es, se trata de su primer intervalo de tiempo de ejecución. De este modo, la próxima vez que la tarea sea elegida para ser planificada, también se

buscarán unos nuevos procesadores para ella, posiblemente más adecuados para ese instante que los escogidos ahora. Si la tarea ya se ejecutó anteriormente (no se trata de su primer intervalo), se mantiene la asignación existente para evitar cambios de procesador innecesarios.

Como ejemplo de las modificaciones, se muestra a continuación un fragmento del código de la función *Planificar()* del algoritmo de camino crítico comparada con la nueva.

Camino crítico:

```
...
# La i tarea es cpu/tapi
SI numCPU≥0 Y numTAPI≥0
   # Los procesadores están disponibles
   SI TareaEn(CPU,numCPU)=-1 Y TareaEn(TAPI,numTAPI)=-1 ENTONCES
      IniciarTarea(CPU, numCPU, i, tiempo, estado, CPUTAPI)
      IniciarTarea(TAPI, numTAPI, i, tiempo, estado, CPUTAPI)
      Fi:=0
      Li:=0

   #Sólo la TAPI está libre
   SINO SI TareaEn(TAPI, numTAPI)=-1 ENTONCES
      # Si la CPU se puede interrumpir
      SI Interrumpible(CPU, numCPU) ENTONCES
         SI Bloquear(CPU, i, numCPU)
            numTareas:=numTareas+1
            CalculaNuevasMatrices()
         FSI

      # La CPU no es interrumpible
      SINO
         tarea:=TareaEn(CPU, numCPU)
         Retardar(i, Cptarea-(tiempo-ComienzoTarea(CPU, numCPU))
      FSI
...
```

Nueva función:

```
...
# La i tarea es cpu/tapi
SI numCPU≥0 Y numTAPI≥0
   # Los procesadores están disponibles
   SI EstadoCPUnumCPU=LIBRE Y EstadoTAPInumTAPI ENTONCES
      j:=EjecNumIntt                # Intervalo que se planifica
      EjecComt,j:=tiempo          # Comienzo del intervalo
      EstadoCPUnumCPU:=estado       # Estado de los procesadores
      EstadoTAPInumTAPI:=estado
      ActualCPUnumCPU:=i            # Tarea actual en los procesadores
      ActualTAPInumTAPI:=i
      Fi:=0
      Li:=0

   #Sólo la TAPI está libre
   SINO SI EstadoTAPInumTAPI=LIBRE ENTONCES
      # Si la CPU se puede interrumpir
      SI EstadoCPUnumCPU=INT ENTONCES
         SI Bloquear(EstadoCPU, ActualCPU, i, numCPU)
            numTareas:=numTareas+1
            CalculaNuevasMatrices()
         FSI

      # La CPU no es interrumpible
      SINO
         tarea:=ActualCPUnumCPU       # Tarea que está en la CPU
         j:=EjecNumIntTarea           # Intervalo de la tarea
         Retardar(i, Cptarea-(tiempo-EjecComtarea,j)
         SI j=1 ENTONCES
            Tabla1,1:=0               # CPU sin asignar
            Tabla1,2:=0               # TAPI sin asignar
```

```
            FSI
        FSI
...
```

5.2.2.4. RetrasarTareas. Retardar todas las tareas listas que no se han planificado

El procedimiento encargado de retrasar todas las tareas listas que no se han podido planificar es muy similar al descrito en el punto 3.3.6 para el camino crítico. Existen las pequeñas diferencias comentadas en el punto anterior para la función *Planificar()*.

Sólo cabe destacar dos aspectos. En primer lugar, para cada tarea que se debe retardar se determinan los procesadores donde debe ejecutarse con las nuevas funciones *SeleccionaCPU()* y *SeleccionaTAPI()* descritas posteriormente en el punto 5.2.2.6, en vez de utilizar la matriz *Tabla* directamente, ya que no hay una asignación espacial previa al algoritmo. En segundo lugar, para cada tarea que se retarda se comprueba si está en su primer intervalo de ejecución, y en tal caso se anula la asignación de procesadores efectuada, tal y como se hace en la nueva función *Planificar()*.

5.2.2.5. Bloquear. Interrumpir la tarea que se ejecuta en un procesador

Si la tarea que hay en el procesador dado por p acaba de empezar, se retrasa la ejecución de la tarea lista a planificar hasta el final de la tarea que hay en el procesador. En caso contrario se bloquea la segunda actualizando sus tiempos de retardo y computo y se marca como tarea lista que no ha acabado. También se actualizan las matrices *EjecFin* y *EjecCom* para representar la actualización del intervalo de ejecución actual y el comienzo del siguiente, y se incrementa el número de intervalos asociado a la tarea almacenado en el vector *EjeNumInt*. El procesador se marca como libre.

Como la función puede ser llamada para actuar sobre un procesador CPU o TAPI, debe recibir como parámetros las matrices con el estado y tarea actual del tipo de procesador adecuado, es decir, *EstadoCPU* o *EstadoTAPI* en el parámetro *EstadoProc*, y *ActualCPU* o *ActualTAPI* en *ActualProc*.

```
FUNC Bloquear(EstadoProc, ActualProc, tareaLista, p)
   i:=ActualProc_p     # Tarea en p
   j:=EjeNumInt_i      # Intervalo de t

   # La tarea que está en el procesador acaba de empezar
   SI EjeCom_i,j=tiempo ENTONCES
      Retardar(tareaLista, Cp_i)
      resultado:=FALSO
   SINO
      duracion:=tiempo-EjeCom_i,j
      EjecFin_i,j:=tiempo          # Fin de intervalo actual
      EjeNumInt_i:=EjeNumInt_i+1   # Incrementa número de intervalos
      EstadoProc_p:=LIBRE          # El procesador queda libre
```

```
        ActualProc_p:=-1
        F_i:=1                          # No ha acabado
        L_i:=1                          # Está lista
        Ret_i:=Ret_i+duracion           # Actualiza tiempo de retardo
        Cp_i:=Cp_i-duracion             # Actualiza Tiempo de computo
        resultado:=CIERTO
    FSI
DEVOLVER resultado
```


5.2.2.6. SeleccionaCPU y SeleccionaTAPI. Seleccionar procesadores

Estas son dos funciones nuevas con respecto al algoritmo de camino crítico, y son las encargadas de realizar la asignación espacial. Dada una tarea lista, buscan el procesador donde debe ejecutarse esa tarea y devuelven su índice. Las dos funciones operan de igual modo, sólo que *SeleccionaCPU()* devuelve un índice de CPU para una tarea cpu o cpu/tapi y *SeleccionaTAPI()* devuelve un índice de TAPI para una tarea tapi o cpu/tapi. Como es lógico, cada función utiliza los valores o matrices relativos a su tipo de procesador.

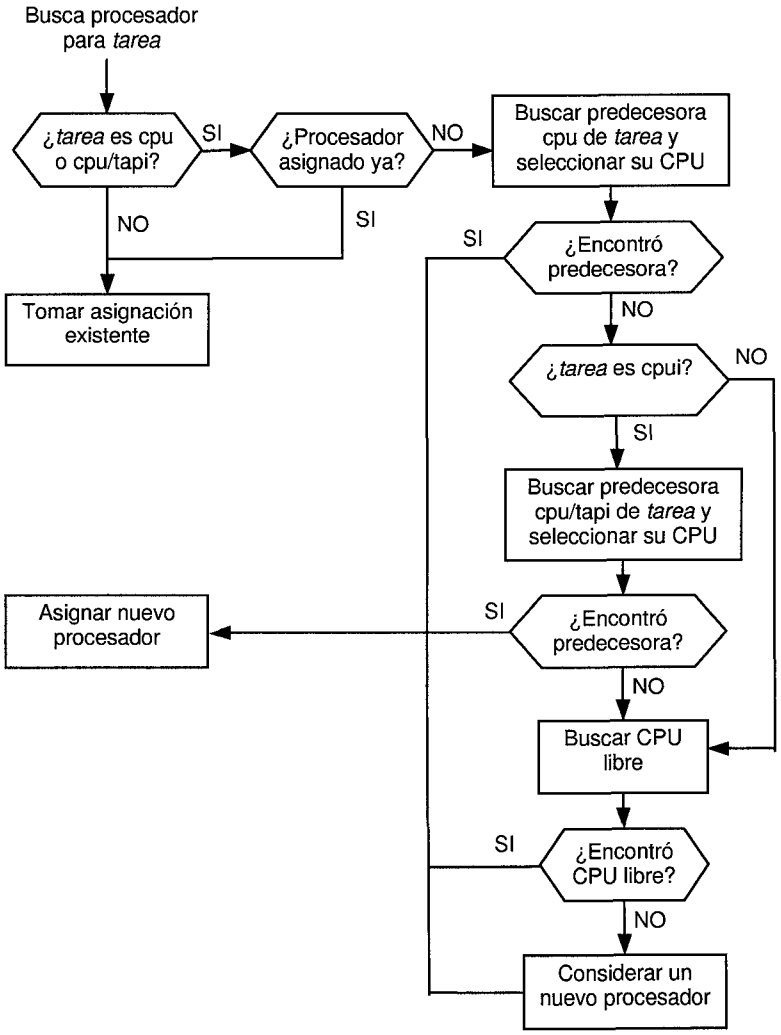


Figura 5-1. Pasos seguidos para seleccionar un procesador CPU.

Como ejemplo, para el caso de *SeleccionaCPU()*, la selección del procesador se realiza según el esquema de la Figura 5-1. En primer lugar se comprueba si la tarea ya tiene asignada una CPU o simplemente no requiere una CPU (es una tarea tapi). En estos casos toma el número de procesador ya asignado (-1 si no requiere CPU). Si no tiene asignada una CPU se continua con las siguientes comprobaciones.

A continuación, se busca una predecesora inmediata de la tarea de tipo cpu para asignarle su mismo procesador. Si no se encuentra esa predecesora y la tarea es cpu/tapi es porque que la predecesora es una tarea tapi, y entonces basta con buscar entre todas las CPUs una que está libre. Si la tarea es cpu, se busca una predecesora cpu/tapi para coger su misma CPU, y si no se encuentra se busca un CPU libre.

En el último caso de no encontrar una CPU libre, lo que se hace es considerar una nueva CPU en el sistema, incrementando el contador de CPUs (*numeroCPUs*). Finalmente se actualiza la matriz *Tabla* con el nuevo procesador encontrado para a tarea.

El código relativo a *SeleccionaCPU()* es el siguiente:

```
FUNC SeleccionaCPU(tarea)
   p:=0

   # Si la tarea no es cpu o cpu/tapi o ya tiene CPU asignada
   SI Tabla_tarea,1<0 O Tabla_tarea,1>0 ENTONCES
      p:= Tabla_tarea,1

   # Es cpu o cpu/tapi y no tiene CPU asignada
   SINO
      # Si la tarea es cpu/tapi busca una predecesora cpu
      SI Tabla_tarea,2≥0 ENTONCES
         PARA i:=1 HASTA N HACER
            SI C_i,tarea=1 Y Tabla_i,1≥0 ENTONCES
               p:=Tabla_i,1
            FSI
         FPARA
      FSI

      # Si no encontró procesador busca una predecesora cpu/tapi. Esa predecesora
      # sólo se puede encontrar si tarea es cpu, y no si es cpu/tapi
      SI p=0 ENTONCES
         i:=1
         MIENTRAS i≤N Y p=0 HACER
            SI C_i,tarea=1 Y Tabla_i,1≥0 Y Tabla_i,2≥0 ENTONCES
               p:=Tabla_i,1
            FSI
            i:=1+1
         FMIENTRAS
      FSI

      # Si no encontró procesador busca una CPU libre
      SI p=0 ENTONCES
         i:=1
         MIENTRAS i≤numeroCPUs Y p=0 HACER
            SI Estado_i=LIBRE ENTONCES
               p:=Tabla_i,1
            FSI
            i:=1+1
         FMIENTRAS
```

```
            #Si no encontró una CPU libre, asigna nuevo procesador
            SI p=0 ENTONCES
               numeroCPUs:=numeroCPUs+1
               p:=numeroCPUs
            FSI
         FSI

         Tabla_tarea,1:=p   # Guarda nuevo procesador asignado
      FSI

      DEVUELVE p
   FIN
```


5.2.2.7. AsignarResultado. Inicializar los objetos procesador

Como último paso del algoritmo antes de compactar los intervalos en los procesadores, queda iniciar los objetos que modelan los procesadores con toda la información sobre los intervalos de tiempo de ejecución de las tareas. Esta información se ha ido guardando durante la planificación en las matrices *Tabla*, *EjeNumInt*, *EjeCom* y *EjeFin*. Mientras que *Tabla* contiene los números de procesadores a los que se asigna cada tarea (asignación espacial), las otras matrices contienen el número de intervalos en que se divide la ejecución de cada tarea y los instantes en que comienzan y acaban estos.

```
PROC AsigarResultado()

   # Para cada tarea
   PARA i:=1 HASTA N HACER
      estado:=EsInterrumpible(i)
      numCPU:=Tabla_i,1
      numTAPI:=Tabla_i,2

      # Para cada intervalo de la tarea
      PARA j:=1 HASTA j≤EjeNumint_i HACER
         # Comienzo y fin del intervalo
         inicio:=EjeCom_i,j
         fin:= EjeFin_i,j

         # Si la tarea es cpu/tapi
         SI numCPU>0 Y numTAPI>0 ENTONCES
            IniciarTarea(CPU, numCPU, i, inicio, estado, TCPUTAPI)
            IniciarTarea(TAPI, numTAPI, i, inicio, estado, TCPUTAPI)
            FinalTarea(CPU, numCPU,fin)
            FinalTarea(TAPI, numTAPI,fin)

         # Si la tarea es cpu
         SINO numCPU>0
            IniciarTarea(CPU, numCPU, i, inicio, estado, TCPU)
            FinalTarea(CPU, numCPU,fin)

         # Si la tarea es tapi
         SINO
            IniciarTarea(TAPI, numTAPI inicio, estado, TTAPI)
            FinalTarea(TAPI, numTAPI,fin)
         FSI
      FPARA
   FPARA
FIN
```

5.2.3. Asignación espacial y planificación temporal con límite de procesadores

El algoritmo descrito en este apartado es una nueva versión del algoritmo de planificación espacio-temporal del apartado anterior (5.2.2), y por lo tanto también está basado en el algoritmo de camino crítico (3.3). Sin embargo presenta cambios importantes respecto a estos. El más destacable es que ahora se considera que el número de procesadores de tipo CPU y el de tipo TAPI están limitados, como sucede en un sistema real. También se modifica en parte el procedimiento principal, y la forma de generar y planificar las tareas listas para adaptarlo a las nuevas características mejorando la eficiencia del código.

Como derivado del algoritmo del camino crítico, considera la interrupción de tareas, aunque suponiendo siempre coste nulo. Los costes de comunicación también se consideran nulos.

El núcleo principal del programa es el siguiente:

```
PROC Espacio_Temporal
   Inicializar()
   numTareas:=TareasListas2()
   MIENTRAS numTareas > 0 HACER
      MIENTRAS numTareas > 0 HACER
         tareaLista:=SeleccionaTareaLista()
         numTareas:=numTareas - 1

         # Asignar procesadores a la tarea
         numCPU:=SeleccionarCPU(tareaLista)
         numTAPI:=SeleccionarTAPI(tareaLista)

         numTareas2:=Planificar(tareaLista,numTareas,numCPU,numTAPI)
      FMIENTRAS
      CalculaMatrices()
      numTareas:=TareasListas()
   FMIENTRAS
   AsignarResultado()
   Compactar()
FIN
```

En comparación con el algoritmo de asignación espacial y planificación temporal del apartado anterior (y con el de camino crítico), desaparecen los procedimientos *ActualizaTiempo()* y *RetrasarTareas()*. El motivo se explica a continuación.

En este nuevo algoritmo, la función *TareasListas2()*, además de obtener las tareas que pueden ser planificadas en el instante de tiempo actual, se encarga también de interrumpir y bloquear las tareas que se ejecutan en todos los procesadores (CPU y TAPI) para hacer que éstas formen también parte del conjunto de tareas listas. Así, en la función *Planificar()*, ya no es necesario bloquear tareas, y por ese motivo ya no se generan nuevas tareas listas no consideradas en un principio que deban ser retardadas. Antes de bloquear las tareas en

ejecución dentro de *TareasListas2()*, es necesario actualizar el instante de tiempo en proceso, por lo que la función *ActualizaTiempo()* forma ahora parte de la primera. Con todo esto se consigue un código más eficiente y claro, y sobre todo, adecuado para gestionar la planificación con limite de procesadores.

Las funciones *SeleccionaTareaLista()* y *Compactar()* no cambian, y siguen siendo las descritas para el algoritmo de camino critico en los puntos 3.3.3 y 3.3.10 respectivamente. Tampoco cambia *AsignaResultado()* con respecto al algoritmo espacio-temporal anterior, siendo la descrita en el punto 5.2.2.7.

Este algoritmo también considera que los intervalos de una tarea generados por la interrupción de ésta pueden asignarse a diferentes procesadores, con lo que se pretende aumentar el paralelismo en la planificación resultante.

Además se introduce el concepto de "prueba de procesador". Con la versión anterior del algoritmo, si no se encuentra un procesador adecuado para una tarea, esto es uno donde se ejecute una tareas predecesora inmediata, se asignaba la tarea al primer procesador libre, lo que no siempre dará el mejor resultado. Ahora, en vez de escoger el primer procesador libre, se irá probando a asignar la tarea en los distintos procesadores hasta que se encuentre un procesador adecuado para ella o directamente sea planificada. Puesto que ahora hay un límite para el número procesadores, las funciones encargadas de la asignación espacial (*SeleccionarCPU() y SeleccionarTAPI()*) ya no pueden considerar nuevos procesadores cuando no se consigue planificar la tarea en los disponibles, y en este caso la tarea se retarda.

Las funciones que presentan cambios con respecto a las versiones anteriores se describen con mas detalle a continuación.

5.2.3.1. Inicializar. Definición de nuevas variables

Además de las variables declaradas en el algoritmo de camino crítico (3.3.1) y en el de asignación espacial y planificación temporal sin limite de procesadores (5.2.2.1), este nuevo algoritmo utiliza las siguientes:

- $PruebaCPU_{(N)}$ y $PruebaTAPI_{(N)}$. Estos vectores representan los contadores de procesadores de tipo CPU y TAPI probados para la asignación espacial de cada tarea. Los contadores se inician a 0, y son utilizados cuando no se encuentra el procesador más adecuado para una tarea.

- $ProcCPU_{(N \times MAX\_INTER)}$ y $ProcTAPI_{(N \times MAX\_INTER)}$. Estas matrices permiten almacenar la asignación espacial que el algoritmo va obteniendo para que, al final, se inicien los objetos que modelan los procesadores según ella. Se han definido estas nuevas variables para poder representar que los intervalos de una tarea se asignen a diferentes procesadores. Esto no es posible con la matriz *Tabla* (ver punto 3.2.4). Para cada tarea (fila) y intervalo de la tarea (columna) las matrices indican el procesador asignado al intervalo. El número de columnas *MAX_INTER* depende del número máximo de intervalos que puede tener una tarea. Todas las componentes se inician con el valor -1 para representar que no hay asignación inicial.

5.2.3.2. TareasListas2. Buscar todas las tareas que pueden planificarse

Esta es una nueva versión de la función que determina las tareas que se pueden planificar en el instante actual. Después de calcularse el vector de tareas listas L (descrito en el punto 3.3.2) para el instante de tiempo actual mediante la función *TareasListas()* original, se actualiza el tiempo actual al tiempo de creación de las tareas listas, que es igual para todas, llamando a *ActaualizaTiempo()* (descrita en el punto 5.2.2.2). Si no hay tareas listas, se actualiza al instante de tiempo infinito para acabar todas las tareas pendientes en los procesadores. Posteriormente, si hay tareas ejecutándose en alguno de los procesadores se interrumpen y bloquean, y también se contabilizan como tareas listas ya que pueden seguir ejecutándose desde el instante actual. Así, todas las tareas quedan en igualdad de oportunidades para ser planificadas. Finalmente se calculan las matrices de tiempos de nuevo (ver punto 3.3.9). Todo este proceso se repite mientras haya tareas a bloquear.

```
FUNC TareasListas2()
   HACER
      numTareas:=TareasListas()

      # Toma tiempo de creación de tareas listas (es igual y mínimo)
      tActual:=INFINITO
      i:=1
      MIENTRAS i≤N Y tActual=INFINITO HACER
         SI Li=1 ENTONCES
            tActual:=Creacióni
         FSI
      FPARA

      # Actualiza al instante de creación de las tareas listas o a infinito
      ActualizaTiempo(tActual)

      hayCambio:=FALSO

      # Bloquear las tareas en todas las CPU interrumpibles
      PARA p:=1 HASTA numeroCPUs HACER
         SI EstadoCPUp=INT ENTONCES
            SI Bloquear(ActualCPU, EstadoCPU, p) ENTONCES
               numTareas:=numTareas+1
               hayCambio:=CIERTO
```

```
            FSI
         FSI
      FPARA

      # Bloquear las tareas en todas las TAPI interrumpibles
      PARA p:=1 HASTA numeroTAPIs HACER
         SI EstadoTAPI_p=INT ENTONCES
            SI Bloquear(ActualTAPI, EstadoTAPI, p) ENTONCES
               numTareas:=numTareas+1
               hayCambio:=CIERTO
            FSI
         FSI
      FPARA

      # Recalcula matrices para una nueva pasada
      SI hayCambio ENTONCES
         CalcularMatrices()
      FSI
   MIENTRAS hayCambio
FIN
```

5.2.3.3. SeleccionaCPU y SeleccionaTAPI. Seleccionar procesadores

Estas son las funciones encargadas de buscar el procesador donde debe ejecutarse una tarea dada como parámetro. Las dos funciones operan de igual modo, sólo que *SeleccionaCPU()* devuelve un índice de CPU para una tarea cpu o cpu/tapi y *SeleccionaTAPI()* devuelve un índice de TAPI para una tarea tapi o cpu/tapi. Son similares a las descritas para el algoritmo de asignación espacial y planificación temporal sin límite de procesadores (punto 5.2.2.6), pero con dos modificaciones.

En primer lugar, antes de buscar un procesador donde se haya ejecutado una tarea predecesora a la de entrada, se comprueba si acaba de finalizar un intervalo de ella, en cuyo caso se escoge como procesador más adecuado el asignado a ese intervalo. Con ello se evitan los cambios innecesarios de procesador para las tareas que podrían surgir debido a que ahora los intervalos se pueden ejecutar en procesadores diferentes.

En segundo lugar, si tras buscar el procesador más adecuado, éste no se encuentra, no se busca el primer procesador libre ni se considera incrementar el número de procesadores. En su lugar se considera probar el siguiente procesador, esto es, se devuelve el número de procesador dado por uno de los contadores *PruebaCPU* o *PruebaTAPI* para la tarea corespondiente, según la función de que se trate, y se incrementa su valor. Ese valor devuelto hace referencia al número de procesador en el que habrá que probar a planificar temporalmente la tarea.

El proceso de selección está esquematizado en la Figura 5-2.

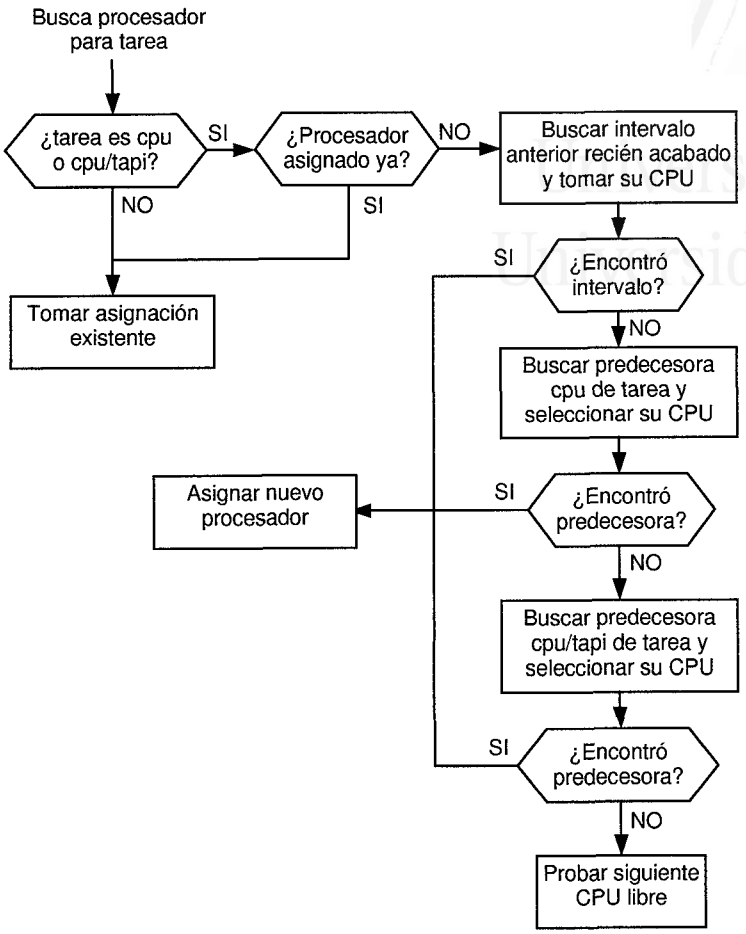


Figura 5-2. Nuevos pasos seguidos para seleccionar un procesador CPU.

La diferencia entre el valor devuelto por las funciones cuando se encuentra un procesador adecuado y cuando se está probando es que en el primer caso se actualiza la *Tabla* de asignación de tareas, pero en el segundo caso no, ya que el procesador no es definitivo, y puede que la tarea acabe planificándose en otro.

Por ejemplo, *SeleccionarCPU()* tiene el código descrito a continuación. *SeleccionarTAPI()* es igual, pero utilizando las variables relativas a procesadores TAPI.

```
FUNC SeleccionaCPU(tarea)
   p:=0

   # Si la tarea no es cpu o cpu/tapi o ya tiene CPU asignada
   SI Tabla_tarea,1<0 O Tabla_tarea,1>0 ENTONCES
      p:= Tabla_tarea,1

   # Es cpu o cpu/tapi y no tiene CPU asignada
   SINO
      # Busca intervalo anterior de la tarea
      j = EjecNumInt_tarea-1;
      SI (j>0) ENTONCES
         # Si intervalo anterior acabó en instante actual
         SI EjecFin_tarea,j = tiempo ENTONCES
            # Toma CPU del intervalo anterior
```

```
            p:=ProcCPU tarea,j;
         FSI
      FSI

      # Si no encontro intervalo
      SI p=0 ENTONCES
         # Si la tarea es cpu/tapi busca una predecesora cpu
         SI Tabla_tarea,2≥0 ENTONCES
            PARA i:=1 HASTA N HACER
               SI C_i,tarea=1 Y Tabla_i,1≥0 ENTONCES
                  p:=Tabla_i,1
               FSI
            FPARA
         FSI

         # Si no encontró procesador busca una predecesora cpu/tapi
         SI p=0 ENTONCES
            i:=1
            MIENTRAS i≤N Y p=0 HACER
               SI C_i,tarea=1 Y Tabla_i,1≥0 Y Tabla_i,2≥0 ENTONCES
                  p:=Tabla_i,1
               FSI
               i:=1+1
            FMIENTRAS
         FSI
      FSI

      # Si encontró procesador guarda nuevo procesador asignado
      SI p>0 ENTONCES
         Tabla_tarea,1:=p

      # Si no encontró procesador, prueba en siguiente procesador de la lista
      SINO
         HACER
            p:=PruebaCPU_tarea
            PruebaCPU_tarea:=PruebaCPU_tarea+1
         MIENTRAS p≤numeroCPUS Y EstadoCPU_p=LIBRE
      FSI
   FSI

   DEVUELVE p
FIN
```

Para la anterior función, el valor devuelto es -1 si la tarea no es del tipo cpu o cpu/tapi, un valor de 1 a *numCPUs* si se encuentra una CPU para asignar o donde probar, o un valor mayor al limite de procesadores (*numCPUs*) si no encuentra CPU. *SeleccionarTAPI()* devuelve valores equivalentes, pero relativos a un procesador TAPI.

5.2.3.4. Planificar. Planificación temporal de una tarea según su tipo

Esta función es la encargada de planificar en el tiempo una tarea sobre los procesadores indicados. Para ello realiza dos acciones básicas, dependiendo si los números de procesadores superan o no el límite de procesadores disponibles.

Si los procesadores están fuera del límite, es que no se encontró un procesador al que asignar la tarea, ni siquiera mediante el procedimiento de prueba. En tal caso su ejecución debe retardarse hasta que finalice la tarea en ejecución que antes acaba. Además de retardarse la tarea, se inician sus contadores de prueba de procesador.

Si por el contrario, los procesadores están dentro del límite, esto es, se pudo asignar la tarea a procesadores disponibles, se planifica según su tipo llamando a una de las tres funciones *PlanificarCPU()*, *PlanificarTAPI()* y *PlanificarCPUTAPI()*. La función también devuelve actualizado el número de tareas listas, ya que puede ocurrir que al intentar planificarla en una de las tres funciones el procesador escogido estuviese ocupado por una tarea no interrumpible.

```
FUNC Planificar(i, numTareas, numCPU, numTAPI)

   # Si no hay procesadores donde planificar t se debe retrasar
   SI numCPU>numCPUs O numTAPI>numTAPIs ENTONCES

      # Si i es tarea cpu/tapi
      SI numCPU≥0 Y numTAPI≥0 ENTONCES
         # Busca cpu que acaba antes
         tareaCPU:=BuscaTareaQueRetrasa(numCPU, numCPUs, ActualCPU)
         interCPU:=EjecNumInt_tareaCPU
         ejeCPU:=EjeCom_tareaCPU,interCPU

         # Busca tapi que acaba antes
         tareaTAPI:=BuscaTareaQueRetrasa(numTAPI, numTAPIs, ActualTAPI)
         interTAPI:=EjecNumInt_tareaTAPI
         ejeTAPI:=EjeCom_tareaTAPI,interTAPI

         # Escoge tarea que más tarde empezó
         SI ejeCPU<ejeTAPI ENTONCES
            tareaAct:=tareaTAPI
         SINO
            tareaAct:=tareaCPU
         FSI

      # Si la tarea es cpu, escoge cpu que acaba antes
      SINO SI numCPU>0 ENTONCES
         tarea:=BuscaTareaQueRetrasa(numCPU, numCPUs, ActualCPU)

      # Si la tarea es tapi, escoge tapi que acaba antes
      SINO
         tarea:=BuscaTareaQueRetrasa(numTAPI, numTAPIs, ActualTAPI)
      FSI

      # Retrasa tarea i, e inicia contadores de procesadores probados
      j:=EjecNumInt_tarea
      Retardar(i, Cp_tarea-(tiempo-EjecCom_tarea,j))
      PruebaProcesadorCPU_i:=0
      PruebaProcesadorTAPI_i:=0

   # Si la tarea tiene asignados procesadores disponibles
   SINO
      # SI la tarea es cpu/tapi
      SI numCPU≥0 Y numTAPI≥0 ENTONCES
         numTareas:=PlanificaCPUTAPI(i, numTareas, numCPU, numTAPI)

      # Si la tarea es cpu
      SINO SI numCPU>0 ENTONCES
         numTareas:=PlanificaCPU(i, numTareas, numCPU)

      # Si la tarea es tapi
      SINO
         numTareas:=PlanificaCPU(i, numTareas, numTAPI)
      FSI
   FSI

   DEVOLVER numTareas
FIN
```

5.2.3.5. PlanificarCPU, PlanificarTAPI y PlanificarCPUTAPI. Planificar una tarea

Estas son las funciones encargadas de planificar una tarea según su tipo en el procesador o procesadores adecuados. Se supone que cada función recibe ya una tarea del tipo adecuado, ya que la clasificación se realiza en la función *Planificar()*, descrita en el punto anterior, que también es quién decide cual de las tres funciones debe ejecutarse.

Como ejemplo, se describe a continuación la función dedicada a planificar una tarea cpu en un procesador CPU. Las dedicadas a tareas tapi o cpu/tapi tienen la misma estructura, pero actúan sobre las variables relativas al tipo de procesador o procesadores implicados.

La función *PlanificarCPU()* recibe como parámetros la tarea a planificar, el número de CPU donde debe hacerse, y el número de tareas listas que hay actualmente. Si el procesador indicado está libre, se inicia en el la ejecución de un nuevo intervalo de la tarea y se marca el procesador como ocupado, con el estado de interrupción asociado a la tarea. La tarea se marca como no lista, y se actualizan las matrices con la asignación espacial (*Tabla*, *ProcCPU* y *ProcTAPI*).

Si, por el contrario, la CPU está ocupada, lo que indica que en ella se ejecuta una tarea no interrumpible (de lo contrario la tarea habría sido bloqueada por la función *TareasListas2()* según se describe en 5.2.3.2), hay que retardar la tarea a planificar. Dependiendo de si el procesador estaba en proceso de prueba o estaba asignado a la tarea se actúa de manera diferente. Si la tarea estaba asignada, directamente se retrasa hasta que acabe el proceso asociado. Si se estaba probando el procesador, y quedan más por probar, se incrementa el contador correspondiente a la tarea ($PruebaProcesadorCPU_t$), se incrementa el número de tareas listas y se deja la tarea sin planificar. Así, podrá ser procesada en otra iteración del bucle principal sobre éste u otro procesador. Si no quedan más procesadores por probar la tarea se retarda hasta que acabe la primera tarea entre todas las que se ejecutan en CPUs.

Cuando una tarea se retarda también se cancela su asignación si estaba en su primer intervalo, con la intención de que en el futuro se pueda escoger una asignación más apropiada al nuevo instante de tiempo.

El código relativo a la función descrita es éste:

```
FUNC PlanificaCPU(i, numTareas, numCPU)
   # Intervalo actual de la tarea
   j:=EjecNumInter_i

   # Si el procesador está libre
   SI EstadoCPU_numCPU=LIBRE ENTONCES
      EjecucionCOM_i,j:=tiempo            # Inicio del nuevo intervalo
```

```
        EstadoCPU_numCPU:=EsInterrumpible(i) # Estado del procesador
        ActualCPU_numCPU:=i

        ProcCPU_i,j:=numCPU          # Datos sobre la asignación
        ProcTAPI_i,j:=-1
        Tabla_i,0:=numCPU

        F_i:=0                       # Tarea planificada
        L_i:=0

    # Procesador ocupado
    SINO
        # Si se está probando procesadores para la tarea
        SI PruebaProcesadorCPU_I>0 ENTONCES
            # Se ha llegado al final y no se encontró procesador en la prueba
            # retrasa la tarea I e inicia el contador de procesadores probados
            SI PruebaProcesadorCPU_I>numProcs ENTONCES
                tarea:=BuscaTareaQueRetrasa(numCPU, numCPUs, ActualCPU)
                j:=EjecNumInt_tarea
                Retardar(I, Cp_tarea-(tiempo-EjecCom_tarea,j))
                PruebaProcesadorCPU_I:=0

                # Cancela asignación si es el primer intervalo de la tarea
                SI EjecNumInt_i=1 ENTONCES
                    Tabla_i,0:=0
                FSI

            # Si quedan procesadores por probar, probará el siguiente
            SINO
                numTareas:=numTareas+1  # Hay que planificar tarea de nuevo
            FSI

        # Si no se está probando procesadores, retrasa tarea t hasta que acabe
        # la tarea ctual en el procesador
        SINO
            tarea:=ActualCPU_numCPU
            j:=EjecNumInt_tarea
            Retardar(i, Cp_tarea-(tiempo-EjecCom_tarea,j))

            # Cancela asignación si es el primer intervalo de la tarea
            SI EjecNumInt_i=1 ENTONCES
                Tabla_i,0:=0
            FSI
        FSI
    FSI

    DEVOLVER numTareas
FIN
```

5.2.3.6. Bloquear. Interrumpir la tarea que se ejecuta en un procesador

Funciona de modo muy similar a las funciones *Bloquear()* del algoritmo anterior y del camino crítico. La principal diferencia es que ahora no recibe como parámetro una tarea lista que deba ser retardada si el bloqueo de la tarea en el procesador indicado no es posible. Es decir, simplemente se bloquea la tarea τ_i que se ejecuta en el procesador dado por el parámetro *p*, finalizando el intervalo actual (*j*) de esa tarea.

Además, cuando la tarea que se ejecuta es interrumpida, se reinicia su asignación de procesadores, poniendo los valores por defecto en los vectores *PruebaProcesadorCPU*, *PruebaProcesadorTAPI* y en la matriz *Tabla*. Con ello se pretende que en la siguiente selección de la tarea para su planificación se vuelva a seleccionar los procesadores más adecuados.

La función requiere como parámetros las matrices *EstadoCPU* o *EstadoTAPI* en *EstadoProc*, y *ActualCPU* o *ActualTAPI* en *ActaulProc*, según el tipo de procesador a bloquear. Devuelve CIERTO si se ha bloqueado una tarea y FALSO en caso contrario.

```
FUNC Bloquear(EstadoProc, ActualProc, p)
   i:=ActualProc_p     # Tarea en p
   j:=EjeNumInt_i      # Intervalo de la tarea

   # La tarea que está en el procesador acaba de empezar
   SI EjeCom_tarea,j=tiempo ENTONCES
      resultado:=FALSO
   SINO
      duracion:=tiempo- EjeCom_i,j
      EjecFin_i,j:=tiempo          # Fin de intervalo actual
      EjeNumInt_i:=EjeNumInt_i+1   # Incrementa número de intervalos
      EstadoProc_p:=LIBRE          # El procesador queda libre
      ActualProc_p:=-1
      F_i:=1                       # No ha acabado
      L_i:=1                       # Está lista
      Ret_i:=Ret_i+duracion        # Actualiza tiempo de retardo
      Cp_i:=Cp_i-duracion          # Actualiza Tiempo de computo

      # Cancela la asignación de procesadores a la tarea
      PruebaProcesadorCPU_i:=0
      PruebaProcesadorTAPI_i:=0
      SI Tabla_i,0>0 ENTONCES
         Tabla_i,0 = 0
      FSI
      SI Tabla_i,1>0 ENTONCES
         Tabla_i,1 = 0
      FSI

      resultado:=CIERTO
   FSI
DEVOLVER resultado
```

5.2.3.7. BuscaTareaQueRetrasa. Determinar tarea a la que se debe esperar

Esta función devuelve la tarea que está retrasando la ejecución de otra tarea en el procesador número *p* dado como parámetro. Si *p* es un procesador dentro de los disponibles (menor o igual al límite dado por el parámetro *numProcs*), se devuelve la tarea que se ejecuta en ese procesador. Si *p* indica un procesador por encima del límite, se busca la tarea que antes acaba de entre todas las que se ejecutan en los procesadores disponibles. El parámetro *ActualProc* hace referencia a la matriz *ActualCPU* o a *ActualTAPI* según sea el tipo de *p*.

```
FUNC BuscaTareaQueRetrasa(p, numProcs, ActualProc)

   # Si el procesador no es válido
   SI p>numProcs ENTONCES
      # Busca tarea que acaba antes en los procesadores
      computoMax:=INFINITO
      PARA i:=1 HASTA numProcs HACER
         tarea:=ActualProc_p
         SI tarea>0 ENTONCES
            j:=EjecNumInt_tarea
            computo:=Cp_t-(tiempo-EjecCom_tarea,j)
            SI computo<computoMax
               computoMax:=computo
               tareaRet:=tarea
            FSI
         FSI
      FPARA
```

```
    # Si el procesador es válido
    SINO
       tareaRet:=ActualProc_p
    FSI

    DEVOLVER tareaRet
FIN
```

5.2.3.8. AsignarResultado. Inicializar los objetos procesador

Esta función es prácticamente igual a la descrita en el punto 5.2.2.7 para el algoritmo espacio-temporal sin límite de procesadores. El único motivo por el que debe mencionarse aquí es para indicar que ahora los números de procesadores a los cuales debe asignarse cada tarea no se obtienen de la matriz *Tabla*, sino de las matrices *ProcCPU* y *ProcTAPI*, lo que permite que intervalos de una tarea puedan asignarse a diferentes procesadores. Aunque este algoritmo no saca provecho de esta característica debido a como se seleccionan los procesadores, puede ser aprovechada en nuevas versiones más completas.

5.3. Asignación espacial y planificación temporal considerando los costes de interrupción

5.3.1. Consideraciones generales

Tomando como base el algoritmo espacio-temporal con límite en el número de procesadores del apartado anterior se han diseñado nuevas versiones que consideran los costes temporales debidos a las interrupciones de las tareas. Esto es algo nuevo en relación al estado del arte sobre los algoritmos de planificación estáticos.

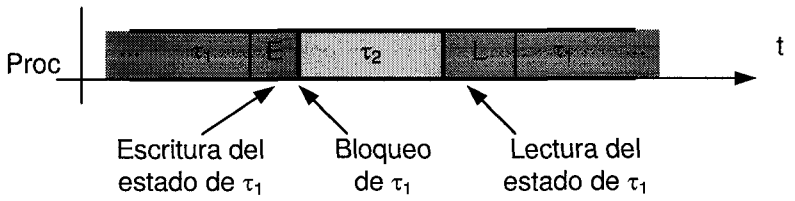


Figura 5-3. Ejemplo del proceso de interrupción y reanudación de una tarea.

Aunque todos los algoritmos de camino crítico expuestos hasta ahora consideran la posibilidad de interrumpir tareas, no tienen en cuenta que la interrupción conlleva unos costes temporales adicionales. Estos costes se deben a la necesidad de almacenamiento del estado de la tarea antes de ser bloqueada y a la recuperación del mismo cuando se reanuda la ejecución, tal y como muestra el ejemplo de la Figura 5-3. A estos dos costes se les denominará costes de escritura y lectura.

Los costes temporales de interrupción son al fin y al cabo valores que hay que sumar al tiempo de proceso de la tarea. Dicho de otro modo, si la tarea se interrumpe, se incrementa su duración, resultando evidente que los costes afectan a la planificación. Un algoritmo como los expuestos a continuación, que consideren esos costes, ofrecerán unos resultados mucho más reales que otros.

En general, los costes que implican la escritura y lectura del estado dependerán del instante del tiempo de computo de la tarea que se esté procesando. Como ejemplo considérese una tarea cuyo proceso consiste en buscar regiones en una imagen y modelarlas como conjuntos de características para devolver éstos. La interrupción de esa tarea tendrá asociados mayores costes conforme más avanzado esté su computo, ya que se dispone de más cantidad de datos generados que hay que guardar. Para modelar esto resulta necesario definir una función para cada tarea del sistema que devuelva sus costes de interrupción en función del tiempo de ejecución completado. Este modelado, aunque realista, resulta complejo y también complica el algoritmo de planificación.

Una aproximación más fácil de abordar, aunque menos realista, es considerar que los costes de escritura y lectura para una tarea son valores constantes. En este apartado se presentan técnicas que contemplan los dos tipos de modelos:

- Asignación espacial y planificación temporal con límite en el número de procesadores y teniendo en cuenta costes de interrupción constantes para cada tarea.
- Asignación espacial y planificación temporal con límite en el número de procesadores y teniendo en cuenta costes de interrupción en función del tiempo de computo de cada tarea.

5.3.1.1. Costes de interrupción de las tareas

Se considera que cada tarea del grafo G de entrada tiene asociados dos valores de coste temporal y una función discreta *Coste(i, t)*.

Los dos valores de coste temporal refieren a los costes de escritura y lectura por interrupción y se consideran constantes para cada tarea. Para acceder a estos valores se dispone de las funciones siguientes:

- *CosteIntEsc(i)*. Devuelve un valor entero correspondiente al coste de escritura de estado en unidades de tiempo necesario para una interrupción de la tarea τ_i.
- *CosteIntLec(i)*. Devuelve un valor entero correspondiente al coste de lectura de estado en unidades de tiempo necesario para reanudar la tarea τ_i.

Por otra parte, la función discreta *CosteInt(i, t)* devuelve un par *(cEsc, cLec)* con los valores de costes de escritura y lectura que implica una interrupción de la tarea τ_i en el instante de tiempo t relativo al tiempo de computo de la tarea. Es una función discreta ya que define valores de coste de forma no continua para diferentes intervalos dentro del tiempo de computo de la tarea para cada instante de tiempo $t \in Z$. La Figura 5-4 muestra un ejemplo de función de costes para una tarea.

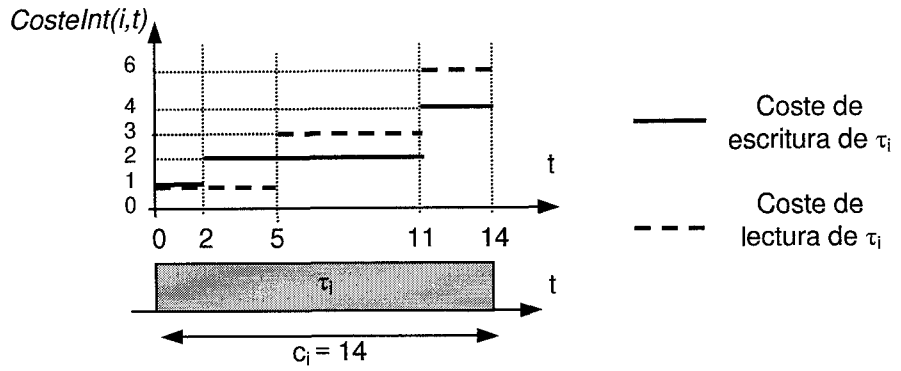


Figura 5-4. Ejemplo de función de costes.

La función de costes discreta se puede expresar conforme a los intervalos de tiempo que dan lugar a distintos costes. Para el ejemplo de la Figura 5-4 se tiene:

$$CosteInt(i,t) = \begin{cases} (1,1) & 0 \le t < 2 \\ (2,1) & 2 \le t < 5 \\ (2,3) & 5 \le t < 11 \\ (4,6) & 11 \le t < 14 \end{cases}$$

Y dado que se está considerando que las unidades de tiempo son valores enteros ($t \in Z$), la anterior función de costes también se puede escribir de otros modos:

$$CosteInt(i,t) = \begin{cases} (1,1) & 0 \le t \le 1 \\ (2,1) & 2 \le t \le 4 \\ (2,3) & 5 \le t \le 10 \\ (4,6) & 11 \le t \le 13 \end{cases} = \begin{cases} (1,1) & t = 0 \wedge t = 1 \\ (2,1) & 2 \le t \le 4 \\ (2,3) & 5 \le t \le 10 \\ (4,6) & 11 \le t \le 13 \end{cases}$$

La función de costes no está definida para $t=c_i$ ($t=14$ en el ejemplo), pero en ese instante finaliza la tarea y no tiene sentido una interrupción en él, como tampoco tiene sentido práctico una interrupción en el instante $t=0$.

5.3.2. Asignación espacial y planificación temporal con costes de interrupción constantes

Como primera aproximación, el algoritmo expuesto a continuación en este punto es una versión del espacio-temporal con límite de procesadores (ver apartado anterior, 5.2.3) que tiene en cuenta unos costes de escritura y lectura distintos y constantes para cada tarea, y dados como unidades de tiempo. Las modificaciones con respecto al algoritmo espacio-temporal con límite de procesadores son pocas, y sólo afectan a los procedimientos encargados de bloquear y planificar las tareas, tal y como se describe a continuación.

5.3.2.1. Inicializar. Definición de nuevas variables

Además de las variables requeridas por el algoritmo de asignación espacial y planificación temporal con límite de procesadores (ver punto 5.2.3.1) se definen las siguientes:

- $Bloqueada_{(N)}$. Vector que indica el estado de cada tarea con respecto a la situación de bloqueo. Recuérdese que una tarea bloqueada es una tarea interrumpida anteriormente y que sigue estando lista para continuar su ejecución. Los posibles valores de la componente correspondiente a una tarea τ_i pueden ser:

$$Bloqueada_i = \begin{cases} 0 & \text{Si } \tau_i \text{ no está bloqueada} \\ 1 & \text{Si } \tau_i \text{ está bloqueada} \\ 2 & \text{Si } \tau_i \text{ fué cancelada en un intento de bloqueo} \end{cases}$$

El valor 2 permite representar que la ejecución del anterior intervalo de una tarea se canceló en un intento de bloqueo debido a que era demasiado corto para poder incluir los costes de interrupción. Esta situación se explica mejor en el punto siguiente. Todas las componentes se inician con el valor 0.

- *UltCosteEsc*$_{(N)}$, *UltCosteLec*$_{(N)}$. Vectores con los valores de coste escritura y lectura para la última interrupción de cada tarea. Inicialmente se inician a 0.

5.3.2.2. Bloquear. Interrumpir la tarea que se ejecuta en un procesador

Esta función bloquear está basada en la función *Bloquear()* de los algoritmos espacio-temporales descritos anteriormente, pero ahora se tiene en cuenta que al bloquear una tarea se incurre en los costes de escritura y lectura de datos. Para ello, cuando se requiere bloquear la tarea de un procesador en el instante actual, se considera que el tiempo de escritura de estado debe estar incluido antes de ese instante. Además, el tiempo de lectura de estado debido a la interrupción debe incluirse en el siguiente intervalo de ejecución de la tarea.

Hay que destacar que la función opera de modo que cuando la tarea de un procesador debe ser bloqueada para el instante actual que marca la variable *tiempo* la tarea finaliza el intervalo actual en ese instante, y el coste de escritura de datos queda incluido en el intervalo. Esto es, se considera que el bloqueo marca la finalización del intervalo de ejecución, no el inicio de la escritura del estado de la tarea. Esto se puede hacer así porque la planificación es estática, previa a la ejecución, y con ello se consigue simplificar el modelado de las interrupciones de tareas.

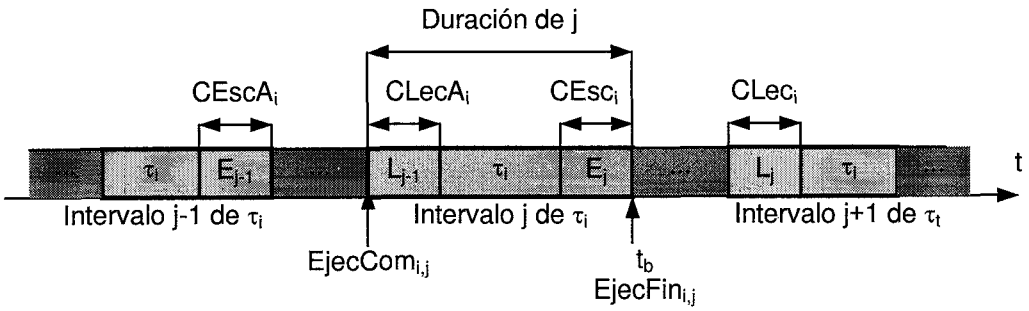


Figura 5-5. Esquema que representa la interrupción de una tarea.

Lo anterior se puede ver con mayor claridad en el ejemplo de la Figura 5-5, donde se representa el bloqueo de la tarea τ_i en el instante de tiempo t_b cuando se ejecuta su intervalo número j, así como los intervalos anterior y posterior a ese. El tiempo de escritura de estado

de la tarea ($CEsc_i$) queda incluido en el intervalo, antes del instante t_b. El tiempo de lectura correspondiente a la interrupción ($CLec_i$) se tiene en cuenta en el siguiente intervalo $j+1$. Además, al comienzo del intervalo interrumpido en t_b aparece el tiempo de lectura correspondiente a una interrupción de un intervalo anterior $j-1$ ($CLecA_i$). Recuérdese además que para el intervalo j, los valores $EjecCom_{i,j}$ y $EjecFin_{i,j}$ representan sus instantes de inicio y final respectivamente (ver punto 5.2.2.1).

Tanto en el ejemplo de la Figura 5-5 como en el algoritmo se diferencian los tiempos que implica la interrupción del intervalo actual ($CEsc_i$ y $CLec_i$) frente a los de la anterior ($CEscA_i$ y $CLecA_i$) para facilitar su comprensión y permitir nuevas mejoras como la consideración de una función de costes de interrupción. Sin embargo, hay que tener en cuenta que en este algoritmo los tiempos de escritura y lectura del estado se consideran constantes, y por lo tanto $CEsc_i=CEscA_i=CosteIntEsc(i)$ y $CLec_i=CLecA_i=CosteIntLec(i)$.

Otro aspecto interesante es que, de cara a la planificación, la ubicación de los tiempos de escritura (al final del intervalo) y de los de lectura (al principio del intervalo siguiente) no es relevante. Lo importante es que dichos tiempos sean incluidos en los tiempos de computo (Cp_i) y de retardo (Ret_i) de la tarea cuando corresponda. Dicho de otro modo, el algoritmo planificador tiene en cuenta los tiempos de la tarea, sin importar en que operaciones los invierte en cada instante.

Considerando todo lo anterior, a la hora de bloquear la tarea de un procesador, destacan tres situaciones según sea la duración del intervalo actual frente a los tiempos de lectura y escritura de datos en el mismo. Éstas están resumidas en la Figura 5-6, y son descritas a continuación.

Los primeros pasos que realiza la función *Bloquear()* son determinar la tarea que hay que bloquear (τ_i) y el número de intervalo actual (j) que se está ejecuando en el procesador p indicado. También es necesario obtener los costes de interrupción para la tarea τ_i en el intervalo j, esto es, el coste de lectura asociado a la anterior interrupción (que será cero con $j=1$) más los asociados a la actual. Además se calcula la duración del intervalo j, lo que se hace restando al instante actual de tiempo t_b el instante de comienzo del intervalo dado por $EjeCom_{i,j}$.

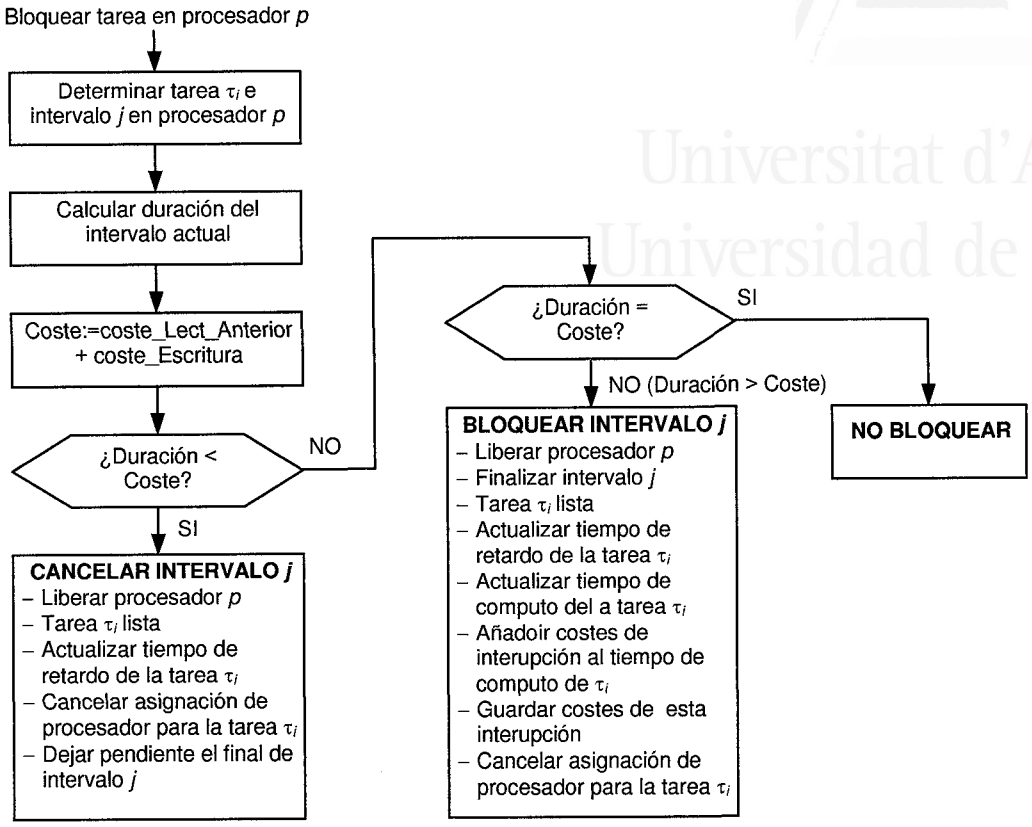


Figura 5-6. Procedimiento de bloqueo de la tarea en un procesador.

La primera situación se da cuando la duración el intervalo es inferior a la suma del coste de lectura que implicó la interrupción anterior más el de escritura que implica la actual. En tal situación, en el intervalo actual no hay tiempo para realizar la lectura y escritura de estado. Así, no es un intervalo válido y debe cancelarse, a no ser que la tarea τ_i sea seleccionada de nuevo para continuar ejecutándose en p dando lugar a un intervalo más largo. Por ello, en este caso, se bloquea la tarea, se actualiza su tiempo de retraso y se marca como lista, además de liberarse el procesador, pero no se actualiza el final del intervalo ni el tiempo de computo de la tarea a la espera de lo que ocurra en el futuro con ella cuando sea planificada. Para saber que la tarea τ_i ha quedado en esta situación se asigna el valor 2 a $Bloqueada_i$. También se cancela la asignación de procesador para que en una nueva elección de la tarea se busque de nuevo el procesador más adecuado con las funciones *SeleccionarCPU()* o *SeleccionarTAPI()* (ver punto 5.2.3.3).

Si la duración del intervalo es mayor que la suma de los costes, se tiene un intervalo válido donde además de ejecutar parte de la tarea da tiempo a realizar la lectura y escritura de estado. Es este caso se bloquea la tarea y se finaliza el intervalo actual, para en el futuro comenzar uno nuevo. Se actualizan los tiempos de computo y retraso de la tarea, se marca

ésta como tarea lista planificable y se libera el procesador. Además se suma el coste que implica la interrupción actual al tiempo de computo de la tarea, y se guardan los costes de escritura y lectura asociados. La tarea se marca como bloqueada asignando el valor 1 a $Bloqueada_i$. A igual que en la situación anterior, también se cancela la asignación de procesador.

La última situación se da cuando la duración del intervalo sólo permite ejecutar la lectura y escritura de estado, pero no parte de la propia tarea. En este caso no se realiza el bloqueo, dando opción a que la tarea comience a ejecutarse. Es el caso equivalente a una tarea que acaba de empezar considerado en las versiones anteriores de la función *Bloquear()*.

A continuación se expone el código equivalente:

```
FUNC Bloquear(EstadoProc, ActualProc, p)
   i:=ActualProc_p     # Tarea en p
   j:=EjeNumInt_i      # Intervalo de i

   t_b:=tiempo

   # Tiempo que lleva ejecutándose el intervalo
   duracion:=t_b-EjeCom_i,j

   # Obtiene costes de interrupción de la tarea
   costeEsc:=CosteIntEsc(i)
   costeLec:=CosteIntLec(i)

   # Si no queda tiempo para ejecutar la interrupción  el intervalo se cancela
   SI duracion < (costeEsc + UltCosteLec_i) ENTONCES
      EstadoProc_p:=LIBRE          # El procesador queda libre
      ActualProc_p:=-1
      F_i:=1                       # La tarea no ha acabado
      L_i:=1                       # La tarea está lista
      Ret_i:=Ret_i+duracion        # Actualiza tiempo de retardo

      Bloqueada_i:=2               # Marca tarea como bloqueada y cancelada

      # Cancela la asignación de procesadores a la tarea
      PruebaProcesadorCPU_i:=0
      PruebaProcesadorTAPI_i:=0
      SI Tabla_i,0>0 ENTONCES
         Tabla_i,0 = 0
      FSI
      SI Tabla_i,1>0 ENTONCES
         Tabla_i,1 = 0
      FSI

      resultado:=CIERTO            # Bloqueo efectuado

   # El intervalo tiene tiempo para ejecución e interrupción
   SINO SI duración > (costeEsc + UltCosteLec_i)
      EjecFin_i,j:=t_b             # Fin de intervalo actual
      EjeNumInt_i:=EjeNumInt_i+1   # Incrementa número de intervalos

      EstadoProc_p:=LIBRE          # El procesador queda libre
      ActualProc_p:=-1
      F_i:=1                       # La tarea no ha acabado
      L_i:=1                       # La tarea está lista
      Ret_i:=Ret_i+duracion        # Actualiza tiempo de retardo
      Cp_i:=Cp_i-duracion          # Actualiza Tiempo de computo

      Bloqueada_i=1                # Marca la tarea como bloqueada

      # Añade el tiempo de cómputo por interrupción y actualiza costes
```

```
        Cpi:=Cpi+costeEsc+costeLec
        UltCosteEsci:=costeEsc
        UltCosteLeci:=costeLec

        # Cancela la asignación de procesadores a la tarea
        PruebaProcesadorCPUi:=0
        PruebaProcesadorTAPIi:=0
        SI Tablai,0>0 ENTONCES
           Tablai,0 = 0
        FSI
        SI Tablai,1>0 ENTONCES
           Tablai,1 = 0
        FSI

        resultado:=CIERTO          # Bloqueo efectuado

     # El intervalo acaba de empezar en este instante
     # (duración = (costeEsc + UltCosteLeci))
     SINO
        resultado:=FALSO           # Bloqueo no efectuado
     FSI

     DEVOLVER resultado
FIN
```

5.3.2.3. PlanificarCPU, PlanificarTAPI y PlanificarCPUTAPI. Planificar una tarea

Como en los algoritmos espacio-temporales, se dispone de estas tres funciones para planificar una tarea de un tipo determinado en un procesador (o procesadores). Éstas son una nueva versión de las utilizadas en el algoritmo espacio-temporal con límite de procesadores descrito en 5.2.3 que incluyen operaciones relativas a la gestión de los costes de interrupción.

Con relación a la función *PlanificarCPU()*, dependiendo de si el procesador especificado por *numCPU* está libre o no, la tarea indicada τ_i se ejecutará según su estado de bloqueo, o se retardará siguiendo el mismo proceso descrito en el punto 5.2.3.5. Así, la novedad está en como se ejecuta la tarea cuando el procesador está libre, por lo que a continuación se describe mejor este caso, el cual está muy relacionado con la forma en que trabaja la función *Bloquear()* descrita en el punto anterior.

Si la tarea está bloqueada ($Bloqueada_i=1$) se examina cual es el procesador asignado al intervalo que fue bloqueado, esto es, el anterior al que va a comenzar en el instante actual ($ProcesadorCPU_{i,inter-1}$). Si coincide con el procesador asignado (*numCPU*), resulta que el intervalo actual se va a ejecutar a continuación del anterior, por lo que la interrupción anterior no resulta necesaria. Entonces se eliminan los costes de esa interrupción dados por $UltCosteEsc_i$ y $UltCosteLec_i$ del tiempo de computo la tarea τ_i. Aunque siguen existiendo dos intervalos, estos serán combinados en el mismo al final del procedimiento principal del algoritmo con *Compactar()*, función descrita en el algoritmo básico de camino critico en el apartado 3.3.10.

Si la tarea tiene asociado el valor $Bloqueada_i=2$ significa que el intervalo actual es uno que fue cancelado en un intento de bloqueo anterior por durar menos que los tiempos requeridos por la propia interrupción (ver punto anterior). En este caso se comprueba el procesador asignado a la tarea es mismo que el tenía cuando la cancelación. De ser así, la tarea puede reanudar el intervalo cancelado con la posibilidad de que ya puedan incluirse en el los costes de una posible futura interrupción. Por eso se deben eliminar los costes de la interrupción anterior del tiempos de retardo de la tarea τ_i. En cambio, si el procesador es otro, se cancela definitivamente el intervalo para empezar uno nuevo en el instante actual.

Existe un tercer caso, en el que la tarea τ_i no estaba bloqueada anteriormente. Esto ocurre cuando se ejecuta su primer intervalo. En esta situación, simplemente se inicia un nuevo intervalo de ejecución en el instante actual de tiempo.

Finalmente, para cualquiera de los tres casos descritos, se acaba marcando la tarea τ_i como no bloqueada, planificada y no lista, y el procesador *numCPU* como ocupado. Además se guarda la asignación del procesador al intervalo actual.

Considerando todo lo anterior, el nuevo código de *PlanificarCPU()* queda como sigue:

```
FUNC PlanificaCPU(i, numTareas, numCPU)
   # Intervalo actual de la tarea
   j:=EjecNumInteri

   # Si el procesador está libre
   SI EstadoCPUnumCPU=LIBRE ENTONCES

      # --- Añadido para los costes de interrupción

      # Tarea bloqueada en su anterior intervalo
      SI Bloqueadai=1 ENTONCES
         # El procesador del intervalo interrumpido es el actual
         SI ProcCPUi,j-1=numCPU ENTONCES
            # Elimina costes de interrupción del tiempo de computo y
            # continúa ejecución del intervalo
            Cpi:=CPi-UltCosteEsci-UltCosteLeci
         FSI
      # Tarea con anterior intervalo cancelado en una operación de bloqueo
      SINO SI Bloqueadai=2 ENTONCES
         # El procesador del intervalo cancelado es el actual
         SI ProcCPUi,j=numCPU ENTONCES
            # Elimina costes de interrupción del tiempo de retraso y
            # continúa ejecución del intervalo
            Reti:=Reti-UltCosteEsci-UltCosteLeci
         SINO
            # Actualiza intervalo cancelado haciendo que comience ahora
            EjecucionCOMi,j:=tiempo
         FSI
      # Tarea no bloqueada: comienza nuevo intervalo
      SINO
         EjecucionCOMi,j:=tiempo
      FSI

      Bloqueadai:=0  # La tarea ya no está bloqueada, está en ejecución

      # --- Fin de lo añadido

      EstadoCPUnumCPU:=EsInterrumpible(i) # Estado del procesador
```

```
        ActualCPU_numCPU:=i

        ProcCPU_i,j:=numCPU          # Datos sobre la asignación
        ProcTAPI_i,j:=-1
        Tabla_i,0:=numCPU

        F_i:=0 # Tarea planificada
        L_i:=0

    # Procesador ocupado
    SINO

        # --- Como antes
        ...

    FSI

    DEVOLVER numTareas
FIN
```

La función *PlanificarTAPI()* incluye los mismos añadidos, pero opera sobre las variables relativas a procesadores TAPI (*EstadoTAPI*, *ActualTAPI*, *ProcTAPI*, etc.).

Por último, hay que destacar que la función *PlanificarCPUTAPI()* no incluye modificaciones con respecto a la utilizada por el algoritmo espacio-temporal, ya que las tareas cpu/tapi no son interrumpibles y nunca serán bloqueadas.

5.3.3. Asignación espacial y planificación temporal con función de costes de interrupción

Este algoritmo supone una importante mejora con respecto al anterior ya que considera que los costes que implica una interrupción de una tarea dependen del instante de la misma que se esté ejecutando. Ésta es una aproximación mucho más real.

Para modelar la dependencia de los costes de interrupción frente al instante de ejecución se considera que cada tarea tiene asociada una función de costes de interrupción como la definida en el punto 5.3.1.1.

Básicamente este algoritmo funciona como el anterior; tiene en cuenta los costes de interrupción a la hora de bloquear las tareas. También se considera que dado el instante en que se debe bloquear el intervalo actual de la tarea presente en un procesador, el tiempo de escritura de estado queda dentro de ese intervalo. Esto es, el instante de bloqueo determina el instante en que la tarea sale del procesador, no cuando se inicia el proceso de interrupción.

Sin embargo se presenta un problema nuevo; dada una tarea y su función de costes, pueden existir más de una opción para interrumpir el intervalo actual de la tarea con respecto al instante en que se inicia la escritura de estado. Esto se puede comprender mejor con el

ejemplo de la Figura 5-7. En ella se representan tres opciones de interrupción del primer intervalo de ejecución de la tarea τ_i, considerando que ésta tiene la siguiente función de costes de interrupción:

$$CosteInt(i,t) = \begin{cases} (8,3) & 0 \le t < 6 \\ (4,3) & 6 \le t < 7 \\ (3,4) & 7 \le t < 8 \\ (2,2) & 8 \le t < 9 \\ (3,3) & 9 \le t < 10 \end{cases} = \begin{cases} (9,3) & 0 \le t \le 5 \\ (4,3) & t = 6 \\ (3,4) & t = 7 \\ (2,2) & t = 8 \\ (3,3) & t = 9 \end{cases}$$

Así, para la tarea τ_i se puede iniciar la escritura de estado del primer intervalo (E_1) en los instantes t_i=8, t_i=7 y t_i=6, ya que estos implican los costes de escritura 2, 3 y 4 respectivamente y para los tres casos el instante final en que es bloqueada es el deseado: t_b=10. Estas tres opciones corresponden a las gráficas A, B y C de la Figura 5-7. No resulta posible comenzar la escritura de estado en los instantes t_i={1, 2, 3, 4, 5, 9} ya que para estos el tiempo que implica la escritura de estado no cabe en el intervalo que debe finalizar en t_b=10.

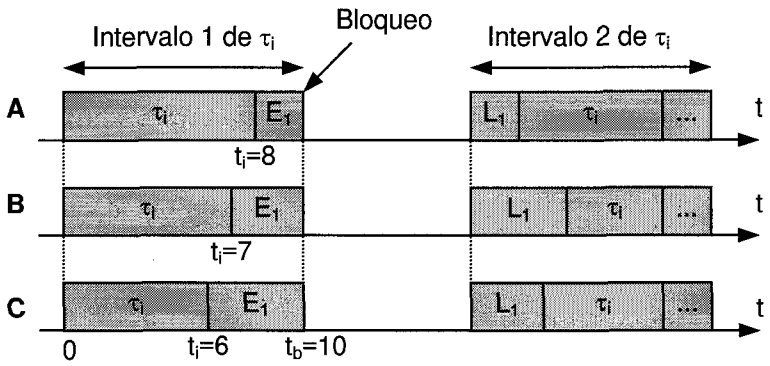


Figura 5-7. Ejemplo de distintos casos a los que da lugar una función de costes de interrupción.

También debe tenerse en cuenta que cada opción puede implicar un coste de lectura diferente (L_1) para el inicio del segundo intervalo, aunque esto no supone ningún problema.

Considérese ahora otro ejemplo con otra función de costes para τ_i:

$$CosteInt(i,t) = \begin{cases} (4,3) & 0 \le t < 5 \\ (5,4) & 5 \le t < 8 \\ (3,2) & 8 \le t < 10 \end{cases} = \begin{cases} (4,3) & 0 \le t \le 4 \\ (5,4) & 5 \le t \le 7 \\ (3,2) & t = 8 \wedge t = 9 \end{cases}$$

Si τ_i se quiere bloquear de nuevo en el instante t_b=10, resulta que ninguno de los instantes de inicio de escritura de estado t_i={1, 2, ...8} permite acabar el intervalo exactamente

en t_b. Además, para $t_i \geq 5$ se tiene que el coste de escritura no cabe en el intervalo. Sólo queda la opción de iniciar la escritura con $t_i \leq 4$, aunque esto supone que el intervalo acaba antes de t_b. Esta opción es válida, ya que al final se cumple el objetivo de que el procesador esté libre en el instante t_b. Para conseguir el mayor tiempo de ejecución efectivo de la tarea se puede tomar $t_i=4$, lo que supone un coste de escritura de 4, acabando el intervalo en el instante $t=8$. La figura Figura 5-8 refleja esta situación.

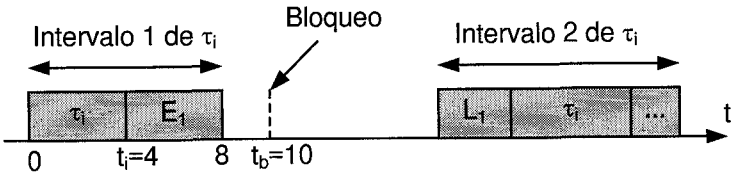


Figura 5-8. Ejemplo de intervalo que concluye antes del instante de bloqueo deseado.

En los dos ejemplos anteriores se consideraba el caso particular de interrumpir el primer intervalo de la tarea, que además comenzaba en $t=0$. En este caso, el instante de inicio de la escritura de estado t_i coincide con el tiempo de computo ya realizado de la tarea que sirve como parámetro para la función de costes de interrupción. Sin embargo, en un caso más general, resultará necesario determinar el tiempo efectivo computado de la tarea τ_i frente a su total c_i para determinar el valor de la función de costes. En ese tiempo efectivo no se tienen en cuenta los costes de escritura y lectura en los intervalos, ni el tiempo en que la tarea está bloqueada.

La Figura 5-9 muestra un ejemplo de una tarea interrumpida 2 veces, y la correspondencia entre sus intervalos de ejecución y su tiempo de computo efectivo, que es el que cuenta al aplicar la función de costes de interrupción.

Considerando los casos descritos, la conclusión a la que se llega es que cuando el algoritmo de planificación requiera realizar el bloqueo de una tarea τ_i en un instante t_b, debe hacer uso de la función *CostesInt(i,t_i)* para determinar el instante de inicio de interrupción t_i relativo al tiempo de computo de τ_i que garantice que la duración del intervalo que se crea sea menor o igual que el hueco de tiempo disponible hasta t_b, a la vez que se maximiza el tiempo dedicado a ejecutar la tarea dentro del intervalo.

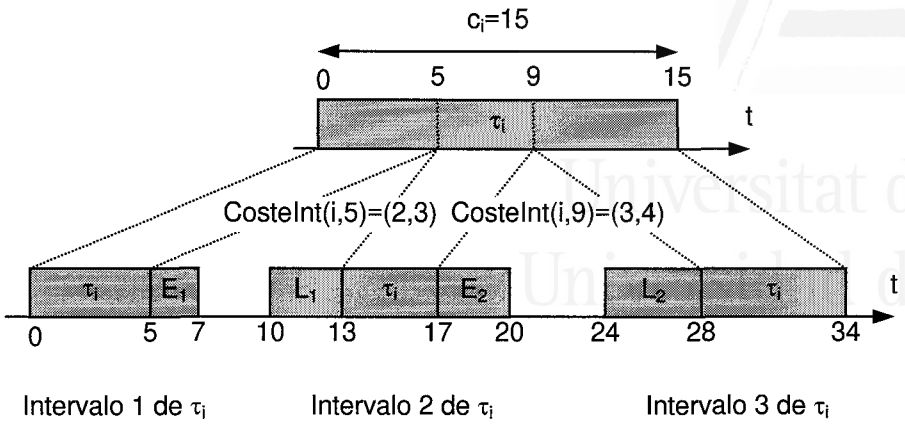


Figura 5-9. Ejemplo de tarea interrumpida dos veces.

5.3.3.1. Inicializar. Definición de nuevas variables

Se requiere definir el siguiente vector además de las variables contempladas en las versión anterior del algoritmo:

- $Computado_{dim}$. Almacena el tiempo de ejecución efectivo hasta el instante actual para cada tarea, esto es, el tiempo computado de la tarea sin contar los costes derivados de las interrupciones. Su utilidad es conocer el instante de ejecución de una tarea en relación a su tiempo de computo para poder aplicar la función de costes de interrupción en ese instante.

5.3.3.2. Bloquear. Interrumpir la tarea que se ejecuta en un procesador

Esta función opera básicamente de la misma forma que la función *Bloquear()* del algoritmo anterior, con costes de interrupción constantes (ver punto 5.3.2.2), permitiendo bloquear el intervalo actual de la tarea del procesador indicado en el instante actual, considerando que el tiempo de escritura de estado debe estar incluido antes de ese instante. También se consideran las tres situaciones descritas para el algoritmo anterior: que el intervalo deba ser cancelado por que no hay tiempo para ejecutar parte de la tarea, que sea bloqueado con éxito o que no se bloquee porque acaba de comenzar a ejecutarse la tarea.

Sin embargo, para considerar los costes de interrupción dependientes del tiempo de ejecución de la tarea, ahora deben tenerse en cuenta los aspectos descritos en el punto anterior. Esto es, los costes de escritura y lectura por la interrupción del intervalo deben obtenerse a partir de la función de costes, considerando de entre las posibles opciones el

instante de inicio de la escritura de estado que de lugar a un mayor tiempo de ejecución efectivo, y existiendo la posibilidad de que el intervalo deba acabar antes del instante actual.

Para buscar el instante de inicio de la escritura de estado se utiliza la función *ObtenerCostes()*, descrita en el siguiente punto. Ésta toma como parámetros el indicador de la tarea (*i*) y la duración máxima disponible para el intervalo (*duración*), y devuelve el tiempo de computo efectivo del intervalo sin considerar los tiempos de interrupción (*ejec*), así como los tiempos de escritura (*costeEsc*) y lectura (*costeLec*) asociados a la mejor opción de interrupción, en base a la función de costes de interrupción para la tarea τ_i.

El valor devuelto para *ejec* indica cuál de las tres situaciones de bloqueo tiene lugar. Con *ejec<0* se tiene que el intervalo debe cancelarse por que no hay tiempo para la ejecución de la tarea y su interrupción para ninguna opción de la función de costes de interrupción. Si *ejec>0* el intervalo es válido y se bloquea, siendo el tiempo de computo efectivo para la tarea el valor de *ejec*. Finalmente el caso *ejec=0* indica que el computo de la tarea acaba de empezar, existiendo tiempo para su interrupción, por lo que no interesa bloquear la tarea en el instante actual.

El código de la función, incluidas la modificaciones descritas, queda como sigue:

```
FUNC Bloquear(EstadoProc, ActualProc, p)
   i:=ActualProc_p      # Tarea en p
   j:=EjeNumInt_i       # Intervalo de i

   t_b:=tiempo

   # Tiempo que lleva ejecutándose el intervalo
   duracion:=t_b-EjeCom_i,j

   # Obtiene costes de interrupción de la tarea
   (ejec,costeEsc,costeLec):=ObtenerCostes(i, duración)

   # Si no queda tiempo para ejecutar la interrupción  el intervalo se cancela
   SI ejec<0 ENTONCES
      EstadoProc_p:=LIBRE          # El procesador queda libre
      ActualProc_p:=-1
      F_i:=1                       # La tarea no ha acabado
      L_i:=1                       # La tarea está lista
      Ret_i:=Ret_i+duracion        # Actualiza tiempo de retardo

      Bloqueada_i:=2               # Marca tarea como bloqueada y cancelada

      # Cancela la asignación de procesadores a la tarea
      PruebaProcesadorCPU_i:=0
      PruebaProcesadorTAPI_i:=0
      SI Tabla_i,0>0 ENTONCES
         Tabla_i,0 = 0
      FSI
      SI Tabla_i,1>0 ENTONCES
         Tabla_i,1 = 0
      FSI

      resultado:=CIERTO            # Bloqueo efectuado

   # El intervalo tiene tiempo para ejecución e interrupción
   SINO SI ejec>0 ENTONCES
```

```
        # Calcula la duración que al final tiene el intervalo
        durFinal:=UltCosteLec_i+ejec+costeEsc

        EjecFin_{i,j}:= EjecCom_{i,j}+durFinal    # Fin de intervalo actual
        EjeNumInt_i:=EjeNumInt_i+1                # Incrementa número de intervalos

        EstadoProc_p:=LIBRE          # El procesador queda libre
        ActualProc_p:=-1
        F_i:=1                       # La tarea no ha acabado
        L_i:=1                       # La tarea está lista
        Ret_i:=Ret_i+duracion        # Actualiza tiempo de retardo
        Cp_i:=Cp_i-durFinal          # Actualiza Tiempo de computo

        Bloqueada_i=1                # Marca la tarea como bloqueada

        # Añade el tiempo de cómputo por interrupción y actualiza costes
        Cp_i:=Cp_i+costeEsc+costeLec
        UltCosteEsc_i:=costeEsc
        UltCosteLec_i:=costeLec

        # Tiempo de ejecución efectivo
        Computado_i:=Computado_i+ejec

        # Cancela la asignación de procesadores a la tarea
        PruebaProcesadorCPU_i:=0
        PruebaProcesadorTAPI_i:=0
        SI Tabla_{i,0}>0 ENTONCES
           Tabla_{i,0} = 0
        FSI
        SI Tabla_{i,1}>0 ENTONCES
           Tabla_{i,1} = 0
        FSI

        resultado:=CIERTO            # Bloqueo efectuado

     # El intervalo tiene el tiempo para interrumpir y no más (ejec=0)
     SINO
        resultado:=FALSO             # Bloqueo no efectuado
     FSI

     DEVOLVER resultado
FIN
```

Un aspecto destacable es que en el caso de bloquear la tarea, ahora hay que determinar la duración y el instante en que acaba el intervalo, que pueden ser menores o iguales a los valores indicados por *duración* y t_b. Para ello se tiene en cuenta el valor de *ejec*, así como el tiempo de lectura para la interrupción anterior almacenado en $UltCosteLec_i$ y el tiempo de escritura de la interrupción actual dado por *costeEsc*.

```
durFinal:=UltCosteLec_i+ejec+costeEsc
EjecFin_{i,j}:= EjecCom_{i,j}+durFinal
```

El valor *durFinal* debe sumarse a $Computado_i$, que indica el tiempo de computo efectivo procesado para τ_i, además de descontarlo de Cp_i.

5.3.3.3. ObtenerCostes. Aplicar la función de costes de interrupción

Dada una tarea τ_i y una duración máxima para un intervalo de ejecución de la misma (*durMax*), esta función busca el instante de inicio de escritura de estado para la interrupción del intervalo que resulte válido según la función de costes, esto es, que de cabida a los

tiempos de lectura de una posible interrupción anterior y de escritura de la actual, y que permita un mayor tiempo de computo efectivo de la tarea. El resultado devuelto es el tiempo de ejecución final para el intervalo así como los costes de la nueva interrupción.

El algoritmo de la función se muestra a continuación. Consiste básicamente en probar para cada posible instante del inicio de tiempo de escritura (*ejec*) dentro del intervalo, desde el final de éste hasta su comienzo, si la suma del valor del tiempo de escritura (*esc*) dado por la función de costes para ese instante ($Computado_i$+*ejec*) más el tiempo de lectura de la anterior interrupción ($UltCosteLec_i$) caben en el intervalo actual, es decir, es menor a *durMax*. La búsqueda finaliza si se encuentra un instante de inicio válido, y ese instante será el que permita un tiempo de ejecución efectivo máximo ya que se comienza a buscar desde el fin del intervalo. También puede ocurrir que no sea posible interrumpir el intervalo dada la función de costes de interrupción para τ_i. Tal caso se da cuando no se encuentra ningún posible instante de inicio para la escritura, incluido el instante inicial del intervalo (*ejec* llega a –1).

```
(ejec,costeEsc, costelec)
FUNC ObtenerCostesInt(i, durMax)
   # Valores devueltos por defecto
   ejec:=-1
   costeEsc:=-1
   costeLec:=-1

   # Si en el intervalo cabe al menos el tiempo de lectura de la int. anterior
   SI durMax>costeLecAnt ENTONCES
      # Busca intervalo y costes de la función de costes de la tarea
      # que encajen en el instante actual con máximo tiempo de ejecución ejec
      ejec:=durMax
      encontrado:=FALSO
      MIENTRAS ejec≥0 Y NO encontrado HACER
         # Obtiene costes para instante de computo efectivo
         (esc,lec):=CosteInt(i,Computado_i+ejec)

         # Si esta es una opción válida (cabe la lectura, ejecución y escritura)
         SI UltCosteLec_i+ejec+esc<=durMax ENTONCES
            costeEsc:=esc
            costeLec:=lec
            encontrado:=CIERTO
         SINO
            ejec:=ejec-1
         FSI
      FMIENTRAS
   FSI

   DEVOLVER (ejec,costeEsc,costeLec)
FIN
```

Se pueden considerar tres casos para los valores del resultado:

- No se encuentra una opción de instante de inició de interrupción adecuado. La función devuelve (-1, -1, -1).
- Se encuentra una opción de instante de inició de interrupción válida. La función devuelve (*ejec, costeEsc, costeLec*), con *ejec>0* y

$$(costeEsc,costeLec) := CosteInt(iComputado_i+ejec)$$

- La interrupción puede efectuarse con *ejec=0*, esto es, en el instante de inicio del intervalo sin opción de ejecutar parte de τ_i. La función devuelve (*0, costeEsc, costeLec*), con

$$(costeEsc, costeLec) := CosteInt(i, Computado_i)$$

5.3.3.4. PlanificarCPU, PlanificarTAPI. Planificar una tarea

Las diferencias que presentan estas funciones con el algoritmo con costes de interrupción constantes (punto5.3.2.3) se presentan en el caso de que la tarea a planificar τ_i estaba bloqueada ($Bloqueada_i=1$) y el intervalo anterior se ejecutó en el procesador asignado a τ_i para el intervalo actual. Como antes, en esta situación, la interrupción anterior no resulta necesaria ya que la tarea sigue ejecutándose a continuación del intervalo anterior, y se debe descontar del tiempo de computo Cp_i los tiempos de escritura y lectura de estado asociados a esa interrupción.

Pero además, ahora hay que considerar otros dos nuevos aspectos. Por una parte, los tiempos de la interrupción anterior se dedican a ejecutar la tarea, por lo que hay que sumárselos al valor de $Computado_i$ para que refleje el verdadero tiempo efectivo procesado.

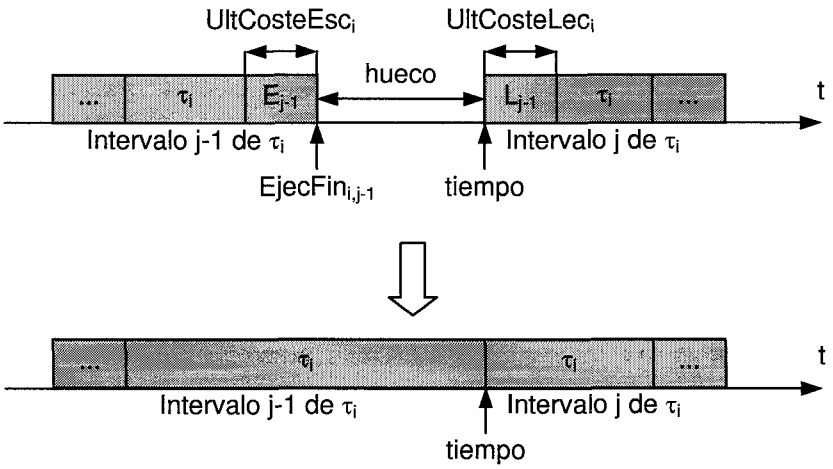


Figura 5-10. Ampliación del intervalo anterior hasta el actualmente planificado.

Por otra parte, es posible que el intervalo anterior hubiera acabado antes del instante de bloqueo deseado, que es el instante de tiempo actual, ya que, según el procedimiento principal, se planifica después de bloquear las tareas. En este caso, existe un hueco de tiempo entre el final del intervalo anterior y el que debe comenzar que puede aprovecharse para

ejecutar la tarea al cancelarse la última interrupción, tal y como refleja la Figura 5-10. Para aprovechar el hueco, su duración debe restarse del tiempo de computo de la tarea Cp_i y sumarse al valor de $Computado_i$ y el final del intervalo anterior debe actualizarse al valor del instante actual (valor *tiempo* en la figura).

Hay que destacar que en la situación descrita siempre quedará tiempo de computo restante de la tarea para completar el hueco hasta *tiempo* y continuar su ejecución, ya que si el intervalo anterior de τ_i se quiso bloquear en el instante *tiempo* (aunque finalmente pudo interrumpirse antes, en el valor dado por $EjecFin_{i,j-1}$) es por que en ese instante aún le quedaba tiempo de computo por procesar.

Considerando lo anterior, la modificación en la función *PlanificarCPU()* con respecto a la descrita en el punto 5.3.2.3 es la siguiente:

```
...
   # Tarea bloqueada en su anterior intervalo
   SI Bloqueada_i=1 ENTONCES
      # El procesador del intervalo interrumpido es el actual
      SI ProcCPU_{i,j-1}=numCPU ENTONCES
         # Hueco de tiempo entre intervalo anterior y actual
         hueco = tiempo - EjecFin_{i,j-1};

         # Amplia hueco anterior hasta instante actual
         EjecFin_{i,j-1}:=tiempo
         Cp_i:=Cp_i-hueco

         // Elimina costes de interrupción
         Cp_i:=Cp_i-UltCosteEsc_i-UltCosteLec_i

         // Tiempo computado efectivo
         Computado_i:=Computado_i+UltCosteEsc_i+UltCosteLec_i+hueco
      FSI
...
```

5.4. Asignación espacial y planificación temporal considerando los costes de comunicación

5.4.1. Consideraciones generales

A parte de las interrupciones y los costes que implican, otro aspecto importante que debería considerarse en la planificación de tareas en sistemas multiprocesador es la necesidad de comunicación de datos entre tareas relacionadas que se ejecutan en procesadores diferentes. Así, cuando dos tareas, una sucesora de otra, se asignan a dos procesadores

diferentes y la sucesora requiere datos generados por la predecesora, es necesario un intervalo de tiempo intermedio entre la ejecución de las tareas dedicado al intercambio de los datos. En ese intervalo los dos procesadores implicados deben estar disponibles simultáneamente.

La comunicación entre tareas es habitual en los sistemas multiprocesador, y puede repercutir en la calidad de la planificación resultante. Sin embargo no es fácil encontrar algoritmos de planificación estática que consideren este hecho, por lo que este apartado también supone una importante aportación.

5.4.1.1. Tipos de tareas

Considerando que se dispone de los dos tipos de procesadores tratados hasta el momento, CPUs y TAPIs, el grafo de tareas de entrada al planificador puede tener dos tipos de tareas básicos, tareas *cpu* y *tapi* según el tipo de procesador requerido para su ejecución. Por otra parte, las comunicaciones dan lugar a intervalos de tiempo que pueden ser considerados como tareas especiales que requieren dos procesadores simultáneamente. Éstas pueden ser de uno de estos tres tipos:

- Tareas *cpu/cpu*; necesarias para comunicar entre sí dos tareas tipo cpu ejecutadas en procesadores diferentes.
- Tareas *tapi/tapi*; necesarias para comunicar entre sí dos tareas tipo tapi ejecutadas en procesadores diferentes.
- Tareas *cpu/tapi*; necesarias para comunicar una tarea cpu con otra tapi o viceversa.

De los tres tipos, las tareas cpu/tapi son las únicas que son imprescindibles siempre, en el sentido de deben existir siempre que se disponga de una tarea cpu que precede a una tapi (o al revés) y existe la necesidad de comunicación entre ambas (que es lo habitual). En cambio, las tareas cpu/cpu y tapi/tapi sólo son necesarias cuando las tareas a comunicar se ejecutan en distintos procesadores. Por ello, en los algoritmos de planificación presentados hasta ahora se ha considerado la existencia de tareas cpu/tapi en el grafo de tareas a procesar.

Sin embargo, para las cpu/cpu y tapi/tapi, resulta más efectivo que sea el planificador el que decida crear las tareas de comunicación según se realiza la asignación espacial, en vez de ser especificadas en el grafo de entrada. En un caso general, el planificador también puede crear tareas cpu/tapi si resulta necesario y éstas no se especifican como entrada. Estas nuevas

tareas de comunicación deben ser insertadas en el grafo de tareas por el planificador, con la correspondientes modificaciones en las relaciones de precedencia (aristas).

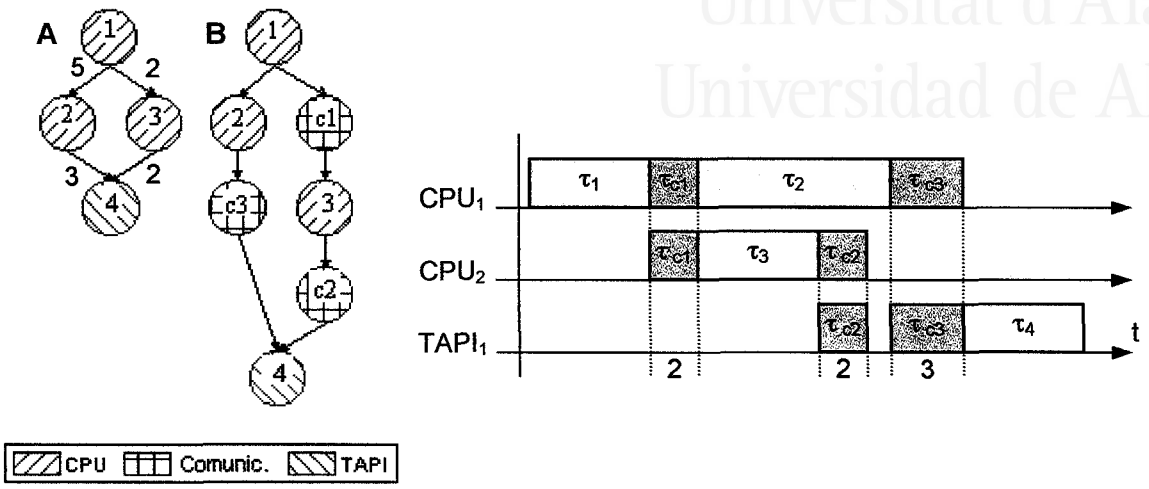


Figura 5-11. Ejemplo de tareas que representan los costes de comunicación.

Por ejemplo, el grafo *A* de la Figura 5-11 representa un grafo con tareas elementales de tipo cpu y tapi. Si las tareas de *A* se planifican sobre dos procesadores CPU y uno TAPI conforme muestra el diagrama temporal de la figura, serían necesarias tres comunicaciones de datos representadas por las nuevas tareas τ_{c1} a τ_{c3}. Estas tres tareas se relacionan con las elementales según muestra el grafo *B*. La tarea τ_{c1} corresponde a la comunicación entre dos CPUs (tarea cpu/cpu) y las τ_{c2} y τ_{c3} representan comunicaciones entre una CPU y una TAPI (tareas cpu/tapi).

En esta tesis se considera que las tareas de comunicación, de cualquiera de los tres tipos indicados, no son interrumpibles, ya que así es habitualmente.

La función *TipoTarea(i)* descrita en el punto 3.2.3 debe actualizarse para considerar los nuevos tipos de tareas que se pueden añadir al grafo, de forma que devuelva una de las siguientes constantes según el tipo de la tare τ_i: TCPU, TTAPI y TCPUTAPI, TCPUCPU y TTAPITAPI.

5.4.1.2. Costes de comunicación

Los costes temporales que representan las operaciones necesarias para la comunicación de datos entre dos tareas ejecutadas en diferentes procesadores se pueden modelar asociando un valor a la arista que las relaciona en el grafo de tareas. Así, se considera

que todas las aristas del grafo tienen asociado el coste de comunicación, aunque éste sólo debe ser tenido en cuenta cuando las tareas relacionadas se ejecutan en diferentes procesadores.

Continuando con el ejemplo anterior relativo a la Figura 5-11, los costes temporales asociados a las tareas de comunicación τ_{c1}, τ_{c2} y τ_{c3} son respectivamente 2, 2 y 3 unidades de tiempo.

En el algoritmo explicado en el siguiente punto se considera que para obtener el coste de comunicación entre tareas relacionadas está disponible de la función *CosteCom(i, j)*, que explora el grafo de tareas de entrada *G=(T,E)* en busca de una arista $(\tau_i,\tau_j, c_{i,j}) \in E$. Si la arista existe (τ_i precede a τ_j) devuelve $c_{i,j}$, que representa las unidades de tiempo correspondientes al coste de comunicación asociado. Si la arista no existe la función devuelve 0.

$$CosteCom(i,j) = \begin{cases} c_{ij} & \text{SI } \tau_i \; R_{prec} \; \tau_j \\ 0 & \text{En otro caso} \end{cases}$$

5.4.2. Asignación espacial y planificación temporal con costes de interrupción constantes

A continuación se describe un algoritmo de planificación estática basado en el de planificación espacio-temporal con límite en el número de procesadores descrito en el punto 5.2.3 (heredando por tanto la filosofía del algoritmo del camino crítico) que también considera los costes derivados de la comunicación entre tareas. Aunque se sigue considerando la interrupción de tareas para lograr una planificación resultado de mayor calidad, este nuevo algoritmo no considera los costes que las interrupciones implican [25]. Además, el algoritmo añade tareas de comunicación (cpu/cpu, tapi/tapi y cpu/tapi) si es necesario, aunque admite que el grafo de entrada tenga ya tareas cpu/tapi.

Para poder realizar la planificación teniendo en cuenta los costes de comunicación y los nuevos tipos de tareas, el nuevo algoritmo presenta cambios en la gestión de los pesos asociados a las tareas con respecto a los basados en el camino crítico (apartado 3.3) y la gestión de los intervalos de tiempo de las tareas y los procesadores asignados a estos se hace más compleja que la descrita en el algoritmo espacio-temporal con límite de procesadores.

Además, no utiliza el sistema de prueba de procesadores de este segundo algoritmo para no complicar innecesariamente la explicación en este apartado.

Aún así, el procedimiento principal mantiene la misma estructura que el algoritmo espacio-temporal con límite de procesadores. Las diferencias son que ahora se utiliza una función para calcular los pesos de las tareas antes de escoger la tarea a planificar, y existe una única función para buscar los procesadores donde ejecutar la tarea, que además devuelve cuatro identificadores de procesador, dos de CPU y dos de TAPI, tal y como se puede ver a continuación.

```
PROC Espacio_Temporal_Comunicaciones
   Inicializar()
   numTareas:=TareasListas2()
   MIENTRAS numTareas > 0 HACER
      MIENTRAS numTareas > 0 HACER
         CalcularPesos()
         tareaLista:=SeleccionaTareaLista()
         numTareas:=numTareas - 1

         (nCPU1,nTAPI1,nCPU2,nTAPI2):=SelecionarProcesadores(tareaLista)

         numTareas:=Planificar(tareaLista,numTareas, nCPU1,nTAPI1,nCPU2,nTAPI2)
      FMIENTRAS
      CalculaMatrices()
      numTareas:=TareasListas()
   FMIENTRAS
   AsignarResultado()
   Compactar()
FIN
```

El hecho de emplear cuatro identificadores de procesador se debe a que ahora existen los nuevos tipos de tareas cpu/cpu y tapi/tapi (además de cpu, tapi y cpu/tapi) que requieren una pareja de procesadores CPU y de TAPIs respectivamente, las cuales no pueden ser referenciadas con sólo una variable para la CPU y otra para la TAPI como se hacía antes. El significado de estos identificadores se describirá mejor en los puntos dedicados a las funciones *SeleccionarProcesadores()* y *Planificar()* (puntos 5.4.2.3 y 5.4.2.6).

La función *Compactar()* es la utilizada por las anteriores versiones de algoritmo de camino crítico, y que fue descrita en punto 3.3.10. Tampoco cambia *TareasListas2()* con respecto a la del algoritmo espacio-temporal con límite de procesadores (ver punto 5.2.3.2). El resto de funciones presentan diferencias respecto a las ya estudiadas por lo que se explican de nuevo de forma resumida, centrando el interés en los aspectos relativos a los costes de comunicación.

Hay que destacar dos aspectos en este nuevo algoritmo. Primero, no se utiliza ya la matriz *Tabla* donde se reflejaba la asignación de las tareas a los procesadores (ver punto 3.2.4), debido a que existen tareas cuyos procesadores no pueden expresarse con esta matriz

(las de tipo cpu/cpu y tapi/tapi), al disponer sólo de dos columnas destinadas a referenciar una CPU y una TAPI.

En segundo lugar, este algoritmo también se basa en la idea de camino-crítico, asignándose unos pesos a las tareas para representar el mínimo tiempo de ejecución necesario para ejecutar completamente cada tarea y sus sucesoras suponiendo el máximo paralelismo posible en la ejecución de todas las tareas. En base a esos pesos se efectúa la elección de la tarea a ejecutar. Sin embargo, frente al algoritmo del camino crítico descrito en 3.3 donde se definían dos valores de peso por tarea, uno relativo a procesadores CPU y otro para TAPI, ahora se utiliza un valor de peso por cada procesador (todas las CPUs y TAPIs) para cada tarea. Esto permite considerar en los pesos, no sólo los tiempos de computo, si no además los de comunicación entre parejas de procesadores, para que sean tenidos en cuenta en la elección de tareas a ejecutar y en la elección del procesador adecuado para ella.

5.4.2.1. Iniciar. Definición e iniciación de las de variables utilizadas

En este algoritmo se utilizan las variables definidas para el de camino crítico básico (punto 3.3.1), exceptuando las relativas a los pesos de las tareas, más las siguientes:

- $EjecNumInt_{(N)}$. Vector con el número de intervalos de tiempo en que se divide la ejecución de cada tarea debido a las interrupciones. Se inicia todo con el valor 1.
- $EjecNumIntAct_{(N)}$. Vector de valores booleanos que permite conocer si se ha actualizado la componente correspondiente del contador de intervalos *EjecNumInt*. Resulta necesario para las tareas cpu/cpu y tapi/tapi. Se inicia con todas sus componentes a FALSO.
- $EjecCom_{(N \times MAX\_INTER)}$ y $EjecFin_{(N \times MAX\_INTER)}$. Estas matrices contienen los instantes de comienzo y de final de cada intervalo (columna) para cada tarea (fila). Todas las componentes se inician a -1 para representar que no hay intervalos.
- $ActualCPU_{(numeroCPUs)}$ y $ActualTAPI_{(numeroTAPIs)}$. Vectores que almacenan las tareas que hay actualmente en cada procesador CPU o TAPI. Todas las componentes de los vectores se inician con el valor -1 para representar que no hay tareas en los procesadores.
- $EstadoCPU_{(numeroCPUs)}$ y $EstadoTAPI_{(numeroTAPIs)}$. Estos vectores indican el estado actual de cada procesador. El valor de cada componente puede ser *LIBRE*, *INT*

(ejecución interrumpible), o *NOINT* (ejecución no interrumpible). Se inician todas sus componentes con el valor *LIBRE*.

- $ProcCPU1_{(N \times MAX\_INTER)}$, $ProcTAPI1_{(N \times MAX\_INTER)}$, $ProcCPU2_{(N \times MAX\_INTER)}$, $ProcTAPI2_{(N \times MAX\_INTER)}$. Estas matrices permiten almacenar la asignación espacial que el algoritmo va obteniendo. Para cada tarea (fila) y intervalo de la tarea (columna) las matrices indican el procesador o procesadores asignados al intervalo. Todas las componentes se inician con el valor -1 para representar que no hay asignación inicial.

- $UltProcCPU_{(N)}$, $UltProcTAPI_{(N)}$. Estos vectores contienen los procesadores asignados CPU y TAPI al último intervalo de cada tarea. Para tareas las de comunicación cpu/cpu y tapi/tapi indican el procesador destino de la comunicación. Se inician con el valor –1.

- $Bloqueadas_{(N)}$. Para cada tarea τ_i la componente $Bloqueadas_i$ vale CIERTO si la tarea ha sido bloqueada alguna vez y FALSO en caso contrario. Se inician todos sus valores a FALSO.

- $ProcPrecCPU_{(N \times N)}$, $ProcPrecTAPI_{(N \times N)}$. Almacenan los procesadores asignados al último intervalo de todas las tareas que son predecesoras de una dada. Por ejemplo, $ProcPrecCPU_{i,j}$ indica el número de procesador CPU asignado al último intervalo de τ_j, predecesora de τ_i. Esas matrices permiten localizar rápidamente los procesadores asignados a todas las tareas predecesoras de una dada. Se inician todas las componentes a -1.

- $PesosAcCPU_{(N \times numeroCPUs)}$, $PesosAcTAPI_{(N \times numeroTAPIs)}$. Los valores de estas matrices se inician, para cada tarea, con el mínimo tiempo de ejecución necesario en cada uno de los distintos procesadores CPU y TAPI para ejecutar completamente todas las tareas sucesoras suponiendo el máximo paralelismo posible en la ejecución de todas las tareas. Esto se calcula con las siguientes operaciones ejecutadas para $i:=N, N-1, \ldots 0$:

$$PesoAcum_{i,p} := \begin{cases} \underset{j:=i+1...N}{MAX}\left(C_{i,j} \cdot PesoTempCPU_j\right) & i < N \\ 0 & i = N \end{cases} \quad \forall p := 1...numeroCPUs$$

$$PesoAcum_{i,p} := \begin{cases} \underset{j:=i+1...N}{MAX}\left(C_{i,j} \cdot PesoTempTAPI_j\right) & i < N \\ 0 & i = N \end{cases} \quad \forall p := 1...numeroTAPIs$$

$$PesoTempCPU_i := \begin{cases} PesoAcum_{i,1} + Cp_i & \text{SI } TipoTarea(i) \in \{TCPU, TCPUTAPI\} \\ PesoAcum_{i,1} & \text{En otro caso} \end{cases}$$

$$PesoTempTAPI_i := \begin{cases} PesoAcum_{i,2} + Cp_i & \text{SI } TipoTarea(i) \in \{TTAPI, TCPUTAPI\} \\ PesoAcum_{i,2} & \text{En otro caso} \end{cases}$$

Durante la ejecución del algoritmo, los valores se actualizarán al acabar una tarea sumando los costes de comunicación en los que se incurre cuando las tareas sucesoras deben ejecutarse en procesadores distintos.

- $PM\_CPU_{(N \times numeroCPUs)}$, $PM\_TAPI_{(N \times numeroCPUs)}$. Representan los pesos de cada tarea calculados en cada paso del algoritmo a partir de los pesos acumulados, para tener también en cuenta que ciertas tareas no deben ejecutarse en determinados procesadores. Esto se indica con el valor de peso no válido –INFINITO en la componente correspondiente. Estos valores se calculan en la función CalcularPesos().
- *CTE_PESO*. Valor constante de peso de magnitud mayor en relación a los valores que las tareas pueden tener como costes de comunicación o computo. Se utiliza para destacar el peso de una componente de peso frente a otras.

Un aspecto importante a tener en cuenta es que con este algoritmo el número de tareas (dado por *N*) puede incrementarse en caso de ser necesario añadir tareas de comunicación. En consecuencia, las matrices o vectores cuyo tamaño estén en función de ese valor pueden crecer, añadiéndose nuevas filas o columnas tras las últimas. Por otra parte, la constante MAX_INTER representa el número máximo de intervalos que puede tener una tarea.

La utilización de las cuatro matrices *ProcCPU1, ProcTAPI1, ProcCPU2, ProcTAPI2* facilita la representación de la asignación espacial de las tareas a los procesadores teniendo en cuenta los nuevos tipos de tareas. Para mostrar mejor su utilización la siguiente tabla refleja los posibles valores de las matrices para un intervalo *j* de una tarea τ_i considerando distintos tipos de tareas:

Tipo de tarea	Procesadores asignados	Valor de $ProcCPU1_{i,j}$	Valor de $ProcCPU2_{i,j}$	Valor de $ProcTAPI1_{i,j}$	Valor de $ProcTAP2_{i,j}$
cpu	CPU p	p	-1	-1	-1
tapi	TAPI p	-1	-1	p	-1
cpu/cpu	CPU p y CPU q	p	q	-1	-1
tapi/tapi	TAPI p y TAPI q	-1	-1	p	q
cpu/tapi	CPU p y TAPI q	p	-1	q	-1

5.4.2.2. ActualizaTiempo. Determinar el siguiente evento

Como en algoritmos anteriores basados en el camino crítico, esta función se encarga de actualizar el instante de tiempo actual al instante indicado, que corresponderá al de la siguiente tarea a planificar. Para ello se finalizan todas las tareas que se están ejecutando en los procesadores y acaban antes de dicho instante, y se actualiza el estado de los procesadores. Esto se hace primero para todos los procesadores CPU y luego para los TAPI.

Además, para las tareas de comunicación hay que tener en cuenta que se ejecutan en dos procesadores. Así al finalizar tareas cpu/cpu o tapi/tapi debe controlarse que sólo se incremente el contador de intervalos $EjecNumInt_i$ de la tarea una vez utilizando los valores de $EjecNumIntAct_i$. Para tareas cpu/tapi, el contador de intervalos se actualiza al explorar las TAPIs.

Otro aspecto nuevo de la función es que, al acabar una tarea τ_i que se ha ejecutado en el procesador número p, se actualizan los pesos acumulados de todas su sucesoras τ_j, de forma que se incrementan todos los valores de peso $PesosAcCPU_{j,q}$ para todos los procesadores q distintos de p con el valor de coste de comunicación entre τ_i y τ_j. Así son tenidos en cuenta los costes de comunicación. Finalmente, también se guarda el procesador p asignado a cada tarea τ_i que finaliza para que sea conocido por todas sus sucesoras τ_j en la componente $ProcPrecCPU_{j,i}$.

```
PROC ActualizaTiempo(inicio)
   HACER
      SI inicio<0 ENTONCES
         inicio:=INFINITO
      FSI

      # Buscar tiempo final más temprano en las CPUs y TAPIs
      fin:=INFINITO
      PARA p:=1 HASTA numeroCPUs HACER
         i:=ActualCPUp
         SI tarea>0 Y fin>Fini ENTONCES fin:=Fini
      FPARA
      PARA p:=0 HASTA numeroTAPIs HACER
         i:=ActualTAPIp
         SI tarea>0 Y fin>Fini ENTONCES fin:=Fini
      FPARA

      porAcabar:=0
```

```
# Si la tarea lista empieza antes de que otras acaben
SI inicio<fin ENTONCES tiempo:=inicio

# Si hay tareas que acaban antes de que empiece la lista
SINO
   tiempo:=fin
   PARA p:=1 HASTA numeroCPUs HACER
      i:=ActualCPU_p
      SI i>0 ENTONCES
         porAcabar:=porAcabar + 1
         SI fin=Fin_tarea ENTONCES
            j:=EjecNumInt_i        # Actualiza fin del intervalo
            EjeFin_i,j:=tiempo
            ActualCPU_p:=-1        # Actualiza estado del procesaodos
            EstadoCPU_p:=LIBRE
            porAcabar:=porAcabar - 1

            # Si la tarea es cpu cambia al siguiente intervalo
            SI TipoTarea(i)=TCPU ENTONCES
               EjecNumInt_i:=EjecNumInt_i+1
            # Si la tarea es cpu/cpu cambia al siguiente intervalo
            SINO SI TipoTarea(i)=TCPUCPU ENTONCES
               # Si no actualizó contador, lo actualiza
               SI EjecNumIntAct_i ENTONCES
                  EjecNumIntAct_i:=FALSE
               SINO
                  EjecNumInt_i:=EjecNumInt_i+1
                  EjecNumIntAct_i:=TRUE
               FIN
            FSI
            # Si es cpu/tapi debe esperar a analizar la TAPI

            # Actualizar pesos de las sucesoras
            PARA j:=1 HASTA N HACER
               SI T_i,j ENTONCES
                  # Incrementa pesos de sucesoras que se ejecutan en otra CPU
                  PARA q:=1 HASTA numeroCPUs HACER
                     SI p<>q ENT PesosAcCPU_j,q:=PesosAcCPU_j,q+CosteCom(i,j)
                  FPARA
                  # Incrementa pesos de todas sucesoras tapi
                  PARA q:=1 HASTA numeroTAPIs HACER
                     PesosAcTAPI_j,q:=PesosAcTAPI_j,q+CosteCom(i,j)
                  FPARA
                  # Guarda procesador de tarea para su sucesora
                  ProcPrecCPU_j,i:=p
               FSI
            FPARA
         FSI
      FSI
   FPARA
   PARA p:=0 HASTA numeroTAPIs HACER
      i:=ActualTAPI_p
      SI i>0 ENTONCES
         porAcabar:=porAcabar + 1
         SI fin=Fin_i ENTONCES
            j:=EjecNumInt_i    # Actualiza fin del intervalo
            EjeFin_i,j:=tiempo
            ActualTAPI_p:=-1       # Actualiza estado del procesaodos
            EstadoTAPI_p:=LIBRE
            porAcabar:=porAcabar - 1

            # Si la tarea es tapi/tapi cambia al siguiente intervalo
            SI TipoTarea(i)=TTAPITAPI ENTONCES
               # Si no actualizó contador, lo actualiza
               SI EjecNumIntAct_i ENTONCES
                  EjecNumIntAct_i:=FALSE
               SINO
                  EjecNumInt_i:=EjecNumInt_i+1
                  EjecNumIntAct_i:=TRUE
               FIN
            # Si la tarea es tapi o cpu/tapi cambia al siguiente intervalo
            SINO
               EjecNumInt_i:=EjecNumInt_i+1
            FSI
```

```
                    # Actualizar pesos de las sucesoras
                    PARA j:=1 HASTA N HACER
                       SI T_{i,j} ENTONCES
                          # Incrementa pesos de sucesoras que ejecutan en otra TAPI
                          PARA q:=1 HASTA numeroTAPIs HACER
                             SI p<>q ENT PesosAcTAPI_{j,q}:=PesosAcTAPI_{j,q}+CosteCom(i,j)
                          FPARA
                          # Incrementa pesos de todas sucesoras cpu
                          PARA q:=1 HASTA numeroCPUs HACER
                             PesosAcCPU_{j,q}:=PesosAcCPU_{j,q}+CosteCom(i,j)
                          FPARA
                          # Guarda procesador de tarea para su sucesora
                          ProcPrecTAPI_{j,i}:=p
                       FSI
                    FPARA
                 FSI
              FSI
           FPARA
        FSI
     MIENTRAS porAcabar>0 Y (inicio:=INFINITO O fin<inicio))
  FIN
```

5.4.2.3. CalcularPesos. Calcula los pesos de cada tarea

A partir de los pesos acumulados de las tareas, esta función determina los pesos que deben utilizarse para la selección de la siguiente tarea. Básicamente se trata de asignar un valor de peso no válido a ciertas tareas con respecto a procesadores donde no interesa ejecutarlas, y para el resto de pesos sumar el coste de ejecutar cada tarea al de los pesos acumulados, que marcan el mínimo coste de ejecutar sus sucesoras con el máximo paralelismo.

Lo anterior se realiza asignando primero los valores de pesos acumulados dados por *PesosAcCPU* y *PesosAcTAPI* a *PM_CPU* y *PM_TAPI*. Luego, para cada tarea τ_i y para cada procesador número p (primero CPUs y luego TAPIs) se comprueba si la tarea está lista para planificarse y si el procesador está libre. Si ambas condiciones se cumplen se suma el coste de ejecución de τ_i a la componente de peso $PM\_CPU_{i,p}$. En otro caso, se asigna un valor de peso no válido (-INFINITO) a $PM\_CPU_{i,p}$ para que en la elección de tarea en *SeleccionarTareaLista()* sea descartada la ejecución de τ_i en p.

Además, el algoritmo también considera la existencia de posibles tareas cpu/tapi en el grafo de entrada. Para estas tareas hay que garantizar que el procesador origen de la comunicación es el mismo que el de la tarea predecesora. Esto se consigue asignando en esta función un valor de peso no válido (-INFINITO) a las componentes $PM\_CPU_{i,q}$ para las que τ_i es una tarea cpu/tapi y q indica un procesador distinto al de la tarea predecesora a τ_i.

```
PROC CalcularModificados()
   # Inicia pesos con los pesos acumulados
   PM_CPU:=PesosAcCPU
   PM_TAPI:=PesosAcTAPI
```

```
        # Actualiza sus pesos para cada tarea
        PARA i:=1 HASTA N HACER

           # Comprueba pesos para cada CPU
           PARA p:=1 HASTA numeroCPUs HACER
              # Si la tarea no está lista: Peso no válido
              SI Li=0 ENTONCES
                 PM_CPUi,p:=-INFINITO
              # Si el procesador no está libre: Peso no válido
              SINO SI ActualCPUp>0 ENTONCES
                 PM_CPUi,p:=-INFINITO
              # Procesador libre y tarea lista: Actualiza peso
              SINO
                 PM_CPUi,p:=PM_CPUi,p+Cpi
              FSI
           FPARA

           # Comprueba pesos para cada TAPI
           PARA p:=1 HASTA numeroTAPIs HACER
              # Si la tarea no esta lista: Peso no válido
              SI Li=0 ENTONCES
                 PM_TAPIi,p:=-INFINITO
              # Si el procesador no está libre: Peso no válido
              SINO SI ActualCPUp>0 ENTONCES
                 PM_TAPIi,p:=-INFINITO
              # Procesador libre y tarea lista: Actualiza peso
              SINO
                 PM_TAPIi,p:=PM_TAPIi,p+Cpi
              FSI
           FPARA

           # Asegura que las posibles tareas cpu/tapi (tapi/cpu)del grafo de entrada
           # se ejecutan en las CPU (TAPI) de sus tareas predecesoras
           SI TipoTarea(i)=TCPUTAPI ENTONCES
              # Busca la tarea predecesora j ejecutándose en CPU (j es cpu/tapi)
              encontrada:=FALSO
              PARA j:=1 HASTA N HACER
                 SI Cj,i Y UltProcCPUj>0 ENTONCES encontrada=CIERTO
              FPARA

              # Para todas las CPU que no son la de la predecesora se invalida pesos
              # para asegurar que la cpu/tapi se ejecuta en la cpu de la predecesora
              SI encontrada ENTONCES
                 PARA q:=1 HASTA numeroCPUs HACER
                    SI q<>UltProcCPUj ENTONCES PM_CPUq:=-INFINITO
                 FPARA
              FSI

              # Busca la tarea predecesora j ejecutándose en TAPI (j es tapi/cpu)
              encontrada:=FALSO
              PARA j:=1 HASTA N HACER
                 SI Cj,i Y UltProcTAPIj>0 ENTONCES encontrada=CIERTO
              FPARA

              # Para todas las TAPI que no son la de la predecesora se invalida pesos
              # para asegurar que la tapi/cpu se ejecuta en la tapi de la predecesora
              SI encontrada ENTONCES
                 PARA q:=1 HASTA numeroTAPIs HACER
                    SI q<>UltProcTAPIj ENTONCES PM_TAPIq:=-INFINITO
                 FPARA
              FSI
           FSI
        FPARA
     FIN
```

5.4.2.4. SeleccionarTareaLista. Selección de la tarea a planificar

Básicamente, la función realiza el mismo trabajo que la descrita en el punto 3.3.3 y utilizada por los anteriores algoritmos de camino crítico. La novedad está en que ahora se dispone de un peso por tarea y procesador, en vez de uno por tarea y tipo de procesador.

En primer lugar se crea un vector que indica que tareas que no tienen otras sucesoras. Utilizando este vector se busca primero una tarea final que tiene un mayor valor de tiempo de finalización. A ésta se la llamará tarea crítica. De esto se encargan las funciones *Crear_Vector_Final()* y *Buscar_Tarea_Crictica()* descritas en el punto 3.3.3. Después, se busca la tarea τ_i lista (L_i=*1*) que es antecesora de la tarea crítica y posee un peso total máximo (es la que retarda más el sistema), comparando indistintamente este máximo con el peso de la tarea τ_i en relación a procesadores CPU o TAPI.

Los pesos por tipo de procesador CPU y TAPI para una tarea τ_i se determinan buscando el mínimo valor entre todos los pesos $PM\_CPU_{i,p}$ para todas las CPU p, y el mínimo valor entre todos los pesos $PM\_TAPI_{i,p}$ para todas las TAPI. En este proceso se descartan las componentes que tienen un valor de peso no válido. Se escoge el procesador con mínimo peso porque será el que de lugar a menores costes de comunicación. Además, ese será el procesador que se seleccione posteriormente para la tarea usando el mismo criterio en la función *SeleccionarProcesadores()*.

En caso de que varias tareas tengan el mismo peso para un tipo de procesador (por ejemplo CPU), se escogerá la que tenga además menor peso para el otro tipo (TAPI), ya que ésta liberará antes los recursos. Este proceso se repite hasta que se consigue escoger una tarea o hasta que no queden más tareas finales.

Si con el proceso anterior no se ha escogido una tarea a planificar, entonces se busca entre todas las tareas finales que están listas la que tiene un mayor peso. Para ellos se vuelve a tener en cuenta todas las tareas finales y el peso máximo establecido anteriormente.

La función devuelve la tarea seleccionada.

```
FUNCION SeleccionaTareaLista(PM_CPU, PM_TAPI)

   # Determinar cuales son las tareas finales (sin sucesoras)
   (numFin,Final):=Crear_Vector_Final()

   seleccion:=-1
   pesoMax:=0
   HACER
      # Buscar tarea final con mayor tiempo de finalización
      tc:=Buscar_Tarea_Crictica(Final)

      # Una tarea final menos tras seleccionar
      numFin:=numFin-1
      Final_tc:=0

      # Para cada tarea lista que es antecesora de la tarea crítica
      PARA i:=1 HASTA N HACER
         SI L_i=1 Y T_i,tc=1 ENTONCES

            # Busca menor peso válido para todas las CPU y TAPI respecto a i
            minCPU:=INFINITO
```

```
            PARA p:=1 HASTA numeroCPUs HACER
               SI PM_CPU_{i,p} >0 Y PM_CPU_{i,p}<minCPU ENTONCES minCPU:=PM_CPU_{i,p}
            FPARA
            minTAPI:=INFINITO
            PARA p:=1 HASTA numeroTAPIs HACER
               SI PM_TAPI_{i,p}>0 Y PM_TAPI_{i,p}<minTAPI ENTONCES minTAPI:=PM_TAPI_{i,p}
            FPARA

            # Comprueba si la tarea lista i tiene mayor peso mínimo CPU y TAPI
            # y actualiza máximos para CPU, TAPI y general
            SI minCPU>pesoMax O minTAPI>pesoMax ENTONCES
               pesoMax:=MAXIMO(minCPU, minTAPI)
               pesoMaxC:=minCPU
               pesoMaxT:=minTAPI
               seleccion:=i
            SINO SI minCPU=pesoMax Y minTAPI<pesoMaxT
               pesoMax:=MAXIMO(minCPU, minTAPI)
               pesoMaxC:=minCPU
               pesoMaxT:=minTAPI
               seleccion:=i
            SINO SI minTAPI=pesoMax Y minCPU<pesoMaxC
               pesoMax:=MAXIMO(minCPU, minTAPI)
               pesoMaxC:=minCPU
               pesoMaxT:=minTAPI
               seleccion:=i
            FSI
         FSI
      FPARA
   MIENTRAS numFin>0 Y selección=-1

   # Si no encontró tarea busca tarea final lista con mayor peso
   SI seleccion=-1
      PARA i:=1 HASTA N HACER
         SI D_i=0 Y L_i=1
            # Busca menor peso válido para todas las CPU y TAPI respecto a i
            minCPU:=INFINITO
            PARA p:=1 HASTA numeroCPUs HACER
               SI PM_CPU_{i,p}>0 Y PM_CPU_{i,p}<minCPU ENTONCES minCPU:=PM_CPU_{i,p}
            FPARA
            minTAPI:=INFINITO
            PARA p:=1 HASTA numeroTAPIs HACER
               SI PM_TAPI_{i,p}>0 Y PM_TAPI_{i,p}<minTAPI ENTONCES minTAPI:=PM_TAPI_{i,p}
            FPARA

            # Comprueba si la tarea i tiene mayor peso
            SI minCPU≥pesoMax O minTAPI≥pesoMax ENTONCES
               pesoMax:=MAXIMO(minCPU,minTAPI)
               seleccion:=i
            FSI
         FSI
      FPARA
   FSI

   DEVOLVER seleccion
FIN
```

5.4.2.5. SeleccionarProcesadores. Selección de los procesadores

El objetivo de esta función es determinar el procesador o pareja de procesadores donde ejecutar una tarea, en función de su tipo. Para una tarea τ_i tipo cpu (o tapi) simplemente se busca el procesador p que ofrece un menor valor $PesosCPU_{i,p}$ (o $PesosTAPI_{i,p}$), esto es, el que implica menores costes de comunicación y ejecución. De igual manera, si la tarea es cpu/tapi se buscan por separado la CPU y la TAPI que ofrecen menores pesos. Finalmente, si la tarea es cpu/cpu o tapi/pai lo que se hace es buscar los dos procesadores del tipo adecuado con los dos menores pesos.

En el anterior proceso se descartan los procesadores con una valor de peso no válido asociado. Aquí se incluyen los procesadores que están ocupados porque no pudieron ser bloqueados es la función *TareasListas2()* del procedimiento principal al ejecutar tareas no interrumpibles.

```
FUNC SeleccionarProcesadores(i)
   # Valores por defecto
   nCPU1:=-1,  nCPU2:=-1,  nTAPI1:=-1,  nTAPI2:=-1

   # Inicia pesos mínimos
   mCPU1:=INFINITO,  mCPU2:=INFINITO,  mTAPI1:=INFINITO, mTAPI2:=INFINITO

   # Si la tarea es válida
   SI i>0 ENTONCES
      # Según el tipo de la tarea
      ELEGIR SEGÚN TipoTarea(i)
         OPCION TCPU
            PARA p:=1 HASTA numeroCPUs HACER
               SI PesosCPU_{i,p}>0 Y PesosCPU_{i,p}<=mCPU1 ENTONCES
                  SI PesosCPU_{i,p}<mCPU1 O (PesosCPU_{i,p}=mCPU1 Y NO Bloqueada_i) ENT
                     nCPU1:=p
                     mCPU1:=PesosCPU_{i,p}
                  FSI
               FSI
            FPARA
         FOPCION

         OPCION TTAPI
            PARA p:=1 HASTA numeroTAPIs HACER
               SI PesosTAPI_{i,p}>0 Y PesosTAPI_{i,p}<=mTAPI1 ENTONCES
                  SI PesosTAPI_{i,p}<mTAPI1 O (PesosTAPI_{i,p}=mTAPI1 Y NO Bloqueada_i) ENT
                     nTAPI1:=p
                     mTAPI:=PesosTAPI_{i,p}
                  FSI
               FSI
            FPARA
         FOPCION

         OPCION TCPUTAPI
            PARA p:=1 HASTA numeroCPUs HACER
               SI PesosCPU_{i,p}>0 Y PesosCPU_{i,p}<=mCPU1 ENTONCES
                  SI PesosCPU_{i,p}<mCPU1 O (PesosCPU_{i,p}=mCPU1 Y NO Bloqueada_i) ENT
                     nCPU1:=p
                     mCPU1:=PesosCPU_{i,p}
                  FSI
               FSI
            FPARA
            PARA p:=1 HASTA numeroTAPIs HACER
               SI PesosTAPI_{i,p}>0 Y PesosTAPI_{i,p}<=mTAPI1 ENTONCES
                  SI PesosTAPI_{i,p}<mTAPI1 O (PesosTAPI_{i,p}=mTAPI1 Y NO Bloqueada_i) ENT
                     nTAPI1:=p
                     mTAPI:=PesosTAPI_{i,p}
                  FSI
               FSI
            FPARA
         FOPCION

         OPCION TCPUCPU
            PARA p:=1 HASTA numeroCPUs HACER
               SI PesosCPU_{i,p}>0 Y PesosCPU_{i,p}<=mCPU2 ENTONCES
                  SI PesosCPU_{i,p}<mCPU1 O (PesosCPU_{i,p}=mCPU1 Y NO Bloqueada_i) ENT
                     nCPU2:=nCPU1
                     nCPU1:=p
                     mCPU2:=mCPU1
                     mCPU1:=PesosCPU_{i,p}
                  SINO SI PesosCPU_{i,p}<mCPU2 O (PesosCPU_{i,p}=mCPU2 Y NO Bloqueada_i)
                     nCPU2:=p
                     mCPU2:=PesosCPU_{i,p}
                  FSI
               FSI
```

```
            FPARA
         FOPCION

         OPCION TTAPITAPI
            PARA p:=1 HASTA numeroCPUs HACER
               SI PesosTAPI_{i,p}>0 Y PesosCPU_{i,p}<=mTAPI2 ENTONCES
                  SI PesosCPU_{i,p}<mTAPI1 O (PesosTAPI_{i,p}=mTAPI1 Y NO Bloqueada_i) ENT
                     nTAPI2:=nTAPI1
                     nTAPI1:=p
                     mTAPI2:=mTAPI1
                     mTAPI1:=PesosTAPI_{i,p}
                  SINO SI PesosTAPI_{i,p}<mTAPI2 O (PesosTAPI_{i,p}=mTAPI2 Y NO Bloqueada_i)
                     nTAPI2:=p
                     mTAPI2:=PesosCPU_{i,p}
                  FSI
               FSI
            FOPCION
         FELEGIR
      FSI

      DEVOLVER (nCPU1,nTAPI1,nCPU2,nTAPI2)
   FIN
```

La función devuelve cuatro identificadores de procesador (*nCPU1, nTAPI1 nCPU2, nTAPI2*) cuyos valores indican...

- *nCPU1*. Número de CPU para una tarea tipo cpu o cpu/tapi, o primer procesador de CPU una tarea cpu/pcu, en el rango [1...*numeroCPUs*] . Vale –1 para otros tipos de tareas o si no se encontró una CPU adecuada disponible.
- *nTAPI1*. Número de TAPI para una tarea tipo tapi o cpu/tapi, o primer procesador CPU de una tarea tapi/tapi, en el rango [1...*numeroTAPIs*] . Vale –1 para otros tipos de tareas o si no se encontró una CPU adecuada disponible.
- *nCPU2*. Segundo procesador CPU para una tarea cpu/cpu en el rango [1...*numeroCPUs*]. Vale –1 para otros tipos de tares o si no se encontró una CPU adecuada disponible.
- *nTAPI2*. Segundo procesador TAPI para una tarea tapi/tapi en el rango [1...*numeroTAPIs*]. Vale –1 para otros tipos de tares o si no se encontró una TAPI adecuada disponible.

5.4.2.6. Planificar. Planificación de la tarea seleccionada

Esta función es la encargada, según el tipo de tarea τ_i indicada, de ejecutar la función de planificación adecuada a ese tipo. Eso se hace si se ha encontrado los procesadores adecuados donde ejecutar la tarea (dados por los parámetros nCPU1, nTAPI1, nCPU2, nTAPI2). Si no se pudo determinar en que procesadores ejecutar la tarea, la ejecución de ésta

será retrasada hasta que concluya la ejecución de la tarea que antes acabe, utilizando la misma función Retardar() descrita en el algoritmo de camino crítico (punto 3.3.8).

La función devuelve además el número de tareas listas actualizado, ya que puede incrementarse debido a la necesidad de añadir tareas de comunicación. Esto puede ocurrir al planificar una tarea cpu o una tarea tapi, dependiendo del tipo de sus predecesoras.

```
FUNC Planificar(i,numTareas,nCPU1,nTAPI1,nCPU2,nTAPI2)

   # Si la tarea tiene asignada CPU y una TAPI; es cpu/tapi
   SI nCPU1>0 Y nTAPI1>0 ENTONCES
      PlanificarCPUTAPI(i,nCPU1,nTAPI1)

   # Si la tarea tiene asignada una sola CPU; es cpu
   ELSE SI nCPU1>0 Y nCPU2<1 ENTONCES
      numTareas:=numTareas+PlanificarCPU(i,nCPU1)

   # Si la tarea tiene asignada una sola TAPI; es tapi
   ELSE SI nTAPI1>0 Y nTAPI2<1 ENTONCES
      numTareas:=numTareas+PlanificarTAPI(i,nTAPI1)

   # Si la tarea tiene asignada dos CPUs; es cpu/cpu
   ELSE SI nCPU1>0 Y nCPU2>0 ENTONCES
      PlanificarCPUTAPI(i,nCPU1,nCPU2)

   # Si la tarea tiene asignada dos TAPIs; es tapi/tapi
   ELSE SI nTAPI1>0 Y nTAPI2>0 ENTONCES
      PlanificarTAPITAPI(i,nTAPI1,nTAPI2)

   # Si no se encontró procesadores libres para la tarea toca retrasarla
   ELSE
      # Para cada tarea en ejecución en una CPU, busca la que acaba antes
      retMin:=INFINITO
      PARA p:=1 HASTA numeroCPUs HACER
         tarea:=ActualCPU_p
         SI j>0 ENTONCES
            j:=EjecNumInt_tarea
            retardo:=Cp_tarea-(tiempo-EjecCom_tarea,j)
            SI retardo<retMin ENTONCES retMin:=retardo
         FSI
      FPARA
      # Para cada tarea en ejecución en una TAPI, busca la que acaba antes
      PARA p:=1 HASTA numeroTAPIs HACER
         tarea:=ActualTAPI_p
         SI j>0 ENTONCES
            j:=EjecNumInt_tarea
            retardo:=Cp_tarea-(tiempo-EjecCom_tarea,j)
            SI retardo<retMin ENTONCES retMin:=retardo
         FSI
      FPARA
      #Retrasa la tarea hasta que concluye la que acaba antes
      Retardar(i,minRet)
   FSI

   DEVOLVER numTareas
FIN
```

5.4.2.7. PlanificarCPU, Planificar TAPI. Planificar las tareas básicas

Las funciones *PlanificarCPU()* y *PlanificarTAPI()* se encargan de la planificación temporal de una tarea básica τ_i, que puede ser cpu o tapi según la función, indicada en el procesador especificado. Básicamente operan de modo que primero comprueban si resulta necesario insertar nuevas tareas de comunicación entre las predecesoras de τ_i y ésta, y si así

es, crean una nueva tarea de comunicación y la planifican, retrasando la ejecución de la propia τ_i. Si no se requieren nuevas tareas, se planifica directamente τ_i.

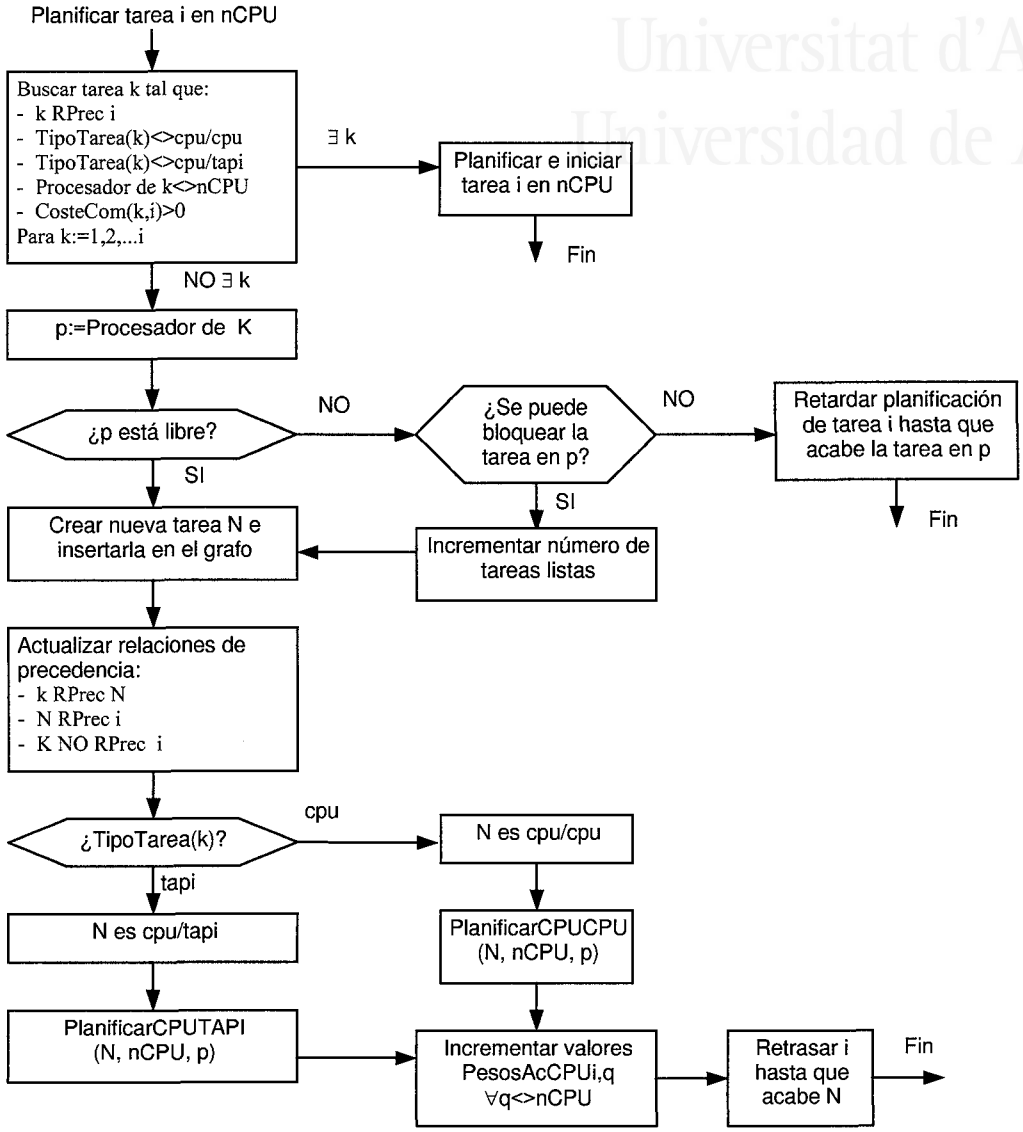


Figura 5-12. Proceso seguido para planificar una tarea cpu.

La Figura 5-12 muestra con mas detalle el proceso llevado a cabo en la función *PlanificarCPU()*. En primer lugar se busca si existe alguna tarea τ_k predecesora de τ_i que implique la necesidad de una tarea de comunicación. Esto ocurre cuando τ_k se ejecuta en un procesador distinto del especificado para τ_i, hay un coste de comunicación entre τ_k y τ_i, y τ_k no es una tarea de comunicación insertada anteriormente. Encontrada τ_k se comprueba si el procesador que requiere está libre. De no ser así se intenta bloquear, y si esto se consigue se dispone de una tarea lista más. Posteriormente se crea una nueva tarea de comunicación, la τ_N, y se añade al grafo, actualizando las relaciones de precedencia para que τ_N quede entre τ_k y τ_i.

Considerando que τ_i es una tarea cpu, según sea el tipo de τ_k, se determina el tipo de τ_N: si es τ_k cpu, τ_N debe ser cpu/cpu y si τ_k es tapi entonces τ_N debe ser cpu/tapi. La nueva tarea de comunicación τ_N se planifica y se retrasa la ejecución de τ_i, finalizando la función.

La función podría realizar el mismo proceso en el futuro para τ_i, si es que ésta tiene otras predecesoras que requieren comunicación. Pero en cada ejecución de la función sólo se crea una tarea de comunicación con el objetivo de planificar una sola tarea en cada iteración del algoritmo.

Si la función no encuentra ninguna tarea τ_k que cumpla las anteriores condiciones, no se requieren tareas de comunicación, y τ_i se planifica directamente.

El código necesario para *PlanificarCPU()* se muestra a continuación:

```
FUNC PlanificarCPU(i,nCPU)
   nuevasTareas:=0   # Número de tareas nuevas
   j:=EjecNumInter_i  # Intervalo de la tarea

   # Para cada tarea k del grafo que sea predecesora de i, no sea una tarea
   # de comunicación, se haya ejecutado en procesador CPU diferente
   # al asignado a i (o en una TAPI), e implique un coste de comunicación
   nueva:=FALSO
   retrasada:=FALSO
   k:=1
   MIENTRAS k≤i Y NO nueva Y NO retrasada HACER
      pC:=ProcPrecCPU_i,k
      pT:=ProcPrecTAPI_i,k
      SI C_k,i Y TipoTarea(k)∉{TCPUCPU,CPUTAPI} Y pC<>nCPU Y CosteCom(k,i)>0 ENT

         # Si el procesador CPU o TAPI de la predecesora está libre
         SI EstadoCPU_pC:=LIBRE O EstadoCPU_pT:=LIBRE ENTONCES
            # Crea una nueva tarea cpu/cpu o cpu/tapi: la última
            nueva:=CIERTO
            N:=N+1
            IniciarComponente(N)

            # Acutailza y matrices de conectividad
            C_k,i=0     # k ya no precede a i
            C_k,N=1     # k precede a N
            C_N,i=1     # N precede a i
            T_N,N=0     # N no precede a N
            T_k,N=1     # k precede a N
            T_N,i=1     # N precede a i
            PARA n:=1 HASTA N HACER
               T_n,N=T_n,k  # Si n precede a k, n precede a N
            FPARA

            # Para la predecesora de i se indica cpu de i para evitar
            # generar más tareas de comunicación en el futuro
            ProcPrecCPU_i,k:=nCPU

            # Si la tarea predecesora era cpu, planifica la nueva cpu/cpu
            SI TipoTarea(k)=TCPU ENTONCES
               NuevoTipoTarea(N,TCPUCPU)
               PlanificarCPUCPU(N,nCPU,pC)
            # Si la tarea predecesora era tapi, planifica la nueva cpu/tapi
            SINO
               NuevoTipoTarea(N,TCPUTAPI)
               PlanificarCPUTAPI(N,nCPU,pT)
            FSI

         # Procesador de la predecesora ocupado
         SINO
```

```
            # Trata de bloquear la tarea del procesador de k
            SI TipoTarea(k)=TCPU ENTONCES
               # k es cpu; se ejecuta en una CPU
               bloq:=Bloquear(EstadoCPU, ActualCPU, pC))
            SINO
               # k es tapi; se ejecuta en una TAPI
               bloq:=Bloquear(EstadoTAPI, ActualTAPI, pT))

            # Si se puede bloquear
            SI bloq ENTONCES
               # Repite iteración para generar tarea de comunicación
               k:=k-1
               # Se tiene una tarea lista más, la bloqueada
               nuevasTareas:=nuevasTareas+1
            # Si no se puede bloquear
            SINO
               # Retrasa la tarea i hasta que acabe la que no se bloquea
               SI TipoTarea(k)=TCPU ENTONCES tarea:=ActualCPU_pC
               SINO tarea:=ActualTAPI_pT FSI
               inter:=EjecNumInter_tarea
               Retardar(i, Cp_tarea-(tiempo-EjecCom_tarea,inter))
               retrasada:=CIERTO
            FSI
         FSI
      FSI
      k:=k+1
   FMIENTRAS

   # Si no se retrasó la tarea i
   SI NO retrasada ENTONCES
      # Se insertó una tarea cpu/cpu o cpu/tapi
      SI nueva ENTONCES
         # Grantiza que en el futuro la tarea i use nCPU
         PARA p:=1 HASTA numeroCPUS HACER
            SI p<>numCPU ENTONCES PesosAcCPU_i,p:=PesosAcCPU_i,p+CTE_PESO
         FPARA
         # Retarda la tarea i hasta que acabe la comunicación
         Retardar(i, Cp_N)

      # No fue necesario insertar tareas: se planifica i
      SINO
         EjecucionCOM_i,j:=tiempo                  # Inicia del nuevo intervalo

         EstadoCPU_nCPU:=EsInterrumpible(i)        # Estado del procesador
         ActualCPU_nCPU:=i

         ProcCPU1_i,j:=nCPU                        # Datos sobre la asignación
         ProcTAPI1_i,j:=-1
         ProcCPU2_i,j:=-1
         ProcTAPI2_i,j:=-1
         UltProcCPU_i:=nCPU
         UltProcTAPI_i:=-1

         F_i:=0              # Tarea planificada
         L_i:=0
      FSI
   FSI
   DEVOLVER nuevasTareas
FIN
```

Una nueva función *IniciarComponente(i)* se encarga de, para todas las matrices usadas en el algoritmo, iniciar la componente *i* a sus valores por defecto. Esto resulta necesario al añadir una nueva tarea. *NuevoTipoTarea(i,tipo)* se encarga de asignar el tipo de tarea a una nueva tarea. La función utilizada para retrasar la ejecución de una tarea, *Retardar()*, es la misma que la descrita en el camino crítico en el punto 3.3.8.

Cabe destacar que en la práctica no resulta necesario añadir las nuevas tareas de comunicación al grafo, con el coste que ello implicaría. Basta con actualizar las matrices de precedencia C y T y otros valores usados por el algoritmo.

PlanificarTAPI() tiene una estructura equivalente a la función descrita, aunque hace uso de las variables relativas a los procesadores del tipo adecuado. De ella cabe destacar que puede crear tareas de comunicación cpu/tapi como la anterior si hay predecesoras de la tarea a planificar de tipo cpu, o tareas tapi/tapi (en vez cpu/cpu) si hay predecesoras de tipo tapi.

5.4.2.8. PlanificarCPUCPU, PlanificarTAPITAPI, PlanificarCPUTAPI. Planificar tareas de comunicación.

La planificación de las tareas de comunicación una vez se han encontrado procesadores donde ejecutarlas es simple. Básicamente se debe actualizar el instante de comienzo de la tarea para el intervalo actual y el estado de los procesadores donde se ejecuta. Además se almacenan los identificadores de los procesadores asignados para al final del algoritmo realizar la asignación de las tareas a los objetos procesador. Cabe destacar como los procesadores donde se ejecutan tareas de comunicación se deben marcar como no interrumpibles.

A continuación se muestra como ejemplo el código de las funciones encargadas de planificar las taeas cpu/tapiy y cpu/cpu.

```
PROC PlanificarCPUTAPI(i,nCPU, nTAPI)
   # Intervalo actual de la tarea
   j:=EjecNumInter_i

   # Inicio del nuevo intervalo
   EjecucionCOM_i,j:=tiempo

   # Estado de los procesadores
   EstadoCPU_nCPU:=NOINT
   EstadoCPU_nTAPI:=NOINT
   ActualCPU_nCPU:=i
   ActualCPU_nTAPI:=i

   # Datos sobre la asignación
   ProcCPU1_i,j:=nCPU
   ProcTAPI1_i,j:=nTAPI
   ProcCPU2_i,j:=-1
   ProcTAPI2_i,j:=-1
   UltProcCPU_i:=nCPU
   UltProcTAPI_i:=nTAPI

   F_i:=0     # Tarea planificada
   L_i:=0
FIN
```

```
PROC PlanificarCPUCPU(i, nCPU1, nCPU2)
   # Intervalo actual de la tarea
   j:=EjecNumInter_i

   # Inicio del nuevo intervalo
   EjecucionCOM_i,j:=tiempo

   # Estado de los procesadores
   EstadoCPU_nCPU1:=NOINT
   EstadoCPU_nCPU2:=NOINT
   ActualCPU_nCPU1:=i
   ActualCPU_nCPU2:=i

   # Datos sobre la asignación
   ProcCPU_i,j:=nCPU
   ProcTAPI_i,j:=-1
   ProcCPU2_i,j:=nCPU2
   ProcTAPI2_i,j:=-1
   UltProcCPU_i:=nCPU2
   UltProcTAPI_i:=-1

   F_i:=0     # Tarea planificada
   L_i:=0
FIN
```

5.4.2.9. Bloquear. Interrumpir la tarea que se ejecuta en un procesador

Como en algoritmos previos, la función requiere como parámetros el identificador de un procesador a bloquear, y la matriz *EstadoCPU* o *EstadoTAPI* en *EstadoProc*, y *ActualCPU* o *ActualTAPI* en *ActaulProc*, según el tipo del procesador. Si hay una tarea que no acaba de comenzar en el procesador se interrumpe ésta finalizando su intervalo actual, marcándola además como tarea lista y se libera el procesador. En este caso devuelve CIERTO. Si la tarea en el procesador acaba de empezar, deja que su ejecución continúe y devuelve FALSO.

La novedad es que ahora, cuando se interrumpe una tarea se deben actualizar sus pesos, para dar prioridad al procesador actual frente al resto cuando se planifique el siguiente intervalo, tratando de evitar así comunicaciones por cambios de procesador innecesarios. Esto se consigue incrementando los pesos de esa tarea con respecto a todos los procesadores distintos del actual en una cantidad constante, ya que al seleccionar procesadores se buscan los de menor peso.

```
FUNC Bloquear(EstadoProc, ActualProc, p)
   i:=ActualProc_p     # Tarea en p
   j:=EjeNumInt_i      # Intervalo de la tarea

   # La tarea que está en el procesador acaba de empezar
   SI EjeCom_tarea,j=tiempo ENTONCES
      resultado:=FALSO
   SINO
      duracion:=tiempo- EjeCom_i,j
      EjecFin_i,j:=tiempo          # Fin de intervalo actual
      EjeNumInt_i:=EjeNumInt_i+1   # Incrementa número de intervalos
      EstadoProc_p:=LIBRE          # El procesador queda libre
      ActualProc_p:=-1
      F_i:=1                       # No ha acabado
      L_i:=1                       # Está lista
      Ret_i:=Ret_i+duracion        # Actualiza tiempo de retardo
      Cp_i:=Cp_i-duracion          # Actualiza Tiempo de computo

      Bloqueada_i=CIERTO           # Marca tarea como bloqueada

      # Si se está bloqueando una CPU
      SI EstadoProc = EstadoCPU ENTONCES
         # Incrementa pesos de otras CPUs para que la próxima vez la tarea
         # se ejecute preferentemente en las misma CPU
         PARA q:=1 HASTA numeroCPUs HACER
            SI p<>q ENTONCES PesosAcCPU_j,q:=PesosAcCPU_j,q+CTE_PESO
         FPARA
      # Si se está bloqueando una TAPI
      SINO
         # Incrementa pesos de otras TAPIs para que la próxima vez la tarea
         # se ejecute preferentemente en las misma TAPI
         PARA q:=1 HASTA numeroTAPIs HACER
            SI p<>q ENTONCES PesosAcTAPI_j,q:=PesosAcTAPI_j,q+CTE_PESO
         FPARA
      FSI

      resultado:=CIERTO
   FSI
DEVOLVER resultado
```

5.4.2.10. CalcularMatrices. Actualizar los valores de tiempo y peso

Se trata de una función similar a la utilizada por los anteriores algoritmos de camino crítico descrita en el punto 3.3.9. La diferencia es que ahora sólo se actualizan los vectores con los tiempos de finalización, y creación, y no las matrices de pesos, que son actualizadas en otras funciones. Realiza estos cálculos:

$$Fin_i := Cp_i + Ret_i + \underset{j:=1..i-1}{MAX}\left(T_{j,i} \cdot Fin_j\right) \quad i := 1, 2, \ldots N$$

$$Creación_i := Fin_i - Cp_i \quad i := 1, 2, \ldots N$$

5.4.2.11. AsignarResultado. Iniciar los objetos procesador

Este procedimiento se encarga de iniciar los objetos que modelan los procesadores con toda la información sobre los intervalos de tiempo de ejecución de las tareas resultante de la planificación. Se trata de una versión del procedimiento descrito en 5.2.2.7 y 5.2.3.8 que tiene en cuanta los nuevos tipos de tareas de comunicación.

```
PROC AsigarResultado()
   # Para cada tarea
   PARA i:=1 HASTA N HACER
      estado:=EsInterrumpible(i)

      # Para cada intervalo de la tarea
      PARA j:=1 HASTA j≤EjeNumInt_i HACER
         # Obtiene procesadores asignados al intervalo
         nCPU1:=ProcCPU1_i,j
         nTAPI1:=ProcTAPI1_i,j
         nCPU2:=ProcCPU2_i,j
         nTAPI2:=ProcTAPI2_i,j

         # Obtiene limites de tiempo del intervalo
         inicio:=EjecCom_i,j
         fin:=EjecFin_i,j

         # Si la tarea tiene asignada CPU y una TAPI; es cpu/tapi
         SI nCPU1>0 Y nTAPI1>0 ENTONCES
            IniciarTarea(CPU, nCPU1, i, inicio, estado, TCPUTAPI)
            IniciarTarea(TAPI, nTAPI1, i, inicio, estado, TCPUTAPI)
            FinalTarea(CPU, nCPU1, fin)
            FinalTarea(TAPI, nTAPI, fin)

         # Si la tarea tiene asignada una sola CPU; es cpu
         ELSE SI nCPU1>0 Y nCPU2<1 ENTONCES
            IniciarTarea(CPU, nCPU1, i, inicio, estado, TCPU)
            FinalTarea(CPU, nCPU1, fin)

         # Si la tarea tiene asignada una sola TAPI; es tapi
         ELSE SI nTAPI1>0 Y nTAPI2<1 ENTONCES
            IniciarTarea(TAPI, nTAPI1, i, inicio, estado, TTAPI)
            FinalTarea(TAPI, nTAPI1, fin)

         # Si la tarea tiene asignada dos CPUs; es cpu/cpu
         ELSE SI nCPU1>0 Y nCPU2>0 ENTONCES
            IniciarTarea(CPU, nCPU1, i, inicio, estado, TCPUCPU)
            IniciarTarea(CPU, nCPU2, i, inicio, estado, TCPUCPU)
            FinalTarea(CPU, nCPU1, fin)
            FinalTarea(CPU, nCPU2, fin)
```

```
            # Si la tarea tiene asignada dos TAPIs; es tapi/tapi
            ELSE SI nTAPI1>0 Y nTAPI2>0 ENTONCES
               IniciarTarea(TAPI, nTAPI1, i, inicio, estado, TTAPITAPI)
               IniciarTarea(TAPI, nTAPI2, i, inicio, estado, TTAPITAPI)
               FinalTarea(TAPI, nTAPI1, fin)
               FinalTarea(TAPI, nTAPI2, fin)
            FSI
         FPARA
      FPARA
   FIN
```

Capítulo 6

Entorno de pruebas

6.1. Introducción

6.1.1. Motivación

Se han propuesto diferentes algoritmos de planificación de tareas para sistemas de visión por computador, aunque la gran mayoría de ellos están destinados a optimizar operaciones concretas sobre determinadas arquitecturas hardware. Además, resulta muy difícil encontrar herramientas que permitan a un desarrollador de aplicaciones de visión por computador especificar un algoritmo cualquiera y aplicarle unas técnicas de planificación, considerando la existencia de un hardware disponible o necesario, para simular y evaluar los tiempos que con ese hardware se obtendrían.

Por otra parte, las herramientas comerciales existentes para la especificación de alto nivel de algoritmos de visión por computador, como Khoros [26], WiT [27] ó Evision [28], no van más allá de una especificación de alto nivel y la simulación, y no suministran información relativa a la planificación de tareas y los tiempos de ejecución para un determinado hardware.

Estos han sido los motivos que han llevado al autor de esta tesis a desarrollar el entorno de herramientas que se describen en este capítulo, el cual permite la especificación de alto nivel de aplicaciones de visión por computador, su simulación, y la novedad de evaluar diferentes planificaciones estáticas dadas unas características hardware.

6.1.2. Orígenes

Dentro del proyecto CICYT coordinado "Construcción de un Entorno para la Investigación y Desarrollo en Visión Artificial" (Ref. TAP96-0629-C04) se ha desarrollado un entorno distribuido llamado EVA (Entorno de Visión Artificial) que consta de una serie de herramientas para trabajar con aplicaciones de visión por computador, las cuales permiten la adquisición de imágenes, operaciones de procesamiento (software y hardware), y visualización de resultados [29]. EVA incorpora un interprete de comandos, un módulo software encargado de ejecutar las operaciones o programas solicitados por los usuarios, y

diseñado para trabajar con bibliotecas de funciones, y en especial, procesamiento de imágenes.

El autor de esta tesis comenzó realizando la interfaz de usuario de EVA dentro del subproyecto "Driver para Tarjeta de Adquisición y Procesamiento de Imágenes y Desarrollo de una Interfaz de Usuario" (Ref. TAP96-0629-CO4-01). Esto es, las aplicaciones que permiten a los usuarios utilizar el interprete de forma amigable. Como interfaz de usuario básica se han desarrollado un conjunto de aplicaciones que trabajan sobre máquinas con el S.O. M.S. Windows 95-98, y que forman una arquitectura distribuida con clientes y servidores, que permite a diversos usuarios trabajar con uno o más interpretes de comandos dentro de una red de ordenadores. Las aplicaciones son el servidor de comandos, que incluye el interprete de comandos; una consola que permite al usuario un acceso directo con un interprete; un servidor de imágenes (o visualizador), encargado de mostrar las imágenes resultado generadas por un interprete; y finalmente el entorno visual, una aplicación en la que se pueden expresar operaciones o programas de forma gráfica, mediante esquemas de objetos gráficos que representan las distintas operaciones a realizar, el intercambio de datos y el flujo de ejecución de forma ilustrativa [30].

6.1.3. Arquitectura del entorno de simulación y planificación

Como extensión al proyecto CYCIT indicado, y aprovechando las características para la simulación del entorno visual y el interprete, se ha reestructurado y mejorado el entorno para incluir en el otras características [30][31]. El objetivo es disponer de una serie de herramientas que además de permitir al usuario una simulación gráfica de aplicaciones de visión por computador, como ya hacen otras herramientas comerciales [26][27][28], incluyan como novedad permitir diseñar y evaluar planificaciones estáticas para las aplicaciones según las técnicas presentadas en los capítulos anteriores.

El entorno final de simulación y planificación se compone de cuatro bloques principales; la aplicación del entorno visual, una base de datos, una serie de módulos que ejecutan las operaciones disponibles para un esquema y la aplicación del planificador estático. Todos ellos se pueden identificar en la Figura 6-1. También existen otras aplicaciones auxiliares. Esta estructura de aplicaciones proporciona un entorno de trabajo muy flexible,

con la interesante novedad de la planificación estática de tareas. En los siguientes apartados se describen con más detalle los distintos módulos.

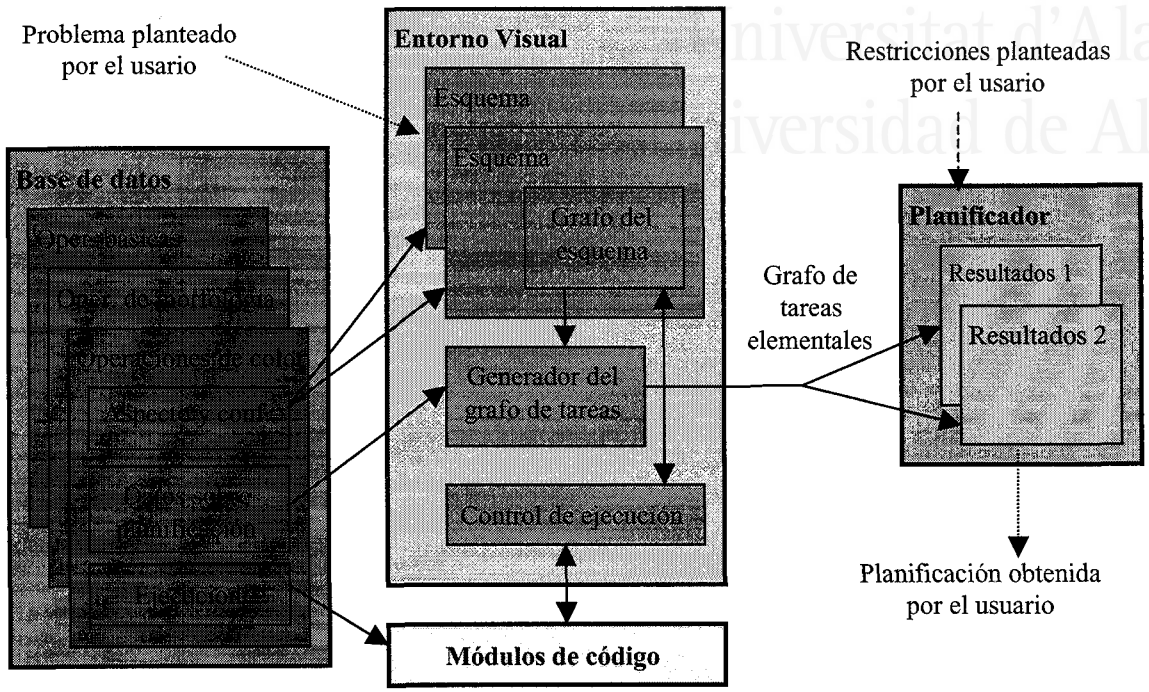


Figura 6-1. Arquitectura del entorno de simulación y planificación estática.

6.2. El entorno visual y la base de datos

En este apartado se describen las características y el funcionamiento básico del entorno visual y de la base de datos, así como la interacción entre ambos módulos. También se describe con más detalle el proceso de generación del grafo de tareas elementales correspondiente a un esquema de alto nivel de abstracción planteado por el usuario.

6.2.1. La interfaz con el usuario

Con el entorno visual el usuario puede especificar un algoritmo de procesamiento de imágenes como un esquema gráfico compuesto por OPIs (Objetos de Procesamiento de Imágenes) para simular su ejecución. El aspecto de la aplicación se muestra en la Figura 6-2.

Cada OPI representa una operación a realizar dentro de un esquema, y posee una serie de entradas a donde llegarán las imágenes que debe operar procedentes de otros OPIs, y de

salidas por las que devuelve los resultados tras su ejecución. Estos objetos se pueden conectar entre sí mediante unas tuberías que representan el intercambio de datos o imágenes. Las tuberías permiten conectar unas salidas con una o más entradas.

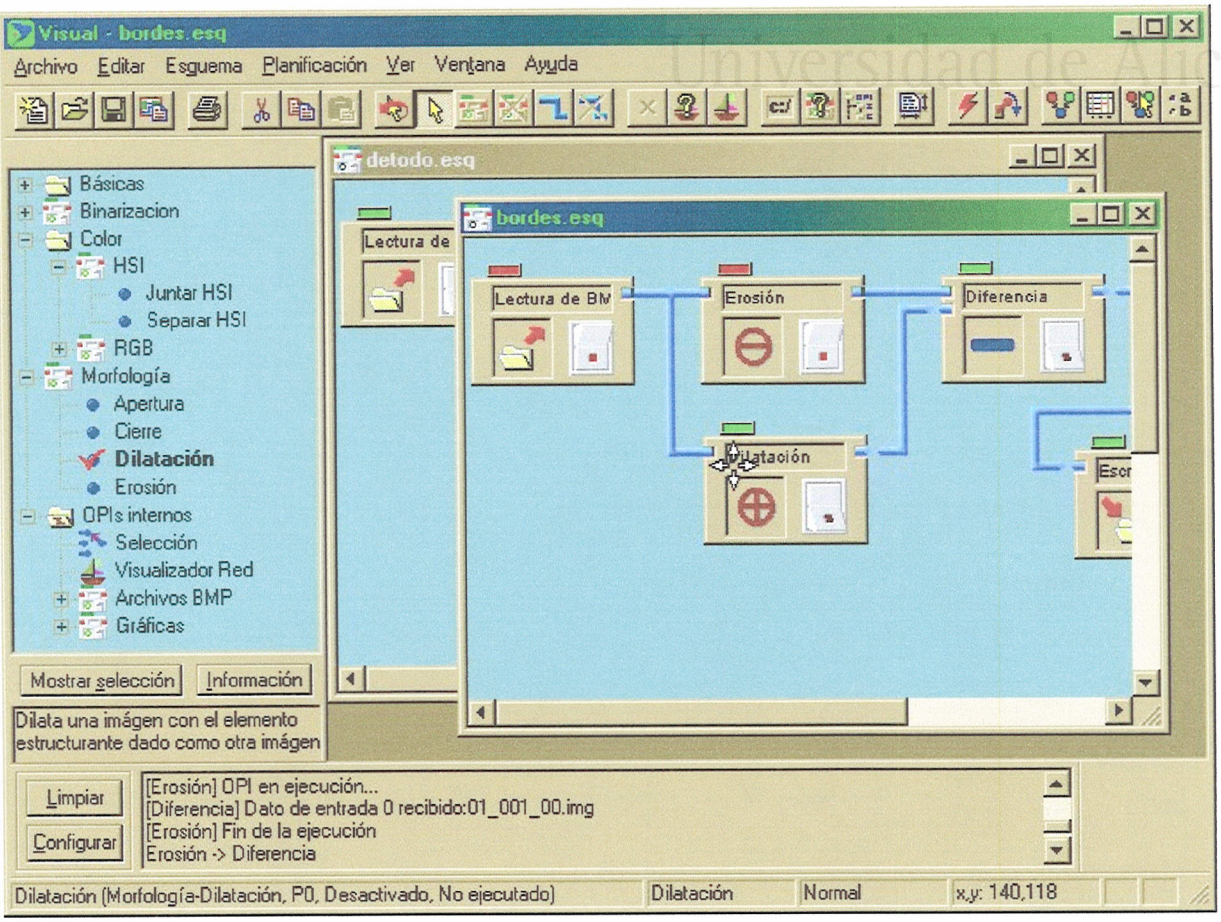


Figura 6-2. Apariencia del entorno visual.

Los OPIs se clasifican en diferentes tipos según la clase de operaciones que pueden realizar. Así, un OPI de un determinado tipo puede ejecutar una operación de entre varias operaciones de una misma clase. Como ejemplo, existe un tipo de OPI dedicado a ejecutar la clase de operaciones aritméticas básicas (suma, resta, producto y división), y que puede ser configurado para realizar una de esas cuatro operaciones. El usuario dispone de una barra de herramientas con un árbol donde aparecen todos los OPI disponibles con sus operaciones de forma clasificada (ver Figura 6-2).

Cada OPI posee unas características o propiedades propias, que se pueden dividir en dos grupos. Por una parte están las propiedades generales del OPI, que representan valores que poseen todos los OPIs como son el número de entradas, número de salidas..., y por otra parte están las propiedades particulares del OPI, que son dependientes de la clase de

operaciones que éste realice. Ejemplos de propiedades particulares pueden ser la selección actual de operación concreta que el OPI debe ejecutar, el nombre de un archivo en un OPI de lectura de disco o el umbral para una binarización.

La ejecución de cada OPI de un esquema se puede controlar manualmente a través del botón de ejecución que dispone. Para que un OPI se ejecute y opere los datos de entrada se debe activar el botón de ejecución. Esta operación la puede realizar el usuario manualmente, actuando sobre el botón del OPI con el ratón, o la puede realizar el Entorno Visual de forma automática cuando está activado el modo de ejecución automática. Además, para que un OPI se ejecute es necesario que los otros OPIs de los que proceden sus datos de entrada se hayan ejecutado ya correctamente y hayan generado salidas. Es decir, un OPI solo se ejecuta cuando tiene disponibles todas la entradas. El estado de la ejecución, los posibles errores y muchas otras situaciones son reflejadas como mensajes en una barra que aparece en la parte inferior de la aplicación (ver Figura 6-2)

La forma en que se representa la dependencia de los datos de entrada a OPI de las salidas generadas por otros OPIs es a través de las conexiones. Éstas se representan en un esquema como una tubería que parten de una salida de un OPI y que llega, a través de posibles ramificaciones, a las entradas de otros OPIs. Cuando un OPI no se ha ejecutado todavía las conexiones que parten de él se dibujan como tuberías cortadas para indicar que al destino todavía no ha llegado un nuevo dato. Por el contrario, cuando un OPI se ejecuta correctamente y envía los datos sus destinos, las conexiones de salida aparecen como tuberías continuas. En general, el flujo de datos en los esquemas siempre va de izquierda a derecha.

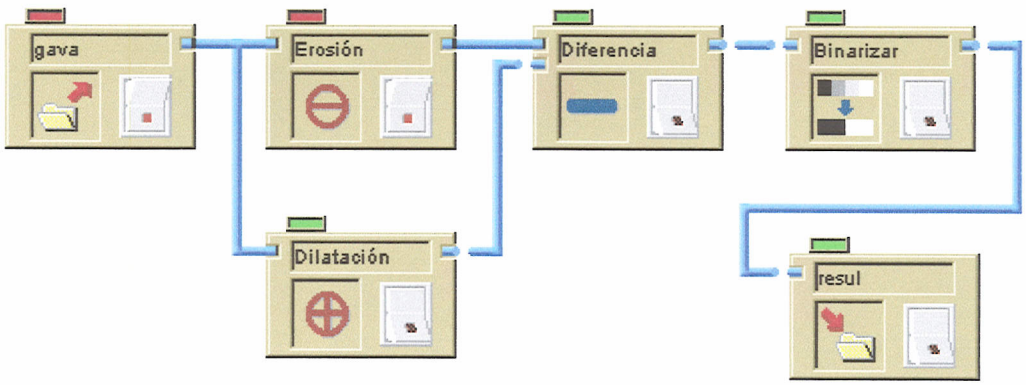


Figura 6-3. Esquema para la detección de bordes con algunos OPIs ejecutados.

La Figura 6-3 muestra un esquema relativo a un sencillo algoritmo de detección de bordes, con algunos OPIs ejecutados en la simulación (los que tienen el indicador de color

rojo). Como se puede ver en la figura, tanto el aspecto de los OPIs como el de las tuberías refleja claramente el estado de ejecución del esquema, ofreciendo el entorno visual un aspecto más ilustrativo que otros programas existentes [30][29][9].

El usuario puede ver en cualquier momento la imagen correspondiente a una salida o entrada de un OPI sin más que activar el ratón sobre ella, como muestra la Figura 6-4. Aparece entonces una ventana de visualizador que también ofrece utilidades como ampliación, exploración de valores, o guardar en archivo la imagen.

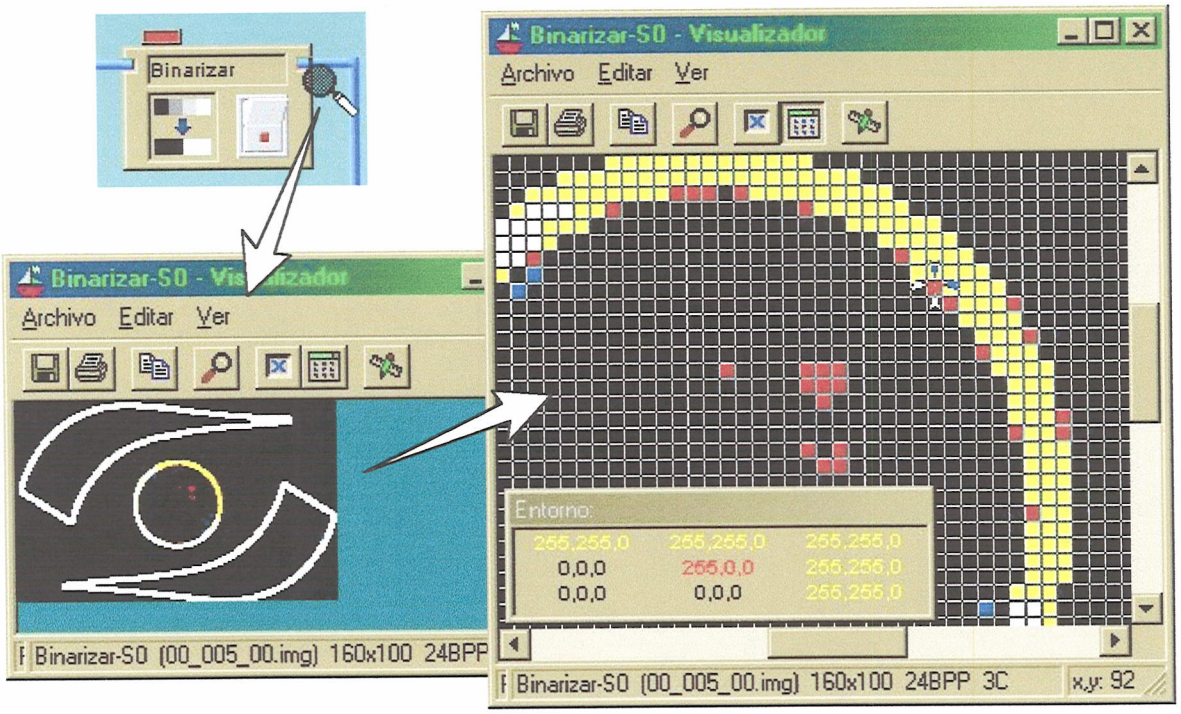


Figura 6-4. Distintas capturas de la ventana del visualizador de imágenes.

Dentro del entorno visual, cada esquema de OPIs se representa como un objeto grafo que contiene las estructuras de datos necesarias. Las dos principales son un vector con los OPIs que forman el esquema y un vector con los objetos que representan las conexiones entre OPIs [30]. La información relativa a los tipos de OPIs disponibles, su aspecto y operaciones se carga dinámicamente al arrancar la aplicación desde la base de datos.

El entorno visual ofrece también opciones para la generación del grafo de tareas que constituyen un esquema, y la ejecución de las aplicaciones del editor de grafos y del planificador. Todo ello se describe en posteriores apartados.

6.2.2. La base de datos

Cada OPI constituye una operación de procesado de imágenes que es posible dividir en bloques elementales a niveles muy diferentes. Un nivel de división alto representaría un pequeño número de operaciones de alto grado de complejidad, mientras que un nivel de división bajo requeriría más operaciones de menor complejidad cada una.

Para este entorno se ha escogido una representación multinivel, de forma que, dentro de un esquema, el usuario puede trabajar con operaciones elementales, o, si lo desea, puede emplear operaciones de alto nivel. De este modo, el usuario no necesita conocer el funcionamiento interno de cada uno de los algoritmos que emplea. Por ejemplo, puede añadir en el sistema un bloque que lleve a cabo un cierto filtro morfológico en vez de añadir bloques para cada una de las operaciones elementales de erosión y dilatación. En cualquier caso, siempre es posible usar bloques elementales para algoritmos muy específicos.

Por otra parte, hay que considerar que el planificador de tareas requiere una representación con el nivel más bajo posible para poder llegar a un resultado eficiente. Cualquier algoritmo se ejecutará más rápidamente si se alcanza el máximo paralelismo en el sistema donde se procesa. Además, el hecho de ejecutar en cierto orden cada uno de los pasos en un algoritmo también puede producir un aumento de velocidad. Pero para llegar a ese punto se requiere conocer en detalle el algoritmo, y el primer dato que debe conocerse son las tareas en las cuales una tarea puede dividirse. Así, cuanto más detallada es la división de tareas, mejores resultados se alcanzan.

Teniendo en cuenta lo anterior, se ha decidido mantener información sobre el subgrafo de tareas elementales para cada operación de los posibles tipos de OPIs, esto es, las tareas más simples de que se componen, las relaciones entre ellas y sus parámetros.

La información de la base de datos se divide en dos partes:

- *Aspecto y simulación de los OPIs.* Mantiene toda la información de los OPIs relativa a un esquema y la simulación, como son las clases de operaciones disponibles, los módulos ejecutables, el aspecto en un esquema, etc. Ésta se almacena en archivos con un formato propio de la aplicación.

- *Tareas elementales de los OPIs.* Contiene los datos relativos a las tareas en que se dividen las operaciones, dentro de un archivo de base de datos M.S. Access (ver Figura 6-5).

La información sobre el aspecto y simulación se carga al arrancar la aplicación del entorno visual, y la concerniente a las tareas se carga cuando el usuario decide generar el grafo de tareas de un esquema.

Grafos : Tabla

TipoOPI	Funcion	Divisible	Nx	Ny	Grafo	Descripción
1	0	☑	2	2	(1482, -2010) -1 0 750	Constante
2	0	☑	2	2	Grafo de procesos. S.T.P.M.	Suma
2	1	☑	2	2	Grafo de procesos. S.T.P.M.	Diferencia
2	2	☑	2	2	Grafo de procesos. S.T.P.M.	Producto
2	3	☑	2	2	Grafo de procesos. S.T.P.M.	Division
3	0	☑	2	2	Grafo de procesos. S.T.P.M.	And
3	1	☑	2	2	Grafo de procesos. S.T.P.M.	Or
3	2	☑	2	2	Grafo de procesos. S.T.P.M.	Xor
3	3	☑	2	2	Grafo de procesos. S.T.P.M.	Not
4	0	☑	2	2	Grafo de procesos. S.T.P.M.	Erosión
4	1	☑	2	2	Grafo de procesos. S.T.P.M.	Dilatación
4	2	☑	2	2	Grafo de procesos. S.T.P.M.	Apertura
4	3	☑	2	2	Grafo de procesos. S.T.P.M.	Cierre
5	0	☑	2	2	Grafo de procesos. S.T.P.M.	SepRGB
5	1	☑	2	2	Grafo de procesos. S.T.P.M.	JuntarRGB
6	0	☑	2	2	Grafo de procesos. S.T.P.M.	SepHSI
6	1	☑	2	2	Grafo de procesos. S.T.P.M.	JuntarHSI
7	0	☑	3	2	Grafo de procesos. S.T.P.M.	Binarizar
30000	0	☐	0	0	Grafo de procesos. S.T.P.M.	Visualizar
30001	0	☐	0	0	Grafo de procesos. S.T.P.M.	EscrBMP

Registro: 1 de 23

Figura 6-5. Información sobre las tareas de los OPIs en la base de datos.

Con la forma anterior de estructurar la información se logran principalmente dos ventajas. Por una parte, el hecho de disponer de la información sobre los OPIs y sus características en una base de datos, y el que está se cargue en tiempo de ejecución, permite la especificación de nuevos OPIs o la modificación de estos de una forma dinámica, sin necesidad de recompilar la aplicación para actualizar su código. Por otra parte, el separar la información relativa a un esquema de la relacionada con las tareas que componen las operaciones permite definir distintas configuraciones de los subgrafos que forman los OPIs y de los parámetros de tareas elementales para el mismo conjunto de OPIs. Esto permite, por

ejemplo, representar como son las tareas elementales de un esquema de alto nivel y sus parámetros de forma individual para varias arquitecturas hardware disponibles.

Sin embargo, la base de datos también presenta un problema; la imposibilidad de cargar correctamente esquemas viejos realizados con OPIs que ya no existen en la base de datos actual o cuyas características se han cambiado. Realmente este problema no es significativo ya que lo habitual es especificar nuevos OPIs o modificar el funcionamiento interno de alguno existente sin cambiar su interface, no eliminar o cambiar los existentes.

La información en la base de datos por cada posible tipo de OPI contiene información sobre cinco aspectos principales [30]:

- *General*. Identificadores del tipo de OPI y de la operación que realiza, además de la descripción, nombre, etc.
- *Parámetros de configuración y aspecto del OPI*. Estos son los parámetros de configuración del OPI y sus tipos de datos (por ejemplo, el umbral para una operación de binarización, que puede ser un valor entero), el aspecto del dialogo de configuración de esos parámetros, esto es los controles que puede utilizar el usuario y su localización (para el caso de la binarización sería un control de edición donde el usuario especifica el valor de umbral), los datos sobre el número de posibles entradas y salidas, y el dibujo que representa el OPI.
- *Módulos para la simulación*. El nombre y ubicación del módulo con el código correspondiente a todas la operaciones de un mismo OPI que es utilizado para la simulación. En la actualizad se trabaja con módulos ejecutables, aunque podrían ser bibliotecas de enlace dinámico DLL o ActiveX.
- *Jerarquía de operaciones*. Organización de las de operaciones en un árbol con las clases de OPIs y sus operaciones que facilite su localización cuando se está diseñando un esquema. Este árbol con las clases y operaciones aparece en la ventana principal de la aplicación según muestra la *Figura 6-2*.
- *Datos sobre planificación*. Se especifica el subgrafo de tareas elementales que ejecuta el OPI, así como un indicador de si el OPI es divisible, es decir, se puede dividir su ejecución en varios flujos de tareas paralelos de menor tamaño, dependiendo del tamaño medio de las imágenes de entrada. En caso de ser divisible se especifica también el número de divisiones en X y en Y en que deben

fraccionarse las imagenes como los valores N_X y N_Y. De este modo una imagen de entrada será dividida $N_X \cdot N_Y$ fragmentos procesados cada uno por una copia del subgrafo.

En relación al subgrafo de tareas elementales, hay que detallar que cada tarea de este tiene indicado el tipo de procesador que requiere para su ejecución, que puede ser una CPU de uso general (tarea tipo cpu) o una TAPI (tarea tipo tapi). También existen tareas de comunicación entre CPU y TAPI que requieren de ambos tipos de procesador a la vez. Esta información será empleada como restricción en la asignación espacial para lograr una planificación más real.

Además las tareas del subgrafo llevan asociados los costes de ejecución y, opcionalmente para tareas interrumpibles, costes relativos a interrupción y reanudación. Los arcos del subgrafo que representan la precedencia mantienen los costes de comunicación. Todos los costes están definidos respecto a un cierto tamaño de imagen predefinido en bytes (*Bc*), y cuando el subgrafo debe procesar imágenes de mayor tamaño, los costes se multiplican por el número de veces que la imagen supere ese tamaño. Esto se describe mejor en el siguiente punto, relativo a la generación del grafo de tareas.

La descripción del subgrafo está expresada como un campo de texto. La Figura 6-6 muestra un ejemplo de descripción para un sencillo subgrafo.

```
Grafo de procesos. S.T.P.M.
# Grafo (ordenación, nº nodos, nº aristas)
1 3 2
# Nodos (posición (x,y), versión, orden, radio)
# (nombre, versión, TMín, TMáx, TInterrupción, TLectura %, tipo, interrumpible)
(2250, -1125) -1 0 750
Tarea1 -2 10 10 10 50.000000 CPU 1
(1125, -3375) -1 1 750
Tarea2 -2 5 5 10 50.000000 CPU/TAPI 0
(3375, -3375) -1 2 750
Tarea3 -2 20 20 10 50.000000 CPU 1
# Aristas (versión, origen, destino, coste comunicación)
-1 0 1 2
-1 0 2 1
```

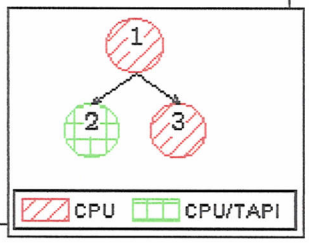


Figura 6-6. Ejemplo de subgrafo y su descripción en la BD.

6.2.3. El generador del grafo de tareas

En el entorno visual están incluidas las funciones que permiten generar un grafo de tareas elementales a partir de las estructuras de datos que representan un esquema de OPIs y

de la información sobre estos presente en la base de datos. Ese grafo de tareas es almacenado en un archivo y contiene los siguientes elementos:

- Lista de las tareas elementales que toman parte en el proceso (generalmente un OPI podrá ser descompuesto en varias tareas de menor nivel), con el tipo de cada una (cpu, tapi o cpu/tapi)
- Datos relativos a los tiempos de computación o coste de las tareas.
- Relaciones de precedencia entre las tareas.
- Relaciones de exclusión. Información sobre las tareas interrumpibles y no interrumpibles.
- Requerimientos de comunicación y sus costes asociados.

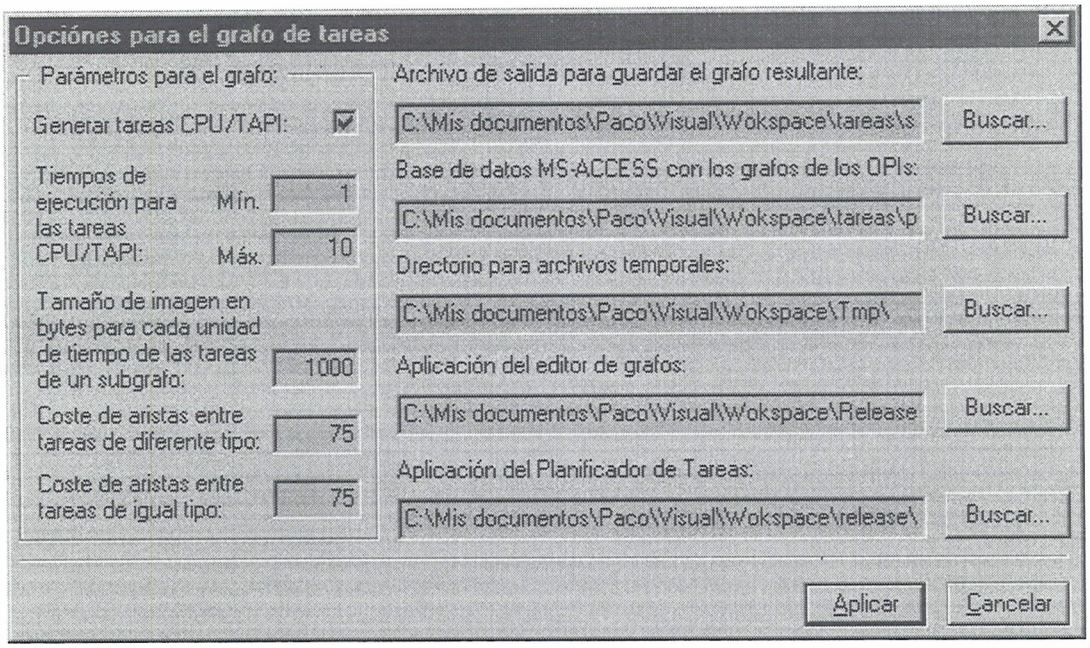


Figura 6-7. Dialogo de configuración de la generación del grafo.

Para la generación del grafo, el usuario puede configurar una serie de opciones en un dialogo como el de la Figura 6-7. Entre las opciones destacan la elección de la base de datos MS. Access que contiene la información sobre los subgrafos con las tareas elementales de los OPIs, la posibilidad de generar tareas cpu/tapi automáticamente y el tamaño de imagen relativo a cada unidad de tiempo especificada en los subgrafos (*Bc*).

El proceso de particionamiento que genera el grafo de tareas se subdivide en tres fases bien diferenciadas [32], las cuales se describen a continuación.

6.2.3.1. Inicialización

Al ejecutar un primer OPI del esquema se inicia la generación del grafo global de tareas elementales. Se abre la conexión con la base de datos para su utilización, y se crea el grafo global resultante, que inicialmente está vacío. Además se asignan los valores a las variables que determinan aspectos como la opción de añadir tareas CPU/TAPI automáticamente, los costes de estas tareas y los costes de comunicación entre ellas y las otras del grafo, o el tamaño de imagen relativo a cada unidad de coste representada en los subgrafos. Todos estos parámetros se obtienen de los datos dados por el usuario en el dialogo de configuración mostrado en la Figura 6-7.

6.2.3.2. Generación de tareas

Este paso se realiza para cada OPI del esquema que se ejecuta correctamente (tiene todos los datos de entrada y da un resultado válido) durante una simulación, cuando la opción de generar el grafo está activada. Así, este paso siempre se lleva a cabo para un OPI que no tiene predecesores, o para uno cuyos predecesores ya ejecutaron esta fase anteriormente.

Para cada OPI ejecutado, en primer lugar se averigua todos los OPIs alcanzables desde él, es decir, los que requieren sus datos de salida. También se determina el tipo de operación que realiza el OPI así como la función a la que corresponde dentro de la clase y el tamaño medio de las imágenes que recibe.

A partir del OPI y de su configuración se determina además si se requiere un área de solape entre los posibles fragmentos en que se dividan las imágenes de entrada para ser operadas. El área de solape depende del tipo de operación del OPI, teniendo sentido en las operaciones que utilicen entornos de vecindad y no en las punto a punto, y del tamaño del área de vecindad o elemento estructurante. Esta área de solape viene determinada por el tamaño s en pixels que debe expandirse cada fragmento para poder ser operado independientemente del resto cuando se requiere trabajar con un entorno de vecindad. La Figura 6-8 muestra un ejemplo de la división de una imagen en cuatro fragmentos considerando un área de solape entre ellos.

Partiendo del tipo de operación y función se busca en la base de datos la información relativa a la planificación del OPI, entre la que destaca el subgrafo de tareas de éste. Además se obtiene si el subgrafo es divisible y el número de fragmentos en que debe dividirse la imagen ($N = N_X \cdot N_Y$). En el caso de que el subgrafo sea divisible, la operación tendrá

asociada una copia del subgrafo leído de la base de datos para cada fragmento en que se divida el proceso de imágenes.

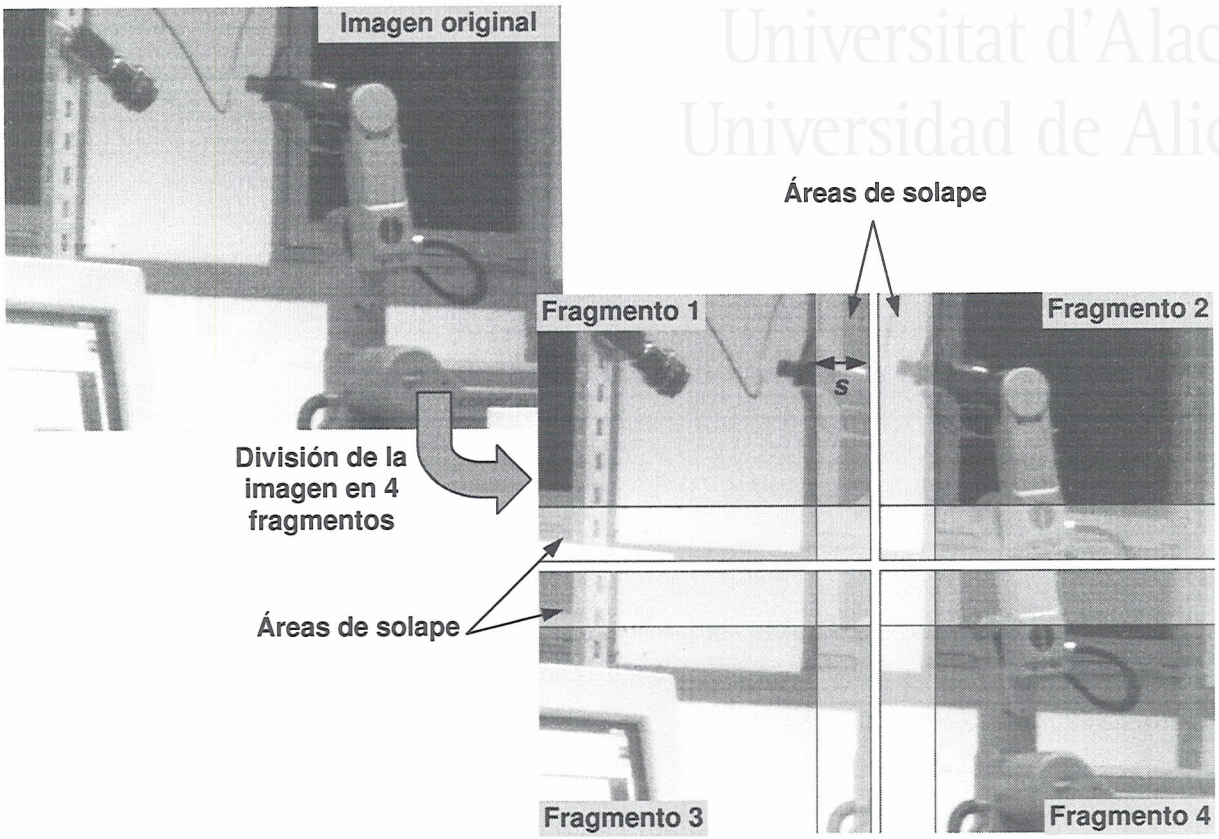


Figura 6-8. Ejemplo de imagen dividida en 4 fragmentos con solape.

Tal y como se esquematiza en la Figura 6-9, se determina el tamaño en bytes a procesar para cada fragmento (*BPF*) en función de las dimensiones medias de las imágenes de entrada (*X*, *Y* y *BPP* o Bytes Por Píxel), del número de fragmentos (*N*), y del tamaño del área de pixels para el solape en ese fragmento (*S*) si es necesaria. Esta área depende de la distancia de solape *s* en pixels, y se considera que existe alrededor de todos los fragmentos, incluidos los que forman parte del exterior de una imagen, y se calcula como el tamaño total del fragmento expandido según el valor *s* menos el tamaño del fragmento sin expandir. El valor de *S* será nulo si *s* es 0, esto es, no hay solape entre los fragmentos.

A continuación se determinan los costes de los nodos y aristas de cada subgrafo (valores $c_{i,j}$), así como de las posibles tareas cpu/tapi que haya que insertar, en función del tamaño de cada fragmento (*BPF*) y los costes relativos del subgrafo leído de la base de datos (valores c_i) dados para cada *Bc* bytes. También se calcula el intervalo de actuación en [0-1]

que corresponde a cada copia de subgrafo que hay que utilizar, teniendo en cuenta que la distribución se realizará en igualdad de condiciones para todos los subgrafos.

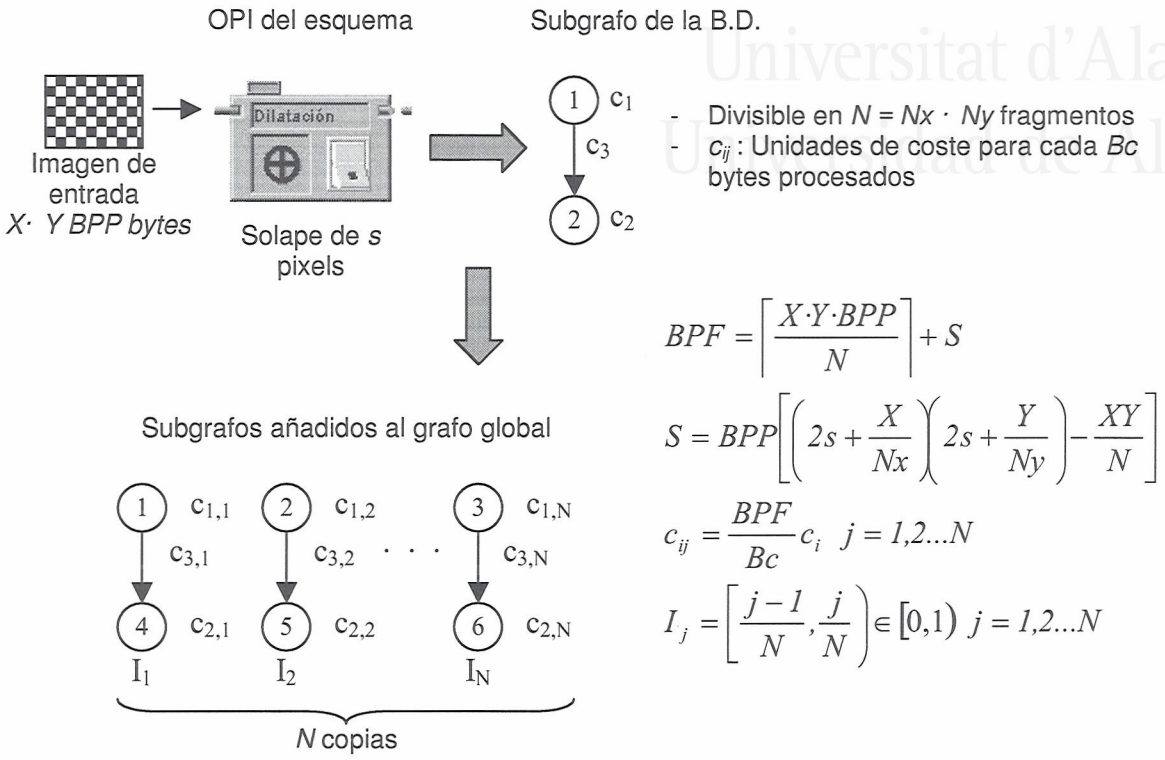


Figura 6-9. Proceso de creación de copias de un subgrafo para un OPI.

Una vez conocido el identificador de cada subgrafo que hay que colocar en el grafo global y ya calculados los intervalos de actuación, se añade al global cada copia del subgrafo. Concluido esto, hay que realizar la conexión con las tareas predecesoras, y para ello hay que añadir aristas en el grafo global teniendo en cuenta de que tareas es sucesora la actual, el tipo de las predecesoras (cpu o tapi) y el tipo de la actual. Aquí se determina si es necesario introducir una tarea cpu/tapi, si el usuario especificó la opción de generar éstas automáticamente. Además si la tarea es divisible se tiene en cuenta el factor de actuación, de tal manera que sólo se conectará con aquellas predecesoras cuya intersección del intervalo de actuación con el propio no sea nula.

Hay que tener en cuenta que las tareas predecesoras al subgrafo del OPI actual ya fueron añadidas al grafo global, al haberse realizado ya este paso para los OPIs predecesores.

Cuando se han añadido las aristas necesarias para el subgrafo, entonces se añade a una lista de sucesores los hijos del OPI en proceso, con un factor de actuación igual al del subgrafo actual. Esta lista permitirá conocer los intervalos de actuación en una próxima ejecución de esta fase.

6.2.3.3. Finalización

Esta fase se realiza cuando ya se ha realizado la fase 2 para todos los OPIs del esquema de trabajo. En ella se almacena en disco el resultado, el grafo global, después de reordenar sus nodos. También se cierra la conexión con la base de datos. El grafo queda disponible en un archivo para ser utilizado por el planificador, o para ser editado antes por la aplicación de editor de grafos descrita en el punto 6.4.1. El formato del archivo es el mostrado en la Figura 6-6.

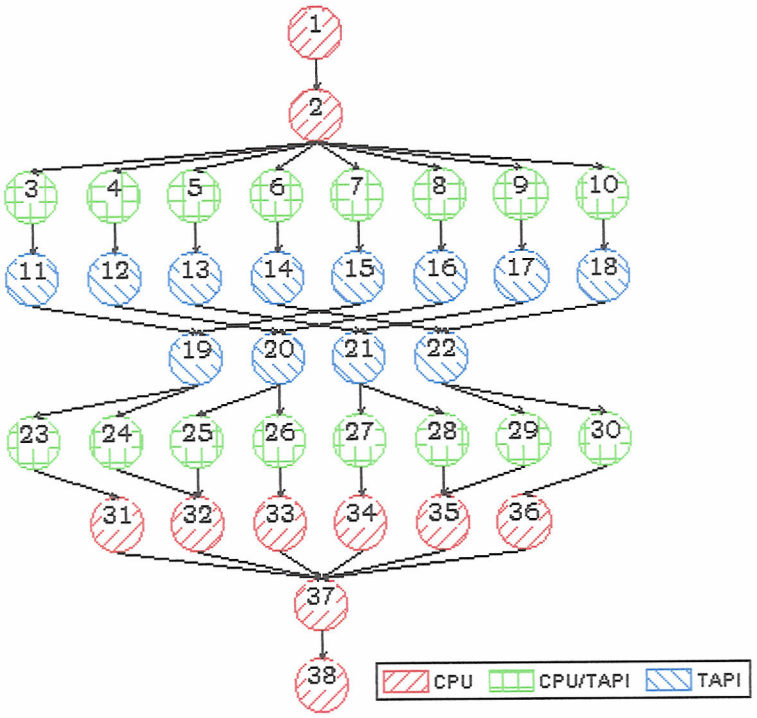
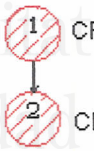
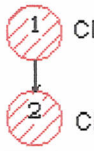


Figura 6-10. Grafo de tareas obtenido a partir del esquema de la Figura 6-3.

Como ejemplo, la Figura 6-10 muestra el grafo obtenido a partir del esquema de la Figura 6-3 con la información de la base de datos de la figura Figura 6-5. Todas las imágenes procesadas son RGB de 160x100 pixels, esto es, de tamaño $160 \cdot 100 \cdot 3 = 48000$ bytes, siendo este también el tamaño medio de imagen N. La información utilizada de la base de datos para este ejemplo se resume en la siguiente tabla.

Operación	Número fragmentos	Pixels de solape	Grafo
Lectura de disco	No divisible	0	
Erosión	N_X=2, N_Y=2, N=4	2	
Dilatación	N_X=2, N_Y=2, N=4	2	
Diferencia	N_X=2, N_Y=2, N=4	0	
Binarizar	N_X=2, N_Y=3, N=6	0	
Escritura en disco	No divisible	0	

La correspondencia entre las operaciones de alto nivel de esquema de la Figura 6-3 y las tareas del grafo de la Figura 6-10, así como el tamaño de los fragmentos procesados, se muestra en esta otra tabla.

Operación del esquema	Lectura de disco	Erosión	Dilatación	Diferencia	Binarizar	Escritura en disco
Tareas	1, 2	11-14	15-18	19-22	31-36	37-38
Subgrafos generados (N)	1	4	4	4	6	1
Bytes de solape en cada fragmento (S)	0	1608	1608	0	0	0
Bytes procesados por subgrafo (BPF)	48000	12000+1608	12000+1608	12000	8000	48000

Finalmente queda destacar las tareas 3 a 10 y 20 a 30 del grafo resultante, que son las tareas de comunicación CPU/TAPI insertadas automáticamente.

6.2.4. La simulación dentro del entorno visual

Para cada clase de OPI existe un módulo de código encargado de realizar las operaciones sobre una o más imágenes de entrada para generar nuevas imágenes. Los módulos son actualmente archivos ejecutables almacenados en un directorio de módulos. En la base de datos se especifica el nombre del archivo de código para cada OPI.

El hecho de trabajar con archivos ejecutables presenta un problema de velocidad en la simulación realizada en el entorno visual, que pude resultar un poco lenta. Sin embargo, ya que se trata de una simulación previa a la ejecución real del algoritmo, normalmente ejecutada paso a paso (u OPI a OPI) para comprobar el correcto funcionamiento de un algoritmo o realizar su depuración, la velocidad no es habitualmente el principal objetivo.

En cambio, los módulos ejecutables presentan dos grandes ventajas: se pueden utilizar independientemente del entorno visual, desde la línea de comandos del S.O. y pueden ser recompilados para utilizarlos en otros sistemas, como por ejemplo Unix o Linux. Además, de este modo resulta más fácil que gente no muy experimentada en programación para M.S Windows pueda desarrollar nuevos módulos.

Otra ventaja derivada del uso de módulos externos es que resulta posible disponer de diferentes conjuntos de módulos para distintos objetivos; simulación en docencia, trabajo sobre hardware de adquisición o procesamiento con bibliotecas propias, interprete programado sobre una máquina con otro S.O., etc [30].

A continuación se resume la forma en que el entorno visual simula un esquema. Cuando el usuario activa el botón de ejecución de un OPI y éste tiene disponible todas las entradas, se identifica el módulo correspondiente y se ejecuta. Al módulo se le pasan como datos de entrada las referencias o nombres de las imágenes a operar, unas nuevas referencias para las imágenes resultado, y los parámetros de configuración del OPI especificadas por el usuario. Hay que destacar que la aplicación del entorno visual no intercambia imágenes con los módulos salvo en el caso de la visualización de resultados. Las imágenes se guardan en un directorio temporal al que acceden directamente los módulos, y el entorno visual solo intercambia con estos referencias a las imágenes de entrada o resultado.

Por otra parte, cuando el entorno visual recibe un aviso sobre el fin de la ejecución de un módulo, comprueba que no hubo errores, y de ser así envía las referencias de las imágenes resultado a los OPIs sucesores. Estos marcan las entradas recibidas para conocer cuales están

disponibles a la hora de su ejecución. Este modo de trabajo también permite al entorno visual ejecutar OPIs en paralelo, ya que el entorno visual puede generar diversas peticiones de ejecución para OPIs cuyas entradas sean independientes antes de recibir los avisos de fin de ejecución . Esto siempre que el S.O. lo permita y no bloquee las peticiones de la aplicación.

En cuanto al paso de datos desde el entorno visual a los módulos (siempre es en este sentido), estos se pueden pasan como parámetros de línea de comando o a través de un pequeño archivo de intercambio. Los módulos únicamente devuelven al entorno visual un código de estado tras acabar su ejecución que indica el éxito u el tipo de error. Los distintos flujos de información se representan en la Figura 6-11.

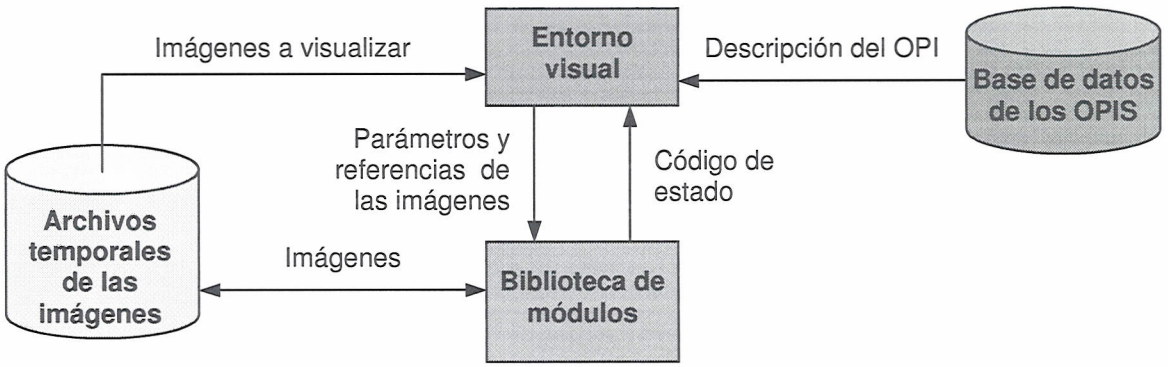


Figura 6-11. Comunicación entre el entorno visual y los módulos.

En el futuro se piensa codificar lo módulos dentro de bibliotecas de enlace dinámico DLL, o controles ActiveX para acelerar la simulación.

6.3. La aplicación del planificador

La aplicación del Planificador de Tareas constituye el último paso en el diseño de sistemas de visión artificial. Dependiendo del hardware disponible se debe calcular una planificación espacial y temporal optima.

6.3.1. Características generales

El planificador diseñado es genérico y realista: un planificador estático que considera interrupción de tareas y tiene en cuenta las relaciones de precedencia. De este modo, se puede

asegurar que ofrecerá una solución factible y posible de llevar la práctica. Uno de los principales objetivos de este planificador es reducir el tiempo final de ejecución de un grupo de tareas, a la vez que se obtiene información sobre la posibilidad de ejecutarlas dentro de ciertos límites de tiempo [30][24].

Se ha optado por la planificación estática porque esta encaja perfectamente en la naturaleza del sistema tratado: un entorno de desarrollo e investigación en visión artificial en el que se genera de antemano el conjunto de tareas a ejecutar y se especifica el hardware disponible. Aún más, en esos sistemas el usuario necesita estimar de un modo fiable los tiempos de ejecución asociados a un cierto esquema, y estos datos no los puede proporcionar una planificación dinámica [8].

Para realizar la planificación se requiere conocer el grafo de tareas en que se divide el sistema especificado en un esquema en el entorno visual, que se genera como ya se ha comentado en el apartado 6.2.3.

Cada tarea del grafo puede pertenecer a uno de los siguientes tres tipos: *cpu*, *tapi* y *cpu/tapi*. Las primeras son tareas a ejecutar un procesador genérico o CPU del sistema destino. Las segundas representan operaciones de bajo nivel a ejecutar en tarjetas de Adquisición y Procesamiento de Imágenes o TAPIs. Las terceras se deben ejecutar en ambos tipos de procesadores y representan un intercambio de información entre estos [8][24].

Las tareas *cpu* o *tapi* pueden ser interrumpibles o no interrumpibles. Las *cpu/tapi* se consideran no interrumpibles. Cada tarea tiene asignado un coste de ejecución y, opcionalmente, unos costes de interrupción y de planificación para tareas interrumpibles. También, se consideran las relaciones de precedencia entre tareas, a las que se les puede asignar un coste, para representar, por ejemplo, una transferencia de datos. La mayor parte de los algoritmos no consideran los deadlines, pero la información suministrada finalmente por estos permite comprobar si se cumplen. Los tiempos de llegada se determinan a partir de las relaciones de precedencia.

6.3.2. Opciones de planificación e información obtenida

El planificador desarrollado permite múltiples posibilidades de algoritmos de asignación y planificación de tareas. El usuario puede elegir entre esas posibilidades de acuerdo a los resultados que desea obtener. Éstas son:

- *Planificación temporal* usando solo un procesador de cada tipo (una CPU y una TAPI). En una planificación muy simple, pero que sirve para establecer comparativas con las otras.
- *Planificación temporal considerando una asignación espacial* previa realizada manualmente. Requiere que se especifique cuantos procesadores y de que tipo hay disponibles (*hardware* disponible).
- *Asignación espacial basada en una planificación temporal previa* calculada para un solo procesador. También requiere que se especifique cuantos procesadores de cada tipo hay disponibles.
- *Planificación y asignación simultáneas*. Estos métodos ofrecen los mejores resultados porque consideran las posibles relaciones entre las distribuciones temporal y espacial.
- Obtener información sobre el *máximo número de procesadores* de cada tipo necesario para una ejecución del menor tiempo posible.

En los siguientes puntos se resumirán las principales características de cada uno de los algoritmos disponibles, que ya han sido descritos con detalle en el capítulo 3.

Con cualquiera de las opciones anteriores, la aplicación ofrece la siguiente información sobre los resultados de la planificación:

- *Gráficas de distribuciones espacial y temporal.* Muestran las tareas que se ejecutan en cada procesador y sus tiempos de inicio y final.
- *Utilización de procesadores.* Indica como de ocupado está cada procesador, es decir, el porcentaje de tiempo que no está parado sin operar.
- *Utilización del sistema.* Tiempo medio de utilización de los procesadores.
- *Tiempo de ejecución del sistema.* Tiempo necesario para ejecutar por completo el esquema propuesto.
- *Grafo de tareas.* Es el grafo empleado para realizar la planificación.
- *Información sobre las tareas.* Hace referencia a la información relativa a cada tarea, mostrándose solo los grupos seleccionados por el usuario.

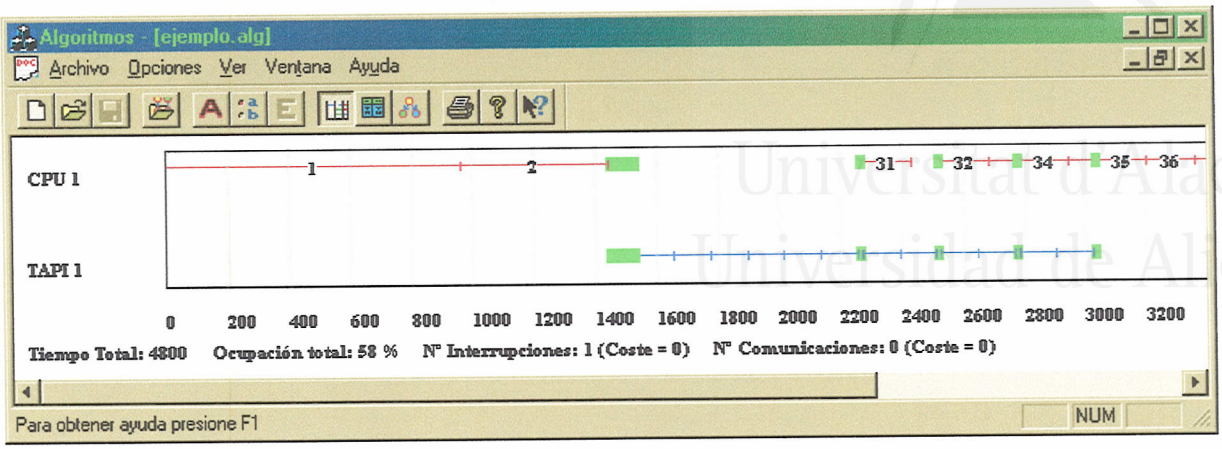


Figura 6-12. Resultado del planificador para el grafo de tareas de la Figura 6-10.

Finalmente se debe destacar que, una vez que se ha realizado una simulación, la aplicación permite obtener los resultados de otra simulación partiendo de los mismos datos de entrada, pero empleando un algoritmo diferente o modificando el hardware disponible.

Como ejemplo, la Figura 6-12 muestra la gráfica con distribución de las tareas generada por el planificador para el grafo de tareas de la Figura 6-10. Se ha considerado un sistema destino con una CPU y una TAPI y utilizando una planificación espacio-temporal con interrupción de tareas. Los pequeños intervalos de tiempo con trazo grueso representan las tareas cpu/tapi, que requieren de ambos procesadores a la vez.

6.3.3. Algoritmo de planificación temporal

Este algoritmo solo realiza una asignación temporal de las tareas a los distintos procesadores, y requiere que la distribución espacial se realice previamente a mano o con otros algoritmos. Tampoco considera el coste de comunicación asociado a las relaciones de precedencia entre tareas.

- *Planificación basada en el Camino crítico.* Planifica primero las tareas que más pueden ralentizar al sistema, es decir, las que pueden hacer que el sistema tarde más en concluir. Con ello se puede lograr reducir el tiempo de ejecución. Considera interrupción de las tareas (aunque con coste nulo) y por ello puede obtener planificaciones con mayor paralelismo y mejores que las ofrecidas por los dos algoritmos anteriores.

El primer algoritmo ha sido obtenido de [6] . El tercero en una versión mejorada del expuesto en [8].

6.3.4. Algoritmos de asignación espacial

Se basan en una planificación temporal previa sobre un procesador de cada tipo (CPU o TAPI), a partir de la cual se realiza una organización espacial de las tareas sobre distintos procesadores según las características de estas y sus relaciones de precedencia. No consideran costes de comunicación ni de interrupción, aunque si permiten tareas interrumpibles si el algoritmo usado para la planificación temporal previa también lo hace. En tal caso, los segmentos de una misma tarea interrumpida se asignan al mismo procesador. El resultado de estos algoritmos de asignación dependerá también de lo bueno que sea el algoritmo empleado para la planificación previa.

- *Espacial con huecos.* Busca una organización espacial óptima de las tareas sobre los dos tipos de procesadores, sin reparar en el número necesario de ellos para concluir la ejecución en el menor tiempo posible. El algoritmo ofrece así el número máximo de procesadores útiles requeridos (más procesadores no aportarán una mejora).
- *Espacial sin huecos.* Basándose en el algoritmo anterior consigue una asignación mejor, aunque con mayor coste computacional, gracias a que trata de evitar huecos sin ejecución en un procesador entre dos tareas. Ofrece planificación con mínimo tiempo de ejecución y el número máximo de procesadores útiles requeridos.
- *Espacial con límite de procesadores.* Versión del algoritmo sin huecos en la que se limita el número de procesadores disponibles para la asignación de tareas, resultando más realista. Cuando no se puede organizar una tarea en los procesadores actuales y no hay más procesadores libres, se asignará dicha tarea al primer procesador que quede libre, es decir, el que tenga la tarea actual con menor tiempo restante de ejecución.

Los tres algoritmos de asignación espacial son originales de este trabajo.

6.3.5. Algoritmos de planificación temporal y asignación espacial

Los algoritmos descritos a continuación permiten una planificación temporal y una asignación espacial de las tareas. Todos consideran la existencia de distintos tipos de procesadores y la posibilidad de interrumpir una tarea. Excepto el último, consideran nulo el coste de comunicación entre tareas.

- *Espacio-temporal.* Versión de algoritmo de planificación temporal de camino crítico que también realiza una asignación espacial de tareas. No se repara en el número de procesadores necesarios, y como resultado ofrece el número de procesadores máximo requerido a partir del cual no hay mejora de tiempo, además de dar el tiempo mínimo de ejecución.
- *Espacio-temporal con límite de procesadores.* En este otro nuevo algoritmo se considera un limite en el número de procesadores. En el momento de planificar se selecciona la tarea que puede retardar más el sistema para asignarla al primer procesador que quede libre, si no hay procesadores libres.
- *Espacio-temporal con interrupciones.* Versión del anterior que considera los costes derivados de la interrupción de tareas. Se dispone de dos variantes, una que considera dichos costes constantes para cada tarea, y otra más realista que determina los costes de interrupción según el instante en que se producen dada una función de costes especificada para cada tarea.
- *Espacio-temporal con coste de comunicación.* Versión del algoritmo espacio-temporal con límite de procesadores que tiene en cuenta además los costes de comunicación entre tareas. Para ello considera que el grafo de entrada no tiene tareas de comunicación cpu/tapi, y éstas son insertadas por el algoritmo cuando resulta necesario con el conste correspondiente.

6.4. Otras herramientas

En este último apartado se comentan brevemente otras herramientas destacables incluidas en el entorno desarrollado; un editor de grafos de tareas, un visualizador de imágenes remoto.

6.4.1. El editor de grafos

Dentro del entorno de trabajo, el usuario también dispone de una herramienta para la creación y edición de grafos de tareas. Ésta permite diseñar gráficamente el grafo de forma sencilla, así como especificar los parámetros para las tareas (tipo de procesador, costes de ejecución e interrupción, función de costes de interrupción, descripción...) y las relaciones de precedencia con sus costes de comunicación. También realiza la reordenación de los nodos del grafo necesaria para que los algoritmos de planificación funcionen correctamente (ver punto 3.4). La aplicación fue desarrollada en el proyecto referenciado en [24]. La Figura 6-13 muestra el aspecto de la aplicación, en la que se están editando dos grafos de tareas.

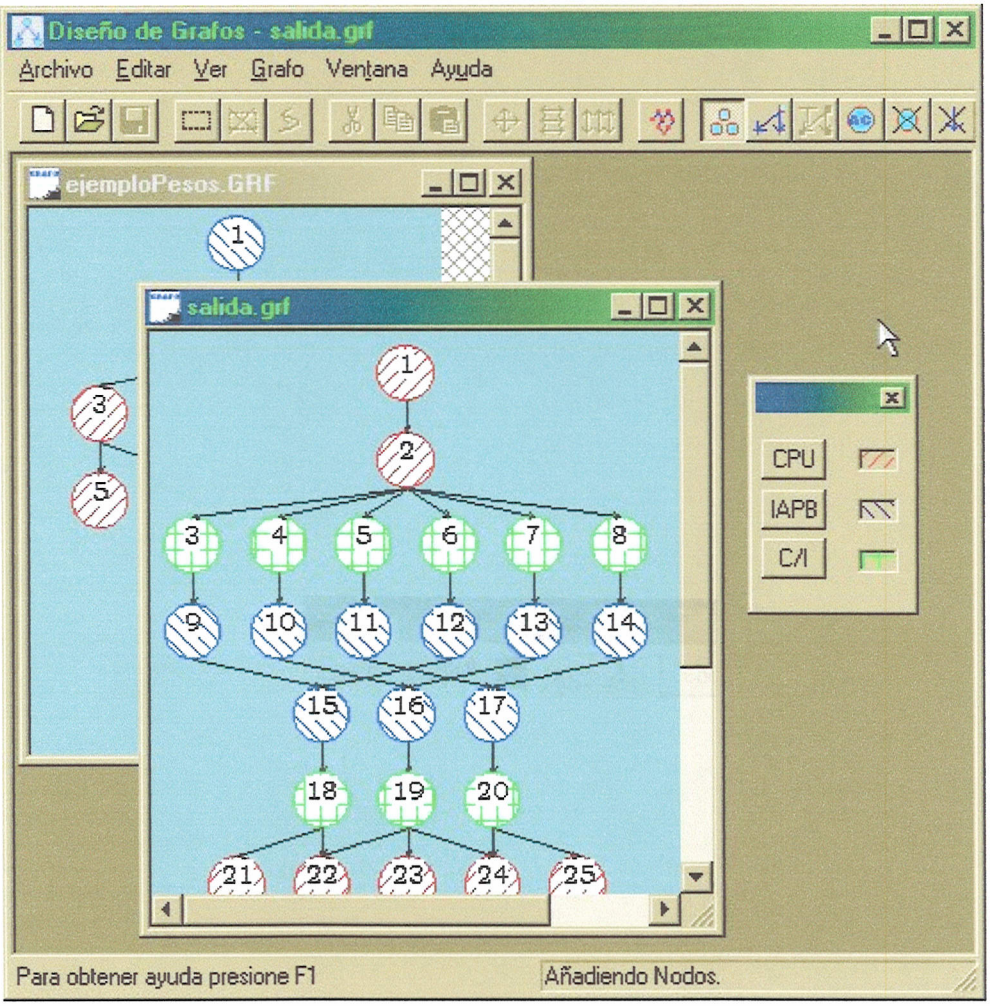


Figura 6-13. Aspecto del editor de grafos de tareas.

Una característica destacable que se ha incluido en la aplicación es la opción de guardar un grafo de tareas dentro de la base de datos junto con la información relativa a su

OPI, permitiendo que el proceso de especificación de subgrafos de tareas para nuevos OPIs sea bastante sencillo.

6.4.1.1. El visualizador remoto

Dentro del entorno EVA [29], se desarrollo una herramienta de visualización remoto, que puede ser utilizada junto con el entorno visual.

El visualizador remoto es una aplicación servidor que atiende conexiones IP-TCP desde aplicaciones clientes, las cuales pueden envían imágenes para que sean mostradas en la máquina donde se ejecuta el servidor. El entorno visual dispone de un OPI especial que funciona como cliente del visualizador distribuido.

Este visualizador remoto tiene muchas utilidades. Por ejemplo, permite mostrar las imágenes en los monitores de otros equipos dentro de la misma subred, con lo que se dispone de todo el área de escritorio para dibujar y simular esquemas. También se puede utilizar para realizar exposiciones o conferencias donde cada asistente con su propio equipo puede ver las imágenes, o para compartir resultados dentro de un grupo de trabajo.

La aplicación del visualizador actúa en segundo plano dentro M.S. Windows 9x, y se puede acceder a su configuración mediante un icono en la barra de tareas como el mostrado en la Figura 6-14. Las imágenes son mostradas de igual forma que por el visualizador incluido en el entorno visual (Figura 6-4).

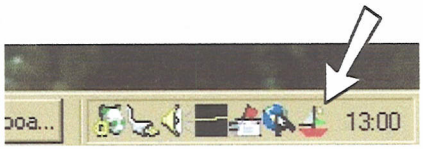


Figura 6-14. Icono del servidor del visualizador distribuido.

6.4.2. Edición e inserción de tipos de OPIs

El editor de grafos permite añadir información sobre las tareas elementales de una operación de un OPI a la base de datos. En cuanto a la información sobre el aspecto y simulación, aunque se puede trabajar directamente sobre los archivos de base de datos, en este momento se está creando una aplicaciones para facilitar a un usuario la incorporación de nuevos OPIs y sus módulos al entorno visual.

Básicamente esta aplicación permite:

Volver al índice/Tornar a l'índex

- Definir los parámetros de entrada y diseñar el aspecto del dialogo de configuración del OPI. Ambos aspectos están muy ligados. También se especifica los posibles números de entras y salidas y una descripción de las operaciones que realiza el OPI.
- Especificar el icono que representa al OPI, así como la ubicación del archivo de su modulo de código.
- Editar el árbol de clases de OPIs y operaciones, para incluir nuevas clases o nuevos OPIs, o modificar su estructura.

Capítulo 7

Aplicación a un algoritmo de correspondencia de dos imágenes estereoscópicas

---

7.1. Introducción

El presente capítulo muestra los resultados que ofrecen los distintos algoritmos estudiados en los capítulos 3, 4 y 5, utilizando el entorno de aplicaciones descrito en el capitulo 6. Para ello se va a considerar un algoritmo de visión por computador como ejemplo, que será especificado en el entorno propuesto y al que se aplicarán los distintos algoritmos de planificación, para evaluar los resultados.

El algoritmo escogido para realizar las pruebas es uno de visión estéreo que realiza la correspondencia entre las características obtenidas a partir de las imágenes adquiridas por dos cámaras. Se ha decidido escoger este algoritmo completo y real antes que algoritmos sencillos ficticios o sin aplicación práctica para que los resultados de las pruebas sea también realistas.

Dentro de este capítulo, el siguiente apartado se dedica a describir el algoritmo escogido, sin pretender profundizar en los aspectos técnicos del algoritmo ni justificar la elección de determinadas técnicas en él, ya que no son objeto de esa tesis. Seguidamente, el apartado 7.3 mostrará cuales son y de que forma se especifican las características de las tareas que componen las operaciones del algoritmo, para obtener el grafo de tareas a planificar. Finalmente, en el apartado 7.4 se mostrarán, evaluarán y compararán las planificaciones que los distintos algoritmos propuestos generan para el ejemplo.

7.2. Algoritmo de correspondencia de dos imágenes estereoscópicas

Como se ha comentado en la introducción, se considerará un algoritmo de correspondencia para visión estéreo que toma dos imágenes de entrada de escalas de grises procedentes de dos cámaras CCD y realiza la correspondencia entre las características de ambas. A continuación se explicará el funcionamiento del algoritmo. Para ello se presentarán primero las operaciones de más alto nivel, que después se irán desarrollando hasta llegar a las más básicas.

La Figura 7-1 muestra un esquema del entorno visual descrito en el capítulo 6 con las operaciones de más alto nivel del algoritmo. Se pueden observar dos entradas, dadas por las operaciones de captura de las imágenes izquierda y derecha. Como primer paso, a cada imagen se le aplica una serie de detectores que generan tres tipos de descriptores de características para cada imagen: esquinas, bordes y regiones. El funcionamiento de estos detectores se describe en el punto 7.2.1. Para cada conjunto de datos de una de estas características se aplica a continuación una serie de técnicas de modelado, correspondencia y verificación.

En el esquema, los OPIs tipo "FUNC" representan bloques que contienen otros OPIs y cuyo funcionamiento se resume a continuación. Los OPIs con el dibujo de los engranajes son nuevas operaciones definidas para este algoritmo.

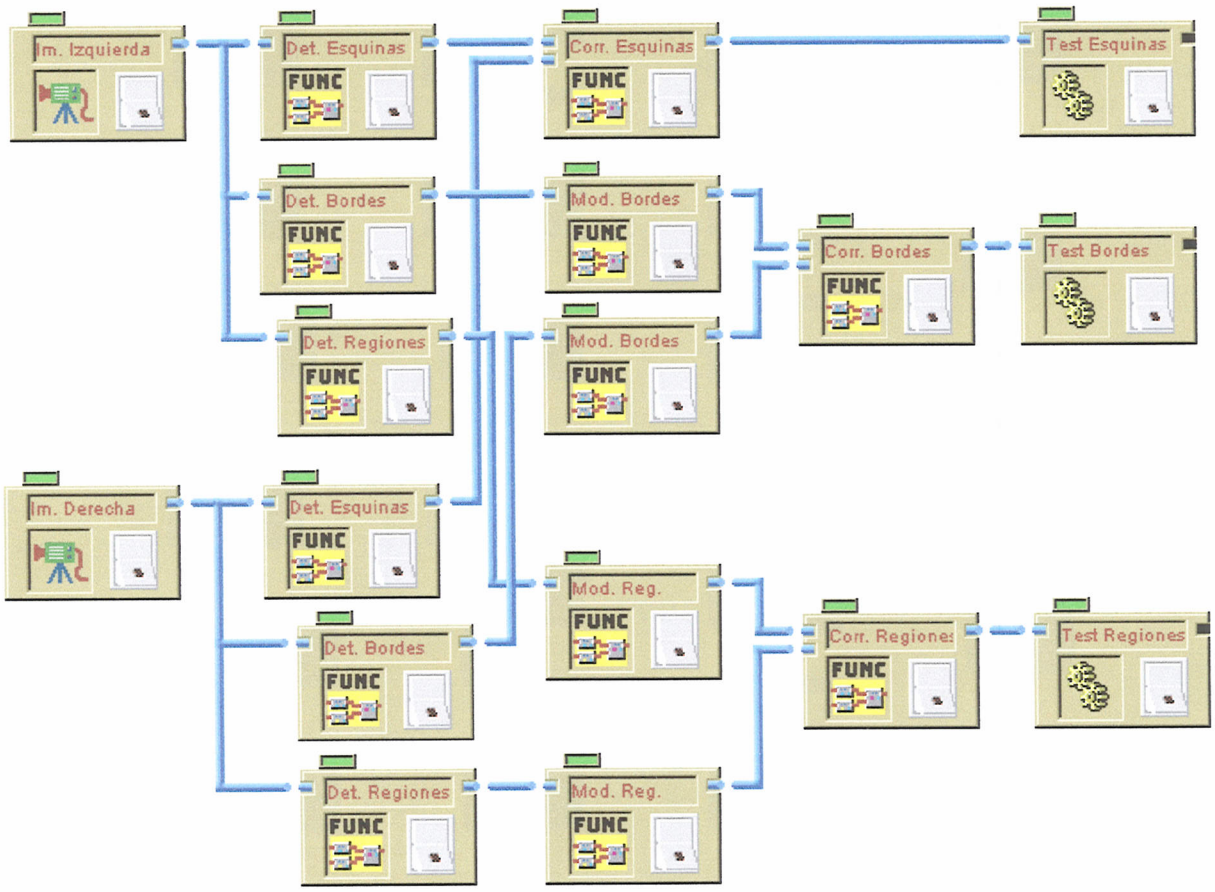


Figura 7-1. Esquema del algoritmo de correspondencia.

En el caso de las esquinas, los datos procedentes de la detección se aplican a una operación de correspondencia de esquinas que básicamente realiza tres pasos: generar pares de puntos en las dos imágenes, reducir el número de pares imponiendo unas restricciones, y

buscar e identificar emparejamientos entre imágenes. El resultado es un conjunto de datos que valora la probabilidad de correspondencia de cada punto de una imagen con la otra. En el punto 7.2.2 se detallarán mejor los pasos de que consta la correspondencia. Finalmente se realiza un test para eliminar las correspondencias de baja probabilidad.

Para los bordes, antes de aplicar una técnica de correspondencia resulta necesaria una operación de modelado de los datos obtenidos de los detectores, la cual pretende identificar y numerar los bordes detectados. Los conjuntos de datos generados por las operaciones de modelado se aplican a una operación de correspondencia de bordes. Ésta, que se describe con más detalle en el apartado 7.2.3, genera emparejamientos de bordes y determina la similitud entre segmentos. Los resultados de la correspondencia se clasifican con un test según su probabilidad.

Finalmente, para las regiones, también resulta necesario una etapa intermedia de modelado antes de realizar la correspondencia. Las regiones se modelan según descriptores de frontera, de región y de tamaño. Los datos generados se aplican a una operación de correspondencia que en primer lugar verifica ciertas restricciones geométricas y después se eligen correspondencias entre pares de regiones de las dos imágenes que cumplen las restricciones. Estas operaciones se describen mejor en el punto 7.2.4. Por último, se realiza un test que elimina las correspondencias de baja probabilidad.

La Figura 7-2 muestra el esquema desarrollado del algoritmo de ejemplo al completo, donde aparecen todas las operaciones, las cuales se describirán en los siguientes apartados.

7.2.1. Los detectores de características

A continuación se detallan un poco más los pasos que realizan cada uno de los tres detectores empleados.

7.2.1.1. Detector de esquinas

La detección de esquinas para una imagen conlleva los pasos representados por el esquema de la Figura 7-3. En primer lugar se realiza un suavizado de la imagen con un filtro gaussiano. A continuación una operación que genera un filtro que identifica el tipo de detector utilizado, el cual se convoluciona finalmente con la imagen suavizada para obtener los cambios bruscos de luminancia que determinan los puntos de esquina.

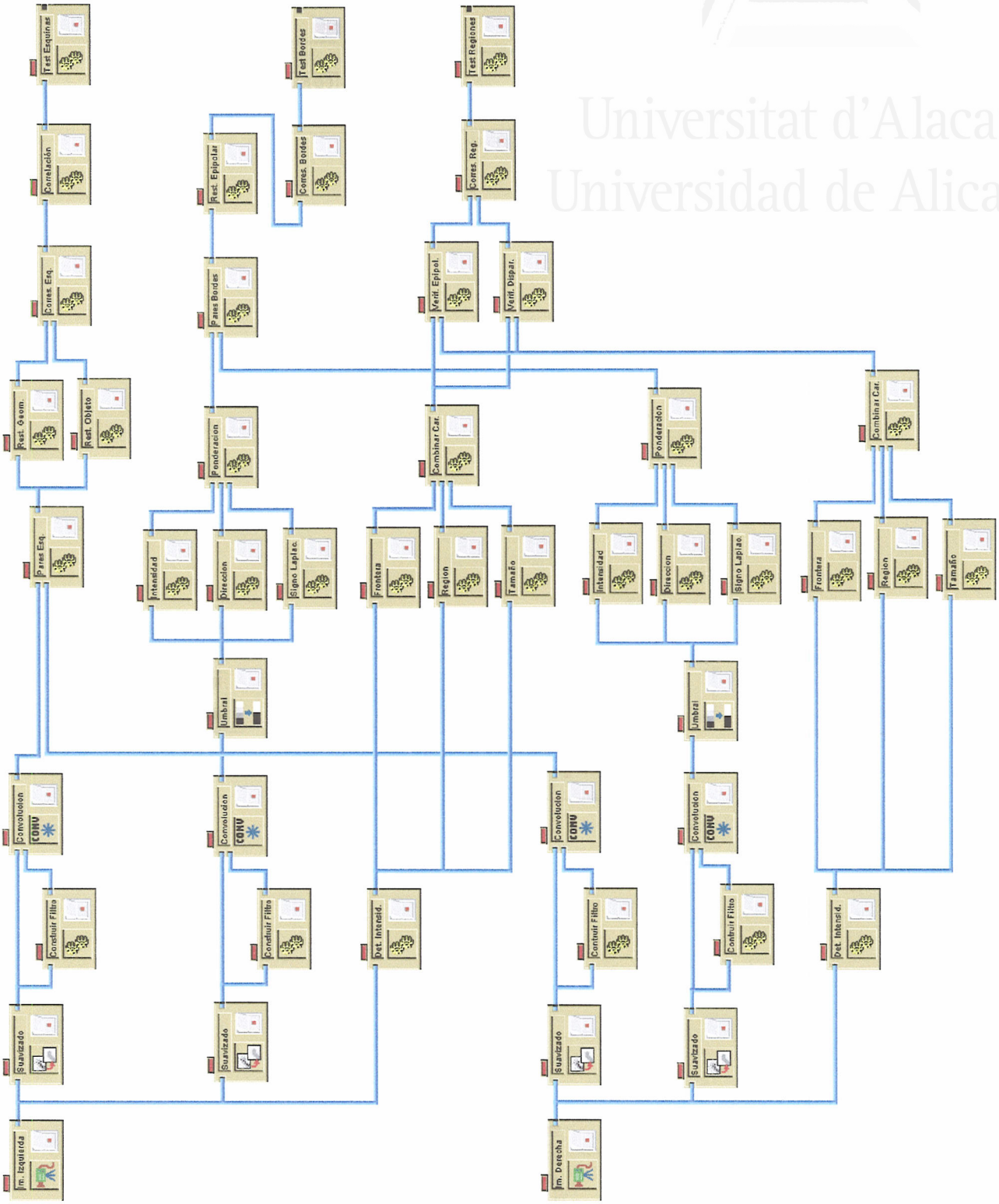


Figura 7-2. Esquema desarrollado del algoritmo de correspondencia

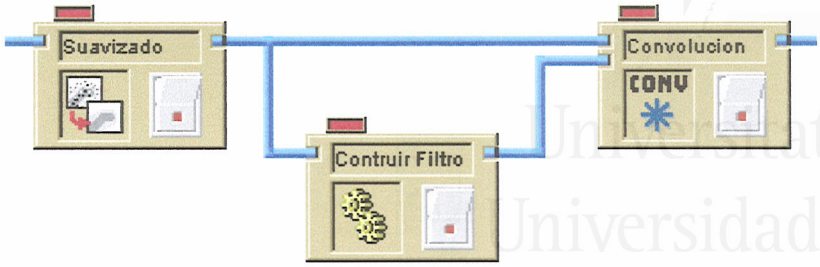


Figura 7-3. Proceso de detección de esquinas.

7.2.1.2. Detector de bordes

Como detector de bordes para una imagen se emplea el mostrado en el esquema de la Figura 7-4, que consta de las siguientes etapas. Primero, la imagen se suaviza con un filtro gaussiano. Con la imagen suavizada se determina el filtro adecuado para este detector, el cual se convoluciona con la imagen suavizada para obtener los bordes. El resultado pasa por una umbralización que elimina los bordes poco significativos en la imagen.

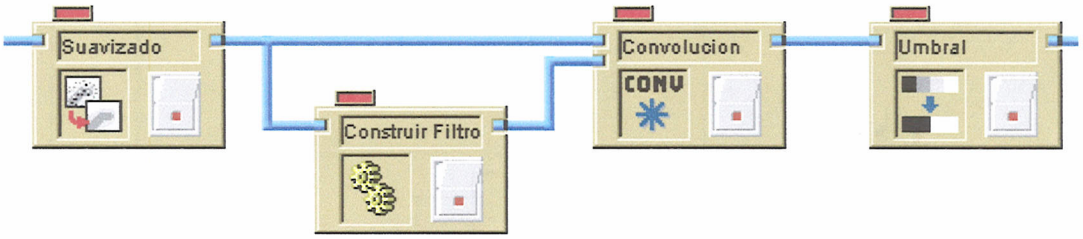


Figura 7-4. Proceso de detección de bordes.

7.2.1.3. Detector de regiones

La detección de regiones se basa en agrupar los pixels de la imagen conforme a la intensidad de los mismos. El resultado es un conjunto de datos con información sobre cada región. La operación correspondiente se muestra en la Figura 7-5.

Figura 7-5. Proceso de detección de regiones.

7.2.2. Tratamiento de las esquinas

Una vez detectadas las esquinas de las dos imágenes de entrada, éstas se aplican al módulo de correspondencia de esquinas, cuyo proceso, mostrado en la Figura 7-6, se describe a continuación.

En primer lugar, se generan las posibles combinaciones de pares de puntos entre ambas imágenes. A continuación, para reducir el número de parejas de puntos y la ambigüedad, se eliminan las combinaciones que no cumplen unas restricciones geométricas y de relación objeto-entorno.

Las restricciones geométricas se basan en la epipolaridad entre imágenes (dado un punto en una imagen, su correspondiente en la otra de estar en la línea epipolar), y en unos límites de disparidad que imponen un rango de profundidad para los objetos de la escena captada. Las restricciones objeto-entorno hacen referencia a la semejanza de profundidad entre pares vecinos, a la ordenación de las proyecciones de los puntos de la superficie de los objetos en las dos cámaras, que deben ser la misma, y a la forma del objeto respecto al sistema de visión (gradiente de disparidad).

Según el grado en que las parejas de puntos cumplen las restricciones se escogen las mejores como posibles correspondencias. A éstas se les aplica una técnica de correlación para determinar la probabilidad de correspondencia de cada punto de una imagen con la otra.

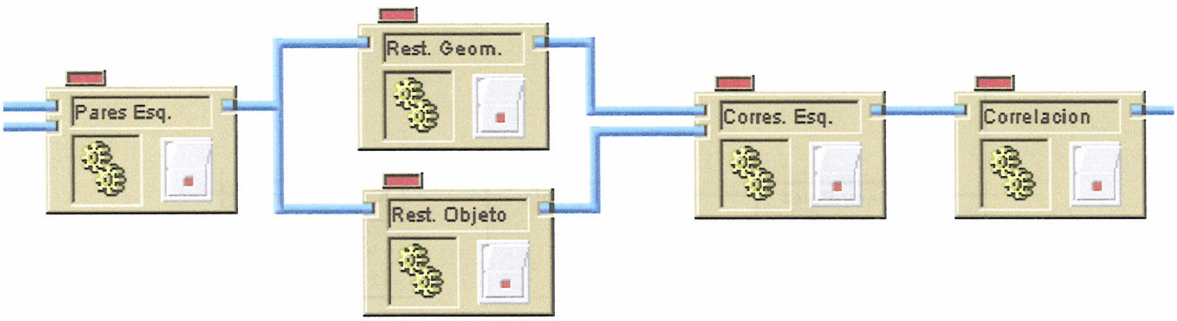


Figura 7-6. Proceso de correspondencia de esquinas.

7.2.3. Tratamiento de los bordes

En primer lugar, para cada una de las dos imágenes con los bordes detectados se requiere un modelado de los mismos que se encargue de identificarlos y numerarlos. Además permite incorporar información adicional que facilite luego la correspondencia. El modelado

de cada borde se realiza conforme unas características como los niveles de intensidad de sus contornos, la dirección del gradiente en sus contornos o el signo de la laplaciana a sus lados. Todas estas características se evalúan y ponderan, según muestra la Figura 7-7.

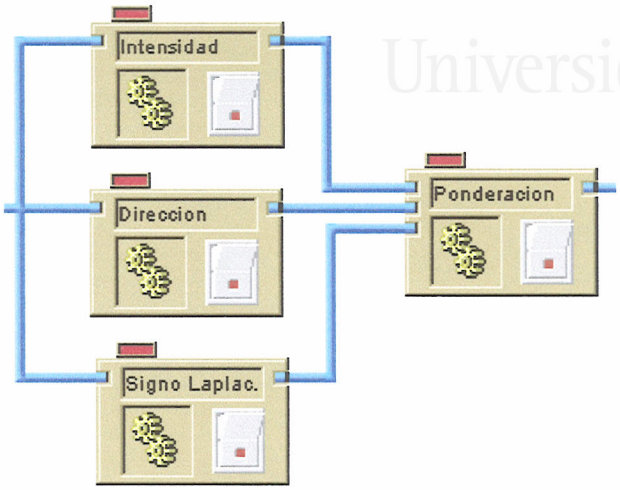


Figura 7-7. Proceso de modelado de bordes.

A partir de los conjuntos de datos con los modelos de los bordes de ambas imágenes, se realiza la correspondencia siguiendo estos pasos. Primero se forman posibles emparejamientos de los bordes entre las imágenes. Después se eliminan los emparejamientos que no superan una restricción epipolar para simplificar su número. Finalmente se realiza una correspondencia entre cada borde de una imagen con respecto a los de la otra, evaluando para ello la similitud entre sus segmentos conforme a la orientación y longitud de los mismos. La correspondencia asigna una probabilidad a cada borde de una imagen con respecto a los de la otra que indica como de semejantes son. El proceso se detalla en el esquema de la Figura 7-8.

Figura 7-8. Proceso de correspondencia de bordes.

7.2.4. Tratamiento de las regiones

Como en el caso del tratamiento de bordes, para la regiones también resulta necesario un modelado previo antes del proceso de correspondencia. Así, para cada imagen se aplica una etapa de modelado que genera una representación de las regiones según descriptores de

frontera mediante códigos de cadena, de región (momentos, longitud de los ejes...) y de tamaño (áreas, perímetros, ...). En la Figura 7-9 se muestra un esquema de estas operaciones.

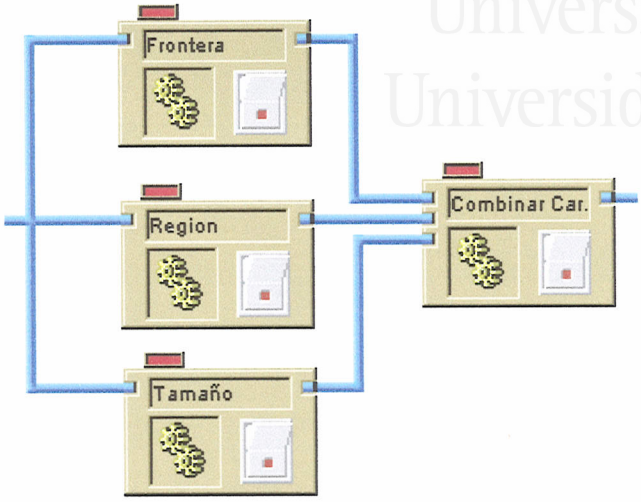


Figura 7-9. Proceso de modelado de regiones.

Ya obtenidos los conjuntos de datos con los modelos de las regiones de ambas imágenes, se aplica a estos el proceso de correspondencia, esquematizado en la Figura 7-10. Éste consta en primer lugar de un bloque de restricciones que trata de identificar posibles emparejamientos entre las regiones de ambas imágenes, eliminando las parejas que no cumplen unos criterios que hacen referencia a restricciones geométricas de epipolaridad y la disparidad máxima. Los pares que superan las restricciones forman el conjunto de datos al que se aplica la correspondencia. Ésta consiste en asignar una probabilidad de correspondencia de cada región de una imagen con la de la otra, conforme a la similitud entre las características de ambas y considerando que su distancia euclídea debe ser mínima.

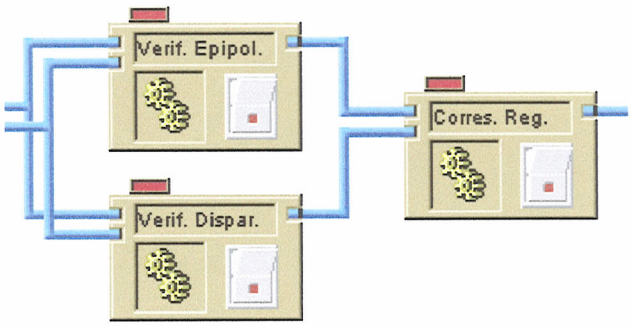


Figura 7-10. Proceso de correspondencia de regiones.

7.3. Especificación de la aplicación

Antes de nada es necesario crear los módulos de las operaciones especificas del algoritmo de visión estéreo de ejemplo y darlos de alta en el entorno visual. A continuación ya se puede especificar el ejemplo como esquemas de OPIs en el entorno visual, obteniendo como resultado el esquema de la Figura 7-2.

Seguidamente, para poder realizar la planificación resulta imprescindible definir las características de las tareas de cada operación en la base de datos. Finalmente, se configura el entorno visual para generar el grafo de tareas que sirve como entrada a los algoritmos de planificación. Estos dos pasos se describen en los puntos siguientes de este apartado.

7.3.1. Hardware disponible

El sistema de destino para el algoritmo de ejemplo consta de dos TAPIs y CPU. Las TAPIs son dos tarjetas Genesis, de la casa Matrox, que incorporan el procesador NOA, un acelerador para las operaciones de vecindad. Como procesador CPU se considera un PC Pentium III a 750 MHz.

La conexión de las TAPIs con la CPU se realiza mediante la interfaz PCI, según la especificación original de este bus que permite un ancho de banda de 132 MBytes/s. Además las TAPIs se conectan entre sí mediante el bus VMChannel, que permite transferencias de hasta 132 Mbytes/s.

7.3.2. Características de las operaciones

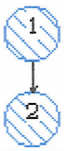
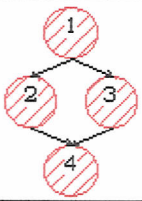
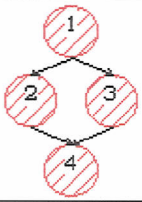
Dentro de la base de datos debe indicarse el subgrafo de tareas asociado a cada operación y los fragmentos en que se dividen los datos si la operación es divisible (ver punto 6.2.2). Para cada subgrafo se especifican los costes de ejecución asociados a cada tarea, los costes de cada comunicación, los costes de interrupción, etc.

Para estimar los costes temporales de las tareas tapi se han utilizado las tablas de tiempo estimados o *benchmarks* que suministra Matrox con la documentación de la tarjeta Genesis. A partir de las ellas se han obtenido los tiempos de computo de dichas tareas, considerando que se trabaja con imágenes de 512 x 512 pixels de 1 byte por píxel (escala de

grises). En cuanto a las tareas cpu, se han realizado pruebas de las operaciones sobre el hardware disponible para estimar los tiempos relativos a las tareas cpu.

Para obtener los costes de comunicación, se considera que las tareas cpu/tapi están determinadas por el interfaz PCI y su ancho de banda de 132 Mbytes/s, Esto supone un coste aproximado de 2ms para una imagen de 512 x 512 pixels con 1 byte por pixel. Lo mismo ocurre para las tareas tapi/tapi relativas al bus VMChannel.

A continuación se muestra una tabla que resume la información de las operaciones y sus tareas elementales.

Nombre de la operación	T/F[1]	Subgrafo[2]	c_i[3]	E/L[4]	Frag.[5]
Capturar imagen	1001/0		c_1=9	-	1
Suavizado	1002/0		c_1=19	-	1
Construir filtro	1002/1		c_1=114 c_2=28	5/5 -	1
Convolución	1002/2		c_1=12	-	1
Umbralizar	1003/0		c_1=1	-	2
Detectar regiones por intensidad	1005/0		c_1=118	5/5	1
Generar pares de esquinas	1006/0		c_1=80	5/5	1
Restricciones geométricas para las esquinas	1006/1		c_1=1 c_2=1 c_3=2 c_4=1	1/1 1/1 1/1 1/1	1
Restricciones de objeto para las esquinas	1006/2		c_1=1 c_2=1 c_3=2 c_4=2	1/1 1/1 1/1 1/1	1

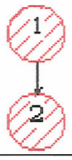
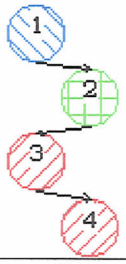
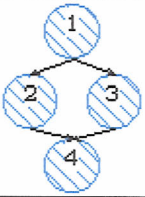
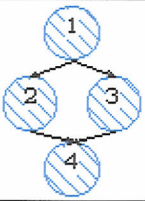
[1] Tipo y función de la operación: Identifican cada operación en la BD.

[2] Tareas *tapi* en azul, *cpu* en rojo, y *cpu/tapi* en verde.

[3] Tiempo de computo requerido por cada tarea en ms.

[4] Costes constantes de escritura y lectura por interrupción en ms. '-' indica que no es interrumpible.

[5] Fragmentos o subgrafos en que se divide la operación (Nx· Ny).

Nombre de la operación	T/F[1]	Subgrafo[2]	c_i[3]	E/L[4]	Frag.[5]
Correspondencia de esquinas	1006/3		c_1=1 c_2=1	1/1 1/1	1
Correlación de esquinas	1006/4		c_1=5 c_2=200 c_3=600	1/1 10/10 10/10	1
Modelado de bordes según intensidad	1007/1		c_1=10	-	1
Modelado de bordes según dirección	1007/2		c_1=10	-	1
Modelado de bordes según signo de la laplaciana	1007/3		c_1=10	-	1
Ponderación de bordes	1007/4		c_1=2	-	1
Generar pares de bordes	1008/0		c_1=12	5/5	1
Restricciones de epipolaridad para los pares de bordes	1008/1		c_1=1	1/1	1
Correspondencia de bordes	1008/2		c_1=1 c_2=1	1/1 1/1	1
Descripción de región por frontera	1009/0		c_1=260 c_2=1 c_3=10 c_4=300	10/10 - 1/1 10/10	1
Descriptores de región	1009/1		c_1=5 c_2=130 c_3=130 c_4=4	- 5/5 5/5 -	1
Descripción de región por tamaño	1009/2		c_1=5 c_2=130 c_3=130 c_4=4	- 5/5 5/5 -	1
Combinar descriptores de región	1009/3		c_1=4	1/1	1
Verificar epipolaridad de regiones	1010/0		c_1=5	1/1	1

Nombre de la operación	T/F[1]	Subgrafo[2]	c_i[3]	E/L[4]	Frag.[5]
Verificar disparidad de regiones	1010/1		c_1=3	1/1	1
Correspondencia de regiones	1010/2		c_1=1	1/1	1
Test para las esquinas	1011/0		c_1=2	1/1	1
Test para los bordes	1011/1		c_1=2	1/1	1
Test para las regiones	1011/2		c_1=2	1/1	1

7.3.3. Grafo de tareas

La Figura 7-11 muestra el grafo de tareas generado por el entorno visual a partir del esquema del algoritmo de correspondecia mostrado en la Figura 7-2, considerando que la base de datos tiene las operaciones indicadas en el punto anterior. A continuación se lista una relación de las tareas del grafo resultante con respecto a las operaciones a las que pertenecen:

Operación	Tareas	Operación	Tareas
Capturar imagen izquierda	1	Capturar imagen derecha	2
Suavizado en det. esquinas izq.	3	Suavizado en det. Esquinas der.	6
Construir filtro det. esquinas izq.	9,19	Construir filtro det. esquinas der.	14,26
Convolución en det. esquinas izq.	33	Convolución en det. esquinas der.	38
Suavizado en det. bordes izq.	4	Suavizado en det. bordes der.	7
Construir filtro det. bordes izq.	10,20	Construir filtro det. bordes der.	15,27
Convolución en det. bordes izq.	34	Convolución en det. bordes der.	39
Umbralizar en det. bordes izq.	43,44	Umbralizar en det. bordes der.	46,47
Mod. bordes intensidad izq.	55	Mod. bordes intensidad der.	58
Mod. bordes dirección izq.	56	Mod. bordes dirección der.	59
Mod. bordes signo izq.	57	Mod. bordes signo der.	60
Ponderación de bordes izq.	78	Ponderación de bordes der.	79
Detectar regiones izquierda	5	Detectar regiones derecha	8
Descripción por frontera izq.	11,21,35,45	Descripción por frontera der.	16,28,40,48
Descripción por región izq.	12,22,23,36	Descripción por der.	17,29,30,41
Descripción por tamaño izq.	13,24,25,37	Descripción por tamaño der.	18,31,32,42
Combinar des. de región izq.	62	Combinar des. de región der.	63
Generar pares de esquinas	61	Verificar epipolaridad regiones	72
Rest. geométricas para esquinas	64,74,75,81	Verificar disparidad de regiones	73
Rest. de objeto para esquinas	65,76,77,82	Correspondencia de regiones	80
Correspondencia de esquinas	85,87	Test para las esquinas	94
Correlación de esquinas	89,91,93	Test para los bordes	92
Generar pares de bordes	83	Test para las regiones	84
Rest. de epipolaridad para bordes	86		
Correspondencia de bordes	88,90		

Destaca como las primeras etapas de procesamiento son de tipo TAPI y las últimas de tipo CPU. Además entre ellas se han insertado las tareas cpu/tapi necesarias, que no aparecían especificadas en los subgrafos de las operaciones.

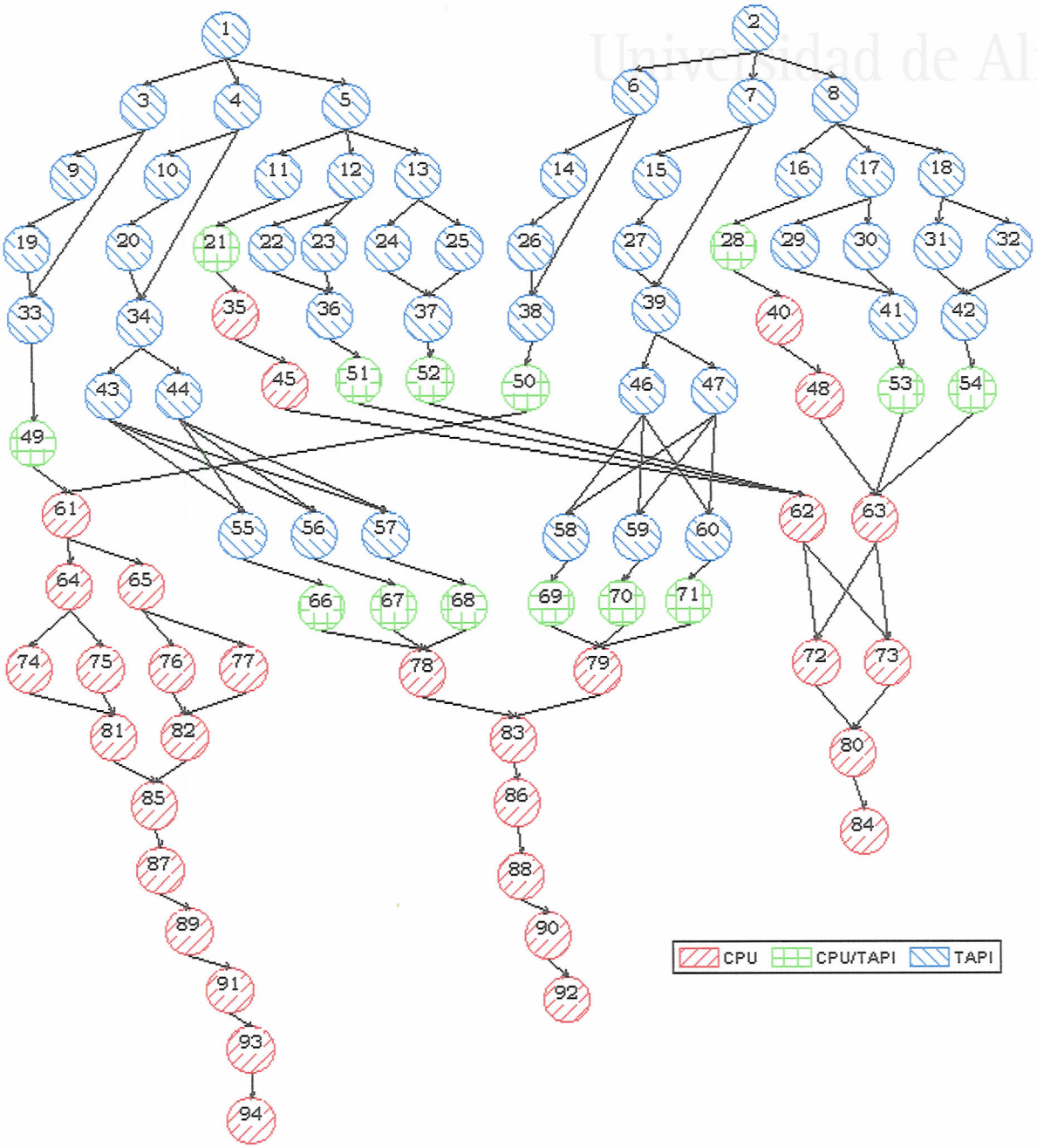


Figura 7-11. Grafo de tareas para el algoritmos de ejemplo.

También se puede comprobar como el subgrafo de la operación divisible de umbralización se ha dividido en dos. Así, para la umbralización de la detección de bordes de la imagen izquierda se tienen las tareas 43, y 44, y en la correspondiente a la derecha tiene las tareas 46 y 47.

7.4. Pruebas de los algoritmos

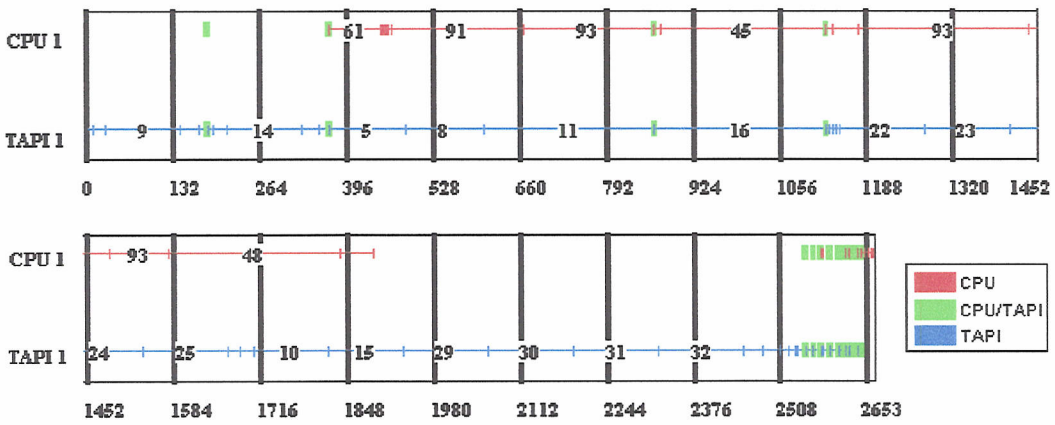
A continuación se muestran y comentan los resultados que ofrecen los distintos algoritmos explicados en los capítulos 3, 4 y 5 al planificar el grafo de la aplicación de ejemplo, descrito en al apartado anterior.

Para los algoritmos que admiten un límite en el número de procesadores se considera que se dispone de un procesador CPU y dos TAPIs. Otras consideraciones propias de algunos algoritmos en particularidad serán detalladas en sus apartados.

7.4.1. Planificación temporal según camino crítico

Este algoritmo, descrito en el capítulo 3, no realiza asignación de tareas. Por ello se han considerado dos evaluaciones distintas, una considerando un único procesador de cada tipo (uno CPU y uno TAPI), con lo que no hace falta una asignación espacial, y otra considerando un procesador CPU y dos TAPIs (los disponibles), pero con una asignación de tareas previa hecha manualmente. Además, no consideraba los costes que implican las interrupciones ni las comunicaciones.

Así, la Figura 7-12 muestra el resultado sin considerar asignación de tareas. Se observa como el tiempo final es 2653ms, y como se han efectuado 8 interrupciones. La tabla de la Figura 7-13 detalla los tiempos de computo, creación y finalización de cada tarea.



Tiempo Total: 2653 Ocupación total: 79 % Nº Interrupciones: 8 Nº Comunicaciones: 0

Figura 7-12. Planificación con el camino crítico sobre un procesador de cada tipo.

Datos de	Procesos																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Cómputo	9	9	19	19	118	19	19	118	114	114	260	5	5	114	114	260	5	5	28	28
Creación	0	184	9	1666	368	193	1685	486	28	1704	604	1126	1131	212	1818	865	1136	1141	142	2452
Finalización	9	193	28	1685	486	212	1704	604	142	1818	864	1131	1136	326	1932	1125	1141	1146	170	2480

Datos de	Procesos																			
	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
Cómputo	1	130	130	130	130	28	28	1	130	130	130	130	12	12	10	4	4	12	12	10
Creación	864	1146	1276	1406	1536	326	2480	1125	1932	2062	2192	2322	170	2508	865	2608	2614	354	2520	1126
Finalización	865	1276	1406	1536	1666	354	2508	1126	2062	2192	2322	2452	182	2520	875	2612	2618	366	2532	1136

Datos de	Procesos																			
	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
Cómputo	4	4	1	1	300	1	1	300	2	2	2	2	2	2	10	10	10	10	10	10
Creación	2620	2626	2532	2533	875	2570	2571	1136	182	366	2612	2618	2624	2630	2534	2546	2558	2572	2584	2596
Finalización	2624	2630	2533	2534	1486	2571	2572	1837	184	368	2614	2620	2626	2632	2544	2556	2568	2582	2594	2606

Datos de	Procesos																			
	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
Cómputo	80	4	4	1	1	2	2	2	2	2	2	5	3	1	2	1	2	2	2	1
Creación	368	2620	2632	448	449	2544	2556	2568	2582	2594	2606	2636	2643	454	450	455	452	2570	2608	2649
Finalización	448	2624	2636	449	450	2546	2558	2570	2584	2596	2608	2641	2646	455	452	456	454	2572	2610	2650

Datos de	Procesos													
	81	82	83	84	85	86	87	88	89	90	91	92	93	94
Cómputo	1	1	12	1	1	1	1	1	5	1	200	2	600	2
Creación	456	457	2610	2652	458	2646	459	2647	460	2648	465	2650	665	1887
Finalización	457	458	2643	2653	459	2647	460	2648	465	2649	665	2652	1887	1889

Figura 7-13. Tabla de tiempos para la planificación con el camino crítico.

Realizando una asignación previa manual, se puede considerar todos los procesadores disponibles del sistema, aunque dicha tarea resulta laboriosa, sobre todo cuando el algoritmo a planificar tiene gran número de tareas. Para el ejemplo tratado aquí se ha considerado la siguiente asignación manual. Todas las tareas tapi y cpu/tapi relativas a la imagen izquierda se procesan en un procesador TAPI, todas las tareas tapi y cpu/tapi relativas a la imagen derecha se procesan en el otro procesador TAPI, y todas las tareas cpu y cpu/tapi se procesan en la única CPU disponible.

El resultado, mostrado en la Figura 7-14, ofrece un tiempo total de 1910ms, bastante menor que en el caso anterior, gracias a la ejecución paralela de gran número de tareas tapi. Además, resulta necesario interrumpir menos tareas. Los tiempos asociados a las tareas con esta planificación se muestran en la Figura 7-15.

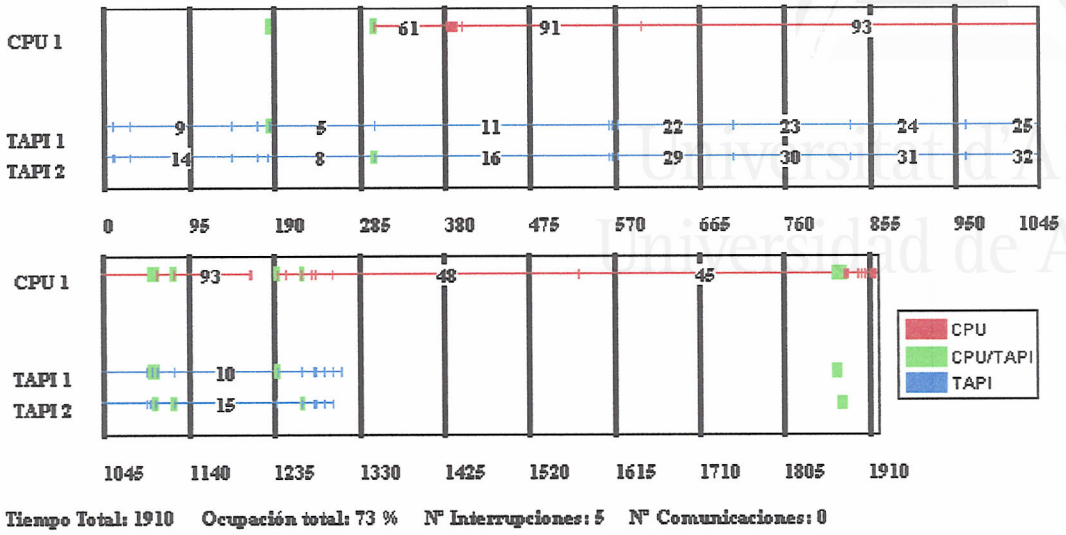


Figura 7-14. Planificación con el camino crítico y una asignación manual previa.

Datos de	Procesos																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Cómputo	9	9	19	19	118	19	19	118	114	114	260	5	5	114	114	260	5	5	28	28
Creación	0	0	9	1104	184	9	1102	182	28	1123	302	562	567	28	1123	302	562	567	142	1238
Finalización	9	9	28	1123	302	28	1121	300	142	1237	562	567	572	142	1237	562	567	572	170	1266

Datos de	Procesos																			
	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
Cómputo	1	130	130	130	130	28	28	1	130	130	130	130	12	12	10	4	4	12	12	10
Creación	1237	572	702	832	962	142	1237	1265	572	702	832	962	170	1266	1238	1092	1098	170	1266	1266
Finalización	1238	702	832	962	1092	170	1265	1266	702	832	962	1092	182	1278	1248	1096	1102	182	1278	1276

Datos de	Procesos																			
	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
Cómputo	4	4	1	1	300	1	1	320	2	2	2	2	2	2	10	10	10	10	10	10
Creación	1092	1096	1278	1279	1248	1278	1279	1276	182	300	1096	1102	1100	1121	1280	1290	1300	1280	1290	0
Finalización	1096	1100	1279	1280	1859	1279	1280	1576	184	302	1098	1104	1102	1123	1290	1300	1310	1290	1300	10

Datos de	Procesos																			
	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
Cómputo	80	4	4	1	1	2	2	2	2	2	2	5	3	1	2	1	2	2	2	1
Creación	302	1887	1891	382	383	1859	1861	1863	1865	1867	1869	1895	1900	388	384	389	386	1871	1873	1906
Finalización	382	1891	1895	383	384	1861	1863	1865	1867	1869	1871	1900	1903	389	386	390	388	1873	1875	1907

Datos de	Procesos													
	81	82	83	84	85	86	87	88	89	90	91	92	93	94
Cómputo	1	1	12	1	1	1	1	1	5	1	200	2	600	2
Creación	390	391	1875	1909	392	1903	393	1904	394	1905	399	1907	599	1207
Finalización	391	392	1887	1910	393	1904	394	1905	399	1906	599	1909	1207	1209

Figura 7-15. Tiempos para la planificación con el camino crítico y una asignación previa.

7.4.2. Asignación espacial

En el capítulo 4 se proponían tres versiones de un algoritmo que realizaba la asignación a partir de una planificación temporal previa sobre un procesador de cada tipo. Para realizar las pruebas de estas versiones y poder compararlas, se tomará como planificación temporal la realizada en el apartado anterior con el algoritmo de camino crítico, mostrada en la Figura 7-12.

El resultado del primer algoritmo de asignación espacial se muestra en la Figura 7-16. Este no consideraba límite en el número de procesadores, y al no aprovechar bien los huecos de tiempo libre existentes en los procesadores da como resultado la necesidad de un gran número de procesadores. El tiempo total requerido es 1077ms, muy bajo debido a la gran cantidad de procesadores. Pero tanto procesador conlleva un gran número de comunicaciones entre ellos, en este caso 11 sin contar las tareas cpu/tapi del grafo de entrada, y en este algoritmo no se considera el coste de las mismas.

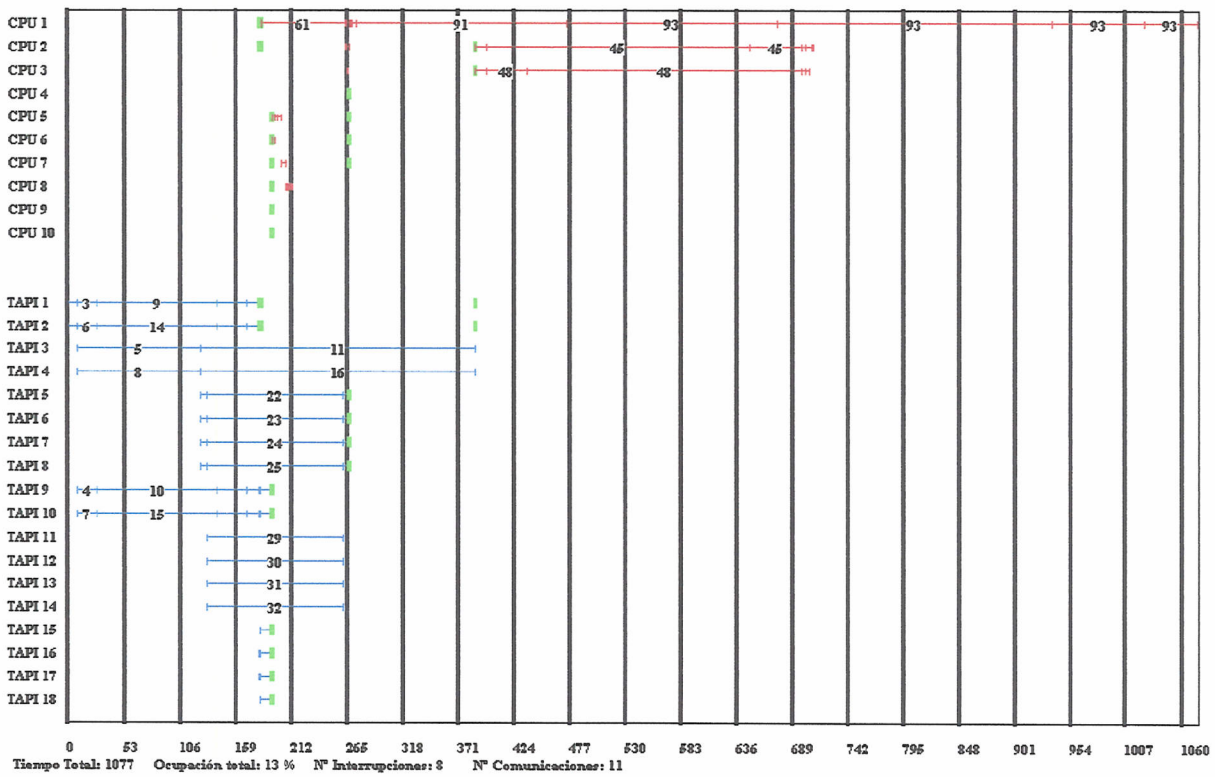


Figura 7-16. Asignación espacial con huecos.

La Figura 7-17 muestra el resultado del algoritmo espacial sin huecos. La mejora que aporta éste con respecto al anterior se ve claramente comparando los resultados de ambos; obteniendose el mismo tiempo total de ejecución, 1077ms, se requieren menos procesadores

de cada tipo y menos comunicaciones, y la ocupación de los procesadores aumenta. De todas formas, como no considera límites para el número de procesadores, la cantidad resultante de éstos sigue siendo poco realista.

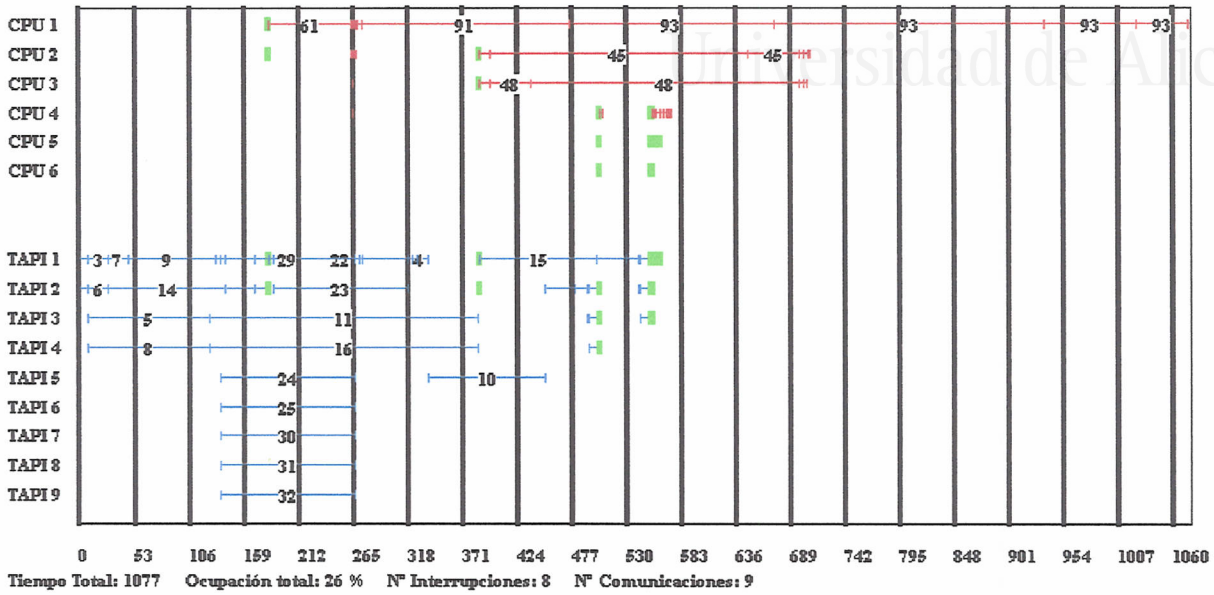


Figura 7-17. Asignación espacial sin huecos.

Finalmente se muestra el resultado de la asignación espacial sin huecos y con límite de procesadores en la Figura 7-18. Se ha logrado un tiempo de ejecución menor (1774ms) que usando el algoritmo de camino crítico con una asignación manual (1910ms), lo que indica que la asignación automática aprovecha mejor los recursos.

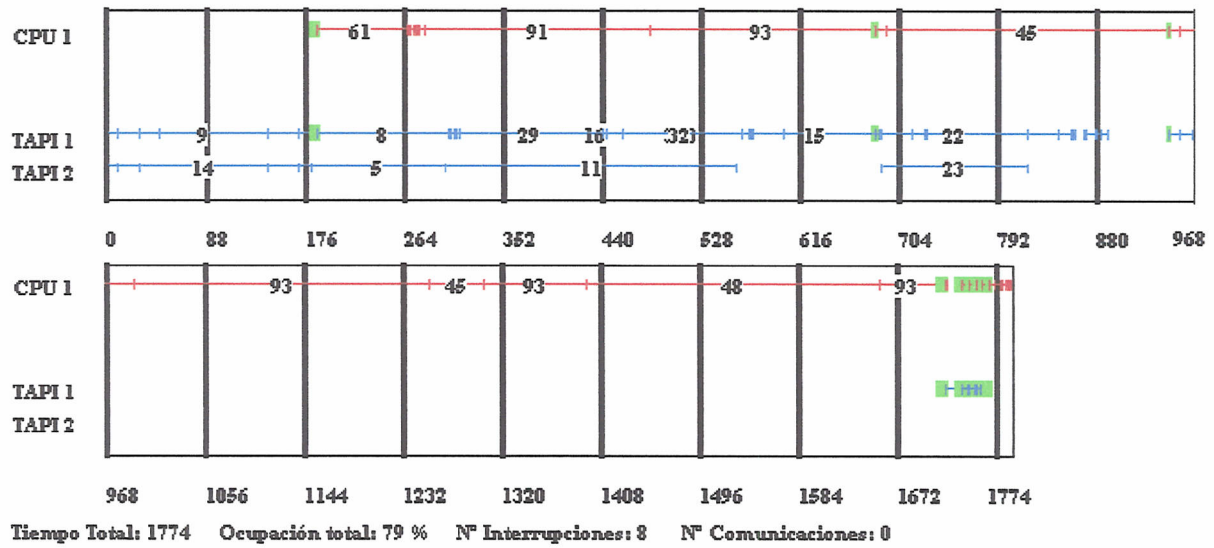


Figura 7-18. Asignación espacial sin huecos con límite de procesadores.

Los tiempos asociados a las tareas para este último algoritmo se muestran en la Figura 7-19.

Datos de	Procesos																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Cómputo	9	9	19	19	118	19	19	118	114	114	260	5	5	114	114	260	5	5	28	28
Creación	0	0	9	440	182	9	28	186	28	459	300	683	305	28	573	304	305	309	142	573
Finalización	9	9	28	459	300	28	47	304	142	573	560	688	310	142	687	564	310	314	170	601

Datos de	Procesos																			
	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
Cómputo	1	130	130	130	130	28	28	1	130	130	130	130	12	12	10	4	4	12	12	10
Creación	682	688	688	310	440	142	818	943	310	440	440	444	170	715	683	880	1729	170	846	944
Finalización	683	818	818	440	570	170	846	944	440	570	570	574	182	727	693	884	1733	182	858	954

Datos de	Procesos																			
	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
Cómputo	4	4	1	1	300	1	1	300	2	2	2	2	2	2	10	10	10	10	10	10
Creación	1735	1741	858	728	693	869	859	954	182	184	1733	1739	1745	1751	944	954	869	1713	870	880
Finalización	1739	1745	859	729	1304	870	860	1655	184	186	1735	1741	1747	1753	954	964	879	1723	880	890

Datos de	Procesos																			
	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
Cómputo	80	4	4	1	1	2	2	2	2	2	2	5	3	1	2	1	2	2	2	1
Creación	186	1741	1753	266	267	1707	1709	1711	1723	1725	1727	1757	1764	272	268	273	270	1713	1729	1770
Finalización	266	1745	1757	267	268	1709	1711	1713	1725	1727	1729	1762	1767	273	270	274	272	1715	1731	1771

Datos de	Procesos													
	81	82	83	84	85	86	87	88	89	90	91	92	93	94
Cómputo	1	1	12	1	1	1	1	1	5	1	200	2	600	2
Creación	274	275	1731	1773	276	1767	277	1768	278	1769	283	1771	483	1705
Finalización	275	276	1764	1774	277	1768	278	1769	283	1770	483	1773	1705	1707

Figura 7-19. Tiempos para la asignación espacial sin huecos con límite de procesadores.

7.4.3. Planificación espacio-temporal

En el capítulo 5 se han descrito varios algoritmos que realizan la asignación espacial y la planificación temporal de forma simultánea. En este apartado se van a evaluar los dos primeros, que no consideraban costes de interrupción ni comunicación.

El primer algoritmo espacio-temporal considera que no hay limite en el número de procesadores. El resultado de aplicarlo al algoritmo de ejemplo se puede ver en la Figura 7-20. El algoritmo reduce el tiempo de ejecución hasta 1077ms, pero considerando gran

cantidad de procesadores para ello, igual que el algoritmo de asignación espacial sin huecos (ver Figura 7-17).

Así, para conseguir ese menor tiempo de ejecución se requieren 7 CPUs y 18 TAPIs, lo cual es poco práctico. Además, se generan muchas comunicaciones entre procesadores, a parte de las tareas cpu/tapi ya indicadas en el grafo.

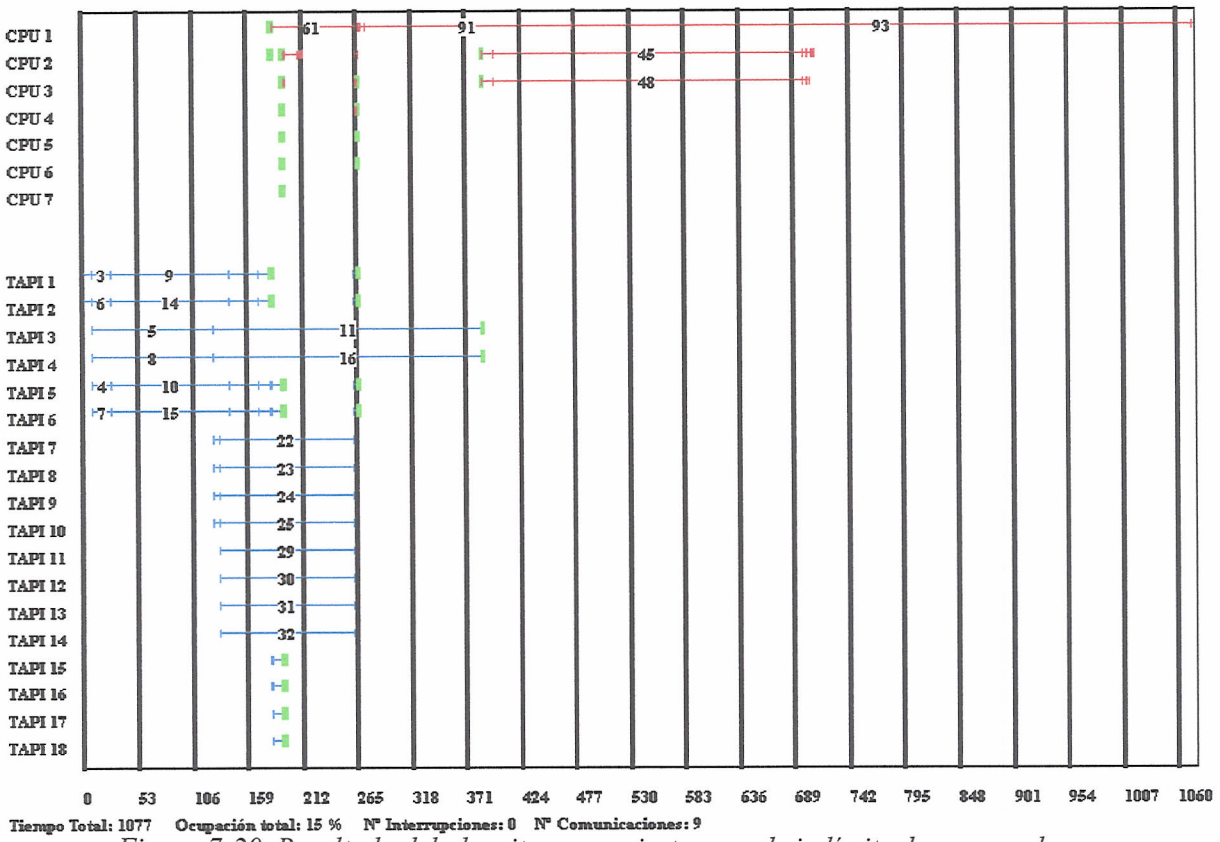


Figura 7-20. Resultado del algoritmo espacio-temporal sin límite de procesadores.

Más prácticos son los resultados a los que lleva la planificación espacio-temporal con límite de procesadores. El resultado, mostrado en la Figura 7-21, indica un tiempo de ejecución de 1766ms, el menor conseguido hasta ahora para el hardware disponible. Aunque para llegar a ese tiempo el algoritmo de planificación realiza 5 interrupciones y 2 comunicaciones cuyos costes no se consideran.

La tabla con los tiempos de las tareas correspondiente a la Figura 7-21 se muestra en la Figura 7-22.

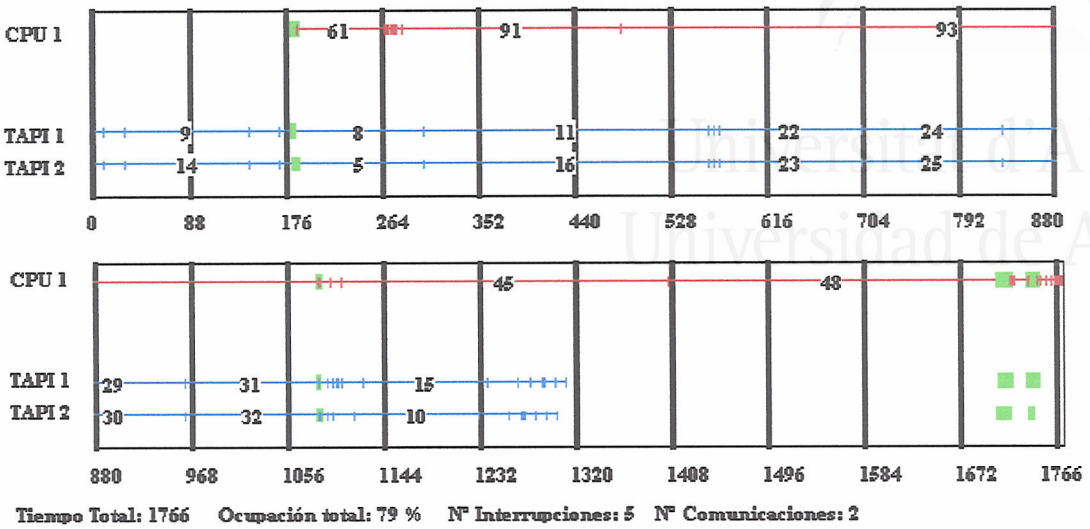


Figura 7-21. Resultado del algoritmo espacio-temporal con limite de procesadores.

Datos de	Procesos																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Cómputo	9	9	19	19	118	19	19	118	114	114	260	5	5	114	114	260	5	5	28	28
Creación	0	0	9	1097	182	9	1105	184	28	1116	302	562	562	28	1124	302	567	567	142	1230
Finalización	9	9	28	1116	302	28	1124	302	142	1230	562	567	567	142	1238	562	572	572	170	1258

Datos de	Procesos																			
	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
Cómputo	1	130	130	130	130	28	28	1	130	130	130	130	12	12	10	4	4	12	12	10
Creación	1083	572	572	702	702	142	1238	1084	832	832	962	962	170	1258	1085	1092	1092	170	1266	1095
Finalización	1084	702	702	832	832	170	1266	1085	962	962	1101	1097	182	1270	1095	1096	1096	182	1278	1105

Datos de	Procesos																			
	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
Cómputo	4	4	1	1	300	1	1	300	2	2	2	2	2	2	10	10	10	10	10	10
Creación	1096	1101	1270	1271	1105	1288	1289	1405	182	184	1733	1735	1737	1739	1272	1278	1282	1290	1292	1300
Finalización	1100	1105	1271	1272	1405	1289	1290	1705	184	186	1735	1737	1739	1741	1282	1288	1292	1300	1302	1310

Datos de	Procesos																			
	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
Cómputo	80	4	4	1	1	2	2	2	2	2	2	5	3	1	2	1	2	2	2	1
Creación	186	1741	1745	266	267	1705	1707	1709	1711	1713	1715	1749	1754	272	268	273	270	1717	1719	1760
Finalización	266	1745	1749	267	268	1707	1709	1711	1713	1715	1717	1754	1757	273	270	274	272	1719	1721	1761

Datos de	Procesos													
	81	82	83	84	85	86	87	88	89	90	91	92	93	94
Cómputo	1	1	12	1	1	1	1	1	5	1	200	2	600	2
Creación	274	275	1721	1765	276	1757	277	1758	278	1759	283	1763	483	1761
Finalización	275	276	1733	1766	277	1758	278	1759	283	1760	483	1765	1083	1763

Figura 7-22. Tiempos de las tareas para el algoritmo espacio-temporal con limite de procesadores.

7.4.4. Planificación espacio-temporal con costes de interrupción

En este apartado se evalúan las versiones del algoritmo espacio-temporal que consideran los costes de escritura y lectura de estado por interrupción. En el capítulo 5 se presentaban dos versiones: una que considera costes constantes y otra según función de costes. Se estudiará primero el caso de los costes constantes, cuyo resultado sobre el ejemplo tratado en este capítulo se puede ver en la Figura 7-23.

El primer hecho destacable es que se incrementa el coste total de ejecución hasta 1838ms, frente a los 1766ms que se obtenían con la versión que no consideraba costes (Figura 7-21). Este aumento viene determinado por el hecho de que ahora las 5 interrupciones que se producen suponen unos costes añadidos a los tiempos de computo de las tareas interrumpidas. Además, el hecho de que esas tareas tengan nuevos tiempos totales de ejecución, hace que esta planificación obtenida ahora difiera de la anterior.

El tiempo de ejecución de 1828ms es más realista que los 1766ms obtenidos anteriormente. Aun así, la planificación requiere una nueva comunicación no especificada en el grafo de entrada, aunque su coste (2ms según lo explicado en el apartado 7.3.2.) es poco siginificativo.

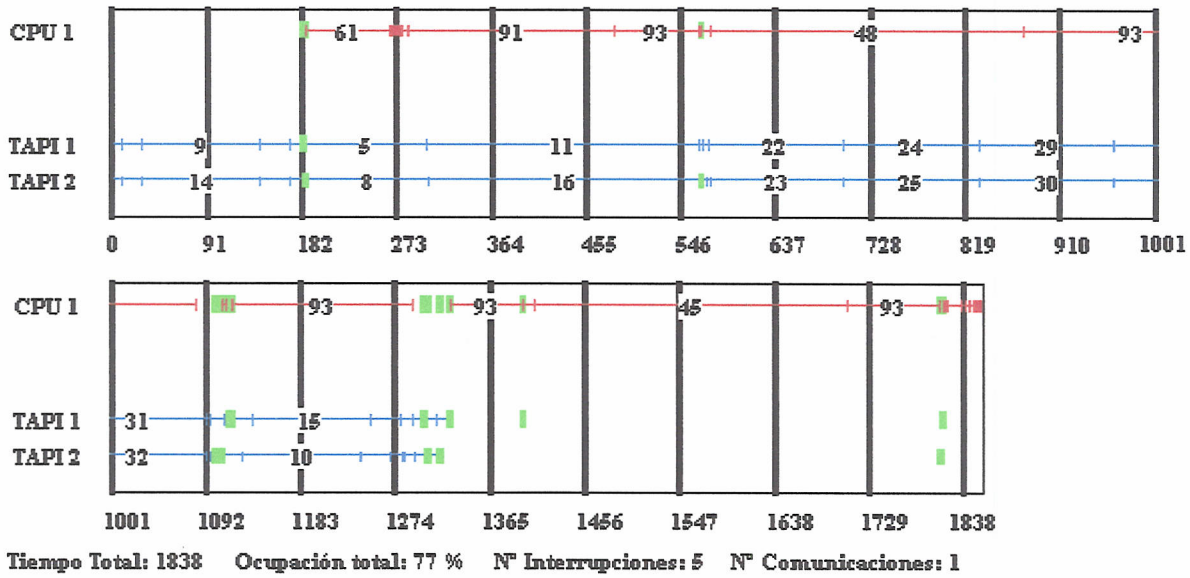


Figura 7-23. Resultado del algoritmo con costes de interrupción constantes.

La tabla con la asignación espacio-temporal correspondiente a la Figura 7-23 se muestra en la Figura 7-24.

Datos de	Procesos																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Cómputo	9	9	19	19	118	19	19	118	114	114	260	5	5	114	114	260	5	5	28	28
Creación	0	0	9	1108	184	9	1117	186	28	1127	302	562	565	28	1136	304	567	570	142	1241
Finalización	9	9	28	1127	302	28	1136	304	142	1241	562	567	570	142	1250	564	572	575	170	1269

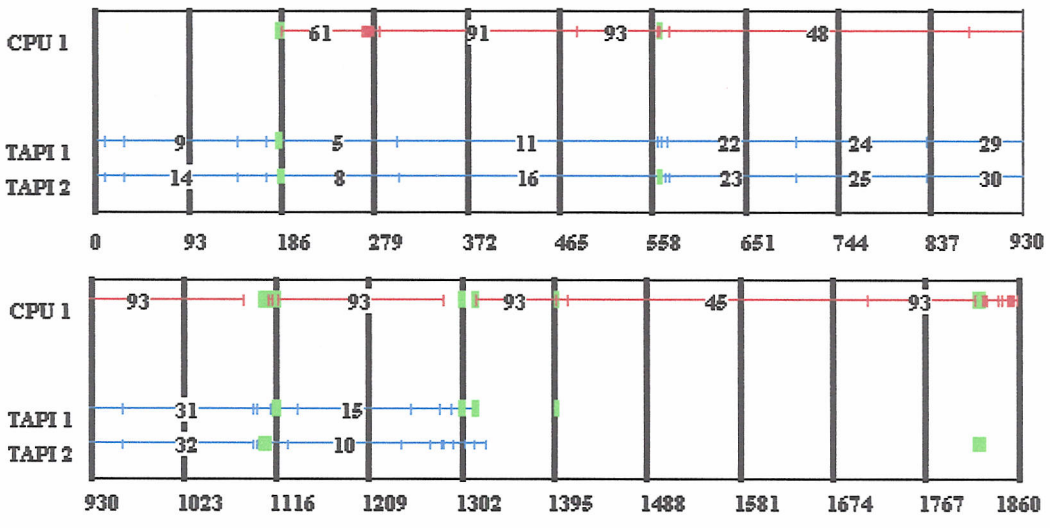
Datos de	Procesos																			
	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
Cómputo	1	130	140	130	130	28	28	1	130	130	130	130	12	12	10	4	4	12	12	10
Creación	1396	572	575	702	702	142	1250	564	832	832	962	962	170	1269	1397	1109	1092	170	1278	565
Finalización	1397	702	1109	832	832	170	1278	565	962	962	1092	1092	182	1281	1407	1113	1096	182	1290	575

Datos de	Procesos																			
	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
Cómputo	4	4	1	1	300	1	1	300	2	2	2	2	2	2	10	10	10	10	10	10
Creación	1096	1102	1281	1282	1407	1302	1303	575	182	184	1113	1115	1100	1106	1283	1290	1293	1304	1305	1314
Finalización	1100	1106	1282	1283	1707	1303	1304	875	184	186	1115	1117	1102	1108	1293	1300	1303	1314	1315	1324

Datos de	Procesos																			
	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
Cómputo	80	4	4	1	1	2	2	2	2	2	2	5	3	1	2	1	2	2	2	1
Creación	186	1817	1108	266	267	1797	1300	1303	1799	1315	1324	1821	1826	272	268	273	270	1801	1803	1832
Finalización	266	1821	1112	267	268	1799	1302	1305	1801	1317	1326	1826	1829	273	270	274	272	1803	1805	1833

Datos de	Procesos													
	81	82	83	84	85	86	87	88	89	90	91	92	93	94
Cómputo	1	1	12	1	1	1	1	1	5	1	200	2	620	2
Creación	274	275	1805	1837	276	1829	277	1830	278	1831	283	1835	483	1833
Finalización	275	276	1817	1838	277	1830	278	1831	283	1832	483	1837	1797	1835

Figura 7-24. Tiempos de las tareas para el algoritmo con costes de interrupción constantes.



Tiempo Total: 1860 Ocupación total: 76 % Nº Interrupciones: 5 Nº Comunicaciones: 1

Figura 7-25. Resultado del algoritmo con función de costes de interrupción.

Para evaluar el algoritmo espacio-temporal que considera las funciones de coste de interrupción, previamente ha sido necesario especificar unas funciones para las distintas tareas interrumpibles de los subgrafos almacenados en la base de datos. Éstas se han estimado conforme a los algoritmos asociados a las distintas operaciones. El resultado, que se puede ver en la Figura 7-25, es similar al del algoritmo que emplea costes constantes, pero con un tiempo total de ejecución algo mayor, 1869ms.

7.4.5. Planificación espacio-temporal con costes de comunicación

Finalmente, se examina el resultado que ofrece la versión del algoritmo espacio-temporal que considera los costes de comunicación sobre el algoritmo de ejemplo.

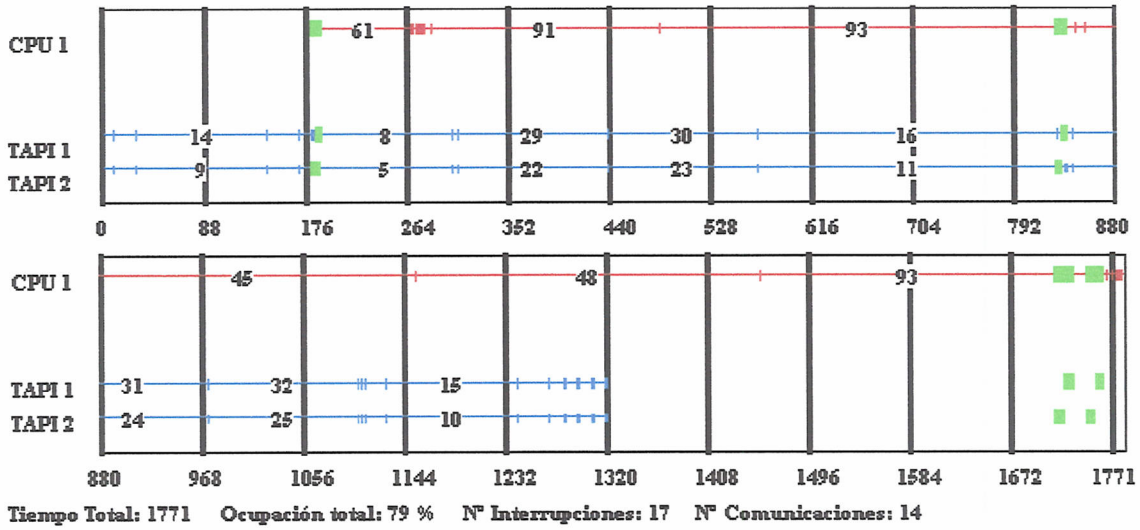


Figura 7-26. Resultado del algoritmo con de costes comunicación.

La Figura 7-26 muestra la planificación resultante y la Figura 7-27 detalla los tiempos de las tareas. En primer lugar, se puede observar que el algoritmo no ha añadido nuevas comunicaciones adicionales a las tareas cpu/tapi especificadas en el grafo de entrada, que son 14. Este hecho se debe a que ahora las comunicaciones implican un coste que se considera dentro de la planificación, por lo cual el algoritmo ha decidido no incluir comunicaciones que se pueden evitar, aunque esto supone que el tiempo final sea de 1771ms, poco mayor al mínimo de 1766ms obtenido por el espacio-temporal que no consideraba el efecto de los costes.

Por otra parte, el algoritmo realiza 17 interrupciones, para las cuales no se considera su coste. Esto hace suponer que el coste final de ejecución considerando también esos costes estará por encima de los 1771ms.

También se ha generado el grafo de tareas del ejemplo con la opción de no incluir las operaciones de comunicación cpu/tapi y se ha planificado con este algoritmo. El resultado obtenido es idéntico al mostrado en la Figura 7-26, ya que el algoritmo inserta automáticamente las 14 tareas de comunicación cpu/tapi necesarias.

Datos de	Procesos																		
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Cómputo	9	9	19	19	118	19	19	118	114	114	260	5	5	114	114	260	5	5	28	28
Creación	0	0	9	1109	186	9	1109	184	28	1128	569	304	838	28	1128	569	304	838	142	1242
Finalización	9	9	28	1128	304	28	1128	304	142	1242	829	309	843	142	1242	829	309	843	170	1270

Datos de	Procesos																			
	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
Cómputo	1	130	130	130	130	28	28	1	130	130	130	130	12	12	10	4	4	12	12	10
Creación	829	309	439	843	973	142	1242	829	309	439	843	973	170	1270	832	832	1103	170	1270	836
Finalización	830	439	569	973	1103	170	1270	830	439	569	973	1103	182	1282	854	836	1107	182	1282	846

Datos de	Procesos																			
	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
Cómputo	4	4	1	1	300	1	1	300	2	2	2	2	2	2	10	10	10	10	10	10
Creación	830	1103	1282	1283	854	1282	1283	1154	182	182	836	1107	836	1107	1284	1296	1308	1284	1296	1308
Finalización	834	1107	1283	1284	1154	1283	1284	1454	184	184	838	1109	838	1109	1294	1306	1318	1294	1306	1318

Datos de	Procesos																			
	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
Cómputo	80	4	4	1	1	2	2	2	2	2	2	5	3	1	2	1	2	2	2	1
Creación	188	1742	1750	268	269	1294	1306	1318	1294	1306	1318	1754	1759	274	270	275	272	1716	1724	1765
Finalización	268	1746	1754	269	270	1296	1308	1320	1296	1308	1320	1759	1762	275	272	276	274	1718	1726	1766

Datos de	Procesos													
	81	82	83	84	85	86	87	88	89	90	91	92	93	94
Cómputo	1	1	12	1	1	1	1	1	5	1	200	2	600	2
Creación	276	277	1726	1770	278	1762	279	1763	280	1764	285	1768	485	1766
Finalización	277	278	1738	1771	279	1763	280	1764	285	1765	485	1770	1710	1768

Figura 7-27. Tiempos de las tareas para el algoritmo con de costes comunicación.

Capítulo 8

Conclusiones

8.1. Introducción

En este último capítulo se describen primero las conclusiones derivadas de aplicar las técnicas de planificación para aplicaciones de visión por computador descritas en los capítulos 3, 4 y 5, parte central de esta tesis, al algoritmo de correspondencia de imágenes estereoscópicas según se describe en el capítulo 7, para después presentar las conclusiones y aportaciones generales de la tesis

8.2. Resultados obtenidos

De las pruebas descritas en el capítulo 7 se desprenden varias conclusiones. La primera es que las técnicas de planificación que ofrecen resultados mejores y más realistas y que tienen en cuenta el hardware disponible, son los algoritmos que realizan la asignación espacial y la planificación temporal de forma simultánea con límite de procesadores y teniendo en cuenta los costes de interrupción.

El algoritmo de planificación que considera costes de comunicación no parece tan útil como los que consideran costes de interrupción, ya que los costes de comunicación afectan mucho menos al tiempo total del sistema que los de interrupción. Esto es así por que las comunicaciones entre procesadores de un sistema multiprocesador (no multicomputador) son muy rápidas frente a los tiempos de muchas de las tareas tratadas en visión por computador y frente a los costes que suponen las interrupciones. Además hay que tener en cuenta otros dos aspectos. Por una parte las técnicas de planificación propuestas realizan interrupción de tareas, por lo que éstas pueden ser numerosas. Y por otra, el número de unidades de proceso disponible habitualmente no es muy alto, y si el algoritmo optimiza la ocupación de los procesadores, las comunicaciones se pueden reducir mucho.

Un aspecto interesante sobre el algoritmo de planificación que considera costes es que es capaz de insertar las tareas de comunicación. Sin embargo, las tareas de comunicación que más frecuentemente aparecen en sistemas de visión por computador son las de tipo cpu/tapi, que también pueden ser incorporadas en el grafo de tareas automáticamente al generar éste.

En cuanto a la utilización de función de costes frente a costes de interrupción constantes, la primera aproximación siempre dará resultados más realistas. Pero no resulta fácil describir esta función para todas las tareas, además de que ese trabajo resulta laborioso, y en muchos casos la aproximación de costes constantes es suficiente.

También ofrece buen resultado el algoritmo de asignación espacial sin huecos con límite de procesadores aplicado sobre una planificación temporal obtenida con el algoritmo básico de camino crítico, algo inferior al que ofrece el espacio-temporal con límite de procesadores. Sin embargo estos algoritmos no tienen en cuenta los costes derivados de las interrupciones, que pueden influir en el tiempo total de ejecución.

En cuanto a los algoritmos que no consideran límite en el número de procesadores, pueden servir para orientar al diseñador cual es el hardware necesario para un tiempo de ejecución mínimo, sabiendo también que añadir más procesadores no mejorará ese tiempo. Sin embargo los resultados de la planificación no son útiles, debido a que estos algoritmos producen gran cantidad de comunicaciones entre procesadores sin tener en cuenta sus costes. En cualquier caso, estos algoritmos han sido un paso necesario para llegar hasta los espacio-temporales que ofrecen mejores resultados.

Finalmente se ha comprobado como el entorno de aplicaciones desarrollado permite al diseñador especificar un algoritmo gráficamente y como se pueden considerar las tareas que forman las operaciones y sus características, lo que permite generar automáticamente el grafo de tareas que sirve de entrada al planificador.

8.3. Conclusiones de la tesis

Los algoritmos de planificación estática desarrollados, destinados a la investigación y desarrollo de aplicaciones de visión por computador para sistemas multiprocesador, son la principal aportación de la tesis. Entre ellos, destacan los algoritmos espacio-temporales que, además de explotar la posibilidad de interrupción de algunas tareas, consideran los costes de escritura y lectura de estado que conllevan.

En general, todos los algoritmos propuestos tienen en cuenta las principales características de los sistemas de visión por computador. Éstas son la distinción entre unidades de procesamiento genéricas (CPUs) y tarjetas de adquisición y procesamiento de

imágenes (TAPIs) y el tipo de tareas que pueden procesar cada una, relaciones de precedencia entre las tareas, y la posibilidad de interrumpir algunas tareas para ejecutar otras.

El principal objetivo de los algoritmos diseñados es obtener una planificación estática, previa a la ejecución de la aplicación desarrollada, que reduzca en lo posible el tiempo de ejecución total dados un cierto número de procesadores de cada tipo. Pero algunas versiones de los algoritmos (las que no consideran límite en el número de procesadores) también permiten al diseñador estimar el tiempo de ejecución mínimo que se podría alcanzar si se dispusiera del hardware necesario e informan sobre el número de procesadores a partir del cual no tiene sentido añadir más de éstos.

También se debe destacar las aplicaciones creadas que permiten al usuario o diseñador especificar una aplicación de visión por computador para aplicar sobre el los diferentes algoritmos diseñados de una forma amigable, pudiendo evaluar y comparar los resultados que ofrecen cada uno de ellos. Se trata del entorno visual, la gestión de la base de datos de operaciones, la aplicación que implementa los algoritmos de planificación diseñados y algunas otras herramientas auxiliares.

Éstas aplicaciones van más allá de la especificación gráfica de alto nivel y la simulación ofrecida por algunas herramientas comerciales, ya que permiten definir la composición de las operaciones en base a sus tareas elementales, generan el grafo de tareas, y enlazan directamente con la aplicación que realiza la planificación.

Referencias

[1] Nimal Nissanke. "*Realtime Systems*". Prentice Hall, 1997.

[2] Behrooz A. Shirazi, Ali R. Hurson, Krishna M. Kavi. *"Scheduling and Load Balancing in Parallel and Distributed Systems"*. IEEE Computer Society Press, 1997.

[3] Jia Xu, David Lorge Parnas. *"On satisfying timing constrains in hard-real-time systems"*. IEEE Transactions on Software Engineering, vol. 19 (1), pp. 74-80, 1993.

[4] W. Zhao, K. Ramamritham, J. A. Stankovic. "*Preemptive scheduling under time and resource constrains*". IEEE Transactions on Computers, vol. 36 (8), pp. 949-960, 1987.

[5] J. A. Stankovic, M. Spuri, M. D. Natale, G. C. Buttazzo. "*Implications of classical scheduling results for real-time systems*". IEEE Computer, vol. 18 (6), pp. 16-25, 1995.

[6] C. M. Krishna, Kang G. Shin. *"Real-Time Systems"*. McGraw-Hill, 1997.

[7] Jia Xu, David Lorge Parnas. *"Priority Scheduling Versus Pre-Run-Time Scheduling"*. The International Journal of Time-Critical Computing Systems, vol. 18, pp. 7-23, 2000.

[8] Fernando Torres. Tesis Doctoral *"Arquitectura Paralela para el Procesado de Imágenes de Alta Resolución. Aplicación a la Inspección de Impresiones en Tiempo Real"*. Universidad Politécnica de Madrid, 1995.

[9] F. Torres, F. A. Candelas, S. T. Puente, L. M. Jiménez, C. Fernández, R. J. Agulló. *"Simulation and Scheduling of Real-Time Computer Vision Algorithms"*. Lecture Notes In Computer Science. Springer, vol. 1542, pp. 98-114, 1999.

[10] C. L. Liu, J. W. Layland. *"Scheduling Algorithms for Multiprogramming in a Hard real-Time Environment"*. Journal of the ACM, vol. 20 (1), pp. 46-61, 1973.

[11] J. Santos, E. Ferro, J. Orozco, R. Cayssials. *"A Heuristic Approach to the Multitask-Multiprocessor Assignment Problem using Empty-Slots Method and Rate Monotonic Scheduling"*. Real-time Systems, vol. 12, pp. 167-199, 1997.

[12] Peter Altenbernd, Hans Hansson. *"The Slack Method: A New Method for Static Allocation of Hard Real-Time Task"*. Real-Time Systems, vol. 15, pp. 103-130, 1998.

[13] Yoshifumi Manabe, Shigemi Aoyagi. *"A Feasibility Decision algorithm for Rate Monotonic and Deadline Monotonic Scheduling"*. Real-Time Systems, vol. 14, pp. 171-181, 1998

[14] N. C. Audsley, A. Burns. *"On Fixed Priority Scheduling, Offsets and Co-Prime Task periods"*. Information Processing Letters, vol. 67, pp. 65-69, 1998.

[15] J. M. Sebastián, F. Torres, R. Aracil, O. Reinoso, L. M. Jiménez, D. García. *"Job-Shop Scheduling Applied To Computer Vision"*. Proc. SPIE, vol. 3166, pp. 158-169, 1997.

[16] Kyunghee Choi, Gihyum Jung, Taegun Kim, Seunhum Jung. *"Real-time Scheduling Algorithm for Minimizing Maximum Weighted Error With O(N logN+cN)*

Complexity". Information Processing Letters, vol. 67, pp. 311-315, 1998.

[17] Dong-Ik Oh, T. P. Baker. *"Utilization Bounds for N-processor Rate Monotone Scheduling with Static Processor Assignment"*. Real-Time Systems, vol. 15, pp. 183-192, 1998

[18] E. Bampis, C. Delorme, J-C. König. *"Optimal Schedules for d-D Grid Graph with Communication Delays"*. Real-Time Systems, vol. 15, pp. 1653-1664, 1998.

[19] S. Y. Lee, J. K. Aggarwal. *"A System Design/Scheduling Strategy for Parallel Image processing"*. IEEE Trans. PAMI, vol. 12, pp. 194-204, 1990.

[20] Cheolwhan Lee, Yuang-Fang Wang, Tao Yang. *"Global Optimization for Mapping Parallel Image Processing task on Distributed memory Machines"*. Journal of Parallel & Distributed Computing, vol. 45 (1), pp. 29-45, 1997.

[21] Cheolwhan Lee, Yuang-Fang Wang, Tao Yang. *"Static Global Scheduling for Optimal Computer Vision and Image Processing Operations on Distributed-Memory Multiprocessors"*. Technical Report TRC94-23, University of California, Santa Barbara, California, 1994.

[22] L. H. Jamieson, E. J. Delp, J. N. Patel, C. C. Wang, A. A. Khokhar. *"A Library-Based Program Development Environment for Parallel image Processing"*. Scalable Parallel Libraries Conference, pp. 187-194, 1993.

[23] J. N. Patel, A. A. Khokhar, L. H. Jamieson. *"Implementation of Parallel Image Processing Algorithm in the Cloner Environment"*. IEEE Work-shop on VLSI Signal Processing, La Jolla, California, pp. 83-92, 1994.

[24] S. T. Puente. *"Asignación de Tareas y Planificación en Entornos Multiprocesador Reales. Aplicación a un Sistema de Inspección Visual"*. Proyecto Fin de Carrera. Escuela Politécnica Superior, Universidad de Alicante, 1998.

[25] S. T. Puente. *"Algoritmo SASEPA con costes de comunicación"*. Informe interno del Grupo de Automática y Visión Artificial. Universidad de Alicante, 2000.

[26] Khoros es una aplicación de Khoral Technologies: http://www.khoral.com.

[27] WiT es una aplicación de Logical Vision: http://www.logicalvision.com.

[28] eVision es un entorno de herramientas de la casa Euresys: http://www.euresys.com.

[29] Informe final del proyecto CICYT TAP96-0629-CO4; *"Construcción de un Entorno para la Investigación y Desarrollo en Visión Artificial"*. Comisión Interministerial de Ciencia y Tecnología, Ministerio de Cultura, 1998.

[30] Francisco A. Candelas. *"Simulación y Planificación en Entorno Distribuido Para Algoritmos en Tiempo Real de Visión Artificial"*. Memoria de Investigación, Universidad de Alicante, 1998.

[31] Francisco A. Candelas, S.T. Puente, F. Torres. *"Metodología y Herramientas de Planificación Estática para Aplicaciones de Visión Artificial"*. Proceedings XX Jornadas de Automática (Salamanca), pp. 19-23, 1999.

[32] F. Torres, F. A. Candelas, S. T. Puente, F. G. Ortiz. *"Graph Models Applied to Specification, Simulation, Allocation and Scheduling of Real-Time Computer Vision Applications"*. International Journal of Imaging Systems & Technology, vol. 11 (5), pp. 287-291, 2000.