UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

DEPARTAMENTO DE ELECTRÓNICA, TELEMÁTICA Y AUTOMÁTICA

TESIS DOCTORAL

METODOLOGÍA DE PARALELIZACIÓN DE ALGORITMOS BASADA EN GRAFOS COLOREADOS

CARMEN NIEVES OJEDA GUERRA

Las Palmas de Gran Canaria, Mayo de 2000

41/1999-00

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
UNIDAD DE TERCER CICLO Y POSTGRADO

Reunido el día de la fecha, el Tribunal nombrado por el Excmo. Sr. Rector Magfco. de esta Universidad, el/a aspirante expuso esta TESIS DOCTORAL.
Terminada la lectura y contestadas por el/a Doctorando/a las objeciones formuladas por los señores miembros del Tribunal, éste calificó dicho trabajo con la nota de SOBRESALIENTE CUM LAUDE (UNANIMIDAD)

Las Palmas de Gran Canaria, a 12 de mayo de 2000.

El/a Presidente/a: Dr.D. Carlos Delgado Kloos,

El/a Secretario/a: Dr.D. Juan Francisco Pérez Castellano,

El/a Vocal: Dr.D. Casiano Rodríguez León,

El/a Vocal: Dr.D. Miguel Valero García,

El/a Vocal: Dr.D. Luis María Díaz de Cerio,

La Doctoranda: Dª. Carmen Nieves Ojeda Guerra,

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

TESIS DOCTORAL

METODOLOGÍA DE PARALELIZACIÓN DE ALGORITMOS BASADA EN GRAFOS COLOREADOS

AUTORA: Carmen Nieves Ojeda Guerra
DIRECTOR: Álvaro Suárez Sarmiento
FECHA: MAYO 2000

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

TESIS DOCTORAL

METODOLOGÍA DE PARALELIZACIÓN DE ALGORITMOS BASADA EN GRAFOS COLOREADOS

PRESIDENTE:

SECRETARIO:

VOCAL:

VOCAL:

VOCAL:

CALIFICACIÓN:

DIRECTOR: ALVARO SUAREZ SARMIENTO

AUTORA: CARMEN N. OJEDA GUERRA

AGRADECIMIENTOS

Como de bien nacido es ser agradecido, ahí van mis agradecimientos a todos aquellos que me han ayudado durante estos años. Especialmente

A Álvaro por su ayuda y dedicación.

A mis compañeros: Cley, Mangel, Cramirez, Francis, Domingo, Elsa, Pablo, Gustavo y Ernestina. También a todos aquellos que no nombro aquí, porque la lista sería demasiado larga. Todos saben quiénes son.

A mi familia, por el poco tiempo que les he dedicado durante estos años de trabajo. A los que están y a los que no están.

A mi marido por su apoyo constante, buenas ideas, ánimo en los momentos difíciles, confianza, paciencia y amor.

Por último, y aunque muchos creían que no me atrevería, a Alejandro y a Itziar por los buenos momentos pasados.

A todos, gracias.

Índice General

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

1.1 Antecedentes

La historia ha atribuido a Von Neumann la invención, en la década de los 40, de la máquina de computación universal denominada tradicionalmente *arquitectura Von Neumann.* Con este nombre se designa a cualquier arquitectura (nivel *RT* e *ISA*) de propósito general que puede ejecutar un programa sin necesidad de modificar el *hardware* (figura 1.1(a)). Las ideas de Von Neumann fueron adoptadas rápidamente, llegando a ser fundamentales en las futuras generaciones de computadores, de tal forma, que ya por esas mismas fechas algunos visionarios pensaban en modelos teóricos de máquinas paralelas capaces de emular el funcionamiento del cerebro humano.

En los años 60 se desarrolló una actividad frenética en el diseño de arquitecturas capaces de realizar varios cálculos concurrentemente, apareciendo las denominadas *arquitecturas vectoriales* [103]. La computación vectorial fue propuesta por los diseñadores del supercomputador CDC Star en 1967, con la idea de reducir la latencia de memoria. Así, se propuso el cálculo sobre grandes conjuntos de datos (vectores) que eran tratados por una CPU con varias unidades funcionales trabajando en *pipeline.* Con estas arquitecturas, los resultados de las primeras operaciones tardaban en producirse, sin embargo, los siguientes aparecían en cada ciclo de reloj. Obviamente estas máquinas se aplicaban a aquellos problemas numéricos en los que la arquitectura Von Neumann era inviable debido a los tiempos de ejecución elevados. De esta forma, nació el concepto

de *Supercomputador* para designar al computador más rápido del momento.

El modelo de programación que se adapta a estas arquitecturas es el modelo de *paralelismo en los datos* [89]. Este modelo se caracteriza por la aplicación de la misma operación sobre múltiples elementos de una misma estructura de datos. La programación de estos computadores se abordó construyendo compiladores capaces de obtener instrucciones vectoriales a partir de código escrito en lenguajes escalares secuenciales, apareciendo así las primeras bibliotecas *software*. En [103] se presenta un resumen histórico detallado de estas arquitecturas, así como una descripción de computadores vectoriales típicos como el CDC Star 100, Cyber 205, Cray-1, etc.

A principios de los 80, los avances de los supercomputadores vectoriales habían llegado a su límite, ya que por problemas tecnológicos no se podía seguir aumentando la frecuencia de reloj. Debido a esto, las tendencias en supercomputación se dirigieron a aumentar el número de CPUs que ejecutaban un único programa [108] (figura 1.1(b)). De esta forma surgió el concepto de *multicomputador* como aquella arquitectura formada por más de un computador (que por lo general sigue el modelo Von Neumann). Cada computador de esta arquitectura se denomina *nodo*, y todos ellos aparecen enlazados mediante una *red de interconexión*. Cada nodo presenta una memoria independiente y puede ejecutar su propio programa. Estos programas no sólo leen y escriben en la memoria local del computador sobre el que se ejecutan, sino que pueden enviar y recibir mensajes a través de la red. Estos mensajes están formados por información o datos provenientes de una memoria externa (memoria local de otro nodo del multicomputador). El acceso a la memoria local siempre será más rápido que el acceso a una memoria externa, por lo que es deseable que la gran mayoría de referencias a memoria se hagan sobre la memoria local. A esto se le denomina *localidad* y es una de las características a tener en cuenta en la programación de multicomputadores. De entre todas estas arquitecturas destacaron las basadas en *Transputer* [101], implementadas utilizando un procesador con enlaces de comunicación con cuatro vecinos, el IBM-SP2 [49], CRAY T3E [19], etc.

Los multicomputadores pueden utilizar básicamente dos modelos de programación: paralelismo en los datos y *paso de mensajes*. Con el primer modelo, los compiladores necesitan algún tipo de ayuda sobre la distribución de datos por parte del programador (mediante directivas propias del lenguaje de programación [46]). El segundo modelo

asocia los datos a *tareas*, de forma que cada una de ellas trabaja con sus datos hasta que necesita un dato externo. En este caso, la tarea debe comunicarse mediante un mensaje con el computador propietario del dato solicitado. Ejemplos de bibliotecas de paso de mensajes se pueden encontrar en [79] y [34].

Además de los multicomputadores, existe otro tipo de máquinas paralelas denominadas *multiprocesadores*. En estas arquitecturas todos los procesadores comparten una memoria común a través de un *bus* o de una *red en bus jerárquica*. Un modelo idealizado de multiprocesador es el PRAM [89], en el que cualquier procesador accede a un dato situado en cualquier módulo de memoria en el mismo tiempo. En la práctica, estas arquitecturas suelen tener algún tipo de memoria jerárquica. Para asegurar el principio de localidad, los procesadores tienen asociada una memoria *cache* que almacena los datos más frecuentemente utilizados (figura 1.1(c)). De entre estas arquitecturas destacan la Silicon Graphics [93], CRAY T90 [20], etc.

El modelo de programación propio de los multiprocesadores se denomina modelo *de memoria compartida*. En este modelo, las tareas comparten un mismo *espacio de memoria*. En este caso existen mecanismos para el acceso a memoria, en exclusión mutua, como los *semáforos* y *monitores*.

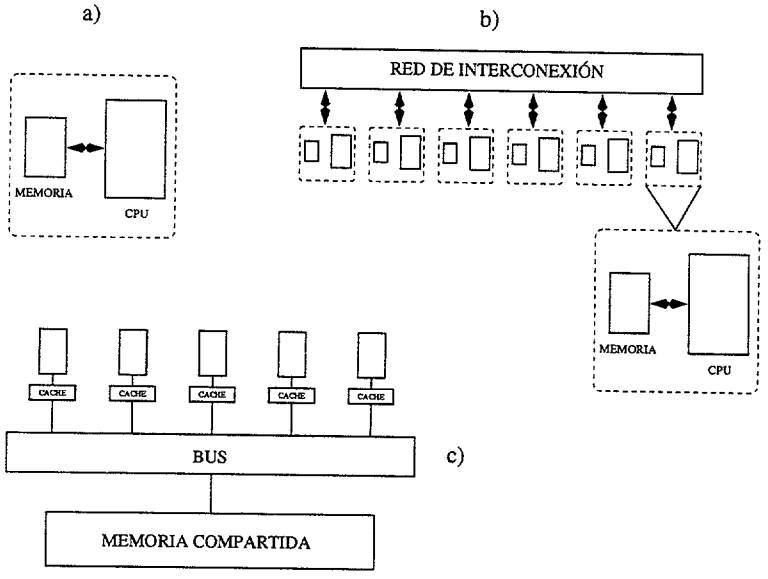


Figura 1.1: a) Arquitectura Von Neumann. b) Multicomputador. c) Multiprocesador

En los años 90, se han popularizado las arquitecturas denominadas *redes de work-*

stations constituidas por agrupaciones de computadores interconectados por una red *Ethernet* [41] o mediante *conmutadores de alta velocidad* [3]. Lo más destacable en este tipo de máquinas es su uso generalizado, aplicándose a problemas tanto numéricos como de índole empresarial. El hándicap de estas arquitecturas es la programación eficiente de las mismas, ya que sólo los programadores expertos pueden abordar este problema con indicios de éxito (generando programas eficientes que aprovechen al máximo las prestaciones de la arquitectura). Ejemplos de estas máquinas se pueden encontrar en [23], [99] (Convex Exemplar SPP-2000X), [78] (NEC SX-4), etc.

El modelo de programación utilizado por estas nuevas arquitecturas es el de paso de mensajes apoyándose en bibliotecas específicas como pueden ser MPI y PVM. Aparentemente, las tendencias futuras caminan en esta dirección, ya que al igual que ocurrió en el pasado, el límite físico de la tecnología en cuanto a diseño de CPUs, hace que los esfuerzos se orienten al desarrollo de redes más rápidas, más que a la mejora del tiempo de reloj o aumentar el número de CPU's.

1.2 Trabajos relacionados

En la actualidad existen problemas cuya solución computacional puede ser resuelta eficientemente en *LANs* o multicomputadores de bajo coste. La programación de estas arquitecturas se realiza usando estrategias a diferentes niveles. A nivel de *sistema operativo* se utilizan *microkernels* que facilitan su programación [1]. A nivel de *lenguaje de alto nivel* se han utilizado lenguajes paralelos como el *HPF* [21] o el *OpenMP* [85]. Los programas escritos en estos lenguajes pueden ser compilados eficientemente considerando distribuciones de datos y particionado de cálculos. A estos compiladores se les denomina *paralelizantes* puesto que el usuario programador escribe un código secuencial y el compilador se encarga de generar un código paralelo semánticamente equivalente. A nivel de *aplicación* se utilizan herramientas de ayuda a la programación paralela [36] que no sólo facilitan la programación a programadores no expertos sino que asisten a los programadores expertos en la optimización de código.

Detrás de cualquiera de estas estrategias está el problema de la distribución de datos y el particionado de los cálculos. Para ello los autores han utilizado diferentes

técnicas (implementadas en cualquiera de los niveles anteriores). Estas aproximaciones tienen en común la utilización de la teoría de grafos [70]. Así, unos autores estudian la distribución y particionado usando la teoría del *empotrado de grafos*. En [18], [45], [65] se plantean estos problemas y mejoras a las métricas de rendimientos que se usan en ese ámbito. Con este empotrado se pretende particionar los cálculos de un programa entre los procesadores de la arquitectura.

Por otro lado, otros autores se centran en técnicas de *alineamiento de datos* en tiempo de compilación. En [62] se introduce un grafo de alineamiento de vectores de varias dimensiones. Los arcos de este grafo indican un posible alineamiento entre los vectores, pesándose éstos en función de la preferencia de alineamiento. En [13] se utiliza un grafo con dos tipos de nodos: de vectores y de lazos. Estos últimos tienen un peso función del tiempo de ejecución del bucle correspondiente. Así, si en un determinado lazo se referencia un determinado vector, existe un arco entre el nodo del lazo y el nodo del vector. El peso de este arco depende del tiempo de comunicación que se produce por la distribución del vector y del bucle en los procesadores correspondientes. En [6] se utiliza un grafo creado a partir del análisis de las sentencias de asignación existentes en el algoritmo. En este grafo existen dos tipos de arcos: de movimiento de datos (que indican un posible alineamiento de los vectores y el coste del movimiento de estos elementos) e hiperarcos de paralelismo (que indican cómo pueden ser paralelizados los bucles y el tiempo que se reduce con tal paralelización).

En cualquier caso, el cálculo de la distribución óptima es un problema NP-completo porque entre los datos, por lo general, existen dependencias [57]. Después de calculada la distribución y posibles redistribuciones de datos se ha de abordar el problema de la comunicación de datos entre los procesadores. El objetivo es eliminarla (aprovechando la localidad de los datos) o reducirla (aprovechando el solapamiento con los cálculos). Sin embargo, este problema se complica debido a la influencia de las dependencias sobre las comunicaciones. El problema del solapamiento de cálculos y comunicaciones ha sido estudiado por varios autores. En [16] se presenta una técnica para solapar cálculos y comunicaciones en algoritmos numéricos, que se basa en reconocer las relaciones existentes entre las comunicaciones y los cálculos a realizar para un problema dado. En este trabajo se describe el *esquema de redistribución intermedia* como aquel en el que se necesita intercambiar resultados, después de calcular una fase y antes de comenzar

la siguiente; el *esquema de repetición de fases* en el que se repiten una serie de cálculos con intercambios de resultados parciales y el *esquema fuertemente dependiente* en el que existen pequeñas tareas de cálculos que necesitan resultados parciales antes de comenzar (el solapamiento de la comunicación se consigue anticipándose a la necesidad de los datos). En [106] se divide la comunicación a realizar en B bloques a comunicar mediante *strip-mining*, donde cada bloque contiene N/B comunicaciones individuales. De igual forma, el bucle de cálculo se divide en B bloques de iteraciones, donde cada bloque de iteración necesita datos que son recibidos en su correspondiente bloque de comunicación. De esta forma, la comunicación del i-ésimo bloque de comunicación se puede solapar con el cálculo del $(i-1)$-ésimo bloque de iteración. En [29] se realiza un solapamiento de comunicaciones en *pipeline*, sobre aquellos algoritmos en los que el cálculo de un elemento dado x_j^i efectuado en la iteración i, depende únicamente del elemento j, calculado en una iteración anterior por uno de sus vecinos.

Por otro lado, otro de los objetivos importante en la programación paralela es el desarrollo de herramientas de visualización, que representen el movimiento de datos, la utilización de los procesadores, las comunicaciones realizadas en el sistema, etc. El primer problema que aparece al diseñar una herramienta de este tipo, es la forma en la que el programador se imagina las relaciones existentes entre los datos y en cómo representar estas relaciones gráficamente. En [44] se referencia una importante bibliografía sobre el problema de la visualización. Asimismo, se indica la necesidad de estas herramientas como ayuda a la pogramación de máquinas paralelas. En [26] se presenta una nueva representación (representación geométrica) de programas paralelos. El punto principal de este trabajo es la forma en la que se expresan los cálculos (encapsulados en objetos geométricos llamados *polytopes*) y las relaciones entre estos cálculos (relaciones espaciales representadas mediante la dimensión del objeto geométrico y la orientación de los cálculos contenidos en él; y relaciones temporales representadas mediante el grafo el dependencias).

Merece especial mención el hecho de que muchos investigadores han desarrollado y analizado compiladores que generan código paralelo. Ejemplos de compiladores paralelizantes o de herramientas basadas en éstos son: parafrase-2 [86] que utiliza un grafo de tareas jerárquico para representar el control de flujo a varios niveles y las dependencias de datos. Asimismo, posee una interfaz gráfica a través de la cual el usuario puede se-

leccionar transformaciones, optimizaciones, etc.; SUIF [97] es un compilador que utiliza un árbol sintáctico abstracto (ASA) para el análisis de las dependencias y representar transformaciones de lazos. Cada procedimiento se representa mediante un árbol de este tipo. A su vez, cada nodo no terminal corresponde con estructuras internas como pueden ser los bucles y cada nodo hoja (nodo terminal) representa instrucciones SUIF. Asimismo este compilador aplica test de dependencias para hacer un análisis de las mismas (análisis escalar, vectorial e interprocedural); ParaScope [17] es una herramienta interactiva para la paralelización de programas que está formado por: a) un sistema de compilación, que informa sobre las dependencias existentes; b) un editor que ayuda al programador en la toma de decisiones; y c) un sistema de depuración que controla los posibles errores en tiempo de ejecución (acceso paralelo para lectura y escritura de la misma variable).

Si bien existen éstos y más compiladores de este tipo, lo cierto es que en la práctica los programadores expertos se muestran reticentes a su uso debido a la poca eficiencia de los códigos generados (en [76] se pueden comparar los resultados obtenidos al generar código paralelo por parte de varios compiladores existentes en el mercado, en relación a los resultados obtenidos en la paralelización manual). Así, varios autores [67] [76], [80], [90], apoyan una solución manual a la paralelización utilizando entornos específicos. Nosotros nos inclinamos por trabajar en esta línea, por lo que hemos desarrollado una herramienta gráfica de ayuda al programador en la paralelización de código secuencial. La herramienta ayuda a distribuir datos y comunicaciones en función de las dependencias existentes entre los datos del programa. A partir de aquí, el programador modifica el código secuencial para realizar la distribución y las comunicaciones propuesta por la herramienta.

1.3 Motivaciones

Hoy día es común el uso de distintas arquitecturas para resolver problemas que demandan gran cantidad de cálculo. Ejemplos de arquitecturas que incluyen más de un procesador son las basadas en procesadores *pentium* [50]. Es normal que en muchos centros de cálculo encontremos LAN de alta velocidad [41],[52] o *clusters* (en [98] se presenta información detallada de *cluster* en linux) o bien máquinas DSM [22].

En este entorno de trabajo es bastante lógico pensar que el usuario programador debe tener conocimientos sobre paralelismo para resolver ciertos problemas, aprovechando el máximo este tipo de arquitecturas. Un ejemplo lo tenemos en los departamentos de *Señales y Comunicaciones* y *Telemática* de la U.L.P.G.C. Los miembros de estos departamentos tienen la necesidad de simular la solución a problemas de comunicaciones o tienen que resolver problemas numéricos que demandan gran cantidad de cálculo. Sin embargo, estos problemas se resuelven en máquinas monoprocesadoras, porque no se tienen los conocimientos adecuados para aprovechar eficientemente la potencia de cálculo de una LAN o una máquina de más de dos procesadores. Por supuesto, este no es un ejemplo aislado ya que ocurre lo mismo en otros departamentos de la Universidad Española y en general de otras Universidades. De igual forma, en algunas empresas del sector informático se pueden encontrar máquinas de este tipo, y aunque no todas resuelven problemas de tipo numérico, si pueden beneficiarse del uso de la programación paralela y de herramientas de ayuda a la programación paralela.

Estos motivos han llevado a considerar el diseño de una metodología de programación de la que el usuario programador no experto se pueda beneficiar, usando herramientas sencillas que le ayuden a aprovechar eficientemente los recursos de la máquina. Es de nuestro interés que sea independiente de la arquitectura y que se pueda aplicar a cualquier tipo de problemas (especialmente a los de Telecomunicación). Asimismo, no deseamos perder de vista las tendencias actuales en *Internet* [51] por lo que consideramos muy interesante diseñar el núcleo de la herramienta de distribución de datos y comunicaciones, de forma que pueda ser utilizada a través de una interfaz *WEB* [105]. Además de estos motivos, al diseñar nuestros algoritmos pretendemos realizar el menor número de comunicaciones posibles (eliminándolas o solapándolas con los cálculos). A partir de aquí, generamos algoritmos eficientes que mejoran en gran medida el tiempo de ejecución secuencial y que, comparativamente con los algoritmos desarrollados por otros autores [39], [100], [48] presentan, en algunos casos, mejores eficiencias.

1.4 Objetivos y aportaciones

El objetivo principal de este trabajo es el desarrollo de una metodología de programación que ayude al usuario programador a usar arquitecturas con más de un procesador.

La metodología presentada consta de los siguientes pasos que se aplican de forma sucesiva:

- *Análisis del algoritmo secuencial.* Partiendo del algoritmo secuencial a paralelizar, se analizan las dependencias existentes entre los datos referenciados en ese algoritmo. Este análisis se realiza de forma gráfica mediante la herramienta desarrollada (GDM).

- *Distribución de datos en los procesadores de la arquitectura paralela.* En base a la información generada por el GDM, se realiza la distribución de datos correspondiente con aquella en la que se consigue máximo paralelismo. Esto puede implicar que exista tanto replicación de datos como desbalanceo de la carga.

- *Establecimiento de las comunicaciones.* Para evitar la replicación de datos y asegurar un balaceo óptimo puede ser necesario las comunicaciones entre procesadores. En este caso, se establece qué procesador se comunica con cuál otro y en qué orden. Asimismo, en la ejecución del algoritmo puede ser necesaria la comunicación de datos entre procesadores. En este caso, esta información la suministra el GDM.

La metodología anterior es una metodología sistemática, con lo que se ayuda al programador a distribuir los datos y conocer las comunicaciones que se deben realizar. Debido a esto, puede ser considerada como una estrategida a *nivel de aplicación.*

Un segundo objetivo ha sido el diseño de una herramienta gráfica que siga los pasos de la metodología. Consideramos que si un programador puede "ver" cómo se utilizan los datos y cómo se comunican estos, le resultará de más ayuda que si esta información se presenta de forma escrita. Asimismo, esta herramienta se ejecuta en cualquier plataforma de computación puesto que ha sido desarrollada usando el lenguaje Java. Nuestra idea es que pueda estar disponible en un servidor WEB e incluso que pueda servir como base para desarrollar una herramienta más completa basada en la tecnología WEB.

Por último hemos creído conveniente demostrar que nuestra metodología es útil para generar la descripción de circuitos electrónicos paralelos que puedan llevar a cabo

las acciones (algoritmos regulares) usando máximo paralelismo, dependiente de las restricciones del *hardware* a utilizar.

Con todo lo anterior el objetivo final es el diseño de una metodología novedosa de programación de multicomputadores, basada en una representación nueva de las dependencias de datos, que se utiliza para distribuir datos de tal manera, que se pueda eliminar las comunicaciones o bien solapar con los cálculos. Además pretendemos que se pueda diseñar *hardware* paralelo ad-hoc utilizando nuestra metodología.

Las aportaciones más importantes de la tesis son las siguientes:

- Aplicamos la metodología a algoritmos del álgebra lineal densos y dispersos de forma sistemática. Las metodologías tradicionales de diseño de algoritmos sistólicos sólo son aplicables a algoritmos densos.
- Obtenemos una representación gráfica de dependencias uniformes y no uniformes.
- Consideramos una amplia gama de algoritmos regulares en los que no se tienen que ejecutar todos los bucles (en algunos casos no se tiene que ejecutar ningún bucle) para conocer la distribución de datos. Además clasificamos los algoritmos a los que podemos aplicar nuestra metodología.
- Formalizamos el cálculo de las dependencias para representarlas en un plano independientemente del número de bucles del algoritmo secuencial. Esto supone una mejora considerable frente a las representaciones tradicionales del *Espacio de Iteraciones* [57] y el *Grafo de Sentencias* [109].
- Se ha estudiado la aplicación a problemas de interés para la Ingeniería de Telecomunicación.

1.5 Estructura de la memoria

El trabajo que hemos llevado a cabo se presenta en los 8 capítulos de la presente memoria. En el capítulo 1 se introducen los antecedentes, estado del arte y los objetivos del trabajo.

En el capítulo 2 se hace una breve descripción de los multicomputadores y su programación, presentando el problema del análisis de las dependencias y el solapamiento de cálculos y comunicaciones. En este capítulo se hace una breve descripción del grafo de dependencias modificado (GDM).

En el capítulo 3 se analiza el GDM en comparación con el espacio de iteraciones y el grafo de dependencias tradicional. En este capítulo se exponen las definiciones y lemas que especifican el grafo objeto de este trabajo.

En el capítulo 4 se estudia el GDM desde el punto de vista de un algoritmo a paralelizar y las relaciones existentes entre los bucles y las dependencias de datos con el grafo generado. Asimismo, en este capítulo se acota el problema presentando distintos tipos de algoritmos sobre el que se aplica la metodología.

En el capítulo 5 se presentan distintos ejemplos del álgebra lineal. Cada uno de estos ejemplos es estudiado respecto a su GDM, estableciendo cómo se distribuyen los datos y qué comunicaciones son necesarias.

En el capítulo 6 se muestran los resultados experimentales de los problemas del capítulo anterior, así como se explican algunos problemas resueltos en el ámbito de la Ingeniería de Telecomunicación.

En el capítulo 7 se explica el desarrollo *hardware* en FPGAs de problemas regulares, alguno de los cuales han sido analizados previamente.

Por último, en el capítulo 8 se presentan las conclusiones y se comentan algunas líneas de trabajo que consideramos interesantes.

Los capítulos anteriores se complementan con la bibliografía y el glosario de términos.

Capítulo 2

Programación de multicomputadores

2.1 Introducción

En la actualidad las arquitecturas paralelas (redes de estaciones de trabajo, multicomputadores de memoria compartida -multiprocesadores- o de memoria distribuida) [89] se muestran como una alternativa barata y adecuada para la resolución de problemas complejos, que necesitan realizar una gran cantidad de cálculos en un tiempo mínimo. Sin embargo, para obtener un rendimiento óptimo de este tipo de arquitecturas hay una serie de tópicos a tener en cuenta, como pueden ser: la *paralelización de los bucles* que constituyen el algoritmo que modela el problema a resolver (*particionado* o *alineamiento* de los bucles y *distribución* de datos y cálculos) y los requisitos de *comunicación*, tópicos muy relacionados entre sí [38].

El diseño de un algoritmo paralelo (aquel que se ejecuta sobre una arquitectura paralela) se puede afrontar siguiendo dos posibles estrategias: la adaptación del algoritmo secuencial óptimo o la creación de un nuevo algoritmo que explote mejor el paralelismo de la arquitectura final. Sea cual sea la estrategia escogida, el objetivo final consiste en reducir el tiempo de ejecución del algoritmo mediante la descomposición del problema (algoritmo) de forma que más de un procesador intervenga en el proceso. Esta descomposición proporciona la distribución de datos y cálculos y establece las necesidades de comunicación. Por lo general, para realizar esta tarea los programadores utilizan su experiencia e intuición y en base a ella, construyen el algoritmo paralelo final. Ante este

panorama, la generación de herramientas de distribución automática de datos, o bien de herramientas que orienten en la distribución de datos, se convierte en una necesidad prioritaria, de manera que se libere al programador de realizar este trabajo.

En cualquier caso, el paralelismo óptimo es aquel que asegura una descomposición del problema tal que los procesadores almacenan los datos necesarios, para que no existan comunicaciones. Sin embargo, esta situación ideal no es la solución real, ya que entre los datos de un algoritmo suelen existir *dependencias* [9] que impiden una solución sin comunicaciones. Según esto, para establecer las necesidades de comunicación de un algoritmo, es importante el análisis de las dependencias existentes entre los datos implicados en el cálculo. Este análisis se puede realizar: a) de forma teórica mediante las distintas formas de representar el *vector distancia* de dependencia [57]: *nivel de dependencia* [2], *vector dirección de dependencia* [109], *dependencias afines* [37]; o b) de forma gráfica mediante el *Espacio de Iteraciones* (*EI*) [57], el *Grafo de Sentencias GS* [109] o el *Grafo de Dependencias Modificado* (*GDM*).

Después del análisis de las dependencias existentes, se realiza la distribución de datos (y consecuentemente de cálculos) en los procesadores de la arquitectura paralela. Tradicionalmente existen dos técnicas de distribución de datos: a) el programador establece dicha distribución utilizando directivas propias de ciertos lenguajes y compiladores paralelizantes [46], o b) el compilador se encarga de resolver este problema (usando las técnicas de alineamiento y distribución) [43]. La primera opción es más utilizada que la segunda, ya que la realización de un compilador de propósito general, que distribuya los datos de forma óptima, es aún un tema a resolver. Sin embargo, sea cual sea la técnica empleada, el problema fundamental que se plantea en la distribución de datos está en encontrar aquella distribución que optimice las comunicaciones, bien eliminándolas o bien solapándolas con los cálculos. Estos cálculos se suelen concentrar mayoritariamente, en los bucles de los programas secuenciales.

2.2 Paralelización de bucles

El código de la mayoría de los problemas científicos está constituido por bucles anidados que presentan un alto grado de paralelismo y, a la vez, consumen la mayor parte del

tiempo de ejecución [30]. De estos bucles, los lazos afines (aquellos cuyos límites son funciones afines de las variables de iteración de otros bucles) son ampliamente utilizados tanto en problemas del álgebra lineal [42] como en problemas típicos de ingeniería [66], [83], [84].

El primer paso en la generación de un algoritmo paralelo consiste en detectar aquellos bucles que puedan ser realizados en paralelo. Así, un bucle se puede ejecutar en paralelo (por más de un procesador) si no existen dependencias entre sus iteraciones o parte de sus iteraciones. La detección de estos bucles se lleva a cabo mediante los algoritmos de paralelización de lazos, cuyo propósito está en transformar los bucles de alguna forma. Así, una transformación lineal, aplicada a un conjunto de bucles, cambia la estructura de un lazo con el objetivo de cambiar el orden de ejecución de las iteraciones. Sin embargo, la estructura de las iteraciones no son alteradas de forma que en una iteración dada, en un nuevo espacio de iteraciones, se realizan las mismas operaciones de cálculo que las que se realizaban en el espacio original.

Existen distintas técnicas de transformación de lazos, también llamadas transformaciones primarias, entre las que se puede destacar [109]: *loop interchange*, *wavefront*, *skewing* y *tiling*. Sin embargo, para conseguir explotar al máximo las características de la arquitectura, es necesario utilizar combinaciones de una o más tranformaciones primarias, dando lugar a una transformación unimodular [8], [59]. Las transformaciones unimodulares son representadas mediante matrices elementales y por tanto, unimodulares [9]. La limitación de las transformaciones unimodulares radica por un lado, en que sólo se pueden aplicar a lazos perfectamente anidados y por otro, a que se deben aplicar las mismas transformaciones a todos los lazos del programa. Con el objeto de eliminar estas restricciones numerosos autores han desarrollado nuevas técnicas que permiten la unificación de diversas transformaciones [55], [58]; o el uso de otro tipo de transformaciones como pueden ser las afines [13].

2.2.1 Análisis de las dependencias de datos

Cualquier transformación aplicada sobre un conjunto de bucles parte del análisis de las dependencias. Así, una sentencia que utiliza para lectura o escritura, un valor calculado por una sentencia de ejecución anterior en el tiempo, se dice que tiene una dependencia

de datos sobre ella.

Existen tres tipos de dependencias: de flujo, antidependencia y de salida [9]. De todas ellas, la más importante es la *dependencia de flujo* que se produce cuando una variable es asignada en una sentencia S_1 y más adelante es utilizada en otra sentencia S_2 ($S_1 \delta S_2$).

Consideremos los siguientes bucles:

```
      do I1 = L1, U1, Step1
         ...
         do Ik = Lk, Uk, Stepk
S1:         A(f1(I), ..., fn(I)) = ...
S2:         ... = A(g1(I), ..., gn(I))
         enddo
         ...
      enddo
```

(2.1)

donde A es un vector de n dimensiones y f_i y g_i son funciones de Z^k a Z donde Z es el conjunto de todos los enteros [87]. En este algoritmo existe una dependencia de flujo $S_1 \delta S_2$ en el bucle más interno, ya que el valor asignado en S_1 es posteriormente utilizado en S_2. Para caracterizar cualquier dependencia de datos se utiliza la *distancia de dependencia*, que corresponde con el número de iteraciones que hay entre la fuente y el sumidero de la dependencia.

En la figura 2.1 se presentan dos bucles anidados y una sentencia interior que tiene una dependencia de flujo sobre sí misma (entre las distintas iteraciones de los bucles). Si desenrollamos los lazos se observa que el valor situado en la posición (i, j) es usado en la iteración (x, y) si $i = x$ y $j = y - 1$, por tanto, el vector distancia de dependencia es (0,1) ya que $x - i = 0$ e $y - j = 1$.

El análisis de las dependencias de flujo se puede realizar visualmente mediante: a) el grafo de dependencias en el espacio de iteraciones (EI) y b) el grafo de sentencias (GS). En el EI, cada vértice representa las operaciones a realizar entre datos, y cada arco, las dependencias existentes entre ellos. La dimensión del EI depende del número de bucles del algoritmo. Cuando el número de bucles es superior a tres, no es posible

la representación gráfica del mismo. Asimismo, cuando existe más de una dependencia de datos en un algoritmo, el número de arcos del grafo añade complejidad extra. En el GS, cada vértice representa una sentencia del algoritmo, utilizándose el peso del arco que las une, para establecer la distancia de dependencia. Esta representación es menos intuitiva ("visual") que la representación en el EI y no ofrece ayuda a los programadores no expertos, ya que con ella se pierde la visión natural del paralelismo existente en el algoritmo. En la figura 2.2 se presenta el algoritmo secuencial de la multiplicación matriz-vector y las dependencias en el EI y en el GS, siendo $n = 5$.

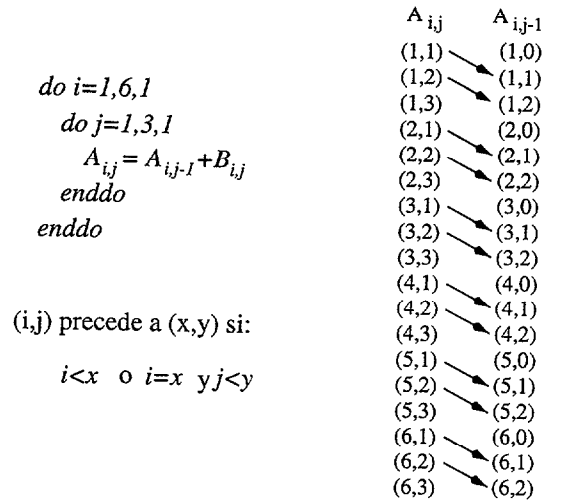


Figura 2.1: Ejemplo de bucles anidados

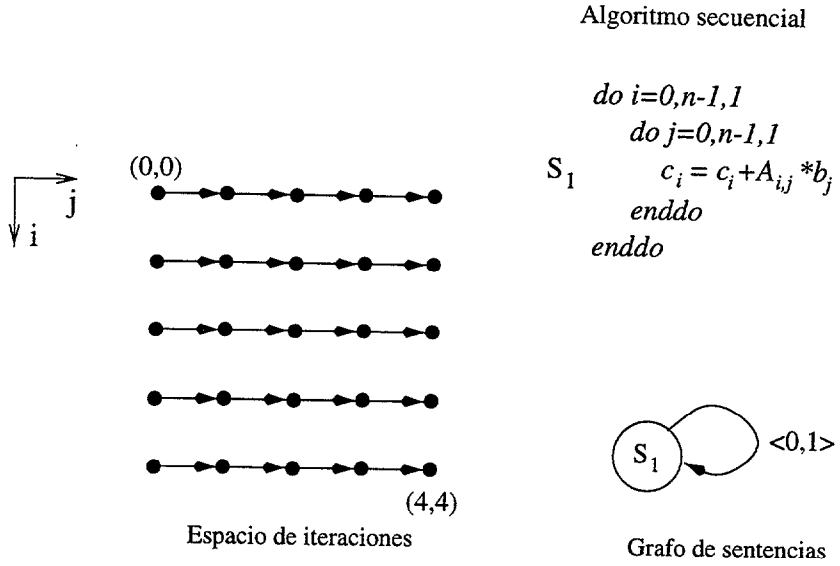


Figura 2.2: Ejemplo de algoritmo y representación de las dependencias

2.2.2 Particionado de algoritmos

Una vez analizadas las dependencias existentes entre los datos del algoritmo a paralelizar, se puede realizar el particionado del algoritmo. Este particionado consiste en distribuir los cálculos a realizar, entre los procesadores de la arquitectura seleccionada, de forma que éstos se ejecuten eficientemente (alineando los bucles en los distintos procesadores).

Un determinado algoritmo puede ser particionado de múltiples formas sobre una determinada arquitectura, teniendo cada una de ellas unos requisitos de comunicación particulares. Asimismo, debe ser posible cambiar de una máquina a otra (de igual o distintas características) de forma eficiente. Algunos autores estudian este problema desde el punto de vista del isomorfismo de grafos y realizan un proceso de empotrado (*embedding*). Una idea que subyace detrás de este proceso es que una red de interconexión de procesadores (arquitectura paralela) se puede ver como un grafo en el cual los nodos son los procesadores y los arcos los enlaces físicos entre ellos. Las principales medidas de calidad de un empotrado son: la dilatación, congestión, expansión y carga máxima [18]. En [65], [45] se presentan empotrados entre diferentes topologías de interconexión de procesadores que intentan mejorar las métricas anteriores.

Por otro lado, se han realizado grandes esfuerzos en el estudio de distintas técnicas de alineamiento en tiempo de compilación. Así, en [62] los autores introducen el Grafo de Componentes Afines (CAG). En este grafo, los nodos se organizan en columnas donde cada una de ellas representa un vector. Entre dos nodos cualesquiera (i_a e i_b) existe un arco, si aparece el mismo subíndice en la dimensión i_a del vector a y en la dimensión i_b del vector b. En este caso se dice que existe una afinidad entre ambas dimensiones. Los arcos de este grafo indican un posible alineamiento entre los vectores, pesándose éstos en función de la preferencia de alineamiento. La idea de los autores se basa en agrupar las columnas en subconjuntos de forma que no existan nodos de una misma columna en el mismo subconjunto. Esta división puede provocar la eliminación del paralelismo (ejecución secuencial de bucles). En [13] se utiliza el Grafo de Alineamiento-Distribución (ADG). En este grafo existen dos tipos de nodos: nodos de vectores (igual que en el CAG) con peso cero y nodos de lazos con un peso que es función del tiempo de ejecución de ese bucle. Así, si en un determinado lazo

se referencia un determinado vector, existe un arco entre el nodo del lazo y el nodo del vector. El peso de este arco depende del tiempo de comunicación que se produce por la distribución del vector y del bucle en los procesadores correspondientes. En [6] se utiliza el Grafo de Paralelismo-Comunicación (CPG) creado a partir del análisis de las sentencias de asignación que existen dentro de los bucles y que actualizan vectores. Los autores suponen vectores de d dimensiones y lazos perfectamente anidados. Los nodos del CPG se organizan de igual forma que los del CAG. Sin embargo, existen dos tipos de arcos: arcos de movimiento de datos (que indican un posible alineamiento de los vectores y el coste del movimiento de estos elementos) e hiperarcos de paralelismo (que indican cómo pueden ser paralelizados los bucles y el tiempo que se reduce con tal paralelización).

2.2.3 Distribución de datos

Estrechamente relacionado con la partición de bucles está la distribución de datos entre los procesadores de la arquitectura. Determinar la distribución óptima de datos para una aplicación específica es una tarea difícil de resolver para la que hay que tener en cuenta numerosos condicionantes. Por un lado está el excesivo coste de las comunicaciones en comparación con los cálculos a realizar internamente en un procesador. Así, los datos se deben distribuir en los procesadores de forma que se elimine (localidad de los datos) o reduzca (solapamiento con los cálculos) el número de comunicaciones.

Como idea básica se puede establecer que la distribución de un espacio $D = (D^1, D^2, ..., D^m)$ de datos sobre un espacio $P = (P^1, P^2, ..., P^n)$ de procesadores se puede modelar mediante una función $f_d : D \rightarrow P$. Si la dimensión y tamaño del espacio P de procesadores es mayor o igual al espacio de datos D ($n \geq m$), la función de distribución puede ser lineal (se entiende por tamaño de un espacio el número de elementos de ese espacio). Sin embargo, éste no es el caso típico. Por lo general, la función de distribución es una función no lineal que incluye operadores de módulo, el entero inmediatamente superior, etc. Por tanto, el problema que se presenta consiste en encontrar una distribución óptima de datos para todas las estructuras de datos vectoriales que forman parte de un programa. Este problema es un problema NP-completo, ya que lo más común es que entre los datos existan dependencias de flujo [57]. Debido

a esto, existen dos tendencias ante la distribución de datos: distribución realizada por el usuario (según la información aportada en el análisis visual de las dependencias) *vs* distribución automática. Siguiendo la segunda alternativa, algunos compiladores paralelizantes (Fortran D, Fortran 90D, Vienna-Fortran [11], Paradigm [7], etc.) han incluido herramientas de distribución automática de datos.

2.2.3.1 Estrategias de particionado: regulares e irregulares

Existen distintas estrategias de distribución de datos que tienden a aprovechar el uso de la memoria y a mejorar la localidad de las referencias en programas paralelos. Estas técnicas se pueden dividir en dos grandes grupos: orientadas a problemas regulares y orientadas a problemas irregulares. Para la distribución de datos en problemas regulares podemos hacer la siguiente clasificación (figura 2.3):

- Distribución por bloques. Consiste en dividir el vector en subvectores contiguos y de igual tamaño asignándolos a diferentes procesadores. Un vector con m elementos se distribuye en n procesadores asignando subvectores de longitud $\lceil \frac{m}{n} \rceil$ a cada procesador.

- Distribución cíclica. Se basa en dividir el vector bidimensional en subvectores unidimensionales que se distribuyen siguiendo la filosofía *round-robin*. A diferencia de la distribución por bloques, esta estrategia maximiza el coste de comunicación ya que, por lo general, elementos vecinos en el vector están en procesadores diferentes.

- Distribución bloque-cíclica que se caracteriza por ser una combinación de las anteriores. A esta distribución también se la conoce como cíclica(g) donde g es el tamaño del bloque, múltiplo de $\frac{m}{n}$.

Sin embargo, para el caso de problemas irregulares los métodos de distribución estudiados anteriormente, generalmente, no son óptimos [88]. El análisis de un programa en tiempo de compilación es insuficiente cuando no se conocen previamente las dependencias de datos y las comunicaciones que se tienen que establecer entre los procesadores. Debido a esto, es necesario realizar un cierto preproceso en tiempo de ejecución antes

de realizar los cálculos de cada iteración. En este preproceso se define la ubicación de los datos y los patrones de comunicación. En la práctica, cada paso de un problema irregular se resuelve siguiendo dos fases: fase del inspector y fase del ejecutor. El inspector se encarga de realizar la etapa de preproceso y traducir los índices globales a locales y el ejecutor realiza los cálculos de esa etapa y comunica los elementos utilizando la información suministrada por el inspector. Para simplificar las fases anteriores existen herramientas de distribución automática para problemas irregulares, que poseen primitivas para la realización de estas operaciones. Entre estas herramientas se puede destacar Chaos/Parti (*Parallel Automated Runtime Toolkit*) [91].

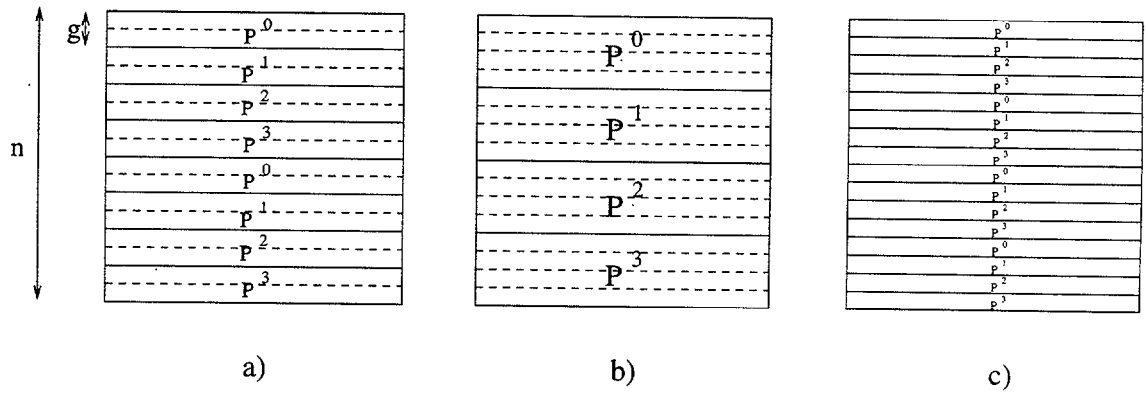


Figura 2.3: Ejemplos de distribución regular de datos donde $n = 4$ y $m = 16$. a) Bloque-cíclica ($g = 2$). b) Distribución por bloques. c) Distribución cíclica

2.3 El problema de la comunicación

En los multiprocesadores, dado que existe una memoria común, el problema de la comunicación se convierte en un acceso eficiente a los datos almacenados en dicha memoria (más de dos procesadores no pueden acceder a la vez, con una orden de escritura o una de escritura y otra de lectura, a una localización de memoria determinada). Por el contrario, en los multicomputadores y en las redes de área local, el problema de la comunicación radica en la topología de interconexión de los procesadores, topología que generalmente es estática. En la figura 2.4 se presentan algunos ejemplos de topologías de interconexión de procesadores.

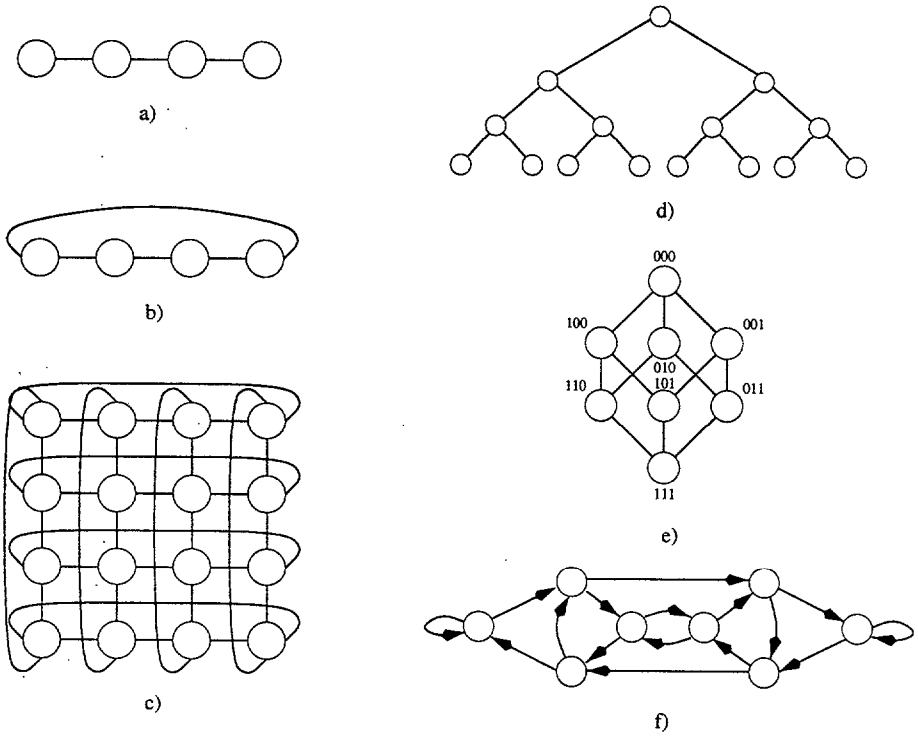


Figura 2.4: a) Línea. b) Anillo. c) Toro 2D. d) Árbol. e) Hipercubo 3D. f) Bruijn

2.3.1 Modelos de comunicación

Un modelo de comunicación establece la forma en que dos procesadores, de una determinada topología de interconexión, se comunican; definiendo el modo en que se utilizan tanto los enlaces como los *buffers* de datos. Los mensajes se pueden transmitir de un procesador a otro siguiendo, básicamente, dos modelos de comunicación: *store-and-forward message passing* (figura 2.5(a)) y *circuit-switched message routing* (figura 2.5(b)) [89]. El primer modelo se basa en que cada procesador que se encuentra en el camino de una comunicación entre dos procesadores, debe almacenar por completo el mensaje y luego comunicarlo (típico de multicomputadores de la primera generación como por ejemplo los multicomputadores basados en transputer T800). El segundo método se basa en la utilización de un módulo de comunicación por computador de forma que éste se encarga de realizar las comunicaciones, independizando de esto al procesador de cálculo. En este caso, se establece un camino de enlace entre los procesadores y se envía el mensaje completo.

Una variante del segundo modelo, conocido como *dynamic circuit switching* [18],

se produce cuando el mensaje comienza a transmitirse aunque el camino no se haya establecido. En este caso, los paquetes transmitidos se van almacenando en los procesadores intermedios hasta que finalmente llegan al procesador receptor, liberándose el enlace completo (figura 2.5(c)). Dentro de este último modelo se puede considerar dos variantes: *wormhole* [73], en el que los procesadores intermedios almacenan los mensajes enviados y los enlaces se van liberando a medida que dejan de utilizarse; y el *virtual cut through*, en el que si se detecta conflicto, los mensajes almacenados en los nodos intermedios se envían al procesador que detectó el problema.

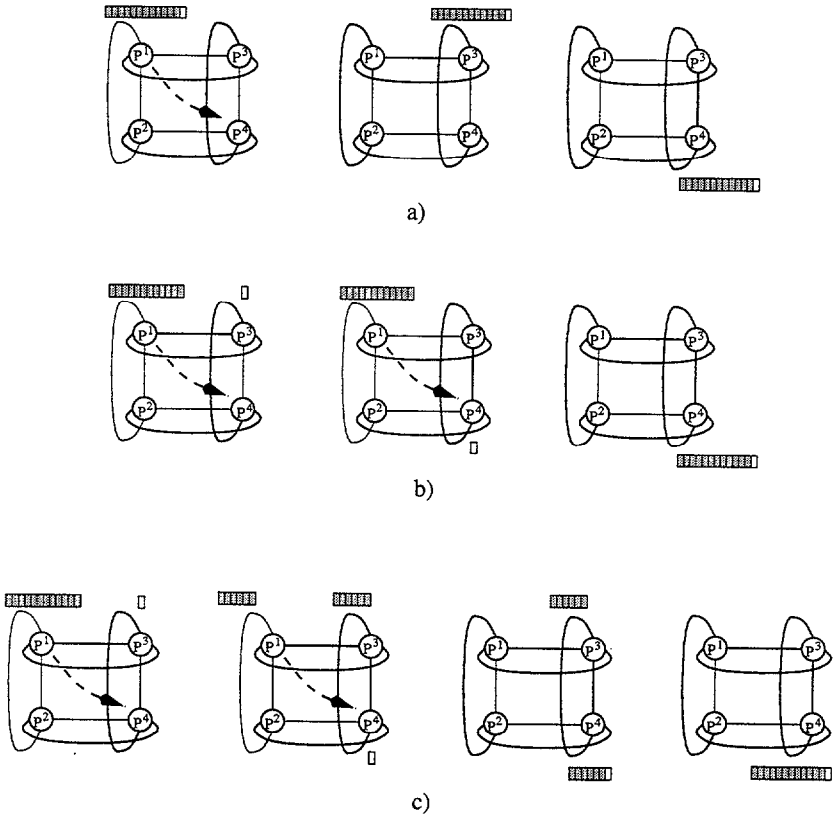


Figura 2.5: a) *Store-and-forward message passing.* b) *Circuit-switched message routing.* c) *Dynamic circuit switching*

2.3.2 Tipos de comunicación

La mayoría de los multicomputadores soportan modelos de paso de mensajes en el cual los procesadores trabajan asíncronamente comunicándose mediante primitivas de

comunicación. Por lo general, los multicomputadores que soportan paso de mensajes utilizan dos tipos de comunicación: punto a punto -*send* y *receive*- (dos procesadores involucrados) y comunicaciones colectivas -*broadcast*, *scatter*, *gather*, *total exchange*, *shift*, *reduction* y *scan*- (un grupo de procesadores involucrados). Los modelos de pasos de mensajes difieren en base a términos de selectividad, ordenación de los mensajes, sincronización y bloqueo [25].

En cuanto a la selectividad y ordenación de los mensajes, los modelos de paso de mensajes permiten establecer qué procesador es el emisor y cuál es el receptor de las comunicaciones (desde un procesador hasta todos los procesadores, pasando por un grupo de estos). Asimismo, los mensajes pueden llevar una etiqueta (*tag*) que los identifica del resto de mensajes. Esta etiqueta se puede utilizar para recuperar el mensaje en el orden adecuado, ya que al receptor le pueden llegar los mensajes de forma aleatoria, en función de las características y comportamiento del sistema de comunicación empleado. Básicamente los parámetros que caracterizan a un sistema de comunicación son la latencia y el *throughput*. La latencia de un mensaje determina de forma restrictiva la granularidad del problema y el *throughput* establece la cantidad máxima de datos que se pueden intercambiar en una unidad de tiempo. En general, el tiempo de comunicación t_{com} de un mensaje de longitud L se puede descomponer en el tiempo de *start-up* β (tiempo necesario para que la cabecera del mensaje llegue al destino) y el τ_t que es el tiempo que tarda el resto del mensaje en ser transmitido, una vez que se ha conseguido establecer el camino entre el emisor y el receptor. De esta forma, $t_{com} = \beta + L \times \tau_{com}$ (donde τ_{com} es el tiempo requerido para transmitir un *byte* y $\tau_t = L \times \tau_{com}$).

Las propiedades de bloqueo y sincronización se refieren al momento en el que se retorna después de la llamada a la primitiva de envío o recepción, existiendo cuatro opciones diferentes: bloqueante asíncrona (no se retorna hasta que el mensaje no haya sido extraído del *buffer* de envío), bloqueante síncrona (no se retorna hasta que el mensaje no haya llegado al receptor), no-bloqueante asíncrona (puede retornar antes de que el mensaje haya sido extraído del *buffer*. Esta opción lleva asociada una primitiva de espera -*wait*- que no retorna hasta que el mensaje haya sido extraído del *buffer*) y no-bloquante síncrona (puede retornar antes de que el mensaje haya sido extraído del *buffer*. Esta opción lleva asociada una primitiva de espera -*wait*- que no retorna hasta

que el mensaje haya llegado al receptor) [25]. Las primitivas no bloquantes presentan dos ventajas con respecto a las bloqueantes: por un lado, eliminan la necesidad de *buffers* en el subsistema de comunicación; y por otro, permiten el solapamiento de los cálculos y las comunicaciones. En la figura 2.6(a) se muestra un ejemplo de envío bloqueante, entre los procesadores de un anillo, cuando no existen *buffers* en el subsistema de comunicación. En este ejemplo, todos los procesadores del anillo intentan enviar el dato pero ningún vecino lo puede recibir porque están bloqueados en su propia operación de envío. Teniendo en cuenta las características de la primitiva *blocking send* que no se considera finalizada hasta que el dato no salga del *buffer* de envío o no llegue al receptor, los procesadores del anillo permanecerán en *deadlock*. Si existen *buffers* en el subsistema de comunicación, este problema no ocurre porque los datos se envían a estos *buffers* (figura 2.6(b)), y todos los procesadores pueden pasar a recibir desde éstos utilizando la primitiva *blocking receive* (figura 2.6(c)).

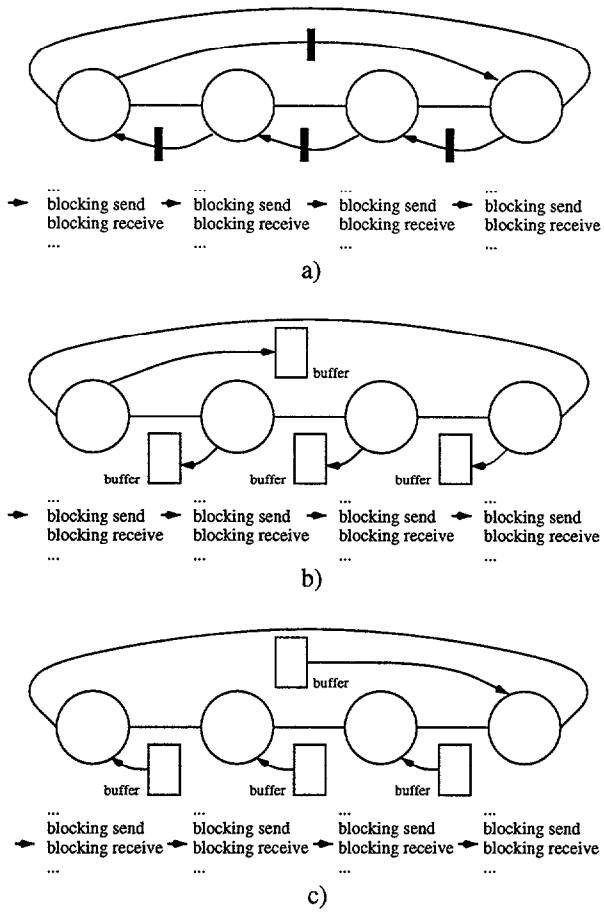


Figura 2.6: a) *Deadlock*. b) Envío con *buffer*. c) Recepción con *buffer*

Por el contrario, la necesidad de *buffers* en el subsistema de comunicación se evita con el uso de primitivas no bloqueantes ya que los procesadores comienzan el envío (almacenando los datos en el *buffer* de envío, interno al procesador) y sin que esta primitiva haya finalizado su ejecución, comienzan la recepción de los datos procedentes de su vecino (almacenándolos en el *buffer* de recepción, interno al procesador).

2.4 Efecto de las dependencias sobre las comunicaciones

Supongamos que las l iteraciones del algoritmo que describe el problema a resolver, se pueden particionar entre los $nProc$ procesadores de una arquitectura paralela, de forma que a cada uno de ellos le corresponden $it = \frac{l}{nProc}$ iteraciones. Asimismo, consideramos que t_0 es el tiempo en el que los procesadores P^i y P^j realizan la primera iteración de su partición, y que $t_{v(w)}$ corresponde con el tiempo en el que cualquiera de ellos realiza la iteración $v(w)$, donde $0 \leq v, w \leq it - 1$. Según esto, se pueden dar los casos siguientes:

- $t_v = t_w$
- $t_v < t_w$

siendo t_v el tiempo en el que el procesador P^i realiza la iteración v y t_w el tiempo en el que el procesador P^j realiza la iteración w.

Asimismo, supongamos que los cálculos realizados en la iteración $v(w)$ por el procesador $P^{i(j)}$ involucran a un conjunto de L^{it} datos (conjunto $D^{v(w)}$) donde $L^{it} < L$ (L es la cantidad de datos que se envían en un mensaje). Según esto, puede ocurrir que:

- $\nexists\, d_1 \delta d_2, \forall d_1, d_2$ tal que $d_1 \in D^v$ y $d_2 \in D^w$. Es decir, no existe dependencia entre datos del conjunto D^v y D^w. En este caso, todos los elementos dependientes se encuentran en el mismo procesador. Si esto ocurre, no se necesitan comunicaciones de datos y los cálculos sobre los datos anteriores se pueden realizar en paralelo.

- $d_1 \delta d_2$, $\forall d_1, d_2$ tal que $d_1 \in D^v$ y $d_2 \in D^w$. Es decir, existe dependencia entre algún dato del conjunto D^v y algún dato del conjunto D^w. En este caso, todos los elementos dependientes no se encuentran en el mismo procesador. Si esto ocurre, se necesitan comunicaciones que pueden ser solapadas con los cálculos (en función de t_v y t_w). Así, el solapamiento de cálculos y comunicaciones se consigue alterando la granularidad del problema y utilizando primitivas no bloqueantes de comunicación, de forma que un procesador pueda calcular a la vez que comunicar, ya que no tiene que esperar por el receptor del mensaje. La elección del tamaño del grano (cantidad de trabajo realizada por procesador entre comunicaciones) es una tarea compleja y depende del problema en sí, de la velocidad del procesador y de la eficiencia de las comunicaciones (cuanto mayor sea el tamaño del grano, menor será el número de mensajes a enviar). Según sea el solapamiento realizado se tienen las siguientes posibilidades (figura 2.7):
  - Las comunicaciones y los cálculos no pueden ser solapadas: $t_{exe} = t_{com} + t_{cal}$.
  - Las comunicaciones y los cálculos se pueden solapar completamente: $t_{exe} = t_{cal}$.
  - Las comunicaciones y los cálculos se solapan parcialmente: $t_{exe} = t_{cal} + t^*_{com}$.

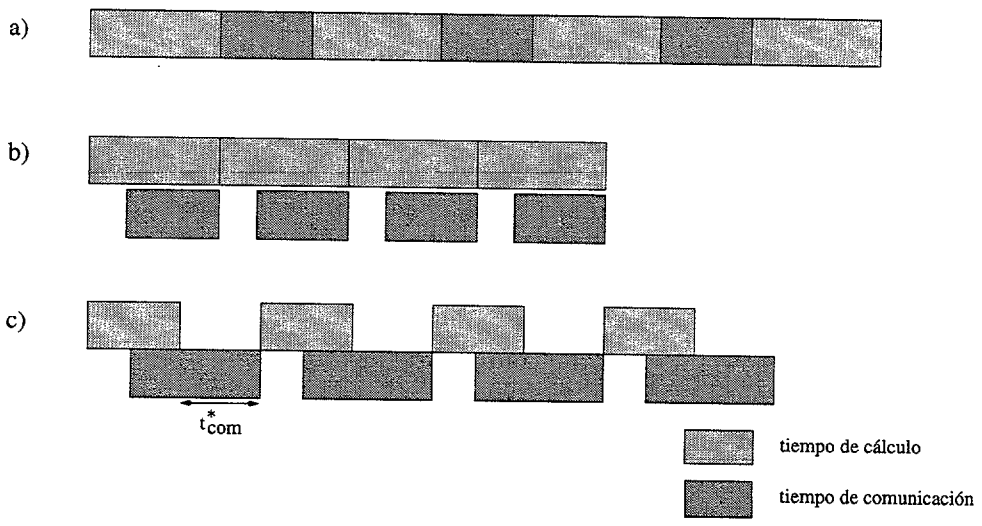


Figura 2.7: a) No existe solapamiento. b) Solapamiento total de comunicaciones y cálculos. c) Solapamiento parcial de comunicaciones y cálculos

En [16], [14] y [106] se presentan distintas técnicas de solapamiento de cálculos y comunicaciones basadas en las relaciones entre las comunicaciones y cálculos a realizar, en la utilización de dos procesadores (uno para comunicar y otro para calcular) y en el *strip-mining* respectivamente. En [56] se presenta un análisis de diferentes transformaciones para detectar cuál de ellas produce paralelismo máximo, minimizando las comunicaciones (mediante solapamiento total o parcial). En cualquier caso, los autores necesitan un análisis muy detallado de las dependencias para que el resultado sea óptimo.

2.5 Metodología de distribución de datos

En este apartado presentamos nuestra metodología de distribución de datos. Se denomina método al modo ordenado de proceder para llegar a un resultado o fin determinado y a metodología se la define como el conjunto de métodos utilizados para alcanzar el fin propuesto. El modo ordenado de proceder se puede traducir también por sistemático, dando a entender la existencia de una relación entre los distintos métodos que forman la metodología. Según esto, podemos definir metodología de distribución de datos como aquellas técnicas realizadas en un orden dado, que consiguen distribuir, de alguna forma, los elementos que se utilizan en un determinado algoritmo.

La generación de un algoritmo paralelo debe venir precedida por la aplicación de una metodología sistemática que pueda llegar a ser automatizada. Si esto es así, se descarga al programador de la tarea tediosa de analizar las dependencias existentes en el algoritmo que modela el problema a resolver, para establecer la distribución de datos y posteriores comunicaciones. La metodología propuesta se puede desglosar en los siguiente puntos:

1. *Análisis del algoritmo secuencial.* De las dos estrategias a seguir para generar un algoritmo paralelo se elige aquella que parte del algoritmo secuencial correspondiente, si es posible, debido a que es mejor utilizar un algoritmo correcto. Así, la detección de paralelismo (conocer qué parte del código se debe ejecutar en paralelo) y comprender el por qué (en función de las dependencias existentes entre los datos) es independiente de la arquitectura paralela que finalmente eje-

cutará el código. Estos conceptos sólo dependen del algoritmo secuencial a ser paralelizado.

El análisis del algoritmo secuencial lleva consigo el estudio de las dependencias existentes entre los datos, para establecer cómo y cuándo deben ser utilizados. Este análisis puede ser realizado de forma gráfica mediante el *EI* o el *GS*. Sin embargo, estas herramientas presentan problemas. Una solución al problema de la representación de las dependencias de datos está en la utilización de una alternativa, que aporte la misma información, pero que no esté limitada al número de bucles y que sea apropiada para programadores no expertos. La representación propuesta es el *GDM* o Grafo de Dependencias Modificado (figura 2.8).

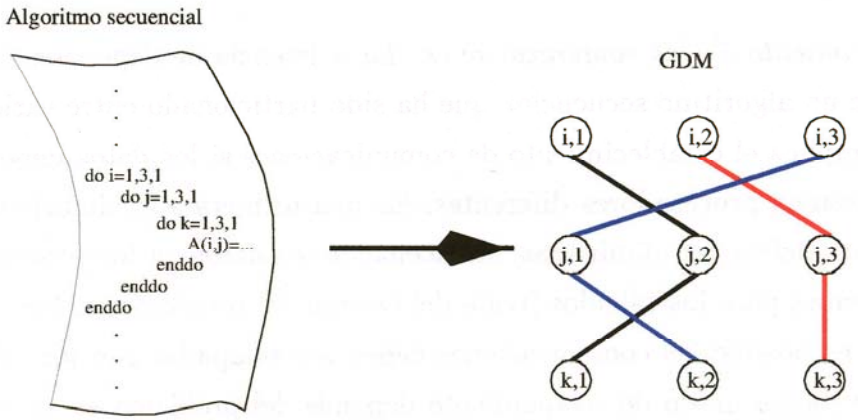


Figura 2.8: Análisis del algoritmo secuencial

2. *Distribución de datos en los procesadores de la arquitectura paralela.* Una vez analizadas las dependencias se realiza la distribución inicial de datos que se desprende del grafo del algoritmo secuencial (*GDM*). Esta distribución de datos corresponde con una distribución para obtener máximo paralelismo (figura 2.9). Sin embargo, la distribución inicial no suele ser la misma en las dististas fases de ejecución del algoritmo paralelo final. Así, si existen dependencias de datos, en la iteración w del procesador P^i se pueden necesitar datos calculados en la iteración v del procesador P^j. Esto implica una nueva distribución de datos o *redistribución* (comunicación de elementos). Esta información también está representada en el grafo del algoritmo secuencial, de forma que este grafo no sólo almacena información sobre la distribución inicial de datos sino también sobre las comunicaciones necesarias.

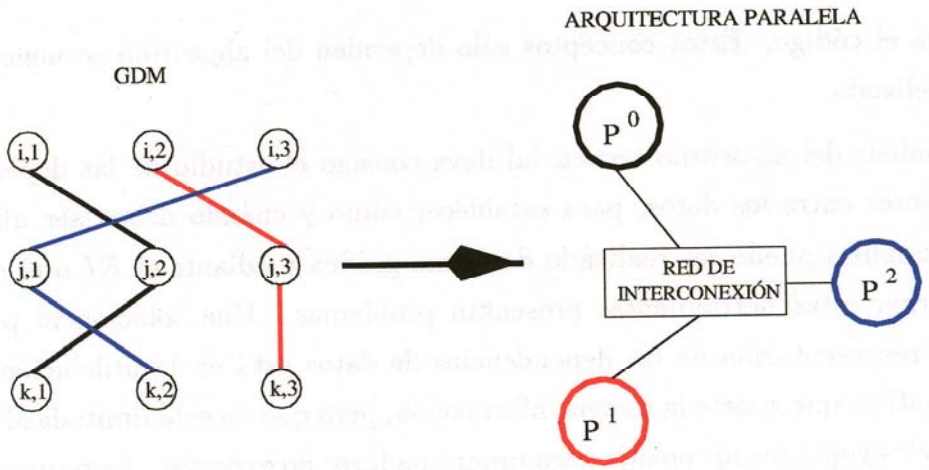


Figura 2.9: Distribución de datos en función del *GDM*

3. *Establecimiento de las comunicaciones.* La existencia de dependencias entre los datos de un algoritmo secuencial, que ha sido particionado entre varios procesadores, implica el establecimiento de comunicaciones si los datos dependientes se encuentran en procesadores diferentes. En una primera aproximación, las comunicaciones deben ser eliminadas, almacenando los datos en los procesadores que los necesitan para los cálculos (regla del cálculo del propietario). Sin embargo, si esto no es posible las comunicaciones deben ser solapadas con los cálculos. Un mayor o menor grado de solapamiento depende del problema en sí. Según esto, se pueden generar dos tipos de algoritmos paralelos (tabla 2.1).

```
do i=1,v_f,Step
   par
      calcular datos^(i-1)
      comunicar datos^i
   endpar
enddo
calcular datos^(v-f)
```

```
calcular datos^0
comunicar datos^0
do i=1,v_f,Step
   par
      calcular datos^(i-1)
      comunicar datos^i
   endpar
enddo
calcular datos^(v-f)
```

Tabla 2.1: Tipos de algoritmos paralelos

En el primer algoritmo de la tabla 2.1, la distribución inicial de datos asegura

la ejecución de la primera iteración sin necesidad de comunicaciones. De esta forma, el primer bloque de cálculo se puede realizar de forma solapada con las comunicaciones (parcial o totalmente, según el valor del tiempo de cálculo t_{cal} y del tiempo de comunicación t_{com}). En el segundo algoritmo de la tabla anterior, es necesario comunicar un bloque de datos, que deben ser previamente calculados, antes de realizar la primera iteración. A partir de esta iteración el solapamiento total o parcial es función de los valores comentados.

2.5.1 Introducción al GDM

El *GDM* está formado por vértices unidos mediante arcos (con dos pesos: "color" y "orden") y tiene una estructura planar. En cada fila del plano se presentan los distintos bucles que rodean a la dependencia analizada; y en cada columna, los posibles valores de los índices de estos bucles.

En el *GDM* se representan las dependencias entre los datos mediante el coloreado de los arcos. De esta forma, arcos de igual "color" representan datos que tienen dependencias y que por tanto, deben estar en el mismo procesador antes de que se realice el cálculo. De esta característica se deduce que existen tantos procesadores como colores diferentes. Asimismo, la ordenación de los cálculos en un procesador se realiza en función de otro peso del arco: el "orden". Supongamos el algoritmo siguiente:

```
do i=0,2,1
   do j=0,2,1
      C_{i,j}=0
      do k=0,2,1
         C_{i,j}=C_{i,j}+A_{i,k} × B_{k,j}
      enddo
   enddo
enddo
```

(2.2)

El *GDM* de este algoritmo se muestra en la figura 2.10. En este grafo se observa que existen tres filas de nodos, ya que existen tres bucles ($n = 3$) que rodean a la

dependencia $C_{i,j} = C_{i,j} + ...$ (i, j, y k) y tres columnas. Así, analizando el algoritmo se observa que el elemento $C_{0,0}$ depende de su valor previo y de los elementos $A_{0,k}$ y $B_{k,0}$, siendo $0 \leq k \leq 2$. Estos datos deben estar en el mismo procesador para que los cálculos se realicen sin necesidad de comunicaciones. En el *GDM* de la figura 2.10 se observa que para $i = 0$, $j = 0$ y $0 \leq k \leq 2$, los vértices $(i, 0)$, $(j, 0)$ y (k, k_0), siendo $0 \leq k_0 \leq 2$ se encuentran unidos mediante arcos de "color" negro. La igualdad del "color" de estos arcos indica que entre los datos proyectados sobre ellos existe dependencia (cuando dos vértices del *GDM* están unidos mediante más de un arco de distinto "color", por motivos de claridad se agrupan todos estos arcos en uno con tantos "colores" como los originales). Asimismo, se observa el "orden" de utilización de los datos, de forma que primero se realiza el cálculo para $k = 0$ $(k, 0)$ y por último para $k = 2$ $(k, 2)$.

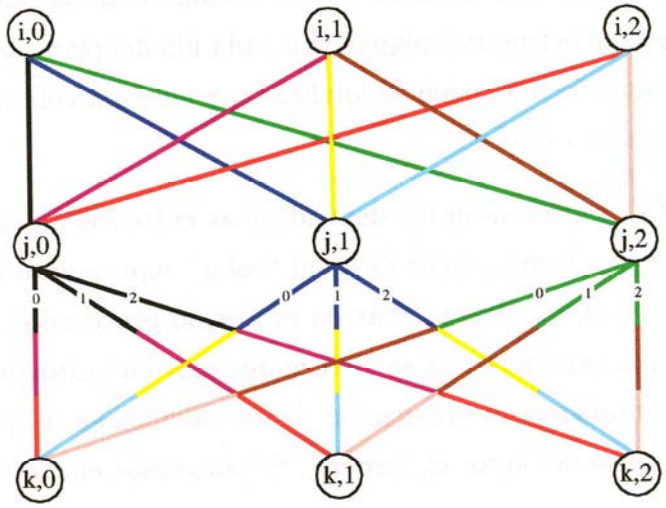


Figura 2.10: *GDM* de la multiplicación matriz-matriz ejemplo

2.5.2 Distribución de datos según el GDM

A partir del *GDM* del algoritmo secuencial se establece la distribución inicial de datos y posteriores redistribuciones, de forma que un dato determinado se almacena en la memoria del procesador P^c, si los $n - 1$ arcos que unen los n vértices del *GDM* donde se proyecta este dato, tienen "color" c. Por ejemplo, para la figura 2.10, el punto $(i = 0, j = 0, k = 0)$ se proyecta sobre los vértices: $(i, 0)$, $(j, 0)$ y $(k, 0)$, que juntos forman un hipervértice; y los dos arcos que unen estas parejas de vértices, que forman un hiperarco. El peso "color" del hiperarco es el negro y el peso "orden" es el cero.

Por otro lado, cada punto (i, j, k) que se proyecta sobre tres vértices y dos arcos del GDM, puede tener asociados más de un dato diferente. Así, los datos $C_{i=0,j=0}$, $A_{i=0,k=0}$ y $B_{k=0,j=0}$ están asociados a los vértices y arcos comentados anteriormente.

Asimismo, los n vértices del GDM sobre los que se proyecta un dato determinado pueden estar enlazados por hiperarcos de distinto "color". En este caso, el dato asociado debe almacenarse en los procesadores emparejados con el "color" del hiperarco (replicación de datos en las memorias de los procesadores).

Figura 2.11: Distribución de datos de la multiplicación matriz-matriz ejemplo

Del GDM del ejemplo (figura 2.10) se puede extraer la distribución inicial de datos (figura 2.11). En este caso, cada elemento de las matrices A y B se encuentran replicados en tres procesadores, mientras que cada elemento de la matriz C está localizado en un único procesador.

2.5.3 Comunicaciones según el GDM

Según la distribución inicial de datos, éstos pueden estar replicados en las memorias de los distintos procesadores, sin embargo, esta replicación puede eliminarse si utilizamos comunicaciones. Supongamos que un dato se proyecta sobre más de un hiperarco de distinto "color". Según esto:

- Si los arcos de distinto "color" tienen distinto "orden", el dato se almacena en la memoria del procesador correspondiente al "color" del arco que tiene menor "orden", y luego se comunica al resto siguiendo el orden establecido por el peso. El solapamiento total o parcial es función del t_{cal} y t_{com} y particular para cada problema y arquitectura.
- Si los arcos de distinto "color" tienen el mismo "orden" puede ser posible la solución anterior (si el "orden" no es restrictivo). Así, el dato se almacena en la memoria de uno de los procesadores y se comunica al resto en un orden establecido por el programador. El solapamiento total o parcial es función del t_{cal} y t_{com} y particular para cada problema y arquitectura.

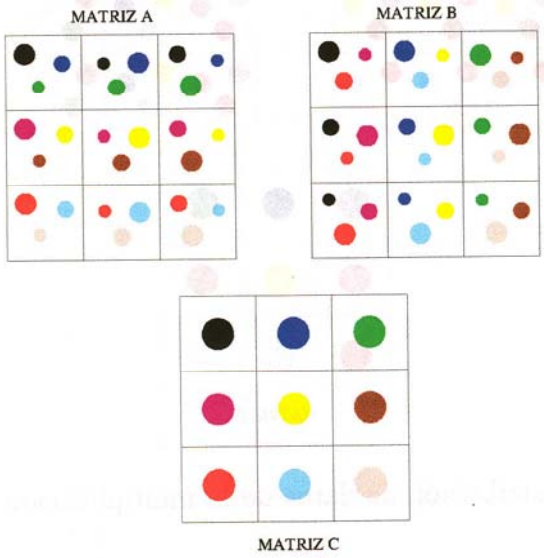


Figura 2.12: Redistribución de datos de la multiplicación matriz-matriz ejemplo

Partiendo del GDM de la figura 2.10 se puede estudiar una posible redistribución de datos (figura 2.12). En este caso, el "orden" de los arcos de distinto "color" sobre los que se proyecta un dato, es el mismo. Así, si tomamos el elemento $B_{k,j}$ siendo $k = 0$ y $j = 0$ existen arcos de tres colores (negro, magenta, rojo) e igual "orden" (0), que unen los vértices $(k, 0)$ y $(j, 0)$. Según esto, este dato debe estar almacenado en la memoria de tres procesadores. Si se quiere evitar la replicación de datos, el elemento puede almacenarse en un único procesador y a la vez que es utilizado, se comunica al siguiente procesador (y así hasta que los tres lo hayan empleado en su cálculo). Esto se puede realizar ya que los cálculos en este caso presentan un orden relajado (la suma es

una operación conmutativa). Esto mismo se puede realizar para el resto de elementos de las matrices A y B, tal y como se indica en la figura 2.12. En esta figura se ha utilizado el tamaño de los puntos (de mayor a menor) para indicar qué dato se utiliza primero en un procesador dado.

2.5.4 Características de la metodología

Las características fundamentales del *GDM* son las siguientes:

- El *GDM* es una herramienta de ayuda al programador que permite descubrir visualmente el paralelismo existente en un determinado código. Así, para ejecutar de forma eficiente (con el menor número de comunicaciones) la dependencia analizada se necesitan tantos procesadores como colores existan en el grafo.
- En función de los colores de los arcos del grafo se realiza una primera distribución de datos, de forma que los datos proyectados sobre los arcos de un "color" (c) se almacenan en el procesador correspondiente a ese "color" (P^c).
- Si en alguna iteración posterior, los vértices sobre los que se proyecta un dato se unen mediante arcos de distinto "color" (f, tal que $c \neq f$), significa que se necesita una redistribución de datos (comunicaciones *punto a punto*). Así, el *GDM* indica cómo deben comunicarse los datos, y que tipo de comunicación hay que realizar. En cualquier caso, un dato a calcular siempre se comunica al procesador que tiene los datos que necesita (no sigue la regla del propietario).
- Si en alguna iteración existen arcos en el *GDM* con el mismo "color" y "orden", los datos proyectados sobre ellos se pueden calcular en paralelo existiendo comunicaciones de tipo *broadcast* de datos dependientes.
- La generación del *GDM* no implica la ejecución de todos los bucles que encierran a la dependencia analizada. El número de bucles a ejecutar depende de las características de la dependencia (uniforme o no uniforme).
- A partir del *GDM* el programador puede distribuir los bucles en los procesadores. Así, los valores de los índices de los bucles asociados a un "color" dado (c) son ejecutados por el procesador correspondiente (P^c).

- Si en el código existe más de una dependencia, debe realizarse un análisis de sus $GDMs$, existiendo un grafo por dependencia. El programador debe estudiar las posibles distribuciones de datos que indica cada $GDM(d_i)$ (siendo d_i la dependencia analizada), eligiendo la distribución más restrictiva.

Capítulo 3

Análisis de las dependencias

3.1 Introducción

Un algoritmo se puede definir como la secuencia de pasos a seguir en la resolución de un problema. Cada uno de estos pasos debe estar perfectamente definido y debe poder ser expresado en un lenguaje de programación y ejecutado sobre uno o más computadores. Los algoritmos se pueden clasificar de distinta forma dependiendo de sus características, sin embargo, en este estudio nos centramos en los algoritmos transformacionales secuenciales imperativos [107].

A nuestros efectos, los algoritmos se clasifican en secuenciales y paralelos. Los algoritmos secuenciales son aquellos que se diseñan para ser ejecutados sobre una máquina secuencial (computador que posee un juego de instrucciones que sólo incluye operaciones sobre variables escalares). Por el contrario, los algoritmos paralelos son aquellos que se ejecutan de forma concurrente o paralela en uno o varios computadores [89].

Al diseñar un algoritmo paralelo, partiendo del algoritmo secuencial, es importante analizar el comportamiento de este último mediante herramientas que permitan representar las dependencias de datos. Algunos autores realizan este análisis visualmente utilizando el EI [57] del algoritmo secuencial y la representación de las dependencias sobre este espacio (*grafo de dependencias*). Sin embargo, esta representación tiene una serie de problemas (sección 2.2.1) que justifican su modificación. En este capítulo se

presenta nuestra solución novedosa: el *grafo de dependencias modificado* o GDM. En una primera aproximación, este grafo se explica comparativamente con la representación del EI.

3.2 Espacio de iteraciones

La estructura de ciertos algoritmos secuenciales (algoritmo 2.1) consiste de un conjunto de bucles que encierran un grupo de una o más sentencias. El conjunto de todos los posibles valores que puede tener el vector formado por las variables índices de los bucles, se denomina *espacio de iteraciones.* Los límites del EI se pueden caracterizar mediante un conjunto de inecuaciones correspondientes a los límites de los bucles, de forma que [57]:

$$S_L \times I \geq l \tag{3.1}$$

$$S_U \times I \leq u \tag{3.2}$$

donde, las ecuaciones 3.1 y 3.2 representan el conjunto de los límites inferiores y superiores de los bucles respectivamente. Si suponemos que existen n bucles, S_L es una matriz triangular inferior de dimensión $n \times n$, S_U es una matriz triangular superior de dimensión $n \times n$, l y u son vectores de dimensión $n \times 1$ e $I = (I_1, I_2, ..., I_n)^t$ es el *vector de iteración* o *vector de índices*, de dimensión $n \times 1$. En la figura 3.1 se presenta un ejemplo del EI de la factorización LU sin pivoteo (espacio piramidal), cuyas coordenadas corresponden con el vector de índices, $I = (k, i, j)$.

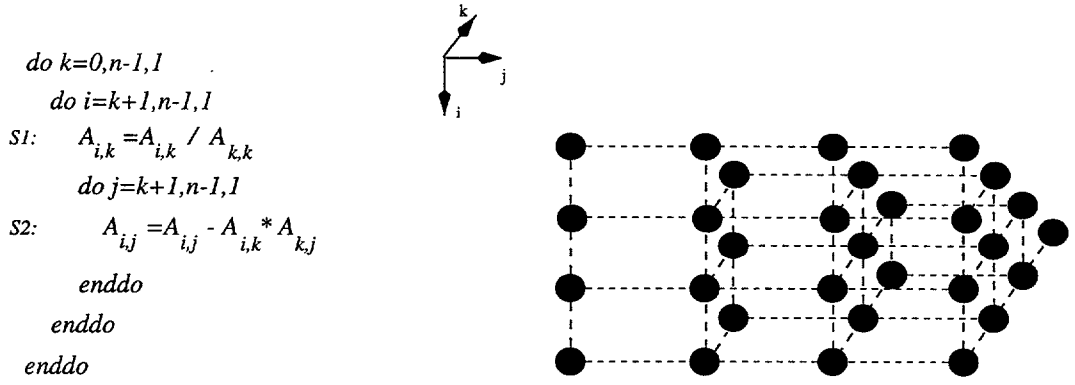


Figura 3.1: Algoritmo secuencial de la factorización LU y su EI ($n = 4$)

Para este ejemplo, las matrices y vectores anteriores son:

$$S_L = \begin{pmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix} \quad S_U = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$I = \begin{pmatrix} k \\ i \\ j \end{pmatrix} \qquad l = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \qquad u = \begin{pmatrix} n-1 \\ n \\ n \end{pmatrix}$$

3.3 Grafo de dependencias

La ejecución de cualquier algoritmo transformacional lleva consigo la modificación de unos datos para obtener unos resultados. La manipulación de estos datos se puede caracterizar mediante un grafo, denominado *grafo de dependencias* del algoritmo secuencial, donde cada vértice representa operaciones a realizar entre datos y cada arco representa las dependencias existentes entre ellos. Básicamente, existen tres tipos de dependencias [9]:

- *Dependencia de flujo.* Se dice que una instrucción S_2 tiene una dependencia de flujo de la instrucción S_1 ($S_1 \delta S_2$) si una variable asignada en S_1 es referenciada posteriormente en S_2 y no hay ninguna instrucción S_3, entre la S_1 y la S_2 que asigne un nuevo valor a esa variable.

- *Antidependencia.* Existe una antidependencia entre las instrucciones S_2 y S_1 si una variable referenciada en S_1 ($S_1 \vec{\delta} S_2$) es posteriormente referenciada en S_2 y no hay ninguna instrucción S_3, entre la S_1 y la S_2 que asigne un nuevo valor a esa variable.

- *Dependencia de salida.* Se dice que una instrucción S_2 tiene una dependencia de salida de la instrucción S_1 ($S_1 \delta^{\circ} S_2$) si una variable asignada en S_1 es posteriormente reasignada en S_2 y no hay ninguna instrucción S_3, entre la S_1 y la S_2 que asigne un nuevo valor a esa variable.

El análisis que se desarrolla a continuación se centra en las *dependencias de flujo* ya que el resto de dependencias son producidas por la reutilización de la misma posición de memoria y pueden ser eliminadas mediante el renombrado de las variables [9], [57]. Por ejemplo, en el algoritmo 3.3 existen dependencias de flujo entre sentencias de una misma iteración del bucle i (S_2 tiene una dependencia de S_1, ya que a_j necesita el dato c_i) y del bucle j (S_2 tiene una dependencia de sí misma, ya que a_{j_2} necesita a a_{j_1} si $j_1 < j_2$, siendo j_1 y j_2 valores diferentes de la variable j) y entre sentencias de distintas iteraciones del bucle i (S_1 tiene una dependencia de sí misma, ya que c_i necesita el dato c_{i-2}). Estos dos tipos de dependencias de flujo se denominan dependencias *loop independent* y dependencias *loop carried*, respectivamente [57].

```
do j=0,n-1,1
   a_j = 0
   do i=0,n-1,1
      S1: c_i = c_{i-2}
      S2: a_j = a_j + c_i
   enddo
enddo
```
(3.3)

$$
\begin{array}{ll}
A_{1,0} = A_{1,0}/A_{0,0} & A_{2,1} = A_{2,1}/A_{1,1} \\
A_{1,1} = A_{1,1} - A_{1,0} \times A_{0,1} & A_{2,2} = A_{2,2} - A_{2,1} \times A_{1,2} \\
A_{1,2} = A_{1,2} - A_{1,0} \times A_{0,2} & A_{2,3} = A_{2,3} - A_{2,1} \times A_{1,3} \\
A_{1,3} = A_{1,3} - A_{1,0} \times A_{0,3} & A_{3,1} = A_{3,1}/A_{1,1} \\
A_{2,0} = A_{2,0}/A_{0,0} & A_{3,2} = A_{3,2} - A_{3,1} \times A_{1,2} \\
A_{2,1} = A_{2,1} - A_{2,0} \times A_{0,1} & A_{3,3} = A_{3,3} - A_{3,1} \times A_{1,3} \\
A_{2,2} = A_{2,2} - A_{2,0} \times A_{0,2} & \\
A_{2,3} = A_{2,3} - A_{2,0} \times A_{0,3} & A_{3,2} = A_{3,2}/A_{2,2} \\
A_{3,0} = A_{3,0}/A_{0,0} & A_{3,3} = A_{3,3} - A_{3,2} \times A_{2,3} \\
A_{3,1} = A_{3,1} - A_{3,0} \times A_{0,1} & \\
A_{3,2} = A_{3,2} - A_{3,0} \times A_{0,2} & \\
A_{3,3} = A_{3,3} - A_{3,0} \times A_{0,3} &
\end{array}
$$

Tabla 3.1: Bucles desenrollados de la factorización LU ($n = 4$)

Por ejemplo, en la factorización LU (figura 3.1), si desenrollamos los bucles suponiendo n=4 (tabla 3.1), se puede observar que existen tres dependencias de flujo de tipo *loop carried* entre distintas iteraciones del bucle k ($A_{i,j} \leftarrow A_{k,j}$, $A_{i,j} \leftarrow A_{i,j}$ -en la sentencia interior-, $A_{i,k} \leftarrow A_{i,k}$ -en la sentencia exterior-) y una dependencia de tipo *loop independent* en cada iteración del bucle i ($A_{i,j} \leftarrow A_{i,k}$ -en la sentencia interior-).

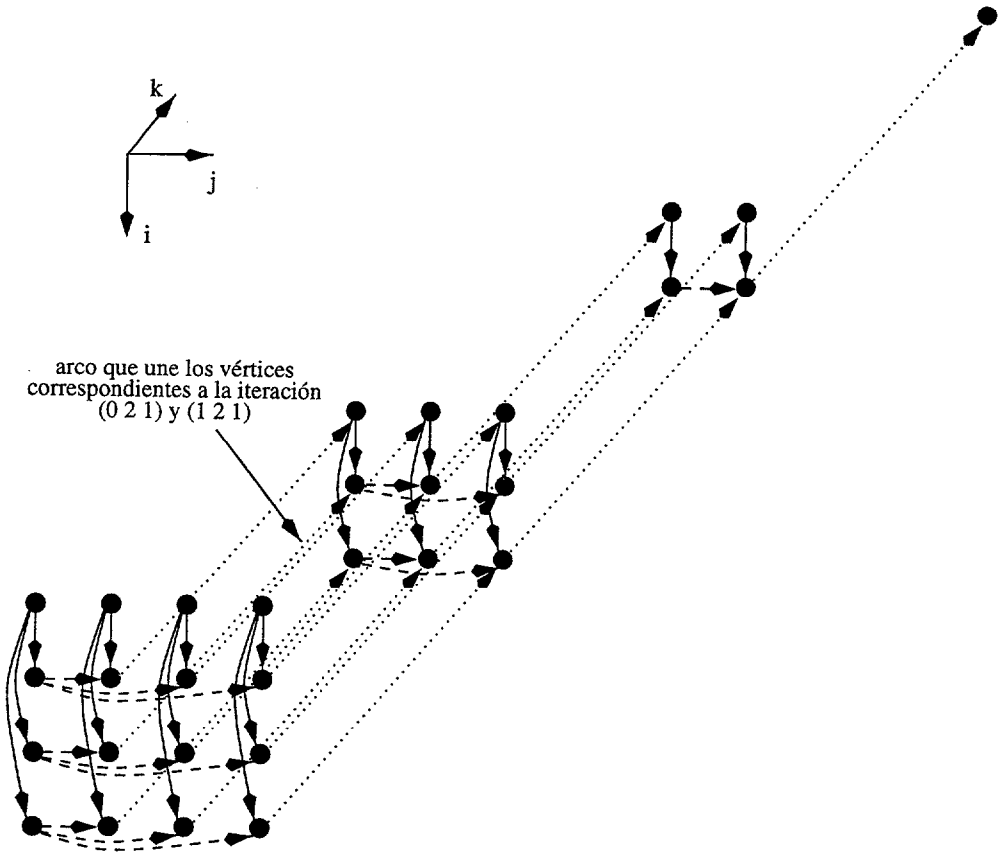


Figura 3.2: Grafo de dependencias de la factorización LU ($n = 4$)

3.3.1 Representación del grafo de dependencias

Partiendo del EI del algoritmo secuencial podemos extrapolar su representación a un grafo, de forma que cada punto del EI es un vértice del mismo. Los arcos del grafo se obtienen al unir los vértices $(\vec{p} = p_1, p_2, ..., p_n)^t$ y $(\vec{q} = q_1, q_2, ..., q_n)^t$ del EI cuyo vector de índices es $I = (I_1, I_2, ..., I_n)^t$, siempre y cuando existan dos sentencias S_v y S_w (no necesariamente diferentes) en el interior de los bucles tal que $S_v[\vec{p}]\delta S_w[\vec{q}]$, para cada

par de iteraciones $\vec{p}$ y $\vec{q}$ si $\vec{p} \neq \vec{q}$ [109]. El arco que une dos nodos del grafo tiene como vértice inicial, el punto correspondiente a la iteración $\vec{p}$ y como vértice final, el punto correspondiente a la iteración $\vec{q}$. En la figura 3.2 se presenta un ejemplo del grafo de dependencias para la factorización LU.

En el algoritmo de la factorización LU (figura 3.1), para $k = k_1$ y $j = j_1$, el dato A_{i,j_1} necesita el elemento A_{k_1,j_1}. De igual forma, para $k = k_1$ e $i = i_1$, el dato $A_{i_1,j}$ necesita el valor A_{i_1,k_1}. Para una iteración $(k\ \ i\ \ j)^t = (k_1\ \ i_1\ \ j_1)^t$, el dato A_{i_1,j_1} tiene una dependencia de A_{i_1,j_1}, calculado en la iteración $k = k_1 - 1$, y de los valores A_{i_1,k_1} y A_{k_1,j_1} comentados anteriormente. De estas relaciones, se establece el grafo de dependencias presentado en la figura 3.2.

3.4 Grafo de dependencias modificado o GDM

La dimensión del grafo de dependencias es función del número de bucles que encierran a la dependencia (en el ejemplo anterior existen tres bucles, luego el grafo es tridimensional). Debido a esto, cuando el número de bucles es superior a tres, no es posible la representación gráfica del mismo siguiendo el método presentado anteriormente. Asimismo, cuando existe más de una dependencia de datos en el algoritmo, el número de arcos del grafo hace que el análisis visual del mismo sea dificultoso, lo que presenta problemas para programadores no expertos.

Una solución al problema anterior consiste en modificar la estructura del grafo de dependencias, consiguiendo una información gráfica más clara de la transformación de los datos en el algoritmo analizado. De esta forma:

- Para cada dependencia (d_i) existente en el algoritmo se obtiene un grafo de dependencias o *GDM(d_i)*.
- Cada *GDM(d_i)* se representa en un plano, por tanto, es independiente del número de bucles que encierran a la dependencia.

El *GDM* se puede considerar como la unión de todos los *GDM(d_i)* para $1 \leq i \leq nud$, donde nud es el número total de dependencias del algoritmo. La información

representada en este grafo debe ser la misma que la representada en el espacio de iteraciones del algoritmo analizado. El grafo de dependencias modificado para una dependencia d_i queda caracterizado mediante un conjunto de vértices y un conjunto de arcos, $GDM(d_i) = (V(d_i), A(d_i))$. Para caracterizar el $GDM(d_i)$, se parte del EI, estableciendo una serie de definiciones. En principio caracterizamos el GDM para cualquier dependencia y explicamos cómo se calculan los conjuntos V y A para esas dependencias.

3.4.1 Vértices del GDM

Dado un EI de dimensión n, que nombramos como $EI(I, n)$, se considera un $EI(I', n-1)$ a aquel espacio que integra a todos los puntos del $EI(I, n)$ y que tiene una de las coordenadas a un valor constante. Sea I_f esta variable de iteración y c_f el valor constante, entonces el vector $I' = (I_1, I_2, ..., I_f = c_f, ..., I_{n-1}, I_n)$ denota a todos los puntos integrados en el $EI(I', n-1)$ para el valor constante de I_f.

Definición 1 *Un superpunto SP_{f,c_f} es la agrupación de puntos del $EI(I, n)$, definido por la proyección:*

$$\vec{I'} \times \vec{e} = c_f$$

donde c_f es un valor constante, $\vec{I'} = (I_1, I_2, ..., I_f = c_f, ..., I_n)$ es un vector fila y $\vec{e} = (e_1\ e_2\ ...\ e_n)$ es un vector columna, tal que $e_f = 1$ y $e_j = 0$ $(\forall f, j = 1..n,\ f \neq j)$.

Nótese que $1 \leq f \leq n$, es decir existe al menos un superpunto por cada una de las variables de iteración del $EI(I, n)$. A su vez, cada variable I_f puede tomar valores dentro del rango definido por las inecuaciones 3.1 y 3.2. Esto significa que por cada variable de iteración pueden existir más de un superpunto (para un valor dado de f, c_f puede tomar valores diferentes definidos por los límites de los bucles y el paso de iteración).

Por ejemplo, en la figura 3.3 se muestran los superpuntos de la dimensión k $(f = 1)$ del algoritmo de la figura 3.1 que se obtienen haciendo la proyección de los vectores

$\vec{I} = (k = k_1 \;\; i \;\; j)$ sobre la dimensión k ($\vec{e} = (1 \;\; 0 \;\; 0)^t$) para los valores $0 \leq k_1 \leq n-1$ ($n = 4$).

$$\begin{pmatrix} k_1 & i & j \end{pmatrix} \times \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = k_1$$

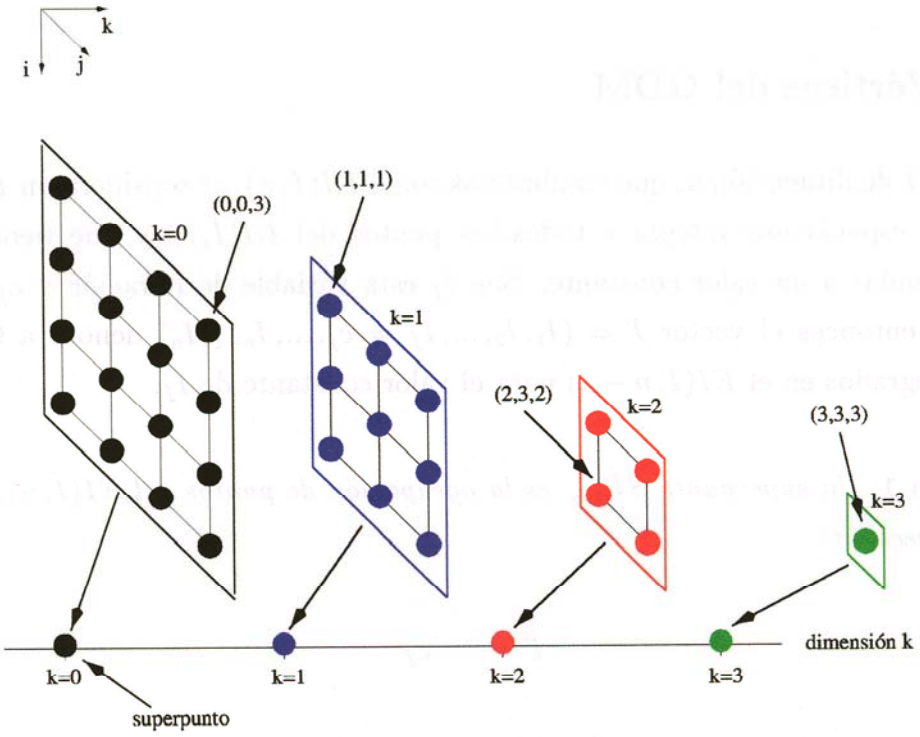


Figura 3.3: Ejemplo de proyección sobre la dimensión k

Asimismo, las proyecciones de los vectores $\vec{I}$ sobre las dimensiones i ($f = 2$) y j ($f = 3$) de este mismo espacio, se consiguen aplicando los siguientes productos escalares:

$$\begin{pmatrix} k & i_1 & j \end{pmatrix} \times \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = i_1 \qquad \begin{pmatrix} k & i & j_1 \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = j_1$$

Obteniéndose los resultados gráficos que se muestran en las figuras 3.4 y 3.5 si $0 \leq i_1, j_1 \leq n-1$ ($n = 4$).

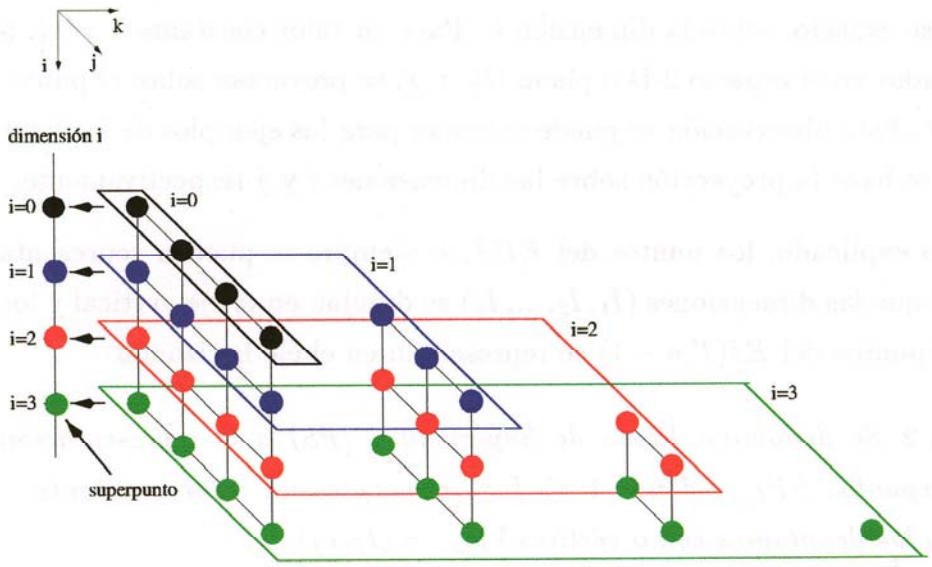


Figura 3.4: Ejemplo de proyección sobre la dimensión i

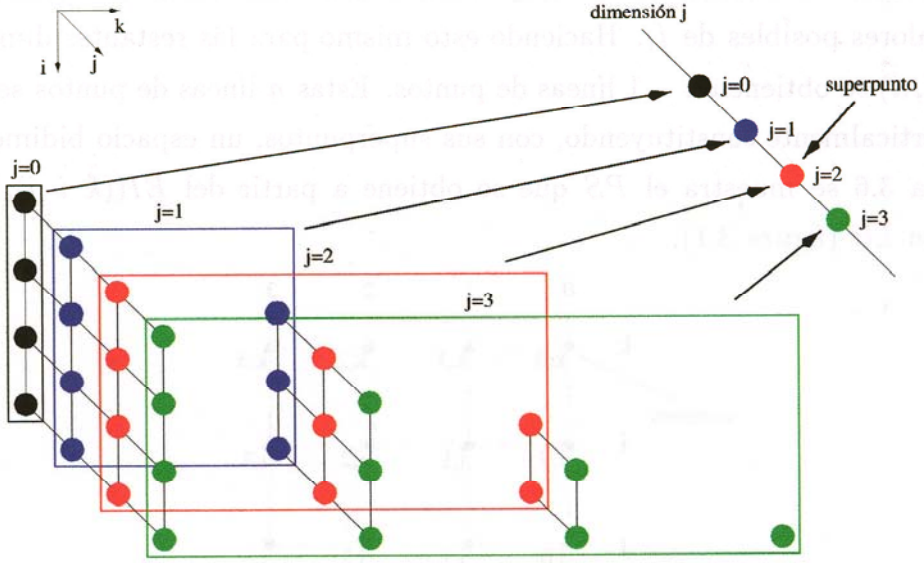


Figura 3.5: Ejemplo de proyección sobre la dimensión j

Cada una de las proyecciones produce la asignación de todos los puntos de un EI $n-1$-dimensional sobre un único punto, ya que para una dimensión I_f, todos los puntos situados en el espacio definido por $(I_1\ I_2\ ...\ I_f = c_f\ ...\ I_n)$ se proyectan sobre el punto c_f de la dimensión I_f $(1 \leq f \leq n)$. Así, en la figura 3.3 se observa un espacio de iteraciones 3-D y la proyección de los vectores, que definen a los distintos

puntos de ese espacio, sobre la dimensión k. Para un valor constante $k = k_1$, todos los puntos situados en el espacio 2-D o plano (k_1 i j) se proyectan sobre el punto k_1 de la dimensión k. Esta observación se puede extender para los ejemplos de las figuras 3.4 y 3.5 cuando se hace la proyección sobre las dimensiones i y j respectivamente.

Según lo explicado, los puntos del $EI(I, n)$ siempre se pueden representar en un plano, en el que las dimensiones $(I_1, I_2, ..., I_n)$ se dibujan en el eje vertical y los valores de los superpuntos del $EI(I', n-1)$ se representan en el eje horizontal.

Definición 2 *Se denomina Plano de Superpuntos (PS) a la representación gráfica de los superpuntos SP_{f,c_f}, donde $1 \leq f \leq n$ y c_f es un valor constante. A estos superpuntos los denotamos como vértices $V_{I_f,c_f} = (I_f, c_f)$.*

Teniendo en cuenta la definición 1, un $EI(I', n-1)$ con $I' = (I_1 = c_1, I_2, I_3, ..., I_n)$, genera una línea de superpuntos correspondiente a la dimensión I_1, donde c_1 toma todos los valores posibles de I_1. Haciendo esto mismo para las restantes dimensiones I_j del $EI(I, n)$ se obtienen $n-1$ líneas de puntos. Estas n líneas de puntos se pueden disponer verticalmente constituyendo, con sus superpuntos, un espacio bidimensional. En la figura 3.6 se muestra el PS que se obtiene a partir del $EI((k\ i\ j), 3)$ de la factorización LU (figura 3.1).

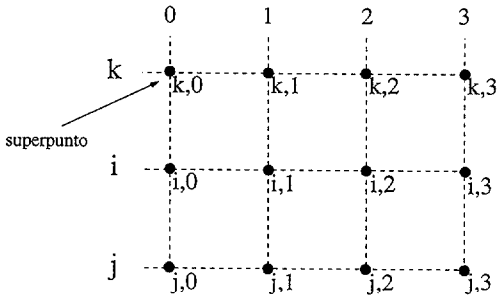


Figura 3.6: Plano de superpuntos de la factorización LU ($n = 4$)

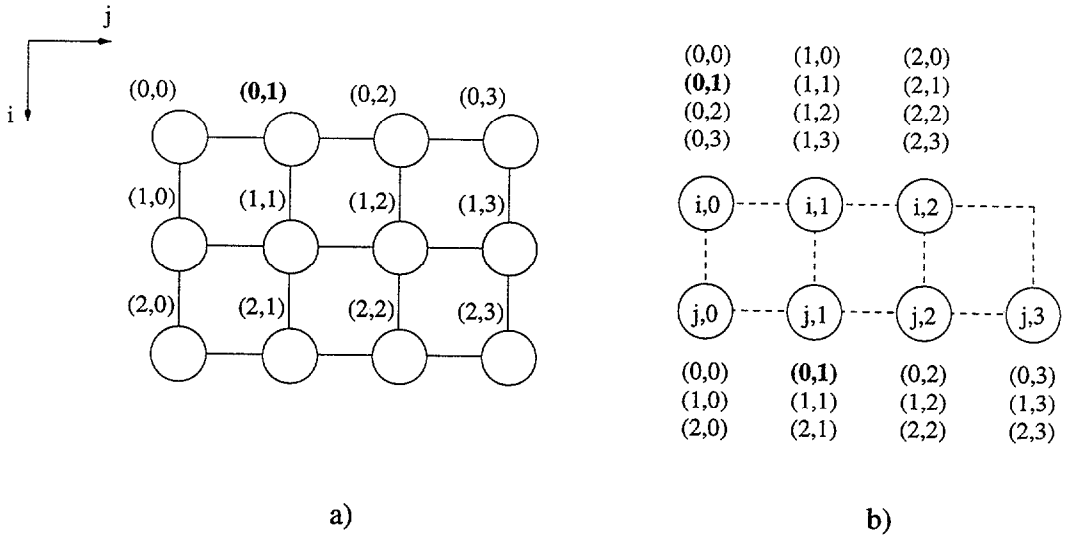
Lema 1 *Sea I^p=($I_1 = p_1, I_2 = p_2, ..., I_n = p_n$) las coordenadas del punto $\vec{p} = (p_1, p_2, ..., p_n)$ en el $EI(I, n)$. Cada punto $\vec{p}$ del $EI(I, n)$ se proyecta en n vértices diferentes del PS. Estos n vértices constituyen un hipervértice ($V_{\vec{p}}$).*

Demostración. Según las definiciones 1 y 2, cualquier punto $\vec{p} = (p_1, p_2, ..., p_n)$ del $EI(I, n)$ está representado en la coordenada I_1 del PS (que corresponde con la dimen-

sión I_1 del $EI(I,n)$ tal que $I_1 = p_1$), pero también lo está en el resto de coordenadas I_l siendo $2 \leq l \leq n$ (que corresponden con las dimensiones I_l del $EI(I,n)$ tal que $I_l = p_l$), apareciendo repetido, por tanto, n veces. Así, el punto $\vec{p}$ del $EI(I,n)$ corresponde con el hipervértice $V_{\vec{p}} = V_{(p_1,p_2,...,p_n)}$ del PS.

En la figura 3.7 se muestra el EI y el PS del algoritmo 3.4 (multiplicación matriz-vector). En esta figura se observa que todos los puntos del espacio original se encuentran proyectados en más de un punto del plano (en concreto en $n = 2$ vértices del PS). Por ejemplo, el punto $\vec{p} = (p_1 = 0, p_2 = 1)$ del EI (figura 3.7(a)) corresponde con el hipervértice $V_{(0,1)}$, es decir, con los vértices $V_{i,0}$ y $V_{j,1}$ del PS (en la figura 3.7(b) se denotan como $i,0$ y $j,1$ respectivamente).

```
do i=0,2,1
   do j=0,3,1
      c_i=c_i+A_{i,k} × b_j
   enddo
enddo
```
(3.4)



a) b)

Figura 3.7: Relación entre los puntos del EI y los vértices del PS del algoritmo ejemplo. a) EI. b) PS

Si suponemos que nv_{I_i} es el número de puntos de la dimensión I_i del $EI(I,n)$, en la representación del EI existen $nv_{I_1} \times nv_{I_2} \times ... \times nv_{I_n}$ puntos y en el PS existen

$nv_{I_1} + nv_{I_2} + ... + nv_{I_n}$ superpuntos. Estos superpuntos del PS constituyen los *vértices* del grafo de dependencias modificado o GDM. Sin embargo, para caracterizar de forma completa este grafo se necesita establecer arcos de unión entre estos vértices.

3.4.2 Arcos del GDM

Supongamos el algoritmo 3.4 y dos iteraciones $\vec{v} = (i_1, j_1)$ y $\vec{w} = (i_2, j_2)$ donde $\vec{v} < \vec{w}$. La iteración $\vec{v} < \vec{w}$ si y sólo si, $(i_1 < i_2)$ o bien $(i_1 = i_2)$ y $(j_1 < j_2)$. Para cualquier pareja de iteraciones $\vec{v}$ y $\vec{w}$ que cumpla lo indicado, el vector $\vec{w} - \vec{v} = (i_2 - i_1, j_2 - j_1)$ se denomina *vector de distancia de dependencia* [57].

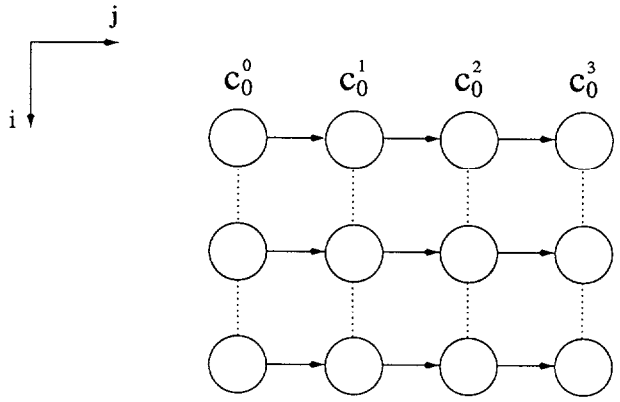


Figura 3.8: Dependencias en el EI de la multiplicación matriz-vector ejemplo

Para que exista una dependencia de datos con respecto a c_i, debe ocurrir que $i_1 = i_2$ (y, por tanto, $j_1 < j_2$) [9]. Analizando el algoritmo se observa que el índice j tiene un incremento de valor uno, luego se puede decir que $j_1 = j_2 - 1$ y por tanto, el vector distancia de dependencia es (0,1). Si cada punto del $EI(I, n)$ representa a una iteración del algoritmo y a los datos que se utilizan en ella (para lectura o escritura), se pueden reflejar las dependencias de datos en este espacio (grafo de dependencias). Estas dependencias se representan mediante arcos en el $EI(I, n)$. Así, en el espacio de iteraciones del algoritmo 3.4 (figura 3.8) se observa que para un valor constante $i = 0$, el dato c_0 se necesita en todos los vértices $(0, j)$ siendo $0 \leq j \leq 3$, ya que existe un arco que une los puntos (i_0, j_0) y $(i_0, j_0 + 1)$ para $0 \leq i_0, j_0 \leq 2$. La dirección de los arcos establece el orden de ejecución de los cálculos asociados a cada vértice del espacio de iteraciones.

3.4.2.1 Unión entre los vértices del grafo

El GDM queda caracterizado mediante los vértices del plano de superpuntos y los arcos que unen a estos vértices. Para representar los arcos en el GDM debemos tener en cuenta que no puede existir pérdida de información con respecto al EI. Sin embargo, si observamos la figura 3.7 (EI y PS del algoritmo 3.4), se puede apreciar que el número de vértices del GDM es menor que el número de vértices del espacio de iteraciones (siete y doce respectivamente), al proyectar más de un punto de este espacio en un único vértice del GDM (lema 1). Para relacionar los vértices del GDM que representan a un punto concreto del EI se enuncia la siguiente definición:

Definición 3 *Cada punto $\vec{p}$ del $EI(I,n)$ se proyecta sobre $n-1$ arcos del GDM. Los $n-1$ arcos constituyen un hiperarco ($A_{\vec{p}}$).*

Según el lema 1, el punto $\vec{p}$ corresponde con los n vértices V_{I_f,p_f}, $1 \leq f \leq n$. Para establecer una relación entre estos vértices, se unen mediante $n-1$ arcos etiquetados como $A_{I_f,(p_f,p_{f+1})}$, cuyo origen siempre es el vértice V_{I_f,p_f} y cuyo destino siempre es el vértice $V_{I_{f+1},p_{f+1}}$ (figura 3.9(a)).

Teniendo como ejemplo el algoritmo 3.4, el punto (i_0,j_0) del $EI((i,j),2)$ corresponde con los vértices V_{i,i_0} y V_{j,j_0} del GDM y con el arco que los une $A_{i,(i_0,j_0)}$, siendo $0 \leq i_0 \leq 2$ y $0 \leq j_0 \leq 3$ (definición 3). Por ejemplo, para un cierto valor i_0 tenemos las correspondencias siguientes para todos los valores j_0:

$$\vec{p}=(i_0,0) \Longrightarrow V_{i,i_0},\ V_{j,0} \text{ y } A_{i,(i_0,0)} \ /\ V_{\vec{p}} = V_{(i_0,0)},\ A_{\vec{p}} = A_{(i_0,0)}$$
$$\vec{p}=(i_0,1) \Longrightarrow V_{i,i_0},\ V_{j,1} \text{ y } A_{i,(i_0,1)} \ /\ V_{\vec{p}} = V_{(i_0,1)},\ A_{\vec{p}} = A_{(i_0,1)}$$
$$\vec{p}=(i_0,2) \Longrightarrow V_{i,i_0},\ V_{j,2} \text{ y } A_{i,(i_0,2)} \ /\ V_{\vec{p}} = V_{(i_0,2)},\ A_{\vec{p}} = A_{(i_0,2)}$$
$$\vec{p}=(i_0,3) \Longrightarrow V_{i,i_0},\ V_{j,3} \text{ y } A_{i,(i_0,3)} \ /\ V_{\vec{p}} = V_{(i_0,3)},\ A_{\vec{p}} = A_{(i_0,3)}$$

Según la relación anterior, cada punto del EI del algoritmo 3.4 está representado mediante dos vértices del GDM y el arco que los une (figura 3.9(b)). Los cuatro arcos de la figura 3.9(b) también se pueden representar tal y como muestra la figura 3.9(c), ya que el vértice inicial de cada uno de ellos es el mismo.

En la figura 3.10 se presenta el GDM para el algoritmo 3.4 uniendo aquellos vértices que están relacionados entre sí, al corresponder al mismo punto del EI.

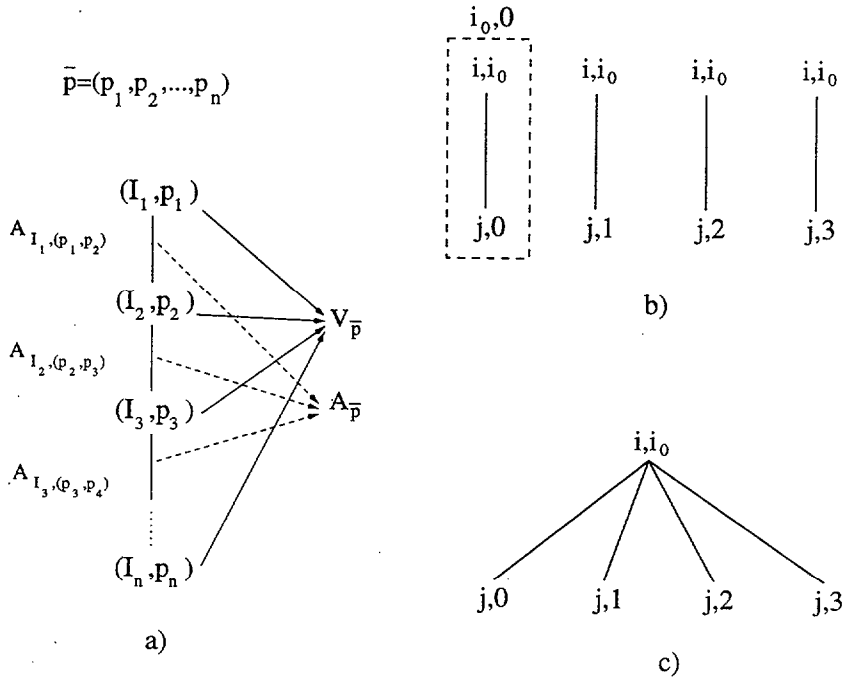


Figura 3.9: a) Unión de n vértices (hipervértice) mediante $n-1$ arcos (hiperarco). b) Arcos entre vértices. c) Arcos con un mismo vértice inicial

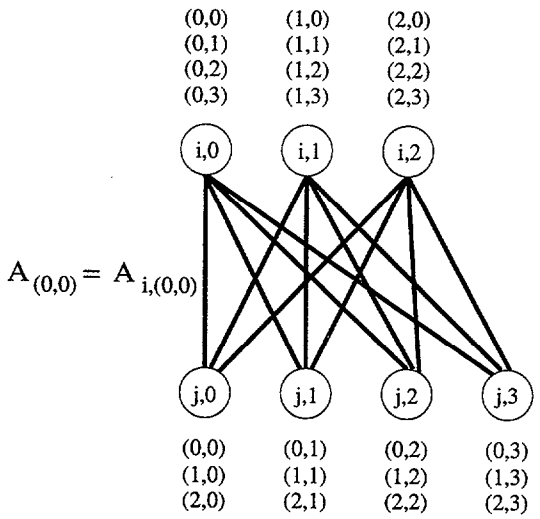


Figura 3.10: Unión de los vértices del *GMD* mediante arcos

Lema 2 *Sean V_{I_i,p_i} y $V_{I_{i+1},p_{i+1}}$ dos vértices del GDM. Entre ellos existe más de un arco si sobre ambos vértices se proyecta más de un punto del EI.*

Demostración. Supongamos dos puntos $\vec{p}$ y $\vec{q}$ del $EI(I,n)$. El punto $\vec{p}(\vec{q})$ corresponde con el hipervértice $V_{\vec{p}(\vec{q})}$ y con el hiperarco $A_{\vec{p}(\vec{q})}$. Si $p_i = q_i$ y $p_{i+1} = q_{i+1}$, entre

los vértices $V_{I_i,p_i(q_i)}$ y $V_{I_{i+1},p_{i+1}(q_{i+1})}$ existen dos arcos: $A_{I_i,(p_i,p_{i+1})}$ y $A_{I_i,(q_i,q_{i+1})}$ (figura 3.11(b)). Esta demostración se puede generalizar al caso en el que más de un punto del EI tengan coordenadas coincidentes. Así, si existen n puntos del espacio n-dimensional, tal que $p_i = q_i = r_i = ...$, y $p_{i+1} = q_{i+1} = r_{i+1} = ...$, existen n arcos que unen los vértices V_{I_i,p_i} y $V_{I_{i+1},p_{i+1}}$.

La unión de los vértices del GDM que corresponden con el mismo punto del EI no representa las dependencias existentes entre los puntos de este espacio. Analizando el EI del algoritmo 3.4 observamos que los puntos (i_0, j_0) e (i_0, j_0+1), para $0 \leq i_0, j_0 \leq 2$, están unidos mediante un arco (figura 3.8). Esta unión indica que entre ambos puntos existe una dependencia, ya que para calcular el valor $c_{i_0}^{j_0+1}$ se necesita el valor $c_{i_0}^{j_0}$. Sin embargo, esta relación de dependencias no aparece reflejada explícitamente como tal (con el mismo significado) en el GDM.

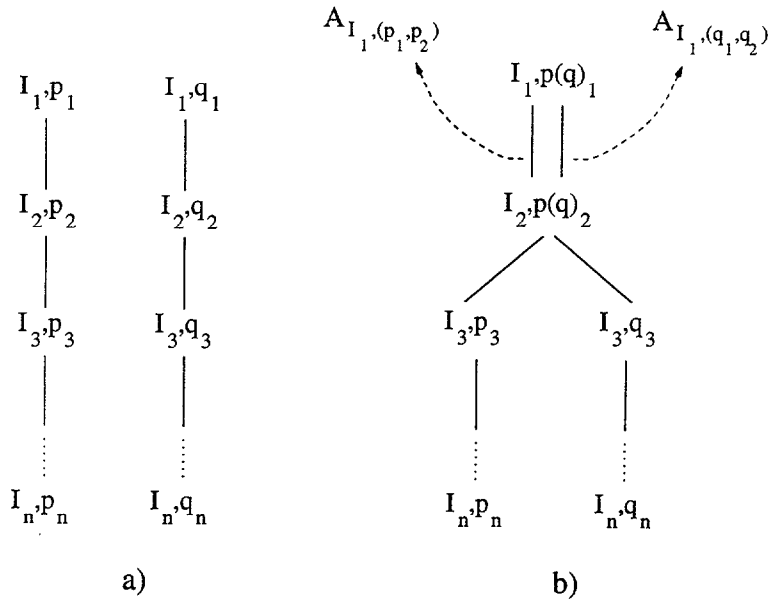


Figura 3.11: a) Ejemplo de vértices y arcos del GDM. b) Arcos dobles entre dos vértices del GDM

3.4.2.2 Coloreado de los arcos

Si existe dependencia entre los datos representados en los puntos $\vec{p}$ y $\vec{q}$, entonces existe un arco en el EI entre ellos, etiquetado como $(\vec{p}, \vec{q})$. Según el lema 1 y la definición 3, los puntos $\vec{p}$ y $\vec{q}$ del EI corresponden con los hipervértices e hiperarcos $V_{\vec{p}}, A_{\vec{p}}$ y $V_{\vec{q}}, A_{\vec{q}}$

respectivamente.

Definición 4 *Sea C^{color} un conjunto de hiperarcos cuyo peso es color (C^{color} : $\{A_{\vec{r}}^{color}$ / $\vec{r} \in EI$ y color $\in \mathcal{N}\}$). Si $\vec{p}$ y $\vec{q}$ son dos puntos del $EI(I,n)$ y existe un arco $(\vec{p},\vec{q})$ en este espacio entonces $A_{\vec{p}}$, $A_{\vec{q}} \in C^{color}$ y se renombran $A_{\vec{p}}^{color}$ y $A_{\vec{q}}^{color}$.*

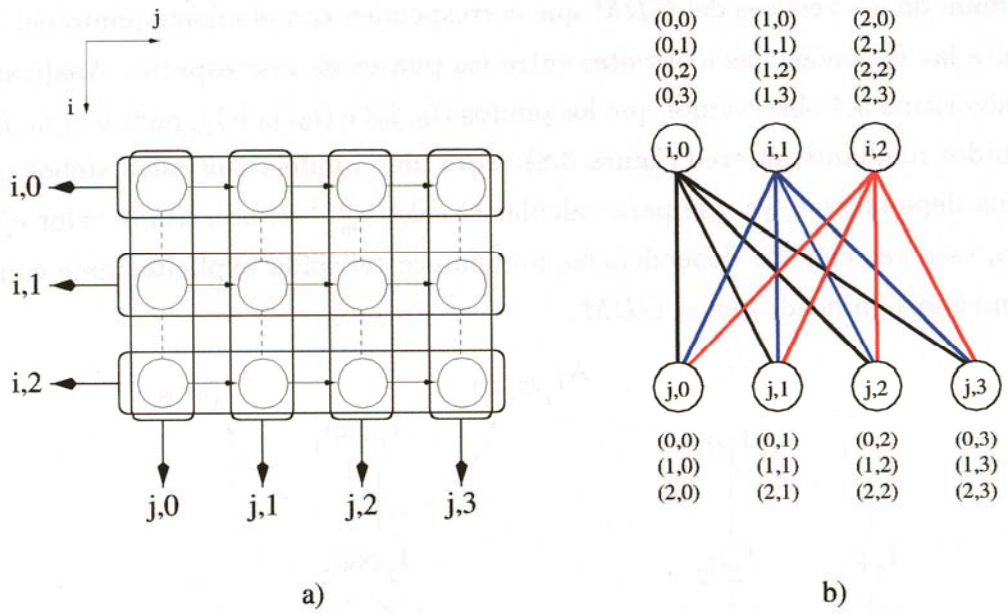


Figura 3.12: Multiplicación matriz-vector. a) Relación entre los vértices del EI y el GDM. b) Hiperarcos del GMD

Supongamos los datos representados en los puntos $(0,0)$ y $(0,1)$ del EI de la figura 3.8. Entre estos datos existe una dependencia por tanto, los hiperarcos relacionados con ellos, $A_{(0,0)}$ para el primer punto y $A_{(0,1)}$ para el segundo punto, deben agruparse en un conjunto con el mismo peso. En la figura 3.12 se observa, para el algoritmo 3.4, la relación existente entre los vértices del EI y del GDM (figura 3.12(a)) y los hiperarcos con su peso correspondiente (figura 3.12(b)). Se puede apreciar que existen hiperarcos de tres colores diferentes. Los hiperarcos de color negro están relacionados con los puntos $(0, j_0)$ siendo $0 \leq j_0 \leq 3$, que tienen una dependencia entre ellos, y por tanto unen los vértices: $(V_{i,0}, V_{j,0})$, $(V_{i,0}, V_{j,1})$, $(V_{i,0}, V_{j,2})$ y $(V_{i,0}, V_{j,3})$. De igual forma, los hiperarcos azules unen los puntos $(1, j_0)$ y los hiperarcos rojos, los puntos $(2, j_0)$.

En general, la existencia de una dependencia entre los datos representados en dos puntos $\vec{p}$ y $\vec{q}$ del EI implica que los hiperarcos que unen los hipervértices del GDM

correspondientes a $\vec{p}$ y $\vec{q}$ ($V_{\vec{p}}$ y $V_{\vec{q}}$) están agrupados en el mismo conjunto, y por tanto, tienen el mismo "color". En la figura 3.13(a), el punto $\vec{p}$ tiene coordenadas (0,0,0) y el punto $\vec{q}$, (0,1,1). Ambos puntos están unidos mediante un arco en el EI, por lo que existe una dependencia entre ellos. Para representar esta dependencia dentro del GDM, se utilizan hiperarcos del mismo "color" que unen los hipervértices asociados a cada punto (figura 3.13(b)), es decir, el hipervértice $V_{(0,0,0)}$ ($(V_{I_1,0}, V_{I_2,0})$ y $(V_{I_2,0}, V_{I_3,0})$ que corresponde con el punto $(0,0,0)$ y el hipervértice $V_{(0,1,1)}$ ($(V_{I_1,0}, V_{I_2,1})$ y $(V_{I_2,1}, V_{I_3,1})$) que corresponde con $(0,1,1)$ (hiperarcos de color negro, figura 3.13(b)). La definición anterior puede extenderse al caso de una cadena de puntos entrelazados dentro del EI.

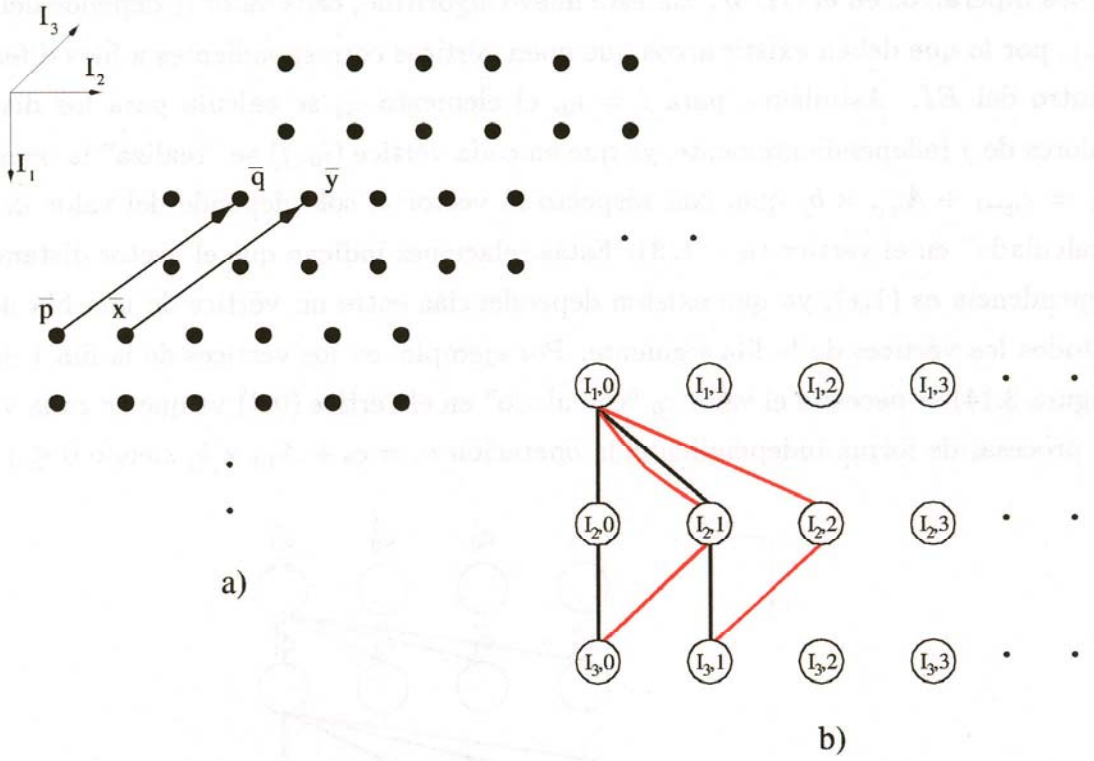


Figura 3.13: a) Arco entre los vértices $\vec{p}$ y $\vec{q}$ en el EI. b) Hiperarcos correspondientes dentro del GDM

De igual forma, la no existencia de una dependencia entre dos puntos $\vec{p}$ y $\vec{x}$ del EI, implica que los hiperarcos que unen a los hipervértices del GDM correspondientes a estos puntos no están agrupados en el mismo conjunto y por tanto tienen diferente "color". Por ejemplo, en la figura 3.13(a) se observa que el punto $\vec{p}$ (0,0,0) no está unido al punto $\vec{x}$ (0,1,0), por tanto, el hiperarco del GDM asociado al punto $\vec{p}$, que une los vértices: $V_{I_1,0}$, $V_{I_2,0}$ y $V_{I_3,0}$, tiene un "color" diferente (color negro) al hiperarco que une

los vértices del GDM que corresponden con el punto $\vec{x}$: $V_{I_1,0}$, $V_{I_2,1}$ y $V_{I_3,0}$ (color rojo).

3.4.2.3 Orden de los arcos

Supongamos una variación en el algoritmo 3.4 tal que la sentencia interior se modifica de la siguiente forma:

$$c_i = c_{i-1} + A_{i,j} \times b_j$$

Esta variación implica dependencias diferentes y por tanto, una nueva representación de los hiperarcos en el GDM. En este nuevo algoritmo, cada valor c_i depende del valor c_{i-1}, por lo que deben existir arcos que unen vértices correspondientes a filas diferentes dentro del EI. Asimismo, para $i = i_0$, el elemento c_{i_0} se calcula para los distintos valores de j independientemente, ya que en cada vértice (i_0, j) se "realiza" la operación $c_{i_0} = c_{i_0-1} + A_{i_0,j} \times b_j$ que, con respecto al vector c, sólo depende del valor de c_{i_0-1} "calculado" en el vértice $(i_0 - 1, 3)$. Estas relaciones indican que el vector distancia de dependencia es $(1,*)$, ya que existen dependencias entre un vértice de una fila del EI y todos los vértices de la fila siguiente. Por ejemplo, en los vértices de la fila 1 del EI (figura 3.14) se necesita el valor c_0 "calculado" en el vértice (0,3) ya que en cada vértice se procesa, de forma independiente, la operación $c_1 = c_0 + A_{0,j} \times b_j$ siendo $0 \leq j \leq 3$.

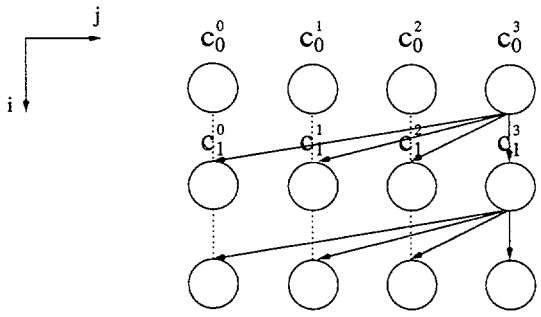


Figura 3.14: Dependencias en el EI del ejemplo modificado

El GDM para este ejemplo (figura 3.15) debe asegurar la misma información obtenida a través del estudio de las dependencias. Así, para un valor $i = i_0$, el hiperarco asociado al punto $(i_0 - 1, 3)$ debe ser del mismo "color" al de los hiperarcos relacionados con los puntos (i_0, j_0), siendo $0 \leq j_0 \leq 3$. En la figura 3.15 se puede observar que todos los hiperarcos del GDM son del mismo "color", ya que la explicación anterior se extiende al resto de puntos del EI, independientemente de la fila en la que se encuentre.

Este hecho supone que existe una dependencia entre puntos de la fila i_0 y puntos de la fila $i_0 + 1$ del EI. Sin embargo, en la figura 3.14 se observa que entre los puntos de una misma fila no existen dependencias. Según esto, los hiperarcos del GDM que unen los hipervértices relacionados con ellos deben tener "color" diferente. Realmente, cada punto de la fila $i_0 + 1$ sólo depende del punto situado en la posición $(i_0, 3)$. Debido a esto, el "color" del hiperarco asociado a este punto es el que impone el "color" de los hiperarcos relacionados con los puntos de la fila siguiente $(i_0 + 1)$.

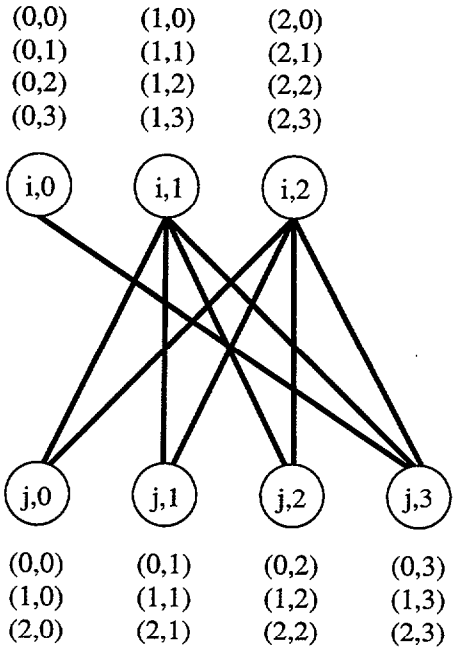


Figura 3.15: Hiperarcos del GMD para el ejemplo modificado

Definición 5 *Sean dos puntos $\vec{p}$ y $\vec{q}$ del $EI(I,n)$ y los hiperarcos $A_{\vec{p}}^{color}$ y $A_{\vec{q}}^{color}$. Si no existe un arco $(\vec{p},\vec{q})$ en el $EI(I,n)$ entonces $A_{\vec{p}}^{color}$ y $A_{\vec{q}}^{color} \in O^{orden}$, tal que $(orden \in \mathcal{N})$ y $(O^{orden} \subset C^{color})$, etiquetándose los hiperarcos de la forma $A_{\vec{p}}^{color,orden}$ y $A_{\vec{q}}^{color,orden}$.*

Definición 6 *Sean dos puntos $\vec{p}$ y $\vec{q}$ del $EI(I,n)$ y los hiperarcos $A_{\vec{p}}^{color}$ y $A_{\vec{q}}^{color}$. Si existe un arco $(\vec{p},\vec{q})$ en el $EI(I,n)$ entonces $A_{\vec{p}}^{color} \in O^{orden_1}$ y $A_{\vec{q}}^{color} \in O^{orden_2}$, tal que $orden_1 < orden_2$ $(orden_1, orden_2 \in \mathcal{N})$ y $(O^{orden_1}, O^{orden_2} \subset C^{color})$, etiquetándose los hiperarcos de la forma $A_{\vec{p}}^{color,orden_1}$ y $A_{\vec{q}}^{color,orden_2}$.*

El peso "orden" establece para un peso "color" dado, el orden de ejecución de las

iteraciones del algoritmo (puntos del EI). Así, si existe el arco $(\vec{p}, \vec{q})$ en el EI y $\vec{p} < \vec{q}$, el "orden" del hiperarco $A_{\vec{p}}$ es menor al "orden" de $A_{\vec{q}}$.

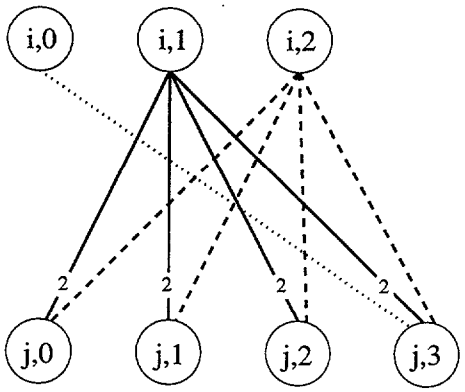


Figura 3.16: Ejemplo del orden de los hiperarcos del *GMD* para el ejemplo modificado

Supongamos los puntos $\vec{r}$, $\vec{p}$ y $\vec{q}$ del $EI(I,n)$ tal que existan los arcos $(\vec{r}, \vec{p})$ y $(\vec{r}, \vec{q})$ y no exista el arco $(\vec{p}, \vec{q})$. Según la definición 4, los hiperarcos $A_{\vec{r}}^{color}$ y $A_{\vec{p}(\vec{q})}^{color} \in C^{color}$, es decir, los hiperarcos del GDM que corresponden con los puntos $\vec{r}$ y $p(q)$ tienen el mismo "color", por la dependencia existente entre los datos representados en ellos. Según la definición 6, el hiperarco que corresponde con $\vec{r}$ tiene un "orden" inferior al hiperarco que corresponde con $p(q)$. Sin embargo, según la definición 5, los hiperarcos que corresponden con $\vec{p}$ y $\vec{q}$ tienen el mismo "orden" ya que no existe arco $(\vec{p}, \vec{q})$ en el $EI(I,n)$. Así, para el ejemplo modificado, el "orden" de los arcos correspondientes a puntos de una misma fila del EI, es el mismo. Este hecho indica que ningún dato referenciado en los puntos (i_0, j_0) del EI, siendo $1 \leq i_0 \leq 2$ y $0 \leq j_0 \leq 3$, es dependiente entre sí. En la figura 3.16 se muestran, en línea continua, los hiperarcos que unen los vértices correspondientes a los puntos de la segunda fila del EI. Estos hiperarcos son del mismo "color" porque dependen del punto $(0,3)$, y tienen el mismo "orden" porque entre ellos no existen dependencias.

En la figura 3.17 se presenta el "orden" de todos los hiperarcos para el ejemplo modificado. El "orden" de los hiperarcos correspondientes a los puntos de la primera y tercera fila del EI siguen el mismo criterio utilizado con los hiperarcos asociados a la fila segunda. En este caso, los hiperarcos correspondientes con los puntos de la primera fila del EI, tienen un "orden" igual a 1, ya que estos puntos no dependen entre sí. Sin embargo, los hiperarcos asociados a los puntos de la segunda fila, tienen un "orden" superior a los anteriores ("orden" 2), ya que dependen del punto $(0,3)$,

situado en la fila anterior (definición 6). El "orden" 2, es el mismo para todos los hiperarcos relacionados con la segunda fila, ya que entre los puntos correspondientes no hay dependencias (definición 5). Lo mismo ocurre para los hiperarcos relacionados con la tercera fila.

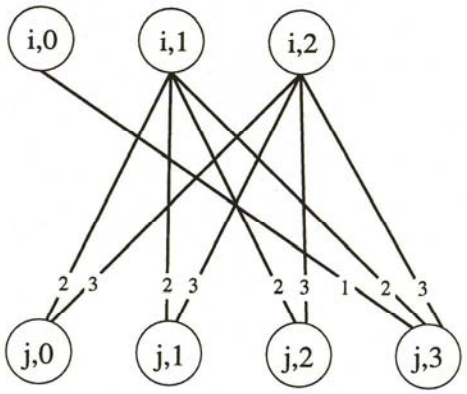


Figura 3.17: *GMD* del ejemplo modificado

Asimismo, en la figura 3.18 se muestra el "orden" de los hiperarcos para el algoritmo 3.4. En esta figura se observa que los hiperarcos de un mismo "color" tienen un "orden" diferente (definición 6). Esto significa que entre los datos representados en los puntos del $EI(I,n)$ relacionados con ellos existe dependencia, siendo el "orden" de la misma, la etiquetada sobre el hiperarco.

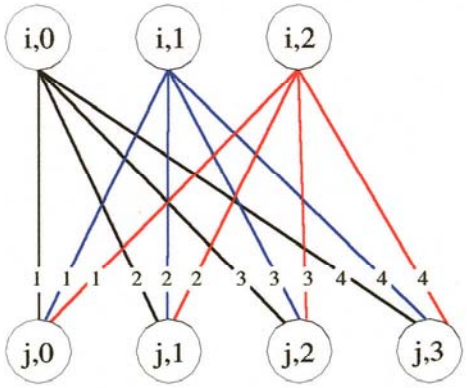


Figura 3.18: *GMD* de la multiplicación matriz-vector ejemplo

Capítulo 4

Metodología de distribución de datos y comunicaciones

4.1 Introducción

La paralelización de algoritmos se puede realizar: a) manualmente (el lenguaje de programación presenta primitivas específicas que el programador utiliza cuando sea necesario); o b) automáticamente (un compilador se encarga de transformar el código fuente en otro código que puede ser ejecutado sobre más de un procesador). La paralelización manual es complicada, por lo que suele ser utilizada por programadores expertos. Así, el programador debe analizar los datos y su utilización, estudiando los posibles problemas que se puedan presentar (redistribución de datos, comunicaciones, etc.). La paralelización automática la realiza un compilador que transforma un código secuencial en otro que es semánticamente equivalente y que puede ser ejecutado en paralelo. De las dos aproximaciones, la paralelización manual es más óptima que la automática, ya que un programador experto paraleliza el código alcanzando mejores prestaciones que un compilador. Sin embargo, el tiempo que emplea en realizar este proceso, también es mayor.

Asimismo, si el programador dispone de información obtenida en tiempo de ejecución puede realizar la distribución de datos y redistribuciones posteriores en base a ella. Esta información se puede obtener utilizando medidas de ciertos parámetros

([15], [75], [90]) o bien ejecutando parte del código secuencial. Según esto, el análisis de las dependencias de datos y del posterior comportamiento de éstos, son los principales objetivos de cualquier herramienta de paralelización centrada en el programador [80].

El estudio de las dependencias de datos se puede realizar mediante técnicas de detección de dependencias ([2], [10], [37], [109]) o mediante alguna herramienta gráfica, como puede ser el grafo de dependencias modificado (GDM). En cualquier caso, el objetivo que se persigue en la paralelización es la reducción del tiempo de ejecución del algoritmo, respecto al tiempo de ejecución sobre una máquina secuencial [89]. Sin embargo, la paralelización puede presentar problemas que producen el efecto contrario al deseado (incremento del tiempo de ejecución). Estos problemas aparecen cuando se necesita establecer comunicaciones entre los distintos procesadores de la arquitectura paralela [14].

Por otro lado, tradicionalmente los multicomputadores se han diseñado siguiendo dos aproximaciones: multicomputadores de topología fija [19] y multicomputadores con conmutador de interconexión, en el que todos los procesadores están conectados con todos [49], [23]. Sin embargo, la tendencia de los últimos años hacia el procesamiento distribuido ha llevado al desarrollo de un nuevo concepto de máquina paralela: las redes heterogéneas de computadores [94]. En estos sistemas los computadores se conectan unos a otros mediante redes y protocolos estándar: Ethernet, FDDI [52]; e incluso mediante redes inalámbricas [92]. Para estos sistemas existen entornos de programación especiales como [34], [79], que también se utilizan en algunos de los multicomputadores que no siguen esta filosofía. Esta configuración de las redes actuales hace que, o bien se pueda establecer una topología a comodidad del programador (multicomputadores o estaciones de trabajo con red de interconexión configurable mediante un conmutador) o bien la topología está fijada de antemano (donde todos los procesadores están conectados a un bus único, cableado o no).

Sea cual sea la configuración de la red que interconecta los procesadores de la arquitectura paralela, el programador de un algoritmo paralelo debe intentar eliminar las comunicaciones. Para ello, los datos deben ser distribuidos en los procesadores de forma que cada uno almacene en su memoria local, aquellos elementos que va a utilizar en sus cálculos (localidad de los datos) [89]. La localidad de los datos puede suponer dos problemas: una mala distribución de la carga y la replicación de datos. Si

la eliminación de comunicaciones no es posible, el programador debe tender a diseñar programas en los que se pueda solapar las comunicaciones. Para ello es necesario: a) obtener cierta información por parte de la herramienta de paralelización empleada (GDM) y b) analizar la arquitectura de la máquina paralela en cuanto a velocidad del procesador y topología de interconexión [18]. En cualquier caso, es necesario establecer cuántos datos pueden ser procesados sin necesidad de comunicación (granularidad del problema), de forma que pueda realizarse un solapamiento de cálculos y comunicaciones.

4.2 Metodología de distribución de datos y comunicaciones

Se denomina método al modo ordenado de proceder para llegar a un resultado o fin determinado y a metodología se la define como el conjunto de métodos utilizados para alcanzar el fin propuesto. El modo ordenado de proceder se puede traducir también por sistemático, dando a entender la existencia de una relación entre los distintos métodos que forman la metodología. Según las definiciones anteriores, consideramos que una *metodología de distribución de datos y comunicaciones* es el conjunto de técnicas realizadas en un orden dado, con las que se consigue: a) distribuir los datos que se utilizan en un determinado algoritmo y b) establecer posteriores comunicaciones entre los procesadores de la arquitectura.

La generación de un algoritmo paralelo puede seguir una metodología sistemática. Si esto es así, la generación del algoritmo puede ser automatizada, con lo que se descarga al programador de la tarea tediosa de analizar las dependencias existentes. En este sentido, nosotros hemos diseñado una metodología que consta de tres pasos: análisis del algoritmo secuencial, distribución inicial de datos y análisis de las comunicaciones.

4.2.1 Análisis del algoritmo secuencial

El análisis del algoritmo secuencial consiste en el estudio de los bucles y dependencias de datos inherentes al código. Este estudio se realiza de forma gráfica, mediante el grafo de dependencias modificado (GDM).

Supongamos un algoritmo secuencial en el que existe un conjunto de bucles y sentencias interiores a ellos, donde M es un vector de n dimensiones y f_i y g_i son funciones de $\mathcal{Z}^k$ a $\mathcal{Z}$ donde $\mathcal{Z}$ es el conjunto de todos los enteros, tal y como se expresa en el algoritmo 4.1.

```
do I_1 = L_1, U_1, Step_1
   ...
   do I_n = L_n, U_n, Step_n
      M(f_1(I_1, ..., I_n), ..., f_n(I_1, ..., I_n)) = M(g_1(I_1, ..., I_n), ..., g_n(I_1, ..., I_n))
   enddo
   ...
enddo
```

(4.1)

4.2.1.1 Bucles y vértices

Supongamos el plano cartesiando de la figura 4.1 en el que las filas corresponden con el vector $I = (I_1\ I_2\ ...\ I_n)$ y las columnas son valores comprendidos en el rango $min(L_i)..max(U_i)$ (siendo L_i y U_i el valor inicial y final respectivamente, del índice del bucle i, donde $1 \leq i \leq n$).

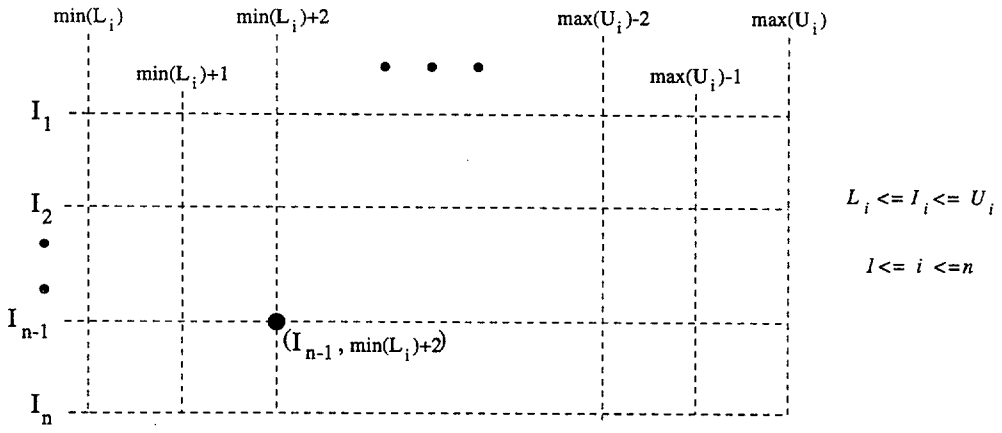


Figura 4.1: Plano cartesiano

Cada punto de este plano está referenciado por el par (I_i, c_i) de forma que $I_i \in I$ y $c_i \in (min(L_i), max(U_i))$. Para especificar los puntos del plano cartesiando, supongamos

el bucle I_i del algoritmo 4.1. Este bucle presenta $\frac{U_i-L_i}{Step_i}+1$ valores diferentes lo que corresponde con $\frac{U_i-L_i}{Step_i}+1$ vértices en la fila I_i del plano. El primer vértice está situado en la columna L_i y el resto de vértices están separados una distancia $Step_i$ (en las columnas $L_i + m \times Step_i$, tal que $L_i + m \times Step_i \leq U_i$).

Por ejemplo, el algoritmo de la tabla 4.1 presenta cuatro bucles anidados con los valores límites siguientes: $1 \leq I_1 \leq 5$ ($Step_1 = 1$), $1 \leq I_2 \leq 7$ ($Step_2 = 1$), $3 \leq I_3 \leq 9$ ($Step_3 = 2$) y $1 \leq I_4 \leq 10$ ($Step_4 = 3$). Las filas y columnas del plano cartesiano se establecen por el número de bucles y los límites de éstos respectivamente. Por otro lado, los bucles I_3 e I_4 tienen un salto mayor a uno, por lo que estos bucles toman los siguientes valores: 3, 5, 7 y 9 para I_3 y 1, 4, 7 y 10 para I_4. En cada uno de los posibles valores de los índices de estos bucles existe un vértice (figura 4.2).

```
do I1=1,5,1
  do I2=I1,I1+2,1
    do I3=3,9,2
      do I4=1,10,3
        ...
      enddo
    enddo
  enddo
enddo
```

Tabla 4.1: Ejemplo de algoritmo

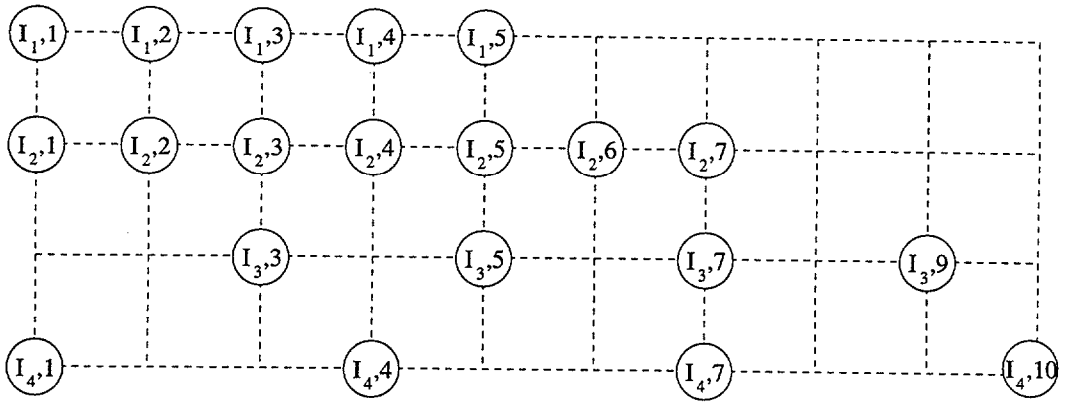


Figura 4.2: Vértices del plano cartesiano para el algoritmo ejemplo

En la figura 4.3 se muestran los vértices del plano para la factorización LU ($n = 4$).

En este ejemplo, las filas del plano para los índices i y j no tienen vértices en la columna 0 ya que el valor inicial de dichos índices es $k+1$.

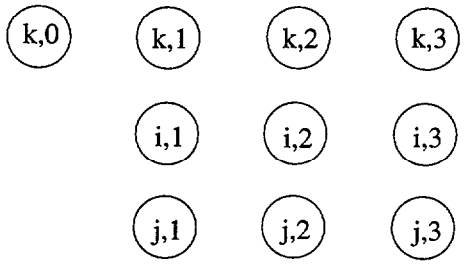


Figura 4.3: Vértices del plano cartesiano de la factorización LU $(n = 4)$

Analizando la representación sobre el plano cartesiano, se puede extrapolar al $GDM(d_i)$ ya que éste es un un grafo plano formado por tantas "filas" como bucles encierran a la dependencia y tantas "columnas" como valores diferentes tengan los índices de estos bucles.

4.2.1.2 Dependencias e hiperarcos

Cada iteración $I^p = (I_1 = p_1, I_2 = p_2, ..., I_n = p_n)$ de un algoritmo lleva asociados una serie de cálculos (cálculos realizados en el punto $\vec{p}$ del EI del algoritmo -cálculo $\vec{p}$-). A su vez, estos cálculos utilizan datos $(M(i_1, i_2, ..., i_m)$ tal que $m \leq n)$ de forma que: el dato $M(i_1, i_2, ..., i_m)$ puede ser utilizado en el cálculo $\vec{p}$ si los índices del conjunto M corresponden con índices del punto $\vec{p}$ $(i_x = f(I_x)$, con lo que $M(i_1, i_2, ..., i_m) = M(i^p))$. Esta afirmación se puede generalizar si existe más de una iteración del algoritmo $(I^p, I^q, I^r, ...)$ que tengan índices coincidentes. Por ejemplo, si $I^p = (I_1 = 0, I_2 = 0, I_3 = 0)$, $I^q = (I_1 = 0, I_2 = 1, I_3 = 0)$ e $I^r = (I_1 = 0, I_2 = 2, I_3 = 0)$ entonces, el dato del conjunto $M(I_1, I_3)$ tal que $I_1 = 0$ e $I_3 = 0$ se utiliza en los cálculos realizados en los puntos $\vec{p} = (0, 0, 0)$, $\vec{q} = (0, 1, 0)$ y $\vec{r} = (0, 2, 0)$.

Según esto, para un algoritmo dado hay que analizar los datos utilizados en cualquiera de sus sentencias. Para ello, hay que definir los conceptos *datos de entrada (IN)* y *datos de salida (OUT)* de una sentencia S. Así, $IN(S)$ está constituido por todos aquellos elementos del vector I, índices de los conjuntos, que van a ser referenciados por la sentencia S y $OUT(S)$ está formado por los elementos del vector I, índices de los conjuntos, que van a ser asignados por dicha sentencia, es decir:

$$IN(S) = \{(I_1^{p^1}, ..., I_n^{p^1}), (I_1^{p^2}, ..., I_n^{p^2}), ...\}$$

$$OUT(S) = \{(J_1^{q^1}, ..., J_n^{q^1}), (J_1^{q^2}, ..., J_n^{q^2}), ...\}$$

donde $I_k^{p^l}(J_k^{q^l})$ es el valor del índice del bucle k en la iteración del algoritmo correspondiente al punto $p^l(q^l)$ del EI, y por tanto está comprendido entre $min(L_k)$ y $max(U_k)$. Por motivos de claridad se ha utilizado $I_k^{p^l}$ para el conjunto IN y $J_k^{q^l}$ para el conjunto OUT.

Supongamos dos iteraciones del algoritmo $I^p = (I_1 = p_1, I_2 = p_2, ..., I_n = p_n)$ e $I^q = (I_1 = q_1, I_2 = q_2, ..., I_n = q_n)$ tal que $I^p < I^q$. Si entre los datos utilizados en estas iteraciones existe dependencia entonces existe un arco $(\vec{p}, \vec{q})$ en el EI del algoritmo. La existencia de esta dependencia y, por tanto de este arco, significa que en la iteración I^q se necesitan datos calculados en la iteración I^p. Estos últimos datos son datos de salida de la sentencia $S_{\vec{p}}$ y de entrada en la sentencia $S_{\vec{q}}$, y por tanto, $I^p \in OUT(S_{\vec{p}})$ e $I^p \in IN(S_{\vec{q}})$. Según [109], entre dos iteraciones I^p e I^q del algoritmo tal que $I^p < I^q$, existe dependencia de flujo $S_{\vec{p}} \delta S_{\vec{q}}$, si y sólo si $OUT(S_{\vec{p}}) \cap IN(S_{\vec{q}}) \neq 0$.

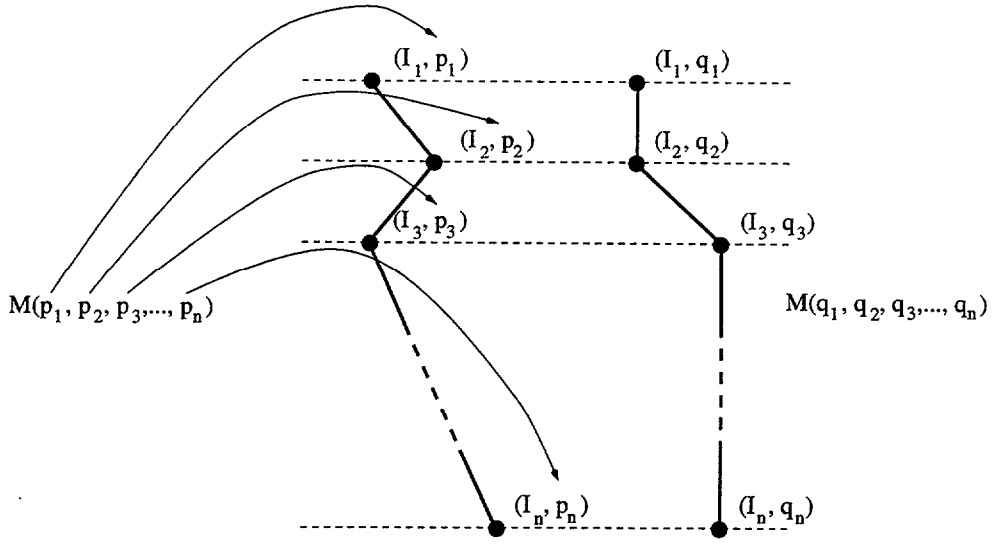


Figura 4.4: Relación entre los elementos del vector y el GDM

Según la definición 4, los hiperarcos del GDM relacionados con los puntos $\vec{p}$ y $\vec{q}$ del EI tienen un mismo "color" ya que existe el arco $(\vec{p}, \vec{q})$ en este espacio. En la figura 4.4 se puede observar que los hiperarcos asociados al punto $\vec{p}$ y al punto $\vec{q}$ tienen el

mismo "color", luego se puede establecer que para actualizar $M(i^q)$ se necesita leer el dato $M(i^p)$, ya que $I^p < I^q$ (en este ejemplo, el conjunto dimensionado tiene el mismo número de índices que los puntos del EI). En la figura 4.5 se observa la utilización del dato $M_{0,0}$ en los cálculos realizados en $\vec{p}$, $\vec{q}$ y $\vec{r}$ (en este caso, el conjunto dimensionado no tiene el mismo número de índices que los puntos del EI).

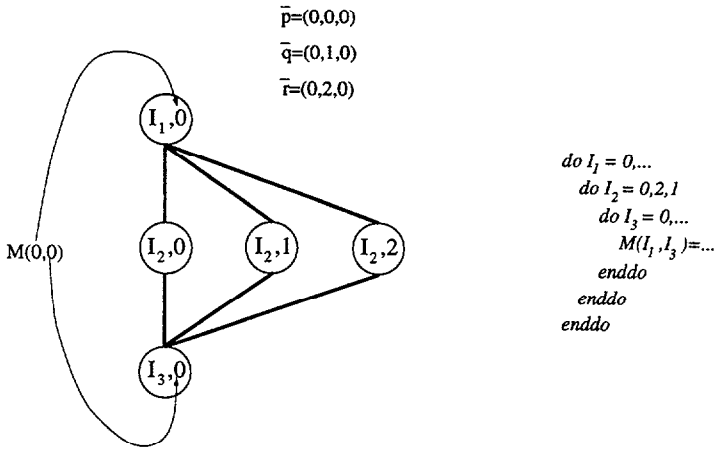


Figura 4.5: Ejemplo de relación entre los elementos del vector M y el GDM

Sin embargo, el peso "color" establece la relación existente entre dos datos pero no define correctamente algunos casos. Supongamos los datos no dependientes $M(i^p)$ y $M(i^q)$ que a su vez dependen del dato $M(i^r)$ (en la figura 4.6 se muestra un ejemplo para tres dimensiones). Aplicando la definición 4, los hiperarcos que unen los vértices del GDM relacionados con estos datos deben tener el mismo "color", ya que los dos primeros dependen del dato $M(i^r)$, sin embargo, $M(i^p)$ y $M(i^q)$ no son dependientes (y por tanto, sus hiperarcos asociados no deben tener el mismo "color"). Para representar este hecho importante, a la hora de saber qué datos pueden ser actualizados a la vez sin depender de otros (en este ejemplo $M(i^p)$ y $M(i^q)$), se estable el peso "orden". De esta forma, según la definición 6, si dos hiperarcos del mismo "color" tienen el mismo "orden" significa que la utilización de los datos asociados es indistinta ya que, aunque ambos dependen del mismo dato, entre ellos no existe dependencia. En la figura 4.6 se observa que los hiperarcos relacionados con los puntos $\vec{p}$ y $\vec{q}$ tienen el mismo "orden". Este peso es mayor al "orden" del hiperarco asociado al punto $\vec{r}$. Con esto se indica que el cálculo sobre $\vec{r}$ se debe realizar antes que el cálculo sobre $\vec{p}$ y $\vec{q}$ (estos últimos se pueden realizar a la vez). Por el contrario, si dos hiperarcos del mismo "color" tienen un "orden" diferente significa que los datos se utilizan en función de este peso, desde

el de menor "orden" hasta el de mayor.

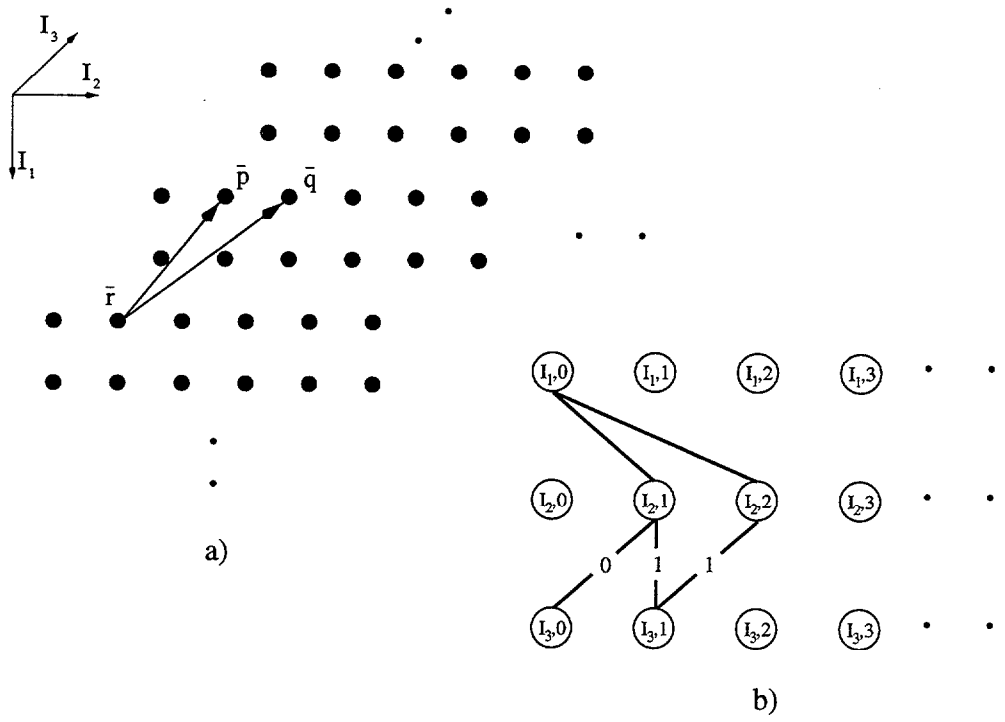


Figura 4.6: a) Dependencia en el EI. b) Dependencias en el GDM

4.2.2 Distribución inicial de datos

El análisis visual del GDM indica una posible distribución inicial de datos, de forma que todos aquellos elementos de un conjunto, cuyos índices I_i están asociados a los vértices extremos de un hiperarco de "color" l_0, deben agruparse ya que dependen entre ellos. Así, si los hiperarcos $A_{\vec{p}(\vec{q})}$ tienen "color" l_0, los elementos del vector referidos a los puntos $\vec{p}$ y $\vec{q}$ deben almacenarse en la memoria del procesador P^{l_0}. A su vez, los datos asociados a hiperarcos del mismo "color" y "orden" pueden almacenarse en las memorias de procesadores diferentes, siempre y cuando el dato del que dependen esté también en esa memoria. El GDM indica cuántos procesadores se necesitan para ejecutar el programa, independientemente de la arquitectura paralela. Debido a esto, la distribución propuesta puede tener problemas de balanceo de la carga y de replicación de datos.

En [64] y [56] se presentan métodos de distribución de datos que utilizan técnicas

de transformación de bucles y que maximiza el paralelismo minimizando las comunicaciones. En el primero se tratan códigos con cualquier número de bucles y referencias afines; y en el segundo se necesita un análisis muy detallado de las dependencias para que el resultado sea óptimo. En [40] se presenta una herramienta de distribución automática de datos basada en el grafo de paralelismo-comunicación (CPG).

4.2.2.1 Balanceo de la carga

Supongamos que nP_h es el número de procesadores según la herramienta y que nP_a es el número real de procesadores de la arquitectura paralela. La distribución de datos establecida por el *GDM* puede producir un mal balanceo de la carga si:

- $nP_h < nP_a$. Existen procesadores en la máquina paralela que no realizan trabajo.
- $nP_h > nP_a$. Se necesita redistribuir los datos que pertenecen a aquellos procesadores que no existen en la máquina paralela (hay que distribuir la carga de forma balanceada).
- $nP_h = nP_a$. Todos los procesadores de la máquina paralela tienen carga, pero unos pueden trabajar más que otros.

Si $nP_h < nP_a$, una redistribución de datos para que existan un balanceo de la carga da lugar o bien a la replicación de datos o al establecimiento de comunicaciones. Si los datos a replicar no son actualizados en ningún procesador, se pueden redistribuir para balancear la carga (considerando que los procesadores tienen memoria suficiente para ello). Sin embargo, si es necesario comunicar datos, hay que estudiar la red de interconexión y los posibles tipos de comunicaciones a realizar (punto a punto, *broadcast*, etc.).

Asimismo, si $nP_h = nP_a$ pueden existir procesadores que estén ociosos. En este caso se redistribuyen los datos (para equilibrar la carga) teniendo en cuenta las mismas consideraciones que en el caso anterior.

Por otro lado, si $nP_h > nP_a$, la redistribución de datos se hace para conseguir: a) que todos los procesadores almacenen el mismo número de elementos; b) que ningún

procesador realice comunicaciones o que el número de comunicaciones sea proporcional en todos ellos. La distribución equitativa de elementos es fácil de conseguir analizando cuántos datos tiene cada procesador (estos valores se obtienen mediante el GDM). Sin embargo, si existen comunicaciones, la distribución tiene que seguir la segunda opción. En este caso, hay que hacer un estudio de las comunicaciones inherentes al algoritmo para saber cuántas veces se comunica un procesador con el resto. A partir de estos datos, se elige la redistribución que asegura un menor número de comunicaciones entre procesadores.

4.2.2.2 Replicación de datos

La distribución de datos realizada por el GDM también puede establecer la necesidad de replicar datos. Los datos se pueden replicar (y no comunicar) si no van a ser modificados por ningún procesador y la memoria de éste es suficiente para almacenarlos. Si esto no es posible, la solución está en el almacenamiento en una única memoria y en posteriores comunicaciones. En el algoritmo de la multiplicación matriz-matriz (algoritmo ejemplo 2.2) se observa como los datos se deben replicar entre los procesadores para que no exista comunicación.

4.2.3 Establecimiento de las comunicaciones

La distribución inicial no suele ser la misma en las dististas fases de ejecución del algoritmo paralelo. Así, si existen dependencias de datos, en la iteración $\vec{w}$ del procesador P^i se pueden necesitar datos calculados en la iteración $\vec{v}$ del procesador P^j. Esto implica una nueva distribución de datos o *redistribución* (comunicación de elementos). Esta información también está representada en el GDM, de forma que este grafo no sólo almacena información sobre la distribución inicial de datos sino también sobre las comunicaciones necesarias.

Supongamos el algoritmo 4.2 que corresponde con la FFT [89]. En este algoritmo existen cuatro dependencias de datos: $A_k \leftarrow A_k\ (d_1)$, $A_k \leftarrow A_{k+l}\ (d_2)$, $A_{k+l} \leftarrow A_k^*\ (d_3)$ y $A_{k+l} \leftarrow A_{k+l}\ (d_4)$ (A_k^* corresponde con el valor previo antes de la modificación), en las que l es un valor variable del algoritmo. Los $GDMs$ generados por la herramienta

para estas cuatro dependencias se presentan en las figuras 4.7, 4.8 y 4.9.

```
do s=1,log(n)
   m=2^s
   w_m = e^(2×π×i/m)
   w=1
   l=m/2
   do j=0,l-1
      do k=j,n-1,m
         t = w × A_(k+l)
         u = A_k
         A_k = u + t
         A_(k+l)=u-t
      enddo
      w = w × w_m
   enddo
enddo
```
(4.2)

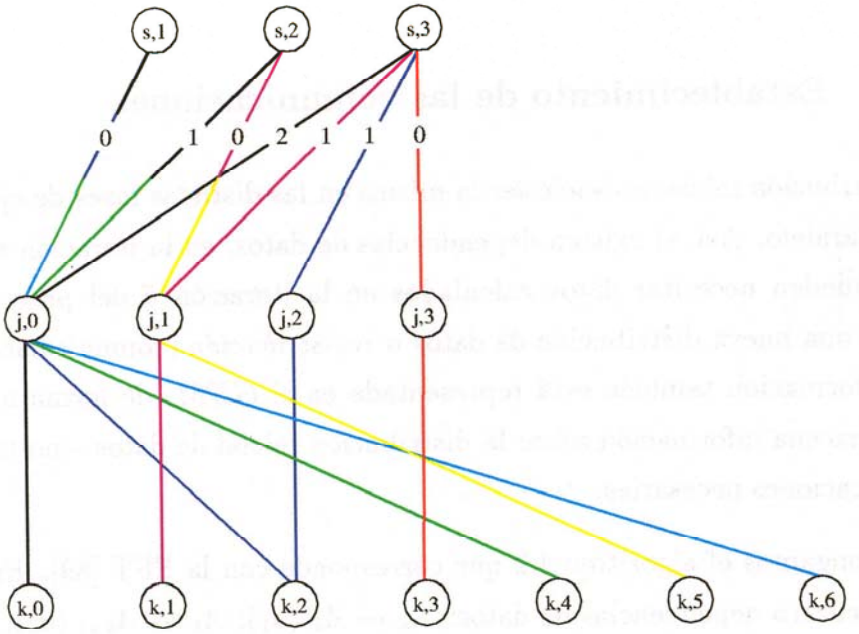


Figura 4.7: $GDM(d_1)$ de la FFT ($n = 8$)

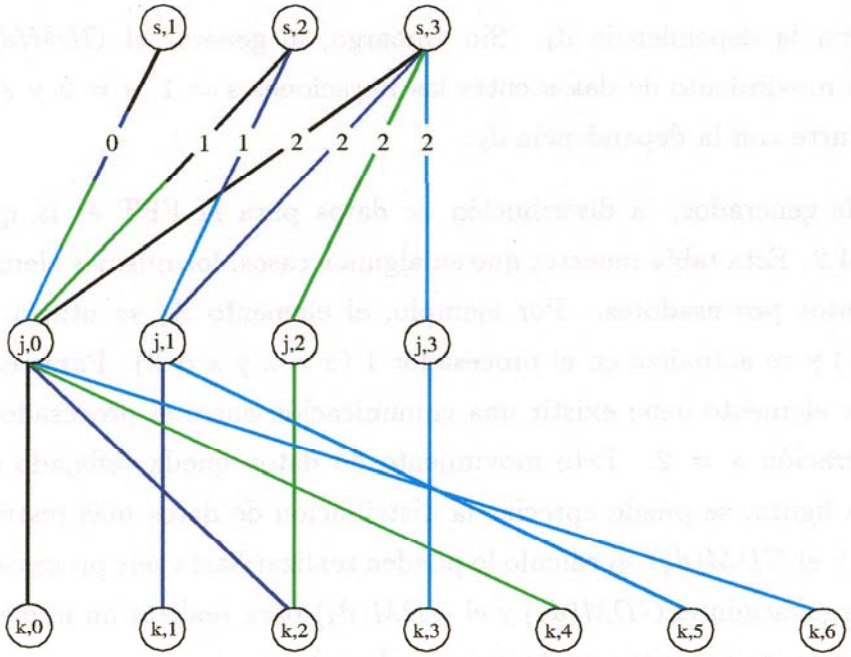


Figura 4.8: *GDM(d_2)* y *GDM(d_3)* de la FFT ($n = 8$)

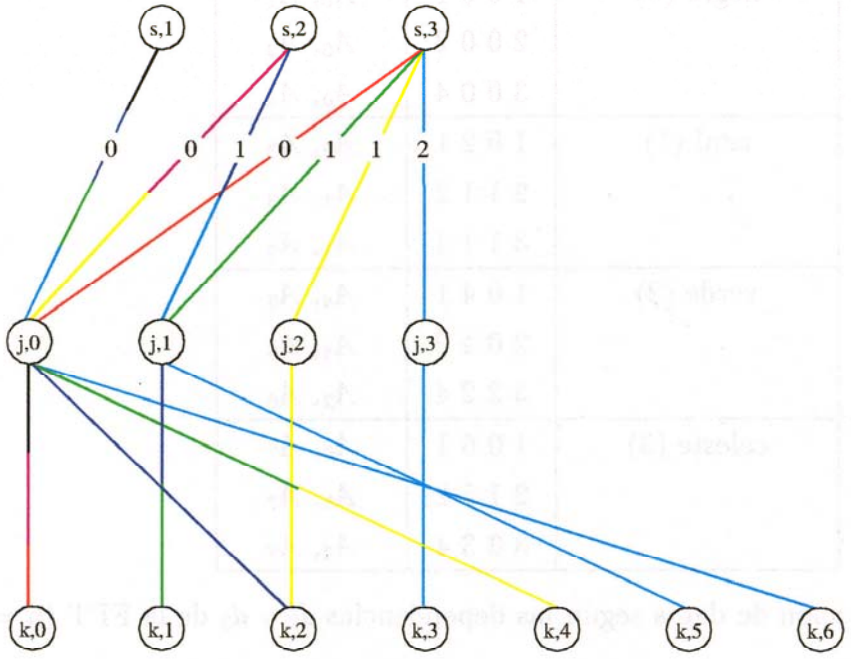


Figura 4.9: *GDM(d_4)* de la FFT ($n = 8$)

Al generar el *GDM(d_1)*, la herramienta no detecta ningún movimiento de datos, pudiéndose realizar el cálculo vinculado a la dependencia d_1 de forma independiente.

Lo mismo ocurre con la dependencia d_4. Sin embargo, al generar el $GDM(d_2)$, la herramienta detecta movimiento de datos entre las iteraciones $s = 1$, $s = 2$ y $s = 2$, $s = 3$. Lo mismo ocurre con la dependencia d_3.

Según los *GDMs* generados, la distribución de datos para la FFT es la que se muestra en la tabla 4.2. Esta tabla muestra que en algunos casos, los mismos elementos se utilizan en distintos procesadores. Por ejemplo, el elemento A_1 se utiliza en el procesador 0 ($s = 1$) y se actualiza en el procesador 1 ($s = 2$ y $s = 3$). Para realizar el cálculo sobre este elemento debe existir una comunicación entre el procesador 0 y el 1 antes de la iteración $s = 2$. Este movimiento de datos queda reflejado en la figura 4.10. En esta figura, se puede apreciar la distribución de datos más restrictiva (según el $GDM(d_1)$ y el $GDM(d_4)$, el cálculo lo pueden realizar hasta seis procesadores diferentes, sin embargo, según el $GDM(d_2)$ y el $GDM(d_3)$ para realizar un número de comunicaciones óptimas, se necesitan cuatro procesadores).

Color/Procesador	*s j k l*	*Elementos*
negro (0)	1 0 0 1	A_0, A_1
	2 0 0 2	A_0, A_2
	3 0 0 4	A_0, A_4
azul (1)	1 0 2 1	A_2, A_3
	2 1 1 2	A_1, A_3
	3 1 1 4	A_1, A_5
verde (2)	1 0 4 1	A_4, A_5
	2 0 4 2	A_4, A_6
	3 2 2 4	A_2, A_6
celeste (3)	1 0 6 1	A_6, A_7
	2 1 5 2	A_5, A_7
	3 3 3 4	A_3, A_7

Tabla 4.2: Distribución de datos según las dependencias d_2 y d_3 de la FFT ($n = 8$)

Sea cual sea la arquitectura paralela, la reducción del tiempo de ejecución se consigue reduciendo el tiempo de comunicación. Para ello se tienden a solapar los cálculos y las comunicaciones siempre que sea posible, de forma que el tiempo de comunicación es ocultado por el tiempo de cálculo, total o parcialmente.

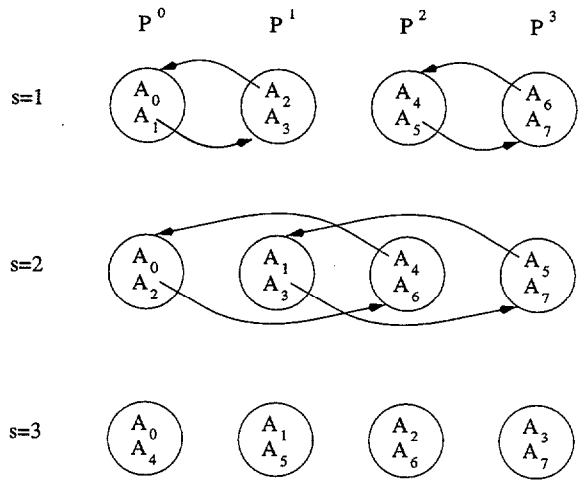


Figura 4.10: Distribución y movimiento de datos en la FFT ($n = 8$)

4.2.3.1 Representación de las comunicaciones en el GDM

La existencia de comunicaciones impone un estudio de las características de la arquitectura paralela sobre la que se implementa el programa. Según [94], las arquitecturas formadas por computadores heterogéneos basadas en conmutador o en bus constituyen las futuras tendencias. En las arquitecturas basadas en conmutador se puede simular cualquier topología mediante primitivas específicas de los entornos de programación, pudiéndose realizar diferentes tipos de comunicación (punto a punto, *broadcast*, etc.). Si los procesadores están conectados mediante un bus, todos comparten el sistema de comunicación por lo que deben primarse las comunicaciones uno a todos (*broadcast*). A continuación se analiza la forma de representar en el GDM, las comunicaciones de tipo *broadcast* y *punto a punto.*

- *Broadcast.* Supongamos el algoritmo 4.3 y las iteraciones ($j = L_j, i = L_i..U_i$), cuyo EI reducido se presenta, junto con las dependencias, en la figura 4.11.

```
do j=Lj, Uj, Stepj
   Aj=...
   do i=Li, Ui, Stepi
      Ai = Aj + ...
   enddo
enddo
```

(4.3)

Analizando el algoritmo 4.3 se observa que, si $L_j = L_i = 0$, los datos relacionados con la iteración $I^p = (j = 0, i = 0)$ se utilizan en las iteraciones del EI tal que $(j = 0, i = 0..U_i)$. De esta forma, una vez actualizado el dato en el punto $\vec{p} = (0,0)$, éste se puede comunicar a aquellos procesadores que realicen los cálculos para el resto de puntos mencionados. Lo mismo ocurre si analizamos los datos asociados a las iteraciones del algoritmo tal que $(j = U_j, i = 0..U_i)$. En cualquier caso, el orden de ejecución de estas iteraciones se conoce examinando el algoritmo secuencial 4.3.

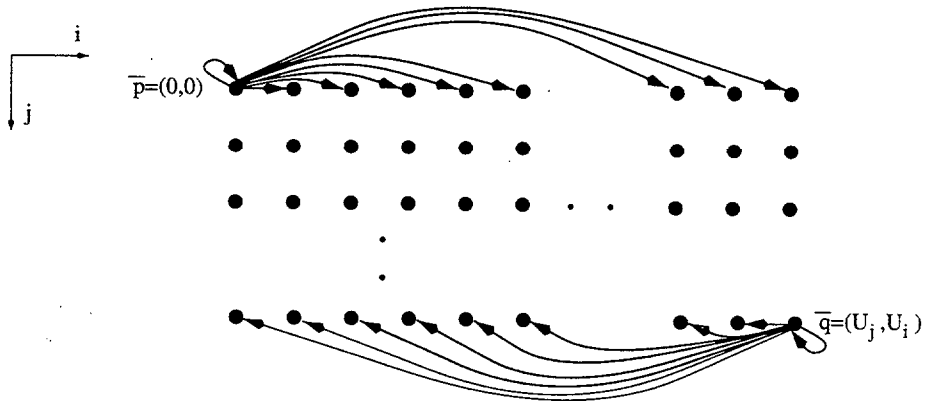


Figura 4.11: EI del algoritmo ejemplo

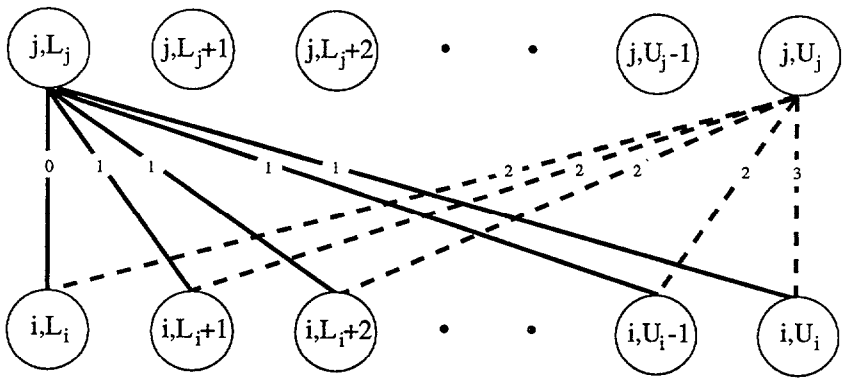


Figura 4.12: GDM del algoritmo ejemplo

El GDM para este ejemplo (figura 4.12 suponiendo $Step_i = Step_j = 1$) aporta la misma información que la indicada en el EI, especificando además, el orden de utilización de los datos (orden de ejecución de las iteraciones). Los hiperarcos sobre los que se proyectan los puntos del EI que están unidos mediante un arco, presentan el mismo "color" (definición 4). Sin embargo, el "orden" de estos

hiperarcos no es el mismo, ya que el "orden" del hiperarco que corresponde con el punto $\vec{p}$ es menor al del resto (cuyos pesos "orden" son iguales), debido a que existe dependencia entre los datos representados en $\vec{p}$ con respecto a los datos representados en el resto de puntos de una misma fila del EI (entre los datos de estos últimos puntos no existe dependencia). Analizando el GDM de la figura 4.12 se puede observar que si bien todos los hiperarcos son del mismo "color", tal y como se explicó en la sección 3.4.2.3, las iteraciones relacionadas con hiperarcos del mismo "color" que tienen un mismo "orden" se pueden ejecutar en procesadores diferentes o en distintos *threads* (el valor del peso "orden" es irrelevante en la figura 4.12, lo único que importa es la relación entre los valores del "orden" de los distintos hiperarcos). En este caso, si existen n hiperarcos con el mismo "color" y "orden", los cálculos asociados con ellos se pueden realizar en n procesadores independientes. Por otro lado, el cálculo asociado al hiperarco de "orden" 0 e igual "color" se debe realizar: a) en todos los procesadores o b) en un procesador diferente al resto. Supongamos este último caso en el que el procesador P^0 ejecuta la iteración $I^p = (j = 0, i = 0)$ y por tanto, calcula el dato A_0. A continuación este dato debe ser comunicado, mediante un *broadcast*, a los procesadores que ejecutan el resto de iteraciones correspondientes a la fila $j = 0$ del EI, de forma que éstos puedan actualizar en paralelo el resto de valores $A_{1..U_i}$. En el caso de que el procesador P^0 ejecute la iteración $I^q = (j = U_j, i = U_i)$ y por tanto, sea el propietario del dato A_{U_i}, antes de actualizarlo debe enviarlo mediante un *broadcast* al resto de procesadores de la fila $j = U_j$ del EI para que éstos actualicen sus valores $(A_{0..U_i-1})$. Esto último se realiza en este orden ya que el peso "orden" de los hiperarcos donde se proyectan los puntos $(U_j, 1..U_i - 1)$ es menor al "orden" del hiperarco asociado con el punto $\vec{q}$.

De la explicación anterior, se puede generalizar que si existen hiperarcos en el GDM con el mismo "orden" y el mismo "color" entonces en los puntos del EI proyectados sobre ellos se utilizan datos que pueden ser comunicados mediante *broadcast*. Así:

Definición 7 *Si existen n hiperarcos* $A_{\vec{q}^j}^{color,orden}$, *tal que* $1 \leq j \leq n$, *el elemento* $M(i^{q^j})$ *se almacena en el procesador* P^j.

Definición 8 *Sean n hiperarcos* $A_{\vec{q}^j}^{color,orden}$, *tal que* $1 \leq j \leq n$. *Si existe un hiperarco* $A_{\vec{p}}^{color,orden_1}$ *y no existe un* $orden_2$ *tal que* $orden_1 < orden_2 < orden$ *entonces el elemento* $M(i^p)$ *se comunica mediante un broadcast, después de ser actualizado, si* $i^p \in IN(S)$. *La fuente del broadcast es el procesador propietario del elemento* $M(i^p)$ *y el destino, los procesadores* P^j.

Definición 9 *Sean n hiperarcos* $A_{\vec{q}^j}^{color,orden}$, *tal que* $1 \leq j \leq n$. *Si existe un hiperarco* $A_{\vec{p}}^{color,orden_1}$ *y no existe un* $orden_2$ *tal que* $orden < orden_2 < orden_1$ *entonces el elemento* $M(i^p)$ *se comunica mediante un broadcast, antes de ser actualizado, si* $i^p \in IN(S)$. *La fuente del broadcast es el procesador propietario del elemento* $M(i^p)$ *y el destino, los procesadores* P^j.

- *Punto a punto.* Supongamos el algoritmo 4.4 y la figura 4.13, en la que se presenta su EI, junto con las dependencias, para $L_k = L_j = 0$, $U_k = U_i = U_j = 2$ y $Step_k = Step_i = Step_j = 1$.

```
m=1
l=0
do k=Lk, Uk, Stepk
   do i=k, Ui, Stepi
      do j=Lj, Uj, Stepj
            Ai,j = Ai-l,j-m + ...
      enddo
   enddo
   l=l+1
   m=m-1
enddo
```

(4.4)

Analizando el algoritmo 4.4 se observa que en las iteraciones ($k = 0, i = k..U_i, j = L_j..U_j$), para actualizar el dato $A_{i,j}$ se necesita el elemento $A_{i,j-1}$. Según esto, los elementos correspondientes a la fila $i = i_0$ del EI se deben almacenar en el mismo procesador P^i. Este hecho queda reflejado en el GDM de la figura 4.14 en el que, si $k = 0$ para calcular los elementos asociados a la fila i se debe seguir un orden (especificado por el peso de los hiperarcos).

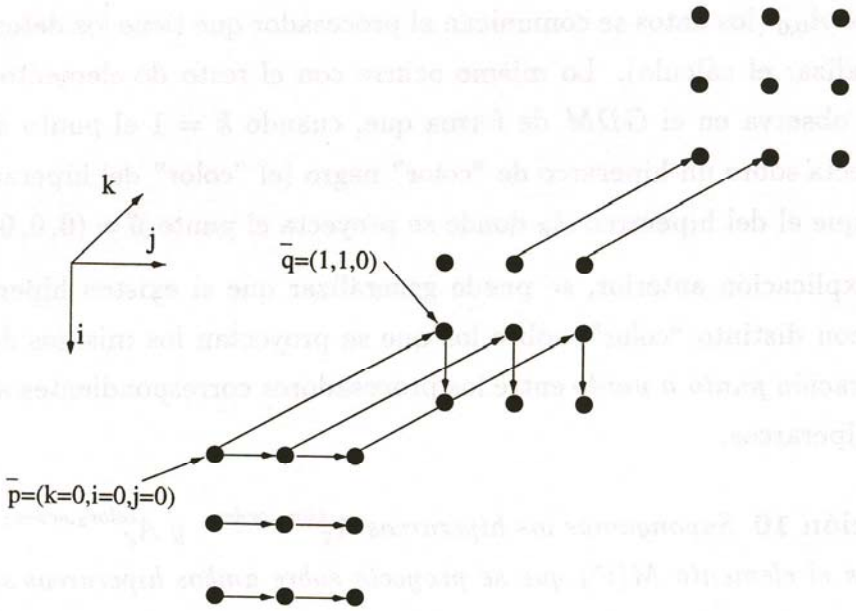


Figura 4.13: EI del algoritmo ejemplo

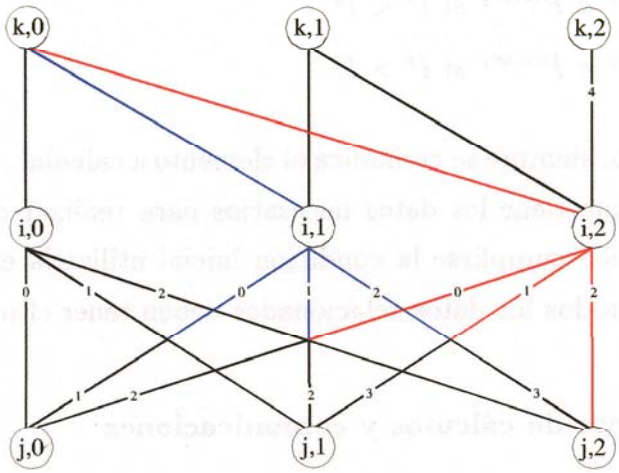


Figura 4.14: GDM del algoritmo ejemplo

En las iteraciones $(k = 1, i = k..U_i, j = L_j..U_j)$, para calcular los valores $A_{i,j}$ se necesitan los elementos $A_{i-1,j}$ calculados previamente y que no se encuentran en el mismo procesador. Por ejemplo, para calcular el elemento $A_{1,0}$ (siendo $k = 1$) se necesita el elemento $A_{0,0}$. El dato a calcular está almacenado en el procesador P^1 (hiperarco de "color" azul) y el dato necesario para este cálculo se encuentra en el procesador P^0 (hiperarco de "color" negro). Teniendo en cuenta esto, el dato a calcular se comunica al procesador P^0, para realizar dicho cálculo con el

elemento $A_{0,0}$ (los datos se comunican al procesador que tiene los datos necesarios para realizar el cálculo). Lo mismo ocurre con el resto de elementos de la fila. Esto se observa en el GDM de forma que, cuando $k = 1$ el punto $\vec{q} = (1, 1, 0)$ se proyecta sobre un hiperarco de "color" negro (el "color" del hiperarco $A_{\vec{q}}$ es el mismo que el del hiperarco $A_{\vec{p}}$ donde se proyecta el punto $\vec{p} = (0, 0, 0)$).

De la explicación anterior, se puede generalizar que si existen hiperarcos en el GDM con distinto "color", sobre los que se proyectan los mismos datos, existe comunicación *punto a punto* entre los procesadores correspondientes a los colores de los hiperarcos.

Definición 10 *Supongamos los hiperarcos* $A_{\vec{p}}^{color_1,orden_1}$ *y* $A_{\vec{q}}^{color_2,orden_2}$. *Si* $i^p = i^q$ *entonces el elemento* $M(i^p)$ *que se proyecta sobre ambos hiperarcos se comunica punto a punto:*

- *desde* P^{color_1} *a* P^{color_2} *si* $I^p < I^q$
- *desde* P^{color_2} *a* P^{color_1} *si* $I^p > I^q$

En cualquier caso, siempre se comunica el elemento a calcular, desde el procesador origen hasta el que tiene los datos necesarios para realizar dicho cálculo. Esto es así, porque debe cumplirse la condición inicial utilizada en la generación del GDM, es decir, todos los datos relacionados deben tener el mismo "color".

4.2.3.2 Solapamiento de cálculos y comunicaciones

Supongamos que el algoritmo que describe el problema a resolver tiene l iteraciones y que existen $nProc$ procesadores en la arquitectura paralela, de forma que cada procesador debe ejecutar $it = \frac{l}{nProc}$ iteraciones. Supongamos, además, que el procesador $P^{i(j)}$ está ejecutando la iteración $v(w)$ siendo $0 \leq v, w \leq it - 1$, y que el conjunto de elementos que utiliza este procesador en esa iteración, conjunto $D^{v(w)}$ de tamaño L^{it} ($L^{it} < L$, siendo L la cantidad de datos que se envían en un mensaje), se puede dividir en s subconjuntos $D_x^{v(w)}$ de tamaño L_x^{it} ($L_x^{it} = \frac{L^{it}}{s}$ y $D_x^{v(w)} \in D^{v(w)}$) siendo $0 \leq x \leq s-1$.

Según esto, el solapamiento de cálculos y comunicaciones, total o parcial, depende del valor de t_v y t_w, de forma que:

- Si $t_v = t_w$ entonces puede o no existir solapamiento. Supongamos el elemento $d_x^{v(w)} \in D^{v(w)}$ utilizado en la iteración $v(w)$ por el procesador $P^{i(j)}$, tal que $d_x^v \delta d_z^w$ y $\not\exists\ d_x^v \delta d_y^v$ ($d_y^v \in D_y^v$ y $0 \leq x, y, z \leq s-1$ tal que $y \neq x$). Si suponemos que: τ_{cal} es el tiempo necesario para que se realice una operación en punto flotante, $s > 1$, l_x es el número de elementos a calcular y L_x^{it} es el número de elementos a comunicar, puede ocurrir que:

  - $\beta + L_x^{it} \times \tau_{com} \leq l_x \times \tau_{cal}$
  - $\beta + L_x^{it} \times \tau_{com} > l_x \times \tau_{cal}$

  En el primer caso, existe solapamiento total de cálculos y comunicaciones, ya que el tiempo de comunicación del bloque D_x^v es menor o igual al tiempo de cálculo en el bloque D_{x+1}^v ($t_{exe} = t_{cal}$). Sin embargo, en el segundo caso, de la expresión se puede extraer que: $t_{exe} = t_{cal} + t_{com}^*$, donde t_{com}^* es el tiempo en el que no existe solapamiento, por lo que en este caso éste es parcial.

  Asimismo, si $s = 1$, no se pueden aplicar las expresiones anteriores, ya que $d^v \delta d^w$ ($d^v \in D^v$ y $d^w \in D^w$). En este caso, la ejecución de la iteración v en el procesador P^i debe realizarse previamente a la ejecución de la iteración w en el procesador P^j, por tanto no existe solapamiento ($t_{exe} = t_{cal} + t_{com}$).

- Si $t_v < t_w$ existe solapamiento. En este caso, si $v < w-1$ y $d_x^v \delta d_z^w$ y $\not\exists\ d_x^v \delta d_y^v$ ($d_x^{v(w)} \in D_x^{v(w)}$, $d_y^v \in D_y^v$ y $0 \leq x, y, z \leq s-1$ tal que $y \neq x$), el solapamiento ocurre cuando:

  - $\beta + L^{it} \times \tau_{com} \leq (w - v - 1) \times l_{it} \times \tau_{cal}$
  - $\beta + L^{it} \times \tau_{com} > (w - v - 1) \times l_{it} \times \tau_{cal}$

  suponiendo l_{it} elementos a calcular en cada bloque $D^{v(w)}$. Asimismo, si $v = w-1$, existe solapamiento si:

  - $\beta + L_x^{it} \times \tau_{com} \leq \sum_{r=x+1}^{s-1} l_r \times \tau_{cal} + \sum_{r=1}^{z-1} l_r \times \tau_{cal}$
  - $\beta + L_x^{it} \times \tau_{com} > \sum_{r=x+1}^{s-1} l_r \times \tau_{cal} + \sum_{r=1}^{z-1} l_r \times \tau_{cal}$

  En cualquiera de los casos, las conclusiones son las mismas que las del punto anterior, es decir, si se cumple la primera expresión existe solapamiento total y si se cumple la segunda, el solapamiento es parcial.

El problema de las expresiones anteriores consiste en encontrar el valor de s y por tanto de L_x^{it}. Así, para que exista solapamiento de cálculos y comunicaciones, el valor de L_x^{it} debe ser el siguiente:

$$L_x^{it} \leq \frac{l_x \times \tau_{cal} - \beta}{\tau_{com}}$$

El valor de τ_{cal} depende del procesador utilizado y los valores de β y τ_{com} dependen del sistema de comunicación. En [72], [110] se presentan estos últimos valores para el sistema de comunicación utilizado en el IBM-SP2 y distintos tamaños de mensajes (L_x^{it}). En [69] se estudia una red de estaciones de trabajo (Berkeley NOW) utilizando como parámetros los indicados por el modelo LogP [24].

4.3 Acotando el problema

Hasta ahora hemos indicado las condiciones generales utilizadas en la metodología de distribución de datos y comunicaciones, sin embargo, la generación del grafo de dependencias modificado siempre depende del problema a resolver.

Las dependencias de datos existentes en un determinado algoritmo condicionan la construcción del GDM, ya que la ejecución de los bucles que encierran a la dependencia analizada puede hacerse innecesaria. Las dependencias de flujo existentes en un algoritmo pueden ser de dos tipos: uniformes o no uniformes. En cualquier caso, la representación de las mismas se realiza mediante su vector *distancia de dependencia*:

Definición 11 *Sea la pareja de iteraciones* $v = (I_1^v, I_2^v, ..., I_n^v) = I^v$ *y* $w = (I_1^w, I_2^w, ..., I_n^w) = I^w$ *tal que* $v < w$, *y la sentencia* $M(i_1^w, i_2^w, ..., i_m^w) = M(j_1^v, j_2^v, ..., j_m^v)$ $(S^v \delta S^w)$. *El vector distancia de dependencia* $I^w - I^v = (i_1^w - j_1^v, i_2^w - j_2^v, ..., i_m^w - j_m^v)$ *se denota como vector desp.*

La distinción entre dependencias uniformes y no uniformes viene reflejada en el vector *desp* ya que si una pareja $(i_l^w - j_l^v)$ puede tomar más de un valor, las dependencias son no uniformes. En este caso se tiene más de un vector *desp* referenciados mediante un subíndice. Al generar el GDM, si las dependencias son uniformes, no se ejecutan los bucles exteriores a la sentencia donde se produce la dependencia, sin embargo, si

las dependencias no son uniformes, existen bucles que deben ser ejecutados. Debido a esto, se requiere un estudio diferenciado de ambos casos.

4.3.1 Distribución de datos con dependencias uniformes

Supongamos el algoritmo secuencial 4.5, formado por n bucles que encierran a una o más sentencias.

```
do I_1 = L_1, U_1, Step_1
   ...
   do I_n = L_n, U_n, Step_n
      M(i_1, i_2, ..., i_m) = M(j_1, j_2, ..., j_m)
      ...
   enddo
   ...
enddo
```

(4.5)

En este algoritmo se accede en l iteraciones a todos los elementos del conjunto dimensionado M, siendo $i_x^w(j_x^w) = f(I_y)^v$ para $1 \leq y \leq n$, $1 \leq x \leq m$ (v y w son dos iteraciones del algoritmo, tal que $v \leq w$ y $1 \leq v, w \leq l$). La sentencia interior a los n bucles del código queda caracterizada por una serie de vectores:

Definición 12 *Sea la sentencia* $M(i_1, i_2, ..., i_m) = ...$, *tal que* $i_x^w = f(I_y)^v$, *siendo* $1 \leq y \leq n$ *y* $1 \leq x \leq m$. *El incremento producido entre los distintos valores de los índices del conjunto dimensionado se denota como inc, y corresponde con los saltos de los bucles del algoritmo, de forma que* $inc_x = Step_y$.

En el análisis realizado a continuación suponemos un vector *inc* constante en todos los casos. Asimismo, en el algoritmo 4.5, el valor inicial de los índices de los bucles $(I_1, I_2, ..., I_n)$ se denota como $(L_1, L_2, ..., L_n)$. El valor L_i puede ser un valor constante $(1 \leq i \leq n)$ o una función de los índices de un bucle anterior $(1 < i \leq n)$. Si L_i es un valor constante, existe una primera iteración en la que L_i corresponde con el valor mínimo de I_i, considerando el valor mínimo como el primer valor que toma I_i cada

vez que se ejecuta el bucle i. Asimismo, si L_i no es constante, existe un número no determinado de iteraciones (x), en la que L_i corresponde con el valor mínimo de I_i. Por ejemplo:

$$
\begin{array}{l}
do\ I_1 = 0,2,1 \\
\quad do\ I_2 = I_1 + 1,3,1 \\
\quad\quad M(I_1, I_2) = ... \\
\quad enddo \\
enddo
\end{array}
$$

El valor mínimo del bucle I_1 es el valor 0, ya que este bucle sólo se ejecuta una vez ($L_1 = 0$). Sin embargo, el bucle I_2 presenta tres valores mínimo, ya que se ejecuta tres veces y L_2 es función del índice del bucle I_1 ($L_2 = 1$, $L_2 = 2$ y L_2=3).

Definición 13 *El valor de los índices del conjunto dimensionado* $(i_1, i_2, ..., i_m)$ *en aquellas iteraciones en las que* L_i *corresponde con el valor mínimo de* I_i $(1 \leq i \leq n)$ *se denomina vector inicial.*

Si L_i no es constante existe más de un vector *inicial*, que se denota utilizando un subíndice. La misma explicación se puede hacer con respecto al valor final de cada bucle, de forma que:

Definición 14 *El valor de los índices del conjunto dimensionado* $(i_1, i_2, ..., i_m)$ *en aquellas iteraciones en las que* U_i *corresponde con el valor máximo de* I_i $(1 \leq i \leq n)$ *se denomina vector final.*

Igualmente, si U_i no es constante existe más de un vector *final*, que se denota utilizando un subíndice. En cualquiera de las definiciones anteriores se supone que $L_i \leq U_i$, $Step_i > 0$ (aunque puede extenderse al caso $L_i > U_i$, $Step_i < 0$).

Según sean las componentes de los vectores *inicial* y *final* se tienen los casos que se presentan a continuación.

4.3.1.1 Caso 1: vectores *inicial* y *final* constantes (caso elemental)

En el algoritmo 4.5, puede ocurrir que:

- $i_x^w = I_y^w$ y $j_x^v = I_y^{w-k}$

  quedando la sentencia del algoritmo de la forma:

$$M(..., I_y, ...) = M(..., I_y - k, ...)$$

- $i_x^{w+k_1} = I_y^{w+k_1}$ y $j_y^v = I_y^w$ (o $j_y^v = I_y^{w-k_2}$)

  quedando la sentencia del algoritmo tal que:

$$M(..., I_y + k_1, ...) = M(..., I_y, ...) \text{ (o } M(..., I_y + k_1, ...) = M(..., I_y - k_2, ...))$$

En cualquier caso, $1 \leq y \leq n$, $1 \leq x \leq m$ y $k, k_1, k_2 \geq 0$. Dadas dos iteraciones v y w del algoritmo 4.5, la distribución de los elementos del conjunto dimensionado sigue el siguiente criterio:

$$\forall elemento \in Conjunto$$
$$S^v \delta S^w \Longrightarrow color(elemento_w) = color(elemento_v)$$
$$\forall v, w \mid 1 \leq v, w \leq l$$

Si $v = v_0$ y $w = v_0 + k$ entonces $S^{v_0}\delta S^{v_0+k}$, $S^{v_0+k}\delta S^{v_0+2\times k}$,..., $S^{v_0+(r-1)\times k}\delta S^{v_0+r\times k}$, siendo v_0 y $v_0 + r \times k = v_l$, la primera y última iteración del algoritmo respectivamente. En general:

$$S^{v_0+(q-1)\times k}\delta S^{v_0+q\times k} \text{ siendo } 1 \leq q \leq l \text{ y } k \geq 0 \tag{4.6}$$

El "color" se asigna a todos los elementos del conjunto dimensionado, a los que se accede en las l iteraciones del algoritmo (el rango $1..l$ se denomina a partir de este punto, *rango total*). La agrupación de elementos bajo un determinado "color" implica la pertenencia a un mismo procesador.

Supongamos el algoritmo ejemplo siguiente:

```
do i = 0,2,1
   do j = 0,2,1
      m_{i+j} = m_{i+j} + a_i × b_j
   enddo
enddo
```

En este ejemplo, existe una dependencia producida por el cálculo y reutilización de elementos del conjunto m ($m_{i_1} = m_{j_1}$, donde $i_1 = (i + j)$ y $j_1 = (i + j)$). Los vectores definidos anteriormente, considerando al índice de iteración i como el índice del conjunto m, tienen dimensión 1 y son: $desp = (0)$, $inc = (1)$, $inicial = (0)$ y $final = (2)$. En la figura 4.15 se muestran, con el mismo color, aquellos datos que son dependientes. Así por ejemplo, el elemento m_2 depende de los elementos a_i y b_i, siendo $0 \leq i \leq 2$. Según la figura 4.15, el paralelismo máximo se consigue con cinco procesadores, ya que existen cinco colores diferentes. Asimismo, a partir del GDM del algoritmo se puede observar esta distribución de datos (figura 4.16).

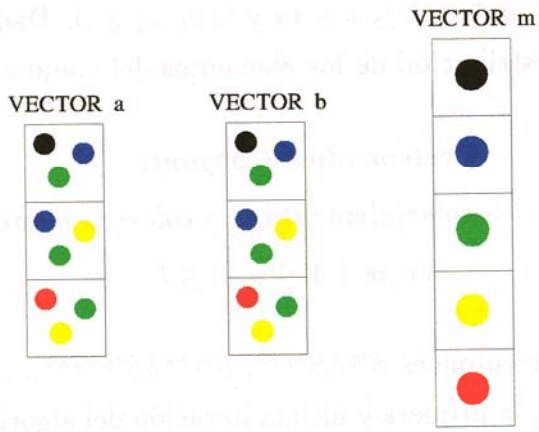


Figura 4.15: Distribución de datos del ejemplo del caso elemental

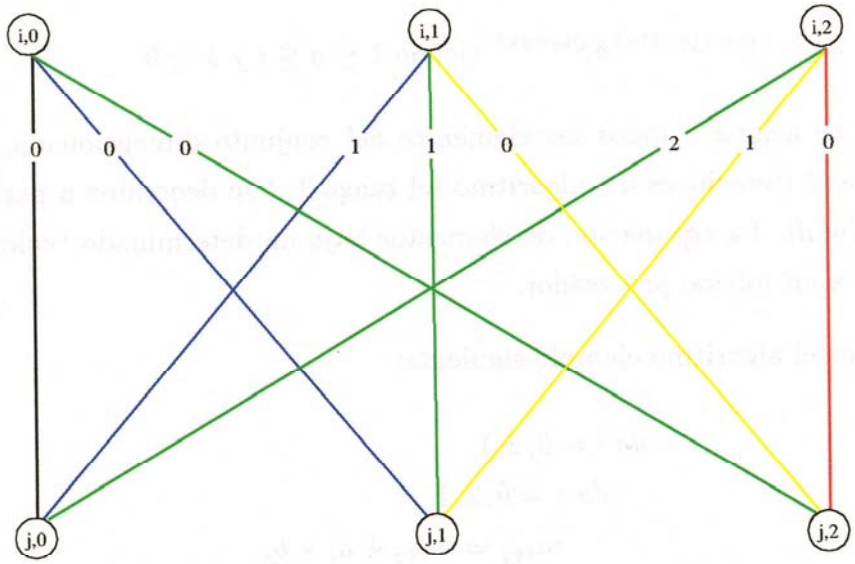


Figura 4.16: GDM del ejemplo del caso elemental

Sin embargo, en la distribución de datos del ejemplo anterior existe replicación de datos, ya que los elementos de los vectores a y b están en más de un procesador. Si en la figura 4.16 se observa el orden de ejecución de los cálculos se puede hacer una nueva distribución que evite esta replicación (figura 4.17). De esta forma, los elementos proyectados sobre los hiperarcos de "orden" 0 se distribuyen en los dintintos procesadores.

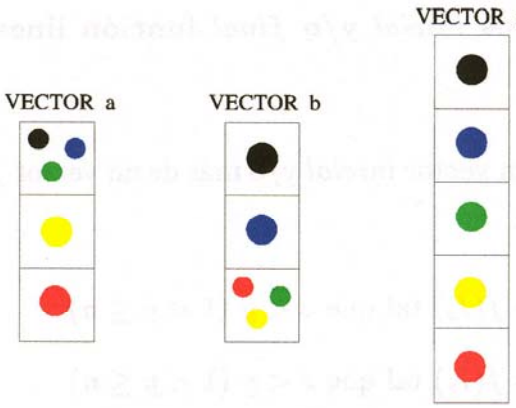


Figura 4.17: Nueva distribución de datos del ejemplo del caso elemental

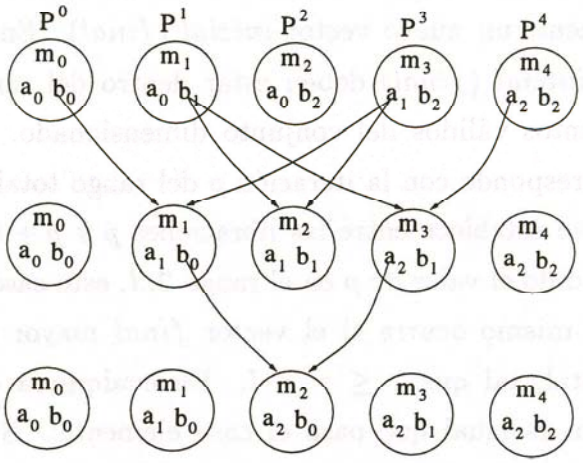


Figura 4.18: Comunicación de datos del ejemplo del caso elemental

Esta nueva distribución implica que cada procesador puede realizar su primer cálculo; y que antes de hacer el segundo debe comunicarse con los procesadores propietarios de los datos que necesita. Así, antes de realizar el segundo cálculo, los procesadores azul (P^1), verde (P^2) y amarillo (P^3) deben tener los datos: a_1 para los dos primeros y a_2 para el último. El valor de a_1 y a_2 lo comunican los procesadores P^3 y P^4 respectivamente. Lo mismo ocurre en el siguiente cálculo en el que hay que comunicar

a_2 (desde el procesador P^3 al procesador P^2). En la figura 4.18 se presentan estas comunicaciones.

La táctica que se sigue en este caso por parte de la herramienta que implementa la metodología, es de no desenrollar los bucles esto es, la distribución se hace directamente ahorrando tiempo de ejecución.

4.3.1.2 Caso 2: vectores *inicial* y/o *final* función lineal de los índices de los bucles

En este caso existe más de un vector *inicial* y/o más de un vector *final*. En el algoritmo 4.5, este caso se produce si:

- $i_x(y_x) = f(I_y)$ y $L_y = f(I_z)$ tal que $z < y$ $(1 < y \leq n)$
- $i_x(y_x) = f(I_y)$ y $U_y = f(I_z)$ tal que $z < y$ $(1 < y \leq n)$

Para cada iteración del algoritmo, una variación sobre I_z implica una variación sobre L_y (U_y) y consecuentemente un nuevo vector *inicial* (*final*). En cualquier caso, los valores de los vectores *inicial* (*final*) deben estar dentro del rango total, para que correspondan con elementos válidos del conjunto dimensionado. Por ejemplo, si el vector *inicial* menor corresponde con la iteración p del rango total, tal que $1 < p \leq l$, la primera dependencia se establece entre las iteraciones p y $p + k$ $(S^p \delta S^{p+k})$, siendo $k \geq 0$. Al estar comprendido el valor de p en el rango $2..l$, este caso es un subconjunto del caso elemental. Lo mismo ocurre si el vector *final* mayor corresponde con la iteración r del rango total, tal que $1 \leq r < l$. En cualquiera de estos supuestos, la táctica de distribución es igual que para el caso elemental. Si este caso se trata como caso elemental, se accede a elementos del conjunto dimensionado que realmente no forman parte del cálculo realizado en el algoritmo. Así, si el primer vector *inicial* menor corresponde con la iteración p y el segundo vector *inicial* menor corresponde con la iteración q, entre ambas iteraciones existen elementos del conjunto dimensionado que van a ser distribuidos (por parte de la herramienta) aunque no se utilizan. La ventaja que presenta el tratar este caso como caso elemental está en que la distribución de datos sigue las mismas pautas, es decir, no se necesita la ejecución de los bucles que encierran a la dependencia.

Supongamos el ejemplo:

$$
\begin{array}{l}
do\ i = 0,3,1 \\
\quad do\ j = i+1,3,1 \\
\quad\quad M_{i,j} = M_{i-1,j} \\
\quad enddo \\
enddo
\end{array}
$$

En este ejemplo $M_{i_1,i_2} = M_{j_1,j_2}$, tal que $(i_1,i_2) = (i,j)$ y $(j_1,j_2) = (i-1,j)$. Los vectores definidos anteriormente tienen dimensión 2 y son: $desp = (1\ 0)$, $inc = (1\ 1)$, $inicial = (0\ i+1)$ y $final = (3\ 3)$. El vector *inicial* corresponde con tres vectores: $inicial_0$=(0,1), $inicial_1$=(1,2) e $inicial_2$=(2,3). La distribución de datos se presenta en la figura 4.19 y su GDM en la figura 4.20.

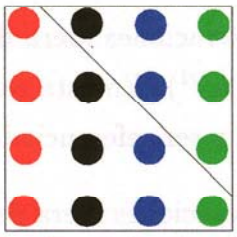


Figura 4.19: Distribución de datos del ejemplo del caso 2

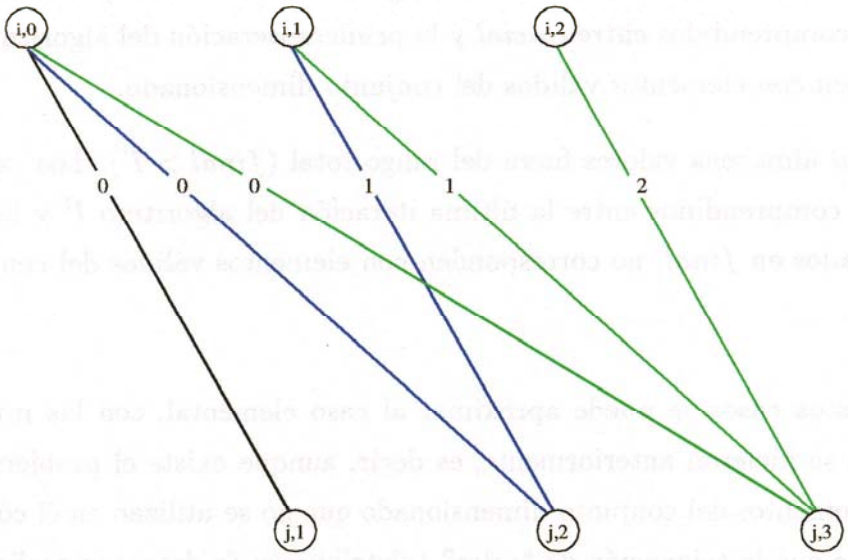


Figura 4.20: GDM del ejemplo del caso 2

Si comparamos las figuras 4.19 y 4.20 se observa que en la primera existen cuatro colores, mientras que en la segunda hay tres. Esto se explica porque en el *GDM* sólo aparecen los elementos de la matriz vinculados a los índices de los bucles, y sin embargo, se han distribuidos elementos de la matriz que no van a ser utilizados, al hacerse una aproximación al caso elemental.

4.3.1.3 Caso 3: vectores *inicial* y/o *final* variables no dependientes de los índices de los bucles

Este caso es parecido al anterior, sin embargo, los valores de los vectores *inicial* (*final*) pueden corresponder a valores que se encuentran fuera del rango total. Suponiendo que el vector *inc* almacena valores positivos y que $inicial \leq final$, puede ocurrir:

- Ambos vectores corresponden con iteraciones fuera del rango total ($inicial < I^1$ y $final < I^1$ o $inicial > I^l$ y $final > I^l$). En este caso, no se necesita distribuir los datos del conjunto ya que no van a ser referenciado en la ejecución del código.
- Ambos vectores corresponden con iteraciones fuera del rango total pero $inicial < I^1$ y $final > I^l$. El rango válido es el rango total.
- El vector *inicial* almacena valores fuera del rango total ($inicial < I^1$). Los valores de los índices comprendidos entre *inicial* y la primera iteración del algoritmo I^1, no corresponden con elementos válidos del conjunto dimensionado.
- El vector *final* almacena valores fuera del rango total ($final > I^l$). Los valores de los índices comprendidos entre la última iteración del algoritmo I^l y los valores almacenados en *final*, no corresponden con elementos válidos del conjunto dimensionado.

Cualquiera de estos casos se puede aproximar al caso elemental, con las mismas consideraciones que se hicieron anteriormente, es decir, aunque existe el problema de la distribución de elementos del conjunto dimensionado que no se utilizan en el código, se tiene la ventaja de que la asignación de "color" (distribución de datos) se realiza sin ejecutar los bucles que encierran a la dependencia analizada.

Supongamos el ejemplo:

```
k=0
do i = 0,3,1
   k=k+1
   do j = k + (2 × k) mod 3,5,1
      M_{i,j} = M_{i,j-2}
   enddo
enddo
```

En este ejemplo $M_{i_1,i_2} = M_{j_1,j_2}$, tal que $(i_1, i_2) = (i, j)$ y $(j_1, j_2) = (i, j-2)$. Los vectores definidos anteriormente tienen dimensión 2 y son: $inc = (1\ 1)$, $inicial = (0\ \ k+(2\times k)\ \ mod\ 3)$, $final = (3\ 5)$ y $desp = (0\ 2)$. El vector $inicial$ se descompone en: $inicial_0 = (0,3)$, $inicial_1 = (1,3)$ e $inicial_2 = (2,3)$.

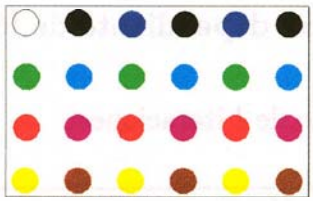


Figura 4.21: Distribución de datos del ejemplo del caso 3

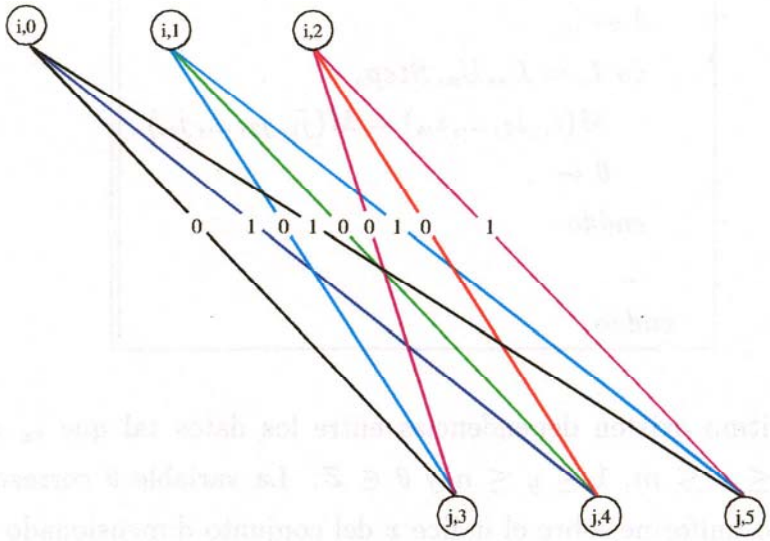


Figura 4.22: *GDM* del ejemplo del caso 3

Al igual que en los casos anteriores, la distribución de los datos utilizados no requiere la ejecución de los bucles y se presenta en 4.21. El *GDM* del ejemplo anterior se muestra en la figura 4.22. Asimismo, al no ser constantes los valores del vector *inicial*, se distribuyen todos los datos de la matriz, incluidos los no utilizados (elementos de las tres primeras columnas).

4.3.2 Distribución de datos con dependencias no uniformes

En este caso, la distribución de los datos no sigue un patrón fijo como ocurre con las dependencias uniformes, y está influenciada por los índices afectados por los desplazamientos no uniformes y por los bucles en los que se asigna valor inicial y se alteran estos desplazamientos. Suponiendo que el vector *desp* es variable en distintas iteraciones del algoritmo 4.5, se pueden dar diferentes casos que se presentan a continuación.

4.3.2.1 Caso 4: vector *desp* no dependiente de los índices de los bucles

Supongamos el siguiente algoritmo de l iteraciones:

```
do I_1 = L_1, U_1, Step_1
   ...
   θ ← ...
   do I_n = L_n, U_n, Step_n
      M(i_1, i_2, ..., i_m) = M(j_1, j_2, ..., j_m)
      θ ← ...
   enddo
   ...
enddo
```

(4.7)

En este algoritmo existen dependencias entre los datos tal que $i_x = I_y$ y $j_x = I_y - \theta$, siendo $1 \leq x \leq m$, $1 \leq y \leq n$ y $\theta \in \mathcal{Z}$. La variable θ corresponde con un desplazamiento no uniforme sobre el índice x del conjunto dimensionado (siempre que se ejecuta el programa el valor θ es igual en una iteración dada). El valor inicial de este desplazamiento se asigna en el bucle I_{n-1} y se altera en el bucle I_n. En cualquier caso,

al ser el desplazamiento variable existen l vectores *desp* que se denotan utilizando un subíndice (donde el vector $desp_v \neq desp_w$ si $v \neq w$ y $1 \leq v, w \leq l$). Asimismo, los n bucles que forman el código del algoritmo se pueden dividir en dos grupos: los bucles que indexan al conjunto dimensionado (bucle r tal que $i_x = I_r$) y los que no lo indexan (bucle r tal que $i_x \neq I_r$).

Supongamos la existencia de q $(1 \leq q \leq m)$ índices del conjunto dimensionado sobre los que se aplica un desplazamiento variable, de forma que al desplazamiento θ^i $(1 \leq i \leq q)$ se le asigna valor inicial en el bucle s^i (en la iteración v, tal que $1 \leq v \leq l$) y se le modifica en el bucle z^i (en iteraciones posteriores a la v), tal que $s^i < z^i$. Los q bucles s^i y z^i constituyen los conjuntos S y Z respectivamente. Respecto a los bucles que encierran a la dependencia analizada se hacen las siguientes definiciones:

Definición 15 *Sean los bucles s_max y t tal que* $1 \leq s\_max < n$ *y* $1 \leq t < s\_max$. *Si* $s\_max \in S$ *entonces* $t \notin S$.

Definición 16 *Sean los bucles z_min y t tal que* $1 < z\_min \leq n$ *y* $z\_min < t \leq n$. *Si* $z\_min \in Z$ *entonces* $t \notin Z$.

Definición 17 *Sea una iteración v tal que* $(i_1^v, i_2^v, ..., i_m^v) = (I_{y_1}^v, I_{y_2}^v, ..., I_{y_m}^v)$ *siendo* $1 \leq y_k \leq n$ $(1 \leq k \leq m)$. *El bucle min corresponde con* y_k $(min = y_k)$ *si* $min > y_j$ $(1 \leq j \leq m$ *e* $k \neq j)$.

Definición 18 *Sea una iteración v tal que* $(i_1^v, i_2^v, ..., i_m^v) = (I_{y_1}^v, I_{y_2}^v, ..., I_{y_m}^v)$ *siendo* $1 \leq y_k \leq n$ $(1 \leq k \leq m)$. *El bucle max corresponde con* y_k $(max = y_k)$ *si* $max < y_j$ $(1 \leq j \leq m$ *e* $k \neq j)$.

Asimismo, en la iteración v tal que $1 \leq v \leq l$, el vector $desp_v$ (de dimensión q) está formado por los valores de los desplazamientos θ^i $(1 \leq i \leq q)$ en esa iteración. Respecto al vector de desplazamiento *desp* se hacen las siguientes definiciones:

Definición 19 *El número de vectores de desplazamiento desp diferentes viene dado por la expresión:*

$$n\_\theta = \prod_{r=s\_max+1}^{z\_min} v_r$$

donde v_r es el número de valores diferentes que puede tomar el índice del bucle r (I_r) y $1 \leq n\_\theta \leq l$.

Definición 20 *El número de iteraciones en las que cualquier vector de desplazamiento ($desp_i$, $1 \leq i \leq n\_\theta$) se mantiene constante viene dado por la expresión:*

$$n\_\theta_i = (\prod_{r=z\_min+1}^{min} v_r)^{i_x=I_r}$$

Definición 21 *El número de iteraciones en las que una misma secuencia de n_θ vectores de desplazamiento (secuencia θ) se repite corresponde con:*

$$n\_sec = (\prod_{r=max}^{s\_max} v_r)^{i_x=I_r}$$

Definición 22 *La distancia (número de elementos del conjunto dimensionado) entre dos secuencias iguales de n_θ vectores de desplazamiento viene dada por la expresión:*

$$d = (\prod_{r=s\_max+1}^{min} v_r)^{i_x=I_r}$$

Según las definiciones anteriores, los m bucles que indexan al conjunto dimensionado se encuentran en el intervalo $max..min$ y los $n-m$ bucles que no lo indexan se encuentran entre 1 y n (incluido el intervalo $max..min$). De esta forma, las l iteraciones del algoritmo 4.7 se pueden agrupar de la forma:

$$(\prod_{r=1}^{n} v_r)^{i_x=I_r} \times (\prod_{r=1}^{n} v_r)^{i_x\neq I_r} = (\prod_{r=max}^{min} v_r)^{i_x=I_r} \times (\prod_{r=1}^{n} v_r)^{i_x\neq I_r} \quad (4.8)$$

El término $(\prod_{r=max}^{min} v_r)^{i_x=I_r}$ de la expresión 4.8, corresponde con los elementos del conjunto dimensionado y el término $(\prod_{r=1}^{n} v_r)^{i_x\neq I_r}$ con las distintas referencias a un mismo elemento de dicho conjunto durante la ejecución del algoritmo 4.7. A su vez, los intervalos $(max..min)^{i_x=I_r}$ y $(1..n)^{i_x\neq I_r}$ se pueden descomponer en varios subintervalos de acuerdo con la posición de los bucles s_max y z_min. De igual forma, sea cual sea la posición de estos bucles, el intervalo $1..s\_max$ se puede separar en los subintervalos: $(1..s\_max)^{i_x=I_r}$ (que pertenece al intervalo $(max..min)^{i_x=I_r}$) y $(1..s\_max)^{i_x\neq I_r}$ (que pertenece al intervalo $(1..n)^{i_x\neq I_r}$). Esto mismo se puede aplicar al intervalo $z\_min+1..n$. Atendiendo a los bucles s_max y z_min se puede establecer los siguientes lemas:

Lema 3 *Sea el bucle t tal que* $z\_min + 1 \leq t \leq n$. *Si* $i_x \neq I_t$ *entonces el bucle t no es considerado en la distribución de los elementos del conjunto dimensionado.*

Demostración. Cualquier iteración de los bucles interiores al bucle z_min presenta un vector de desplazamientos $desp_i$ constante, ya que z_min es el mayor bucle del conjunto Z en el que se modifica este vector. El intervalo $z\_min + 1..n$ se puede descomponer en dos subintervalos: $(z\_min + 1..n)^{i_x = I_r}$ y $(z\_min + 1..n)^{i_x \neq I_r}$. Así, cualquier elemento del conjunto dimensionado que tenga fijos los índices del intervalo $z\_min + 1..n$ se referencia $(\prod_{r=z\_min+1}^{n} v_r)^{i_x \neq I_r}$ veces. Las referencias a un mismo elemento del conjunto, se caracterizan porque tienen el mismo vector de desplazamiento $desp_i$. Esto implica que cualquier sentencia que actualice este elemento (con este vector de desplazamiento) utiliza siempre los mismos datos (para lectura y escritura), por lo que sólo es necesaria una única iteración de los bucles $(z\_min + 1..n)^{i_x \neq I_r}$ para estudiar la utilización de los elementos (respecto a la distribución de datos y posteriores comunicaciones). Según esto, $(\prod_{r=z\_min+1}^{n} v_r)^{i_x \neq I_r} = 1$.

Lema 4 *Sea el bucle t tal que* $1 \leq t \leq s\_max$. *Si* $i_x \neq I_t$ *y* $t < s\_max < max$ *o* $t < max < s\_max$, *entonces el bucle t no es considerado en la distribución de los elementos del conjunto dimensionado.*

Demostración. Dividamos el intervalo $1..s\_max$ en: $(1..s\_max)^{i_x = I_r}$ y $(1..s\_max)^{i_x \neq I_r}$. Un elemento del conjunto dimensionado, que tenga fijos los índices comprendidos en el intervalo $1..s\_max$, es referenciado $(\prod_{r=1}^{s\_max} v_r)^{i_x \neq I_r}$ veces con la misma secuencia θ, ya que s_max es el menor bucle del conjunto S en el que se asigna valor inicial al vector de desplazamiento $desp_i$. Si $s\_max < max$, en el intervalo $1..s\_max$ sólo existen bucles que no indexan al conjunto dimensionado, luego cualquier elemento del conjunto (cuyos índices están en el intervalo $max..min$) es referenciado $\prod_{r=1}^{s\_max} v_r$ veces con la misma secuencia θ. Según esto, cualquier sentencia que actualiza un elemento del conjunto dimensionado, utiliza siempre los mismos datos (para lectura y escritura) en cualquiera de las $\prod_{r=1}^{s\_max} v_r$ referencias al elemento. Esto supone que no es necesario realizar más de una iteración de los bucles $1..s\_max$, ya que a efectos de la distribución de datos y posteriores comunicaciones, se obtiene la misma información. Si $s\_max > max$, la explicación anterior se puede extender a cualquier bucle en el intervalo $1..max$.

Si aplicamos los lemas 3 y 4, las l iteraciones del algoritmo 4.7 pueden quedar agrupadas, considerando los bucles s_max, z_min, max y min, de la forma siguiente:

$$(\prod_{r=lim\_in+1}^{z\_min} v_r)^{i_x \neq I_r} \times (\prod_{r=max}^{min} v_r)^{i_x = I_r} \tag{4.9}$$

donde:

- $lim\_in = max$ si $s\_max \geq max$.
- $lim\_in = s\_max$ si $s\_max < max$.

Asimismo, los bucles comprendidos en el intervalo $(lim\_in + 1..z\_min)^{i_x \neq I_r}$ pueden agruparse en n_r bloques tal que max_i y min_i corresponden con el bucle más exterior e interior del $bloque_i$ respectivamente $(1 \leq i \leq n\_r)$ y no existe un bucle t $(max_i < t < min_i)$ que cumpla que $i_x = I_t$ (figura 4.23).

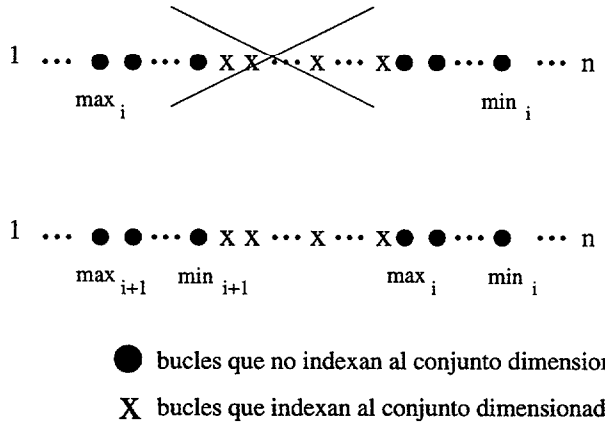


Figura 4.23: Formato de aparición de bucles

Para estudiar cómo se aplican los desplazamientos sobre los elementos del conjunto dimensionado es preciso analizar los bucles que no lo indexan, ya que su existencia implica distintas referencias a un mismo elemento a distancias no constantes (referencias no consecutivas). Así, tenemos las siguientes definiciones:

Definición 23 *El número de iteraciones en las que un mismo elemento e del conjunto dimensionado es referenciado de forma consecutiva se obtiene por la expresión:*

$$n\_d = \prod_{r=min+1}^{z\_min} v_r$$

Definición 24 *El número de veces en las que un bloque de elementos es referenciado de forma no consecutiva viene dado por la expresión:*

$$n\_s = (\prod_{r=lim\_in+1}^{z\_min} v_r)^{i_x \neq I_r}$$

Definición 25 *Si $n\_\theta_i > n\_d$ ($z\_min < min \leq n$) entonces el número de iteraciones en las que a un bloque de elementos diferentes y consecutivos se les aplica el mismo vector de desplazamiento $desp_i$ viene dado por:*

$$n\_b_\theta = \frac{n\_\theta_i}{n\_d} = (\prod_{r=z\_min+1}^{min} v_r)^{i_x = I_r}$$

Definición 26 *Si $n\_\theta_i < n\_d$ ($min < z\_min \leq n$) entonces el número de vectores de desplazamiento diferentes $desp_i$, que se pueden aplicar sobre un elemento e corresponde con:*

$$n\_b_{\neq\theta} = \begin{cases} \prod_{r=min+1}^{z\_min} v_r & : \quad n\_d < n\_\theta \\ n\_\theta & : \quad n\_d > n\_\theta \end{cases}$$

Utilizando estas definiciones se puede sistematizar el procedimiento de asignar "color" a los distintos elementos del conjunto dimensionado, según los valores de los vectores *desp* variables. Este procedimiento se basa en la posición de los bucles *s_max* y *z_min* respecto a los bucles *max*, *min* y a los bucles que no indexan al conjunto dimensionado, dentro del algoritmo secuencial. Así, se pueden dar las siguientes variantes:

- $n\_r = 0$. Se consideran los bucles pertenecientes al intervalo *max..min* (ecuación 4.9). Todos los bucles de este intervalo indexan al conjunto dimensionado ($i_x = I_r$) ya que $n\_r = 0$. Partiendo de estas condiciones iniciales se tiene que $n\_d < n\_\theta$, por lo que $n\_b_{\neq\theta} = 1$. Si consideramos que existen l_0 iteraciones ($l_0 < l$) en los bucles del intervalo *max..min*, se pueden dar los siguientes casos:
  - $n\_b_\theta = 1$. Es un caso elemental para cada n_θ vectores de desplazamiento. En un instante dado, se aplica un vector de desplazamiento determinado $desp_i$ a *n_sec* elementos diferentes del conjunto dimensionado separados una distancia *d*. Por tanto, si l_0 corresponde con el número de iteraciones a ejecutar, se necesitan $\frac{l_0}{n\_sec}$ iteraciones para asignar "color" a todos los elementos.

- $n\_b_\theta \neq 1$. Es un caso elemental para cada n_θ vectores de desplazamiento. En cada instante se aplica un vector de desplazamiento diferente $desp_i$ a n_b_θ elementos consecutivos del conjunto dimensionado. De las l_0 iteraciones se deben ejecutar $\frac{l_0}{n\_b_\theta}$ para asignar "color" a los elementos.

- $n\_r > 0$. En este caso existen bucles que no indexan al conjunto dimensionado en el intervalo $lim\_in + 1..z\_min$ (ecuación 4.9). Si consideramos que existen l_0 iteraciones ($l_0 < l$) en los bucles representados en la ecuación 4.9, se pueden dar las siguientes variantes en función de la posición del bucle z_min:

  - $n\_b_\theta \neq 1$ y $n\_b_{\neq\theta} = 1$. Para este caso $z\_min < min$. De las l_0 iteraciones se necesitan ejecutar $\frac{l_0}{n\_b_\theta}$. Esto es así porque cada vector de desplazamiento ($desp$) se puede asignar en bloques de n_b_θ elementos consecutivos.
  - $n\_b_\theta = 1$ y $n\_b_{\neq\theta} \neq 1$. Para este caso $z\_min > min$. De las l_0 iteraciones se necesitan ejecutar $\frac{l_0}{n\_b_{\neq\theta}}$ ya que, sobre un mismo elemento del conjunto dimensionado se aplican, de forma sucesiva, distintos vectores de desplazamiento ($n\_b_{\neq\theta}$ vectores). De estos vectores de desplazamiento, únicamente el último produce un efecto en la asignación de "color".
  - $n\_b_\theta = 1$ y $n\_b_{\neq\theta} = 1$. Para este caso $z\_min = min$. Debido a las características del problemas se deben ejecutar las l_0 iteraciones de los bucles comprendidos en el intervalo $max..min$, si $max \leq s\_max$; o en el intervalo $s\_max + 1..min$, si $s\_max < max$.

Supongamos el ejemplo:

```
do i=0,2,1
   dsp=0
   do j=0,1,1
      do k=0,2,1
         do p=0,3,1
            M_{j,k} = M_{j,k-dsp}+...
         enddo
      enddo
      dsp=dsp+1
   enddo
enddo
```

En este ejemplo $M_{i_1,i_2} = M_{j_1,j_2}$, tal que $(i_1, i_2) = (j, k)$ y $(j_1, j_2) = (j, k - dsp)$. De las $l = 72$ iteraciones iniciales del algoritmo, según la ecuación 4.9 se ejecutan los bucles comprendidos en el intervalo $max..min$, siendo $l_0 = 6$ el número final de iteraciones. El número de vectores *desp* diferentes corresponde con $n\_\theta = 2$: $desp_x = (0, x)$, siendo $0 \leq x \leq 1$ ($s\_max = 1$ y $z\_min = 2$).

Asimismo, en el intervalo $s\_max + 1..min$ no existen bucles que no indexan al conjunto dimensionado. Según esto, $n\_r = 0$ y $n\_b_\theta = v_3 = 3$ (al ser $z\_min < min$), por lo que se aplica cada vector de desplazamiento *desp* a tres elementos consecutivos del conjunto dimensionado. Según lo explicado para este tipo de problemas, la asignación de "color" a los elementos se puede hacer en $\frac{l_0}{n\_b_\theta} = 2$ iteraciones.

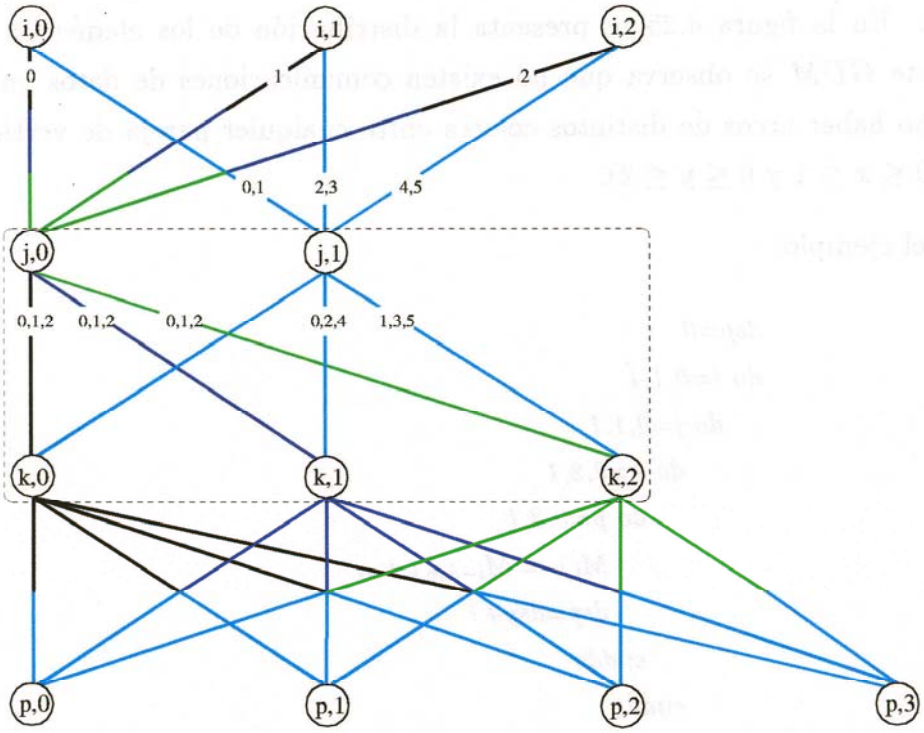


Figura 4.24: *GDM* ejemplo

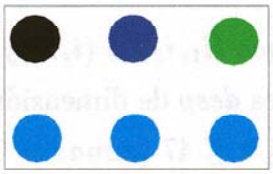


Figura 4.25: Distribución de los elementos de la matriz

En la figura 4.24 se presenta el GDM generado para este ejemplo. En este grafo existe un arco de "color" negro, azul y verde que unen los vértices $V_{j,0}$ y $V_{k,0}$, $V_{j,0}$ y $V_{k,1}$ y los vértices $V_{j,0}$ y $V_{k,2}$ respectivamente, por lo que los elementos $M_{0,0}$, $M_{0,1}$ y $M_{0,2}$ se deben almacenar inicialmente en los procesadores negro (P^0), azul (P^1) y verde (P^2). Lo mismo se puede decir del resto de elementos. Asimismo, en cada hiperarco del GDM se observa el "orden" de ejecución de los cálculos. Si bien en la herramienta diseñada cuando se quiere ver el "orden" de un determinado hiperarco se tiene la opción de marcarlo, apareciendo resaltado junto con su peso; en la gráfica mostrada (figura 4.24) se presentan los pesos de los hiperarcos que comparten vértices, separados por coma. Así, los hiperarcos $A_{(0,0,2,0)}$, $A_{(1,0,2,0)}$ y $A_{(2,0,2,0)}$ tienen peso "orden" 0, 1 y 2 respectivamente, por lo que en el GDM aparece una marca $0, 1, 2$ sobre el arco $A_{j,(0,2)}$ de "color" verde. En la figura 4.25 se presenta la distribución de los elementos de la matriz. En este GDM se observa que no existen comunicaciones de datos entre procesadores al no haber arcos de distintos colores entre cualquier pareja de vértices $V_{j,x}$,$V_{k,y}$ (siendo $0 \leq x \leq 1$ y $0 \leq y \leq 2$).

Supongamos el ejemplo:

```
dsp=0
do i=0,1,1
   do j=0,1,1
      do k=0,2,1
         do p=0,3,1
            M_{i,k} = M_{i-dsp,k}+...
            dsp=dsp+1
         enddo
      enddo
   enddo
enddo
```

En este ejemplo $M_{i_1,i_2} = M_{j_1,j_2}$, tal que $(i_1, i_2) = (i, k)$ y $(j_1, j_2) = (i - dsp, k)$. Al haber $l = 48$ iteraciones, existen l vectores $desp$ de dimensión 2, de los cuales $n\_\theta = 48$ son diferentes: $desp_x = (0, x)$, siendo $0 \leq x \leq 47$ ($s\_max = 0$ y $z\_min = 4$).

Los bucles se encuentran repartidos de forma que $n\_r = 2$, estando el $bloque_1$ formado por el bucle 4 y el $bloque_2$ por el bucle 2. Teniendo en cuenta la posición de

los bucles *min* y *z_min*, $n\_b_\theta = 1$ y $n\_b_{\neq\theta} = v_4 = 4$ (al ser $z\_min > min$), por lo que a un mismo elemento del conjunto dimensionado se le aplican cuatro ($n\_b_{\neq\theta}$) vectores de desplazamiento *desp* diferentes, consecutivamente. De estos vectores, sólo se tiene en cuenta el último, a efectos de asignación de "color". En este ejemplo se deben ejecutar $\frac{l}{n\_b_{\neq\theta}} = 12$ iteraciones de las l iniciales.

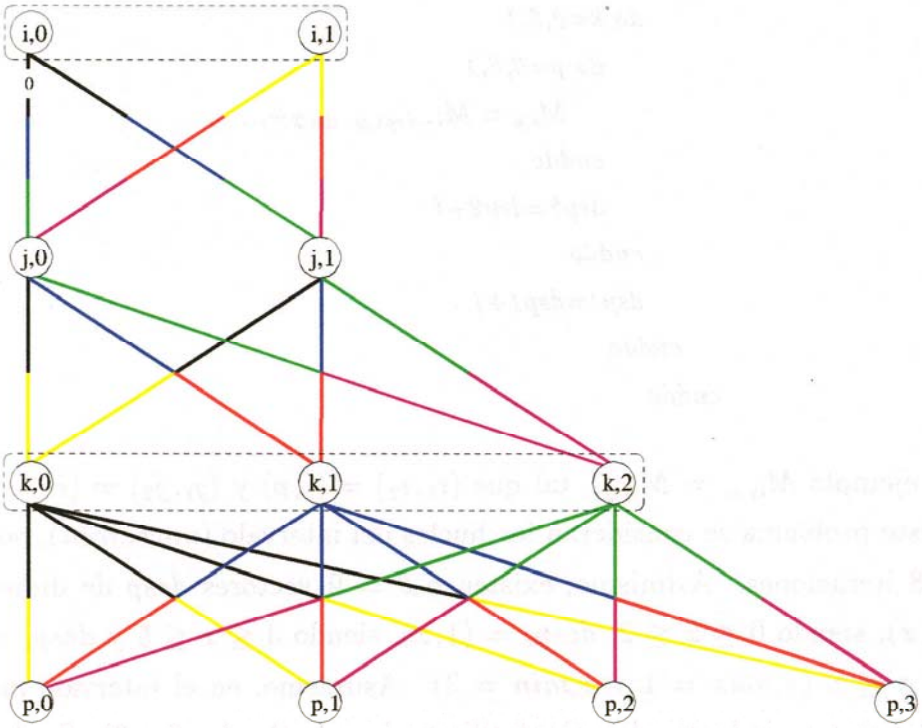


Figura 4.26: *GDM* ejemplo

En la figura 4.26 se presenta el *GDM* del ejemplo anterior en el que se observa que existen seis colores diferentes, de forma que cada elemento de la matriz se calcula en un procesador independiente. Esto es así, porque los valores de los vectores de desplazamiento hacen que los datos referenciados en el algoritmo dependan de elementos no válidos (no existiendo comunicaciones -figura 4.27-).

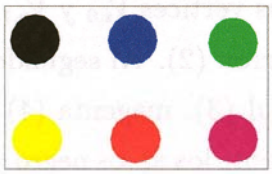


Figura 4.27: Distribución de los elementos de la matriz

Supongamos el ejemplo:

```
do i=0,2,1
   dsp1=0
   dsp2=0
   do j=0,2,1
      do k=0,2,1
         do p=0,3,1
            M_{i,p} = M_{i-dsp1,p-dsp2}+...
         enddo
         dsp2=dsp2+1
      enddo
      dsp1=dsp1+1
   enddo
enddo
```

En este ejemplo $M_{i_1,i_2} = M_{j_1,j_2}$, tal que $(i_1, i_2) = (i, p)$ y $(j_1, j_2) = (i - dsp1, p - dsp2)$. En este problema se consideran los bucles del intervalo $(max..min)$, por lo que hay $l = 108$ iteraciones. Asimismo, existen $n\_\theta = 9$ vectores $desp$ de dimensión 2: $desp_x = (0, x)$, siendo $0 \leq x \leq 2$, $desp_x = (1, x)$, siendo $3 \leq x \leq 5$ y $desp_x = (2, x)$, siendo $6 \leq x \leq 8$ ($s\_max = 1$ y $z\_min = 3$). Asimismo, en el intervalo $max..min$ existen bucles que no indexan al conjunto dimensionado (bucles 2 y 3). Según esto, y la posición de z_min respecto de min, se tiene que: $n\_r = 1$, $n\_b_\theta = v_4 = 4$ y $n\_b_{\neq\theta} = 1$. De esta forma, sobre los elementos del conjunto dimensionado se aplican los vectores de desplazamiento en bloques de n_b_θ elementos consecutivos.

En la figura 4.28 se presenta el GDM de este ejemplo, en el que existe comunicación de datos. La comunicación puede ser detectada en el GDM si observamos los arcos que salen de los vértices $V_{i,x}$ (siendo $0 \leq x \leq 2$) y los arcos que llegan a los vértices $V_{p,y}$ (siendo $0 \leq y \leq 3$). Supongamos los vértices $V_{i,0}$ y $V_{p,1}$. Del primer vértice salen arcos de "color" negro (0), rojo (1) y marrón (2). Al segundo vértice llegan arcos de "color" negro (0), rojo (1), marrón (2), azul (3), magenta (4) y rosa (5). Si comparamos los arcos anteriores, vemos que únicamente los arcos negro y azul tienen como vértice inicial y final los vértices $V_{i,0}$ y $V_{p,1}$, luego el dato $M_{0,1}$ debe estar en la memoria del P^0 y del P^3 en algún momento de la ejecución del algoritmo. Al ser la iteración (0,0,0,1), que

corresponde con el hiperarco $A^{3,0}_{(0,0,0,1)}$, anterior en el tiempo a la iteración (0,0,1,1), que corresponde con el hiperarco $A^{0,1}_{(0,0,1,1)}$, el elemento $M_{0,1}$ se debe almacenar inicialmente en la memoria del P^3 y después de ser actualizado debe ser comunicado al procesador P^0 (definición 10). En la figura 4.29 se presenta la distribución inicial de datos y las comunicaciones realizadas entre los procesadores.

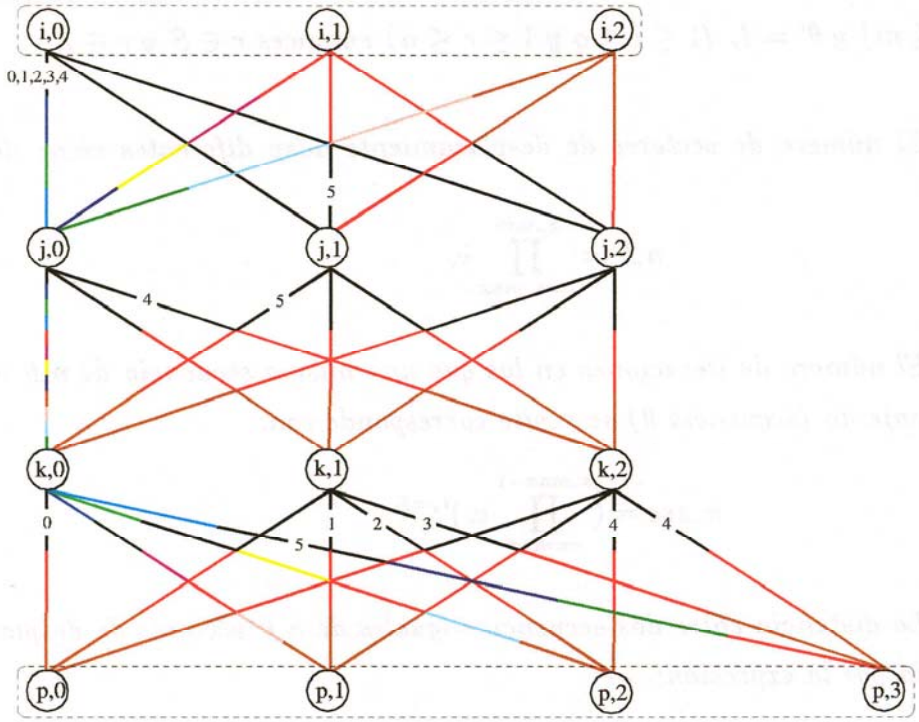


Figura 4.28: *GDM* ejemplo

distribución inicial distribución de datos después de la comunicación

Figura 4.29: Distribución de los elementos de la matriz

4.3.2.2 Caso 5: vector *desp* dependiente de un índice de los bucles

Supongamos el algoritmo 4.7 en el que el desplazamiento $\theta^i = I_r$ ($1 \leq i \leq q$ y $1 \leq r \leq n$, siendo q el número de índices del conjunto dimensionado cuyo desplazamiento es

variable). Este problema es un caso particular del caso 4, siendo $s^i = z^i$ (la asignación inicial y posterior modificación de un desplazamiento θ^i se realiza en el mismo bucle y por lo tanto, $s^i = r$ y $z^i = r$). Debido a esto se realizan las siguientes modificaciones a definiciones previas:

Definición 27 *Si existen q índices en el conjunto dimensionado con desplazamiento variable* $(1 \leq q \leq m)$ *y* $\theta^i = I_r$ $(1 \leq i \leq q$ *y* $1 \leq r \leq n)$ *entonces* $r \in S$ *y* $r \in Z$.

Definición 28 *El número de vectores de desplazamiento desp diferentes viene dado por la expresión:*

$$n\_\theta = \prod_{r=s\_max}^{z\_min} v_r$$

Definición 29 *El número de iteraciones en las que una misma secuencia de* n_θ *vectores de desplazamiento (secuencia* θ*) se repite corresponde con:*

$$n\_sec = (\prod_{r=max}^{s\_max-1} v_r)^{i_x=I_r}$$

Definición 30 *La distancia entre dos secuencias iguales de* n_θ *vectores de desplazamiento viene dada por la expresión:*

$$d = (\prod_{r=s\_max}^{min} v_r)^{i_x=I_r}$$

Supongamos el ejemplo:

```
do i=0,2,1
   do j=0,2,1
      do k=0,2,1
         do p=0,3,1
            M_{j,k} = M_{j,k-p}+...
         enddo
      enddo
   enddo
enddo
```

Al igual que en los casos anteriores se tiene que $(i_1, i_2) = (j, k)$ y $(j_1, j_2) = (j, k-p)$. Para este ejemplo, $s\_max = z\_min = 4$ ya que el único desplazamiento se produce sobre el índice k del conjunto y corresponde con la variación del bucle p (por tanto el valor del desplazamiento se asigna inicialmente en el bucle p y se modifica en ese mismo bucle). Al haber $l = 108$ iteraciones, existen l vectores $desp$ de dimensión 2, de los cuales $n\_\theta = 4$ son diferentes: $desp_x = (0, x)$, siendo $0 \leq x \leq 3$.

Los bucles a considerar están en el intervalo $max..z\_min$ (2..4), existiendo $l_0 = 36$ iteraciones correspondientes a esos bucles. En este intervalo existen dos bucles que indexan al conjunto dimensionado (j y k) y un bucle que no lo indexa (p), por lo que $n\_r = 1$. Al ser $z\_min > min$ se tiene que $n\_b_\theta = 1$ y $n\_b_{\neq\theta} = v_4 = 4$. Según esto, a un mismo elemento del conjunto se le aplican consecutivamente, cuatro vectores de desplazamiento, de forma que para asignar "color" a los elementos hay que ejecutar $\frac{l_0}{n\_b_{\neq\theta}} = 9$ iteraciones de las $l = 108$ iniciales. Como se puede observar, es un caso similar al anterior salvo las modificaciones realizadas sobre los valores n_θ, n_sec y d.

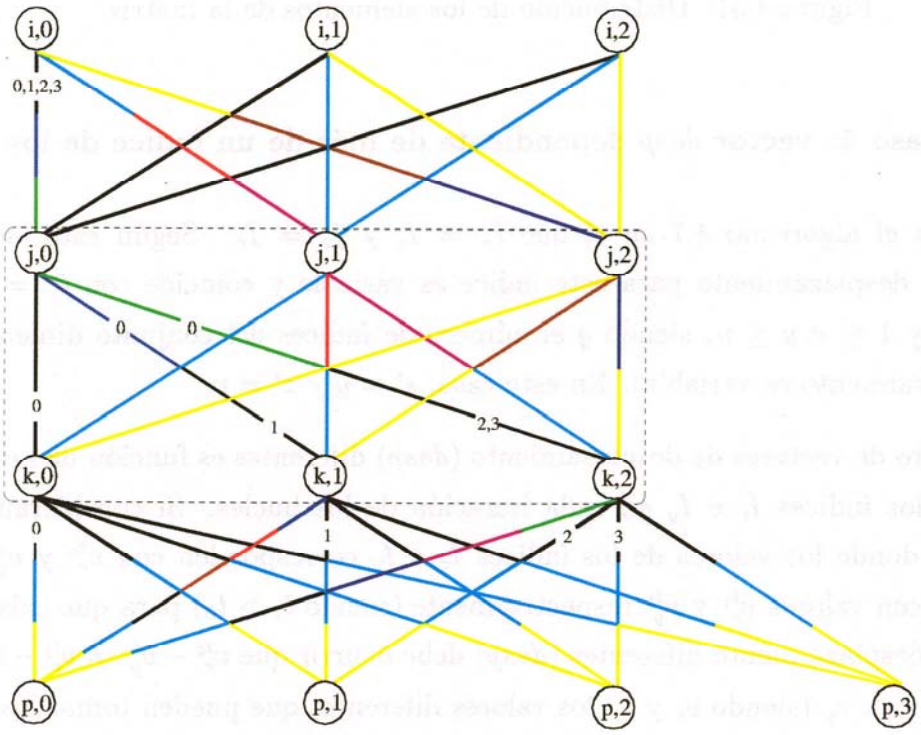


Figura 4.30: *GDM* ejemplo

En la figura 4.30 se presenta el *GDM* para este ejemplo en el que se muestra el "orden" de los hiperarcos negro, azul y verde. Las dependencias producidas por la

variación de p da lugar a los hiperarcos que unen los vértices del grafo. Así, se puede apreciar que existen arcos de "color" negro y azul entre los vértices $V_{j,0}$ y $V_{k,1}$. El hiperarco de "color" azul (1) corresponde con una iteración anterior en el tiempo a la del hiperarco de "color" negro (0), luego el dato $M_{0,1}$ debe almacenarse inicialmente en el P^1 y posteriormente comunicarse (después de ser actualizado) al P^0. Lo mismo ocurre con el dato $M_{0,2}$ (se almacena inicialmente en el procesador verde (P^2) y después del cálculo se comunica al P^0) y con el resto de elementos proyectados sobre vértices unidos por arcos de más de un "color". En la figura 4.31 se muestra la distribución inicial de datos y posteriores comunicaciones.

distribución inicial

distribución de datos después de la comunicación

Figura 4.31: Distribución de los elementos de la matriz

4.3.2.3 Caso 6: vector *desp* dependiente de más de un índice de los bucles

Supongamos el algoritmo 4.7 en el que $i_x = I_y$ y $j_x = I_r$. Según esto, se puede decir que el desplazamiento para este índice es variable y coincide con $\theta^i = I_y - I_r$ ($1 \leq i \leq q$ y $1 \leq r, y \leq n$, siendo q el número de índices del conjunto dimensionado cuyo desplazamiento es variable). En este caso, $s^i = y$ y $z^i = r$.

El número de vectores de desplazamiento (*desp*) diferentes es función de los valores que toman los índices I_r e I_y en cada iteración de los bucles. Si consideramos una iteración t_0 donde los valores de los índices I_r e I_y corresponden con $v_r^{t_0}$ y $v_y^{t_0}$ y una iteración t_1 con valores $v_r^{t_1}$ y $v_y^{t_1}$ respectivamente (siendo $t_1 > t_0$) para que existan n_θ vectores de desplazamiento diferentes (*desp*) debe ocurrir que $v_r^{t_0} - v_y^{t_0} \neq v_r^{t_1} - v_y^{t_1}$ para $1 \leq t_0, t_1 \leq v_r \times v_y$ (siendo v_r y v_y los valores diferentes que pueden tomar los índices I_r e I_y). Si no es así, existen n_θ_6 vectores de desplazamiento diferentes que siempre estará en el rango $1 \leq n\_\theta_6 \leq n\_\theta$. Según esto no se puede establecer el valor de n_θ_6 para este caso, por lo que las definiciones 19, 21 y 22 realizadas para los casos 4 y 5 no pueden ser aplicadas. Debido a esto, la clasificación realizada según el valor n_r se

debe modificar si $n\_r = 0$ (ya que únicamente en algún caso cuando $n\_r = 0$ se utilizan los valores de n_θ, n_sec y d para asignar "color") de forma que:

- $n\_r = 0$ y $n\_b_\theta = 1$. Al ser $n\_b_{\neq\theta} = 1$, se deben ejecutar l_0 iteraciones del algoritmo. En este caso se deben realizar l_0 iteraciones de las l iniciales, ejecutándose los bucles comprendidos en el intervalo $max..min$, si $max \leq s\_max$; o en el intervalo $s\_max + 1..min$, si $s\_max < max$.

Si suponemos que $A(i_1, i_2, ..., i_m) = A(j_1, j_2, ..., j_m)$ y algún x tal que $1 \leq x \leq m$ corresponde con un índice del caso 6, el resto de definiciones realizadas para el caso 4 se pueden utilizar en cualquier ejemplo de este tipo. El único problema que puede existir al aplicar las mismas definiciones se produce en la definición 26, si $n\_d < n\_\theta$, ya que en este caso $n\_b_{\neq\theta} = n\_\theta$ (y este valor no puede ser calculado a priori), sin embargo:

- Si $i_x < j_x$ $(y < r)$ entonces $s\_max \leq min$. Al ser $i_x < j_x$ se tiene que $s^x = y$ y además siempre ocurre que $y \leq min$, por lo que $s^x \leq min$. Por la definición 15, $s^x > s\_max$ así que $s\_max \leq min$.

- Si $i_x > j_x$ $(y > r)$ entonces $s\_max \leq min$. Al ser $i_x > j_x$ se tiene que $s^x = r$ y además siempre ocurre que $y \leq min$, por lo que $r \leq min$ $(s^x \leq min)$. Por la definición 15, $s^x > s\_max$ así que $s\_max \leq min$.

Según lo anterior y teniendo en cuenta las definiciones 19 y 23, siempre se cumple que $n\_\theta > n\_d$ por lo que $n\_b_{\neq\theta}$ nunca toma el valor n_θ.

Supongamos el ejemplo:

```
do i=0,2,1
  do j=0,2,1
    do k=0,2,1
      do p=0,3,1
        M_{i,k} = M_{k,k}+...
      enddo
    enddo
  enddo
enddo
```

En este ejemplo se tiene que $(i_1, i_2) = (i, k)$ y $(j_1, j_2) = (k, k)$. En este caso, $\theta^i = i - k$ por lo que se puede decir que el único desplazamiento se asigna inicialmente en el bucle i y se modifica en el bucle k. De esta forma, $s\_max = 1$ y $z\_min = 3$. Al haber $l = 108$ iteraciones, existen l vectores *desp* de dimensión 2.

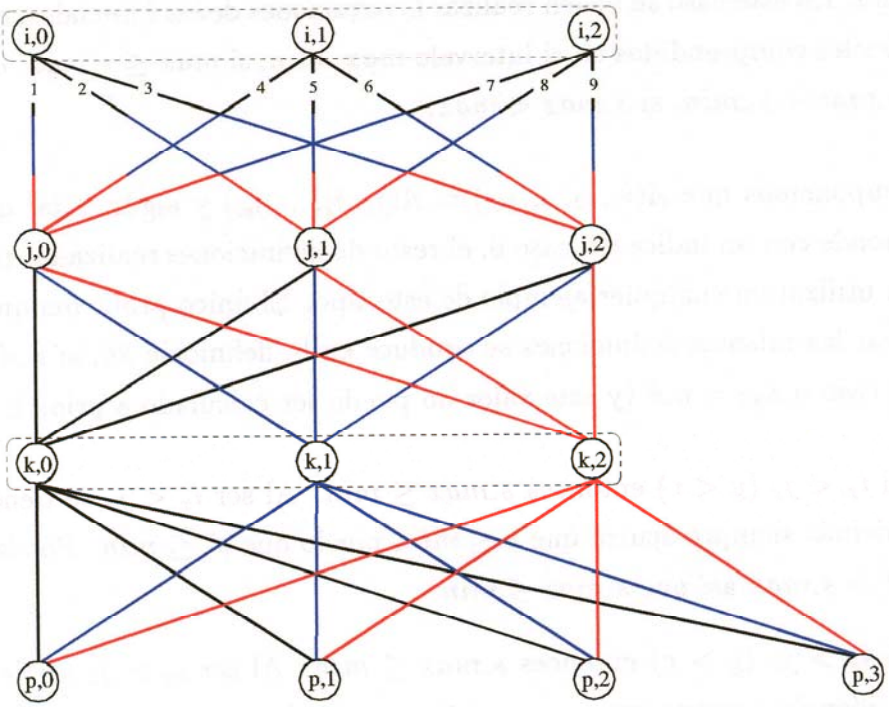


Figura 4.32: *GDM* ejemplo

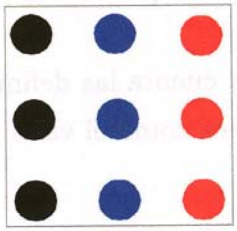


Figura 4.33: Distribución de los elementos de la matriz

De los n bucles del código, los bucles a considerar están en el intervalo $max..z\_min$ (1..3), existiendo $l_0 = 27$ iteraciones correspondientes a esos bucles. En este intervalo, los bucles que indexan al conjunto dimensionado son i y k. Asimismo, existe un bucle que no indexa al conjunto (j), por lo que $n\_r = 1$. Al ser $z\_min = min$ se tiene que $n\_b_\theta = 1$ y $n\_b_{\neq\theta} = 1$, por lo que hay que ejecutar las l_0 iteraciones comentadas para asignar "color" a los elementos. En cualquier caso, al ser $l_0 < l$ se ejecutan menos iteraciones que las establecidas por los bucles iniciales.

En la figura 4.32 se presenta el GDM del algoritmo anterior en el que existen tres procesadores (tres colores) que unen los vértices correspondientes a la fila i y a la k. Asimismo se puede observar que no existen comunicaciones entre procesadores ya que un dato $M_{i,k}$ se almacena en un mismo procesador para las distintas iteraciones de los bucles j y p (figura 4.33).

Capítulo 5

Análisis de algoritmos numéricos típicos

5.1 Introducción

En las aplicaciones científicas actuales se emplea una gran variedad de algoritmos numéricos. La disminución del tiempo de ejecución de estos algoritmos es el verdadero caballo de batalla de los desarrolladores de *software*. Aunque los sistemas son cada vez más rápidos y eficientes, es responsabilidad del programador aprovechar la potencia de cálculo de los modernos procesadores al igual que las ventajas de las nuevas arquitecturas.

Gran cantidad de los esfuerzos del programador se invierte en diseñar un algoritmo, del que se conoce su estructura secuencial, para poderlo ejecutar en paralelo sobre una arquitectura multiprocesadora. En este capítulo se utiliza la metodología de paralelización de estos algoritmos basada en el grafo de dependencias modificado (GDM). Esta metodología analiza las características fundamentales de los programas paralelos como pueden ser: la distribución inicial de datos, los cálculos a realizar en cada procesador del sistema y las comunicaciones necesarias entre ellos. A partir de estas características y teniendo en cuenta que las comunicaciones entre procesadores es el verdadero cuello de botella en la ejecución del algoritmo, se paraleliza el algoritmo secuencial, no sólo descomponiendo el problema en trozos que se puedan ejecutar en paralelo, sino inten-

tando que el tiempo necesario para comunicar los datos entre los procesadores tenga la menor influencia posible sobre el tiempo total. La disminución del efecto de las comunicaciones en el tiempo de ejecución se puede conseguir, básicamente de dos formas: disminuyendo el número de éstas o solapando las comunicaciones con los cálculos. Con esta última técnica y un buen diseño del algoritmo paralelo se puede disminuir la latencia (tiempo perdido en ejecución cuando un procesador espera la conclusión de una comunicación), y en algunos casos anularla.

5.2 Multiplicación matriz-vector

En la resolución de sistemas lineales de ecuaciones una de las operaciones más comunes es la multiplicación de una matriz por un vector. En esta sección se utiliza esta operación para introducir, de forma práctica, la metodología seguida en la paralelización de algoritmos. Dada una matriz $A = (A_{i,j})$ de dimensión $n \times n$ y un vector $b = (b_j)$ de dimensión n siendo $n > 1$, se obtiene la multiplicación $A \times b$ como [42]:

$$c_i = \Sigma_{j=0}^{n-1} A_{i,j} \times b_j \quad \forall \quad i = 0...n-1$$

siendo $c = (c_i)$ un vector de n elementos.

5.2.1 Análisis del algoritmo secuencial. Generación del GDM

El algoritmo secuencial de complejidad $O(n^2)$ que resuelve este problema numérico es el siguiente:

```
do i = 0, n − 1, 1
    do j = 0, n − 1, 1
        c_i = c_i + A_{i,j} × b_j
    enddo
enddo
```

El algoritmo secuencial anterior está compuesto por dos bucles imbricados con una sentencia interior. Cada fila $A_{i,j}$, siendo $j = 0..n-1$, y el vector b_j están relacionados

mediante un producto escalar. El GDM generado para este algoritmo se muestra en la figura 5.1.

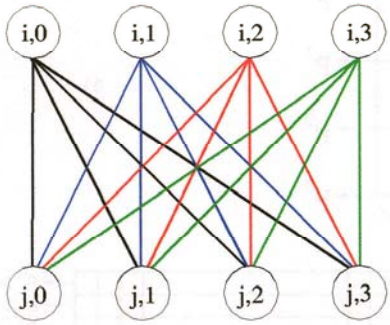


Figura 5.1: Algoritmo ejemplo de la multiplicación matriz-vector ($n = 4$)

5.2.2 Distribución inicial de datos

El GDM de la multiplicación matriz-vector presenta hiperarcos de cuatro "colores". Supongamos el hipervértice $V_{(0,0)}$ y el hiperarco $A_{(0,0)}$ de "color" negro. Este hiperarco significa que todos los elementos de los conjuntos dimensionados referenciados en el algoritmo, cuyos índices i y j sea 0, se deben almacenar en la memoria de un mismo procesador. Así, los elementos $A_{0,0}$, b_0 y c_0 se almacenan en el P^0. De igual forma, los elementos cuyo índice i es igual a 0 y cuyo índice j es igual a j_0, siendo $1 \leq j_0 \leq 3$, se almacenan en el mismo procesador anterior (debido al "color" de los hiperarcos correspondientes).

Esto implica que los elementos $A_{i_0,j}$, los elementos b_j y el elemento c_{i_0}, siendo $0 \leq j \leq n-1$ e i_0 un valor constante comprendido en $(0..n-1)$, se almacenan en la memoria del P^{i_0}. La distribución de datos de la matriz A se realiza por filas o bloques de filas (figura 5.2). En esta figura se puede apreciar que si existen n filas en la matriz y n procesadores, cada uno de ellos puede computar una fila de la matriz por el vector y obtener un elemento del resultado (vector c). Sin embargo, si no existen procesadores suficientes (np procesadores), se deben redistribuir los datos (en este caso, para que la carga en ellos sea la misma). El efecto de esta redistribución en el GDM, es la existencia de un menor número de "colores" diferentes. El objetivo final debe ser la existencia de tantos "colores" diferentes, en los hiperarcos del GDM, como procesadores existan en la arquitectura seleccionada. En la figura 5.3 se presenta el

GDM para la multiplicación matriz-vector si $n = 4$ y $np = 2$.

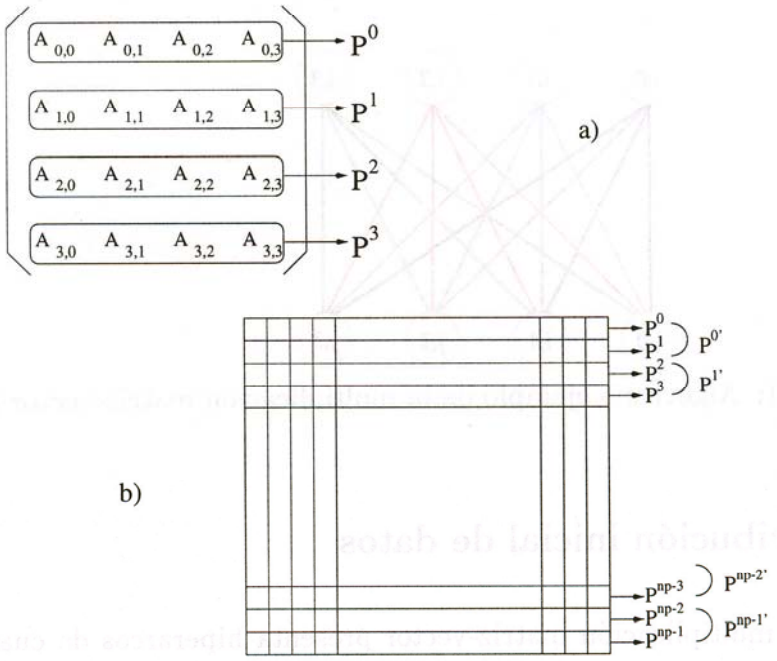


Figura 5.2: a) Distribución de datos de la multiplicación matriz-vector ($n = 4$). b) Distribución de datos, caso general (np procesadores). Redistribución de datos en función del número de procesadores

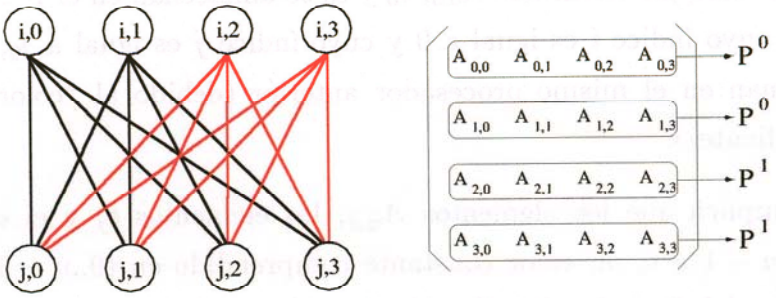


Figura 5.3: Distribución de datos de la multiplicación matriz-vector ($n = 4$, $np = 2$)

5.2.3 Comunicaciones necesarias

Partiendo de la distribución inicial de datos, cada procesador debe almacenar un bloque de elementos de la matriz A, de dimensión $p \times n$, un bloque de elementos de c, de dimensión $p \times 1$, siendo $p = \frac{n}{np}$, y todos los elementos del vector b. De esta forma, cada procesador puede realizar el cálculo del producto escalar entre su submatriz y el vector

b. Según esta distribución, las comunicaciones entre los procesadores no son necesarias, ya que éstos tienen todos los datos que precisan.

Supongamos que el vector b se vuelve a distribuir siguiendo un nuevo patrón, que evite la replicación de datos en la memoria de los distintos procesadores. En este caso, cada procesador almacena un bloque de p elementos del vector b (figura 5.4). De esta forma, cada procesador únicamente puede realizar el cálculo del producto escalar entre su submatriz y su subvector. Sin embargo, los procesadores necesitan utilizar el resto de subvectores b para el cálculo completo de los elementos del vector resultado c, que le corresponde.

En este punto, se necesita la comunicación de datos entre los procesadores. Esta comunicación se hace mediante primitivas de paso de mensajes, *send/receive* no bloqueantes, de forma que los procesadores puedan realizar el cálculo submatriz-subvector y, a la misma vez, comunicar los elementos de los subvectores. Así, en un instante k, el procesador P^x realiza el cálculo de su submatriz (A_x) por el subvector actual (b_k) y recibe del procesador P^y, el subvector siguiente b_{k+1} $(0 \leq x, y \leq np-1)$.

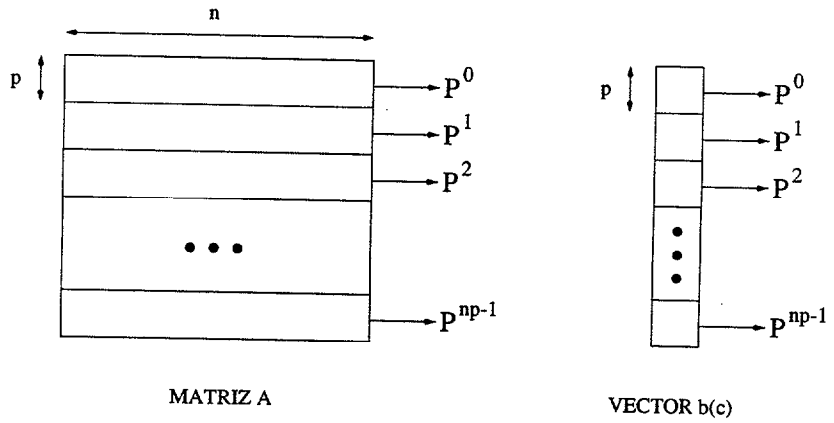


Figura 5.4: Distribución de datos de la multiplicación matriz-vector en bloques de filas consecutivas

5.3 Multiplicación matriz-matriz

Dadas dos matrices $A(A_{i,k})$ y $B(B_{k,j})$ de dimensión $n \times n$, se define la multiplicación matricial $A \times B$ como [42]:

$$C_{i,j} = \Sigma_{k=0}^{n-1} A_{i,k} \times B_{k,j} \ (\forall i, j = 0..n-1)$$

donde $C = (C_{i,j})$ es una matriz de dimensión $n \times n$.

5.3.1 Análisis del algoritmo secuencial. Generación del GDM

El algoritmo secuencial de complejidad $O(n^3)$ que resuelve este problema es:

```
do i=0,n-1,1
   do j=0,n-1,1
      C(i,j)=0
      do k=0,n-1,1
         C(i,j)=C(i,j)+A(i,k) × B(k,j)
      enddo
   enddo
enddo
```

Este algoritmo está compuesto por tres bucles anidados con una sentencia interior. El cálculo de un elemento $C_{i,j}$ es el producto escalar del vector fila $\overrightarrow{A}_{i,k}$ y el vector columna $\overrightarrow{B}_{k,j}$, $(0 \leq k \leq n-1)$. El *GDM* generado a partir de este algoritmo se muestra en la figura 5.5.

5.3.2 Distribución inicial de datos

El *GDM* ejemplo de la figura 5.5 muestra hiperarcos de nueve "colores" enlazando los distintos hipervértices del grafo. Supongamos el hipervértice $V_{(0,0,0)}$. Los elementos de los conjuntos dimensionados referenciados en el algoritmo, cuyos índices i, j y k son 0, se deben almacenar en la memoria de un mismo procesador. Así, los elementos $A_{0,0}$, $B_{0,0}$ y $C_{0,0}$ se almacenan en el P^0. Generalizando, los elementos cuyo índice i y j sean igual a 0 y cuyo índice k sea igual a k_0, siendo $0 \leq k_0 \leq n-1$, se almacenan también en el mismo procesador (debido al "color" de los hiperarcos correspondientes). Según esto, los datos se distribuyen en los procesadores siguiendo el patrón mostrado en la figura 5.6.

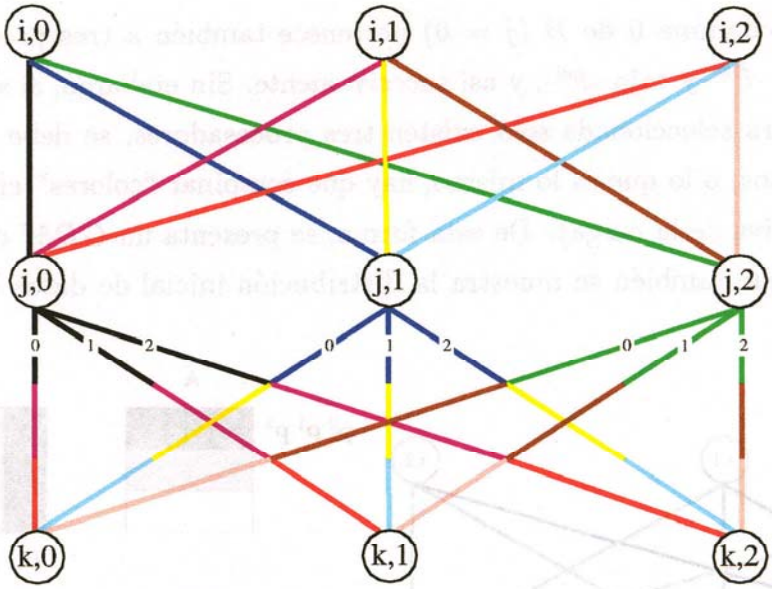


Figura 5.5: GDM de la multiplicación matriz-matriz ($n = 3$)

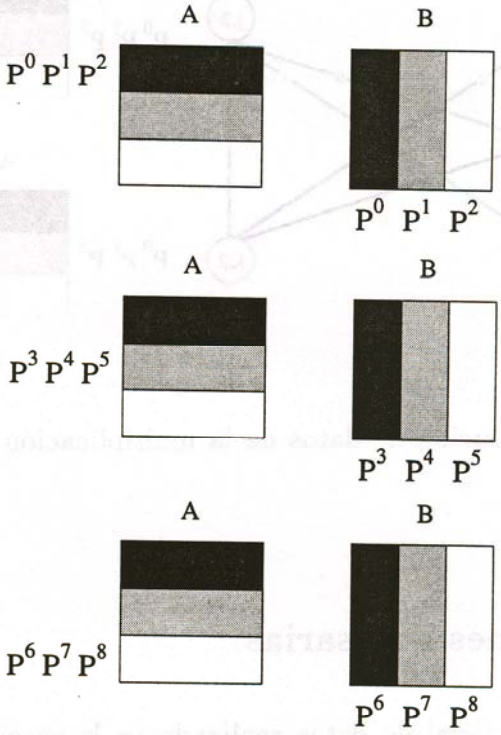


Figura 5.6: Distribución de datos de la multiplicación matriz-matriz ($n = 3$)

Teniendo en cuenta que en el GDM existen nueves "colores", existirán nueve procesadores (etiquetados del 0 al 8). Si analizamos el GDM, se observa que la fila 0 de A ($i = 0$), pertenece a tres procesadores: negro -P^0-, azul -P^1- y verde -P^2-. De

la misma forma, la columna 0 de B ($j = 0$) pertenece también a tres procesadores: negro -P^0-, magenta -P^3- y rojo -P^6-, y así sucesivamente. Sin embargo, si suponemos que en la arquitectura seleccionada sólo existen tres procesadores, se debe hacer una redistribución de datos, o lo que es lo mismo, hay que combinar "colores" en el GDM (distribución equitativa de la carga). De esta forma, se presenta un GDM como el de la figura 5.7, en la que también se muestra la distribución inicial de datos bajo estas condiciones.

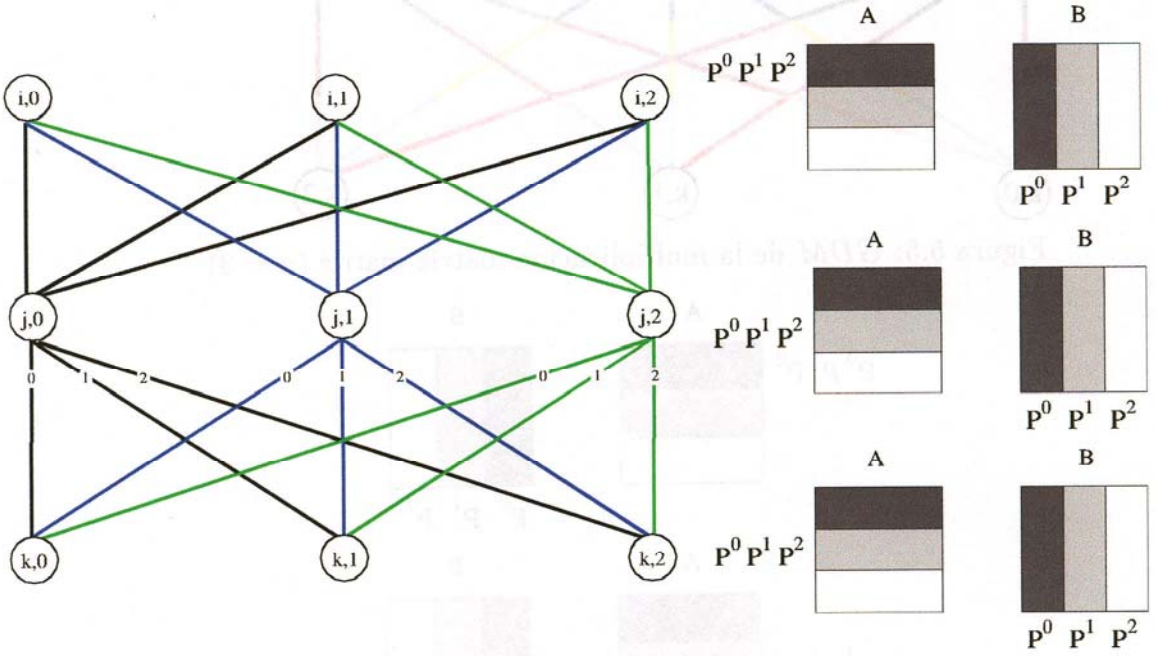


Figura 5.7: GDM y distribución de datos de la multiplicación matriz-matriz ($n = 3$, $np = 3$)

5.3.3 Comunicaciones necesarias

En base a la distribución inicial de datos realizada en la sección anterior, cada procesador almacena la matriz A completa y una columna de la matriz B para hacer el cálculo de la primera fila de la matriz C. Si generalizamos el problema de forma que a cada procesador le corresponda un bloque de $p = \frac{n}{np}$ columnas(filas) consecutivas de la matriz $B(C)$, cada procesador puede realizar el cálculo de una submatriz resultado final, de dimensión $n \times p$.

5.3.3.1 Primera redistribución de datos

Supongamos que la matriz A se vuelve a distribuir siguiendo un nuevo patrón, que evite la replicación de datos en la memoria de los distintos procesadores. En este caso, cada procesador almacena un bloque de p elementos de las matrices A y C (figura 5.8). De esta forma, cada procesador únicamente puede realizar el cálculo de la submatriz resultado de dimensión $p \times p$.

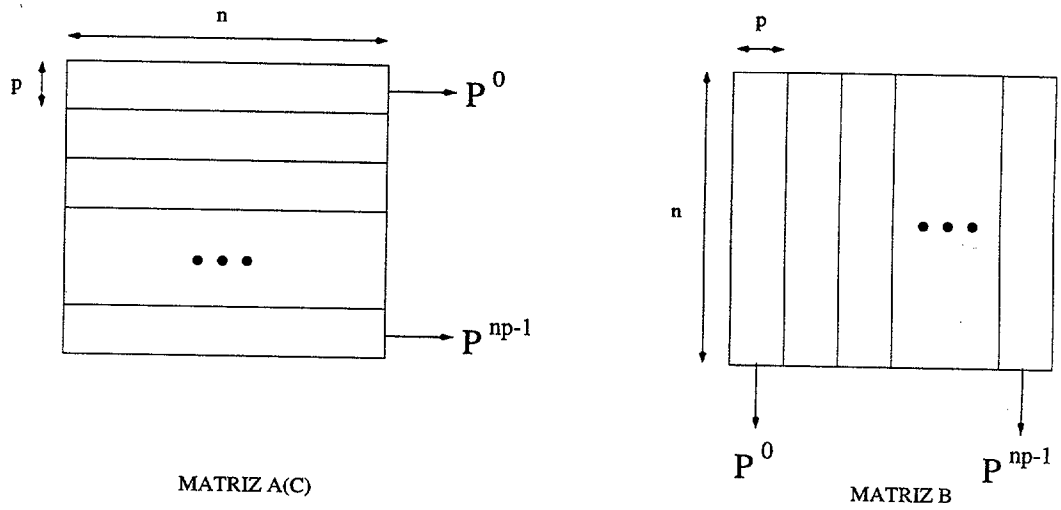


Figura 5.8: Distribución de datos de la multiplicación matriz-matriz en bloques de filas/columnas consecutivas

Sin embargo, los procesadores necesitan utilizar el resto de submatrices B, de dimensión $n \times p$ para el cálculo completo de los elementos de C. Debido a esto, se necesita comunicar los elementos de la matriz B entre los procesadores del sistema. Así, en un instante k, el procesador P^x realiza el cálculo de su submatriz (A_x) por la submatriz actual (B_k) y recibe del procesador P^y la submatriz siguiente B_{k+1} $(1 \leq x, y \leq np-1)$. Estas comunicaciones se realizan con primitivas no bloqueantes para asegurar el solapamiento de cálculos y comunicaciones.

5.3.3.2 Segunda redistribución de datos

Supongamos que las matrices A y B se redistribuyen de la forma indicada en la figura 5.9. En este caso, cada procesador (para el GDM de la figura 5.5) almacena una submatriz A y B de dimensión $p \times p$.

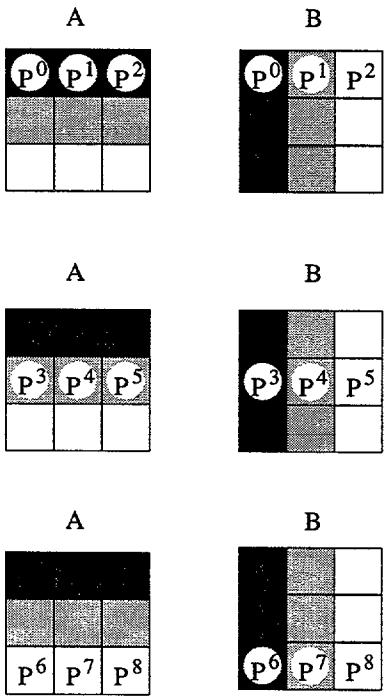


Figura 5.9: Distribución de datos de la multiplicación matriz-matriz en bloques de submatrices de dimensión $p \times p$

Utilizando esta distribución de datos, los procesadores necesitan comunicar submatrices A y B de dimensión $p \times p$. Si cada procesador de la arquitectura elegida tienen cuatro vecinos, los procesadores deben comunicarse la matriz A con los vecinos horizontales y la matriz B con los vecinos verticales, de esta forma, un procesador x, y envía su submatriz $A(B)$ al vecino $x, y-1(x-1, y)$ y recibe la submatriz $A(B)$ del vecino $x, y+1(x+1, y)$.

5.4 Resolución de sistemas de ecuaciones

Sean $x(x_j)$ y $b(b_i)$ dos vectores de n elementos y $A(A_{i,j})$ una matriz densa, de $n \times n$ elementos, donde $A(A_{i,j})$ y $b(b_i)$ son conocidos y $x(x_j)$ es un vector a calcular de forma que se cumpla que $A \times x = b$. La solución de sistemas de ecuaciones se puede realizar utilizando dos métodos diferentes: *directos* e *iterativos*. Los métodos directos se basan en transformar la matriz A en un producto de matrices que cumple unas ciertas condiciones (factorización LU, QR, etc.), mientras que los métodos iterativos

tienden a aproximarse a la solución final, después de aplicar una serie de pasos, que no son conocidos a priori [42]. En este apartado se analiza la resolución de sistemas de ecuaciones basados en el método de la factorización LU. Esta factorización genera dos matrices triangulares (L triangular inferior y U triangular superior tal que $A = L \times U$), que facilitan el cálculo del vector x. De esta forma, encontrar una solución al problema $A \times x = b$ queda reducido a la resolución de dos sistemas triangulares (ecuaciones 5.1 y 5.2). Es decir, si:

$$A \times x = b \quad \text{y} \quad A = L \times U \;\Rightarrow\; L \times U \times x = b$$

$$L \times y = b \tag{5.1}$$

$$U \times x = y \tag{5.2}$$

El vector solución del sistema 5.1 es el vector de términos independientes del sistema 5.2. Una vez resuelto el sistema 5.2 se tiene el vector solución final x del problema original $A \times x = b$.

5.4.1 Factorización LU

En esta sección se analiza la solución para la factorización LU.

5.4.1.1 Análisis del algoritmo secuencial. Generación del GDM

Un algoritmo secuencial que calcula la factorización de Doolittle $A = L \times U$ donde L es una triangular inferior unitaria ($L_{ii} = 1,\ i = 0..n-1$) y U es una triangular superior, es el siguiente:

$$\begin{array}{l} do\ k = 0, n-2, 1 \\ \quad do\ i = k+1, n-1, 1 \\ \quad\quad A_{i,k} = A_{i,k}/A_{k,k} \\ \quad\quad do\ j = k+1, n-1, 1 \\ \quad\quad\quad A_{i,j} = A_{i,j} - A_{i,k} \times A_{k,j} \\ \quad\quad enddo \\ \quad enddo \\ enddo \end{array}$$

Si $i > j$ entonces $A_{i,j}$ almacena los valores de la matriz $L(L_{i,j})$ y si $i \leq j$ entonces $A_{i,j}$ almacena los valores de la matriz $U(U_{i,j})$. En este algoritmo se especifica el orden de ejecución de los cálculos de los elementos de las matrices L y U. Así, los elementos de estas matrices deben ser obtenidos previamente al cálculo de los elementos internos de la matriz A (bucle más interno del algoritmo secuencial).

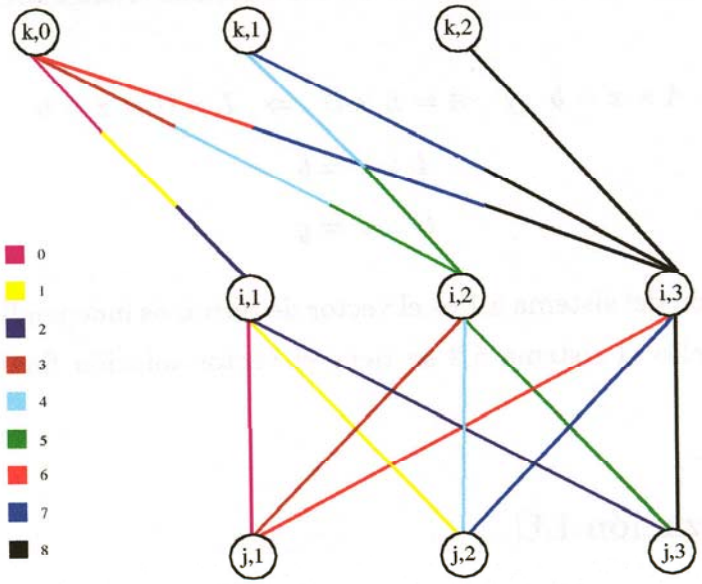


Figura 5.10: $GDM(d_1)$ de la factorización LU $(n = 4)$

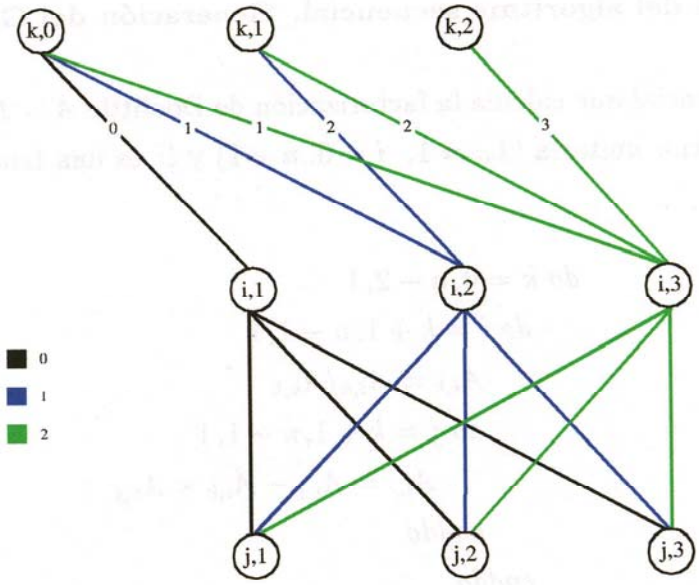


Figura 5.11: $GDM(d_2)$ de la factorización LU $(n = 4)$

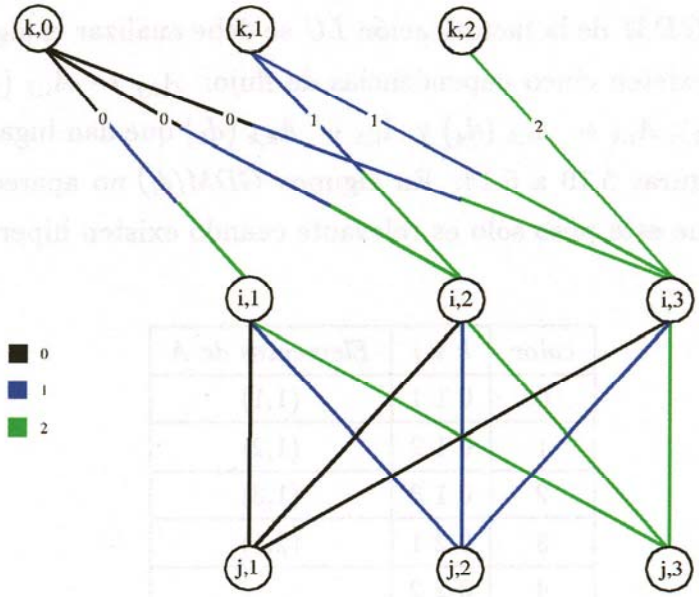


Figura 5.12: $GDM(d_3)$ de la factorización LU $(n = 4)$

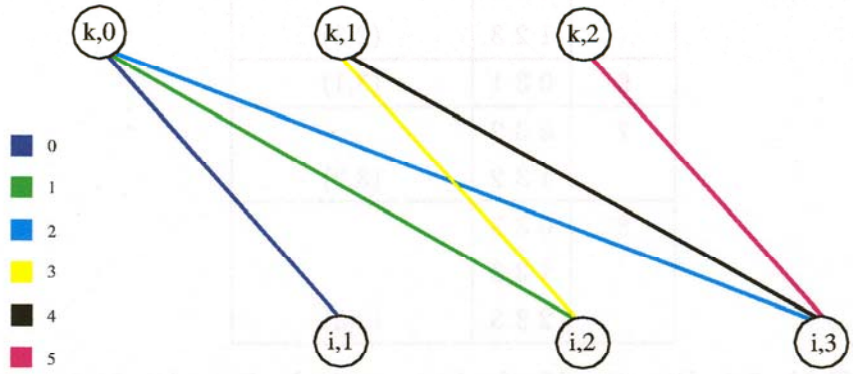


Figura 5.13: $GDM(d_4)$ de la factorización LU $(n = 4)$

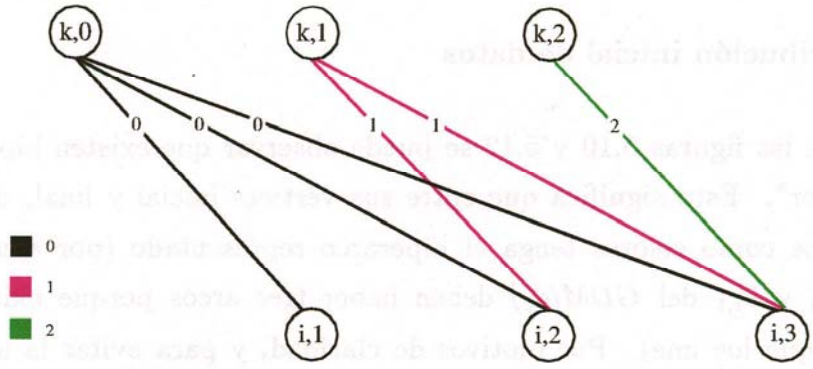


Figura 5.14: $GDM(d_5)$ de la factorización LU $(n = 4)$

Para generar el GDM de la factorización LU se debe analizar el algoritmo anterior. En este algoritmo existen cinco dependencias de flujo: $A_{i,j} \leftarrow A_{i,j}$ (d_1), $A_{i,j} \leftarrow A_{i,k}$ (d_2), $A_{i,j} \leftarrow A_{k,j}$ (d_3), $A_{i,k} \leftarrow A_{i,k}$ (d_4) y $A_{i,k} \leftarrow A_{k,k}$ (d_5) que dan lugar a los $GDM(d_i)$ mostrados en las figuras 5.10 a 5.14. En algunos $GDM(d_i)$ no aparece el "orden" de los hiperarcos porque este peso sólo es relevante cuando existen hiperarcos del mismo "color".

color	*k i j*	*Elementos de A*
0	0 1 1	(1,1)
1	0 1 2	(1,2)
2	0 1 3	(1,3)
3	0 2 1	(2,1)
4	0 2 2	
	1 2 2	(2,2)
5	0 2 3	
	1 2 3	(2,3)
6	0 3 1	(3,1)
7	0 3 2	
	1 3 2	(3,2)
8	0 3 3	
	1 3 3	
	2 3 3	(3,3)

Tabla 5.1: Distribución de datos según la dependencia d_1

5.4.1.2 Distribución inicial de datos

En los grafos de las figuras 5.10 y 5.12 se puede observar que existen hiperarcos con más de un "color". Esto significa que entre sus vértices inicial y final, deben haber tantos hiperarcos como colores tenga el hiperarco representado (por ejemplo, entre los vértices $V_{k,0}$ y $V_{i,1}$ del $GDM(d_1)$ deben haber tres arcos porque existe un arco de tres colores que los une). Por motivos de claridad, y para evitar la acumulación de hiperarcos que compliquen el estudio del grafo, se ha optado por representar un único arco con diferentes colores. Analizando el $GDM(d_1)$, podemos ver que existen nueve colores (numerados del 0 al 8) que están relacionados con los elementos de la

matriz A tal y como indica la tabla 5.1. Como cada "color" representa elementos no dependientes, las sentencias que actualizan los elementos del vector referenciados por distintos colores, pueden ser computadas por procesadores diferentes (etiquetados como $P^0..P^8$), de forma que los datos asociados al "color" i se almacenan en la memoria local del P^i.

color	*k i j*	*Elementos de A*
0	0 1 1	(1,1)
	0 1 2	(1,2)
	0 1 3	(1,3)
		(1,0)
1	0 2 1	(2,1)
	0 2 2	(2,2)
	0 2 3	(2,3)
	1 2 2	(2,2)
	1 2 3	(2,3)
		(2,0)
		(2,1)
2	0 3 1	(3,1)
	0 3 2	(3,2)
	0 3 3	(3,3)
	1 3 2	(3,2)
	1 3 3	(3,3)
	2 3 3	(3,3)
		(3,0)
		(3,1)
		(3,2)

Tabla 5.2: Distribución de datos según la dependencia d_2

Si observamos el *GDM(d_2)*, podemos ver que existen tres colores (numerados del 0 al 2) que están relacionados con los elementos de la matriz A tal y como indica la tabla 5.2. De los elementos de esta tabla, los últimos de cada "color" corresponden con los elementos $A_{i,k}$, y el resto con los elementos $A_{i,j}$. Si analizamos el algoritmo de la factorización LU secuencial vemos que para calcular los elementos de la fila i en una iteración dada k, se necesita el elemento situado en la columna *k-ésima*. Fijándonos

en el "color" 0 ($k = 0$), los elementos de la fila 1, necesitan para su actualización el elemento (1,0), ya que ese es el situado en la columna 0. Al igual que en el grafo anterior, los datos asociados al "color" i se almacenan en la memoria local del P^i.

color	*k i j*	*Elementos de A*
0	0 1 1	(1,1)
	0 2 1	(2,1)
	0 3 1	(3,1)
		(0,1)
1	0 1 2	(1,2)
	0 2 2	(2,2)
	0 3 2	(3,2)
	1 2 2	(2,2)
	1 3 2	(3,2)
		(0,2)
		(1,2)
2	0 1 3	(1,3)
	0 2 3	(2,3)
	0 3 3	(3,3)
	1 2 3	(2,3)
	1 3 3	(3,3)
	2 3 3	(3,3)
		(0,3)
		(1,3)
		(2,3)

Tabla 5.3: Distribución de datos según la dependencia d_3

Asimismo, en la figura 5.12, también existen tres colores formados por los elementos de la tabla 5.3. Al igual que en el caso anterior, de los elementos de la tabla, los últimos de cada "color" corresponden con los elementos $A_{k,j}$, y el resto con los elementos $A_{i,j}$. Analizando el algoritmo de la factorización LU vemos que para calcular los elementos de la columna j en una iteración dada k, se necesita el elemento situado en la fila *k-ésima*. Fijándonos en el "color" 0 ($k = 0$), los elementos de la columna 1, necesitan para su actualización el elemento (0,1), ya que ese es el situado en la fila 0.

En la figura 5.13 se observa el *GDM(d_4)*, en el que existen seis colores asociados

a los elementos indicados en la tabla 5.4. Al igual que en los casos anteriores, los datos asociados al "color" i se almacenan en la memoria local del P^i. Asimismo, en la figura 5.14 se representa el *GDM(d_5)* en el que existen tres colores asociados a los elementos indicados en la tabla 5.5. El último elemento de cada "color" corresponde con el elemento $A_{k,k}$, y el resto con los elementos $A_{i,k}$. Analizando el algoritmo de la factorización LU vemos que para calcular los elementos de la columna k en una iteración dada k, se necesita el elemento situado en la fila *k-ésima.* Fijándonos en el "color" 0 ($k = 0$), los elementos de la columna 0, necesitan para su actualización el elemento (0,0).

color	*k i*	*Elementos de A*
0	0 1	(1,0)
1	0 2	(2,0)
2	0 3	(3,0)
3	1 2	(2,1)
4	1 3	(3,1)
5	2 3	(3,2)

Tabla 5.4: Distribución de datos según la dependencia d_4

color	*k i*	*Elementos de A*
0	0 1	(1,0)
	0 2	(2,0)
	0 3	(3,0)
		(0,0)
1	1 2	(2,1)
	1 3	(3,1)
		(1,1)
2	2 3	(3,2)
		(2,2)

Tabla 5.5: Distribución de datos según la dependencia d_5

A partir de estos cinco *GDMs* se deduce la distribución inicial de datos en función de los colores asociados a cada elemento de la matriz A. En la figura 5.15(a) se muestra la matriz $A_{4\times4}$ y cómo se distribuye según los grafos: *GDM(d_1)*, *GDM(d_2)* y *GDM(d_3)*. Así, los elementos rodeados por un círculo negro representan los datos que

son distribuidos a procesadores diferentes según lo indicado en el $GDM(d_1)$ (tabla 5.1); los elementos rodeados por un borde rojo representan los datos que son distribuidos a procesadores diferentes según lo indicado por el $GDM(d_2)$ (tabla 5.2); y los elementos rodeados por un borde azul representan los datos que son distribuidos a procesadores diferentes según lo indicado por el $GDM(d_3)$ (tabla 5.3). En la figura 5.15(b) se muestra la matriz $A_{4\times4}$ y cómo se distribuye según los grafos: $GDM(d_4)$ y $GDM(d_5)$. Así, los elementos rodeados por un círculo amarillo representan los datos que son distribuidos a procesadores diferentes según lo indicado por el $GDM(d_4)$ (tabla 5.4); y los elementos rodeados por un borde marrón representan los datos que son distribuidos a procesadores diferentes según lo indicado por el $GDM(d_5)$ (tabla 5.5).

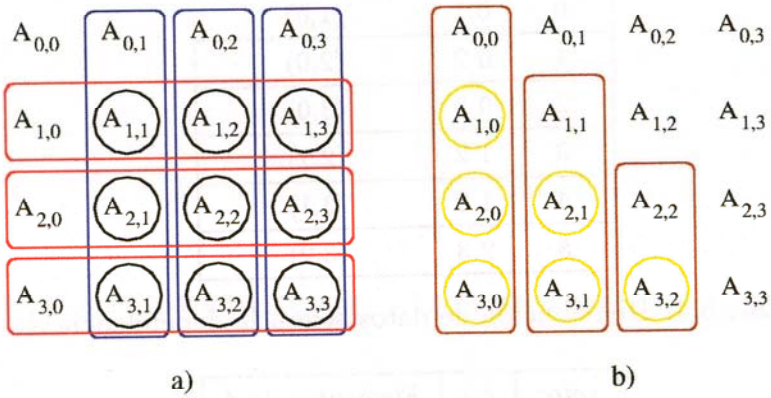


Figura 5.15: Distribución de datos de la factorización LU ($n = 4$): a) Según la sentencia interior. b) Según la sentencia exterior

5.4.1.3 Comunicaciones necesarias

Analizando la figura 5.15 se puede ver que los elementos de cada fila de la matriz deben almacenarse en el mismo procesador según la dependencia d_2, por tanto, al ser más restrictiva esta distribución que la marcada por las dependencias d_1 y d_4, se van a agrupar los datos de cada fila de la matriz en un procesador dado. Asimismo, los elementos de cada columna deben almacenarse en un mismo procesador según las dependencias d_3 y d_5. La distribución de datos queda finalmente según indica la figura 5.16(a). Con esta primera redistribución, se necesitan siete procesadores ($2 \times n - 1$ procesadores para el caso general), en los que existe replicación de datos. Así, cuando un procesador quiera actualizar el elemento $A_{i,j}$, necesita los elementos $A_{k,k}$, $A_{i,k}$ y

$A_{k,j}$. De los datos mencionados, el procesador que almacena una columna de la matriz es el propietario de los elementos $A_{i,j}$ y $A_{k,j}$; y el procesador que almacena una fila de la matriz tiene en su memoria los datos $A_{i,j}$ y $A_{i,k}$. En ambos casos, existe necesidad de comunicar elementos para realizar el cálculo ($A_{k,k}$ y el correspondiente $A_{i,k}$ o $A_{k,j}$ dependiendo del caso).

Para reducir el número de comunicaciones, se puede modificar la distribución anterior de forma que no exista replicación de datos. Esto se consigue distribuyendo los elementos por filas (columnas) o bloques de ellas (figura 5.16(b)). Si se realiza una distribución por filas (columnas), en la iteración k, el procesador propietario de la fila (columna) k debe comunicarla al resto de procesadores, una vez actualizada. De esta forma, el resto de procesadores dipondrá de los elementos $A_{k,j}$, $A_{k,k}$ ($A_{i,k}$, $A_{k,k}$) necesarios según la distribución realizada. Aplicando esta nueva distribución, el número de procesadores no es dependiente de la dimensión de la matriz.

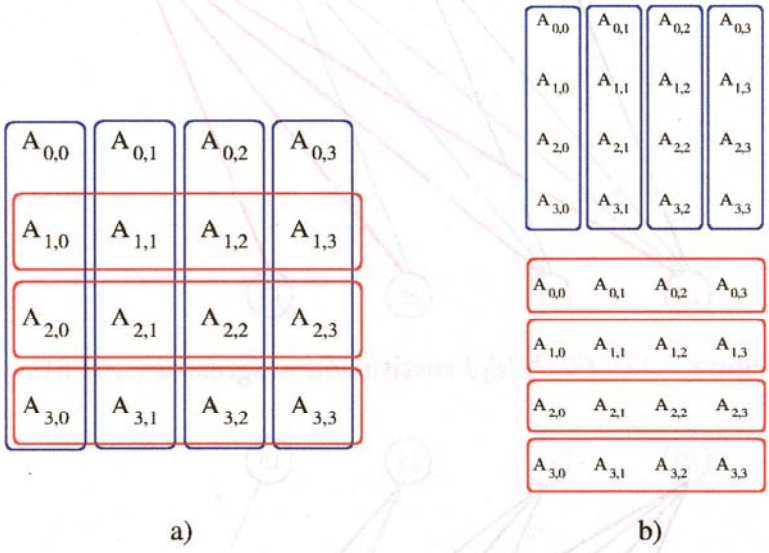


Figura 5.16: Distribución de datos de la factorización LU ($n = 4$): a) Primera redistribución. b) Segunda redistribución

5.4.2 Sistemas triangulares

En esta sección se presenta la solución para los sistemas de ecuaciones triangulares.

5.4.2.1 Análisis del algoritmo secuencial. Generación del GDM

Los algoritmos de sustitución progresiva y regresiva, suponiendo que en la matriz A se encuentran almacenadas las matrices L y U, resultado de la factorización, son:

```
do i = 0, n − 1, 1
    y_i = b_i
    do j = i − 1, 0, −1
        y_i = y_i − A_{i,j} × y_j
    enddo
enddo
```

```
do i = n − 1, 0, −1
    x_i = y_i / A_{i,i}
    do j = i + 1, n − 1, 1
        x_i = x_i − (A_{i,j} / A_{i,i}) × x_j
    enddo
enddo
```

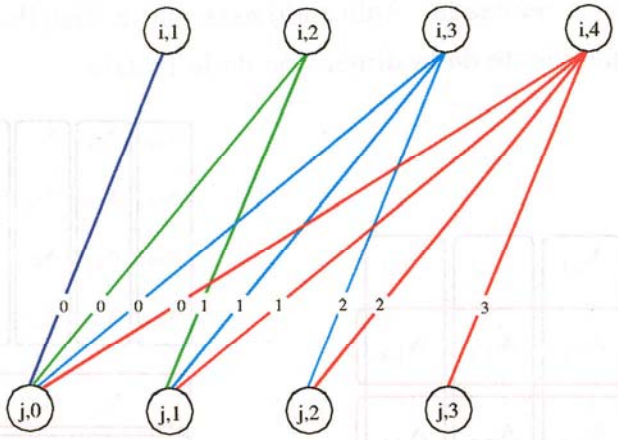


Figura 5.17: $GDM(d_1)$ sustitución progresiva $(n = 5)$

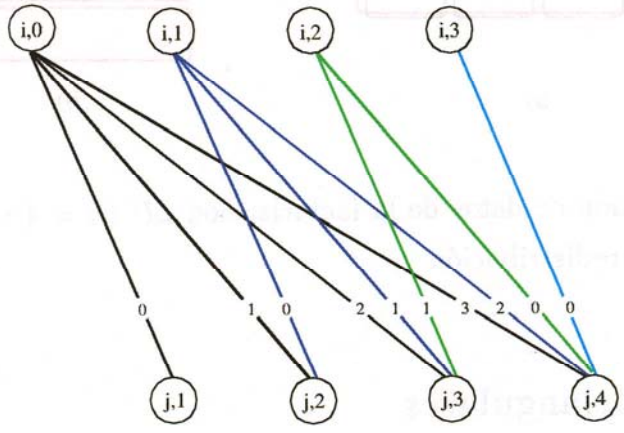


Figura 5.18: $GDM(d_1)$ sustitución regresiva $(n = 5)$

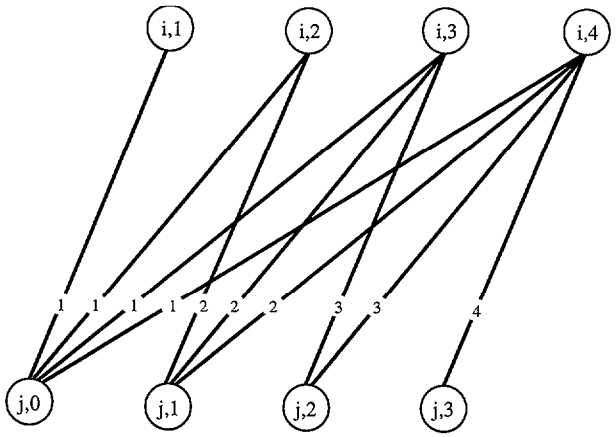


Figura 5.19: *GDM(d_2)* sustitución progresiva ($n = 5$)

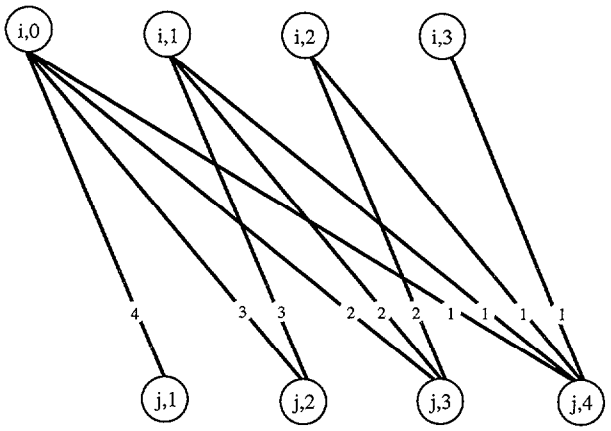


Figura 5.20: *GDM(d_2)* sustitución regresiva ($n = 5$)

Analizando los algoritmos anteriores, se ve que existen dependencias del tipo $y_i \leftarrow y_i(d_1)$ y $y_i \leftarrow y_j(d_2)$. Los *GDMs* de la dependencia d_1 generados para ambos algoritmos se presentan en las figuras 5.17 (sustitución progresiva) y 5.18 (sustitución regresiva) y los de la dependencia d_2 en las figuras 5.19 (sustitución progresiva) y 5.20 (sustitución regresiva).

5.4.2.2 Distribución inicial de datos

Analizando la sustitución progresiva, el *GDM(d_1)* presenta cuatro colores diferentes, ya que cada valor y_i $(1 \leq i \leq n-1)$ se puede actualizar independientemente en un procesador distinto (en el procesador P^i), sin embargo, si observamos el *GDM(d_2)* sólo existe un color. Por tanto, la restricción viene impuesta por la dependencia d_2, que

indica que un único procesador, en principio, debe realizar todo el cálculo. Así, cada valor y_i depende de los valores anteriores y_j $(0 \leq j \leq i-1)$ y del valor previo de y_i.

Debido al hecho de que todos los elementos del vector y dependen del y_0, todos los hiperarcos, sobre los que estos datos se proyectan, son del mismo "color" al del hiperarco asociado al elemento y_0. Sin embargo, analizando el algoritmo se puede ver que la actualización de y_i no tiene porque hacerse secuencialmente. Así, se puede actualizar y_i $(1 \leq i \leq n-1)$ respecto a y_0 en paralelo. A continuación, se pueden actualizar y_i $(2 \leq i \leq n-1)$ respecto a y_1 también en paralelo, e igual con el resto de elementos del vector y. Esto indica que entre los distintos elementos del vector y las dependencias no son muy restrictivas, ya que cierto cálculo puede realizarse, o bien en procesadores diferentes (en un multicomputador), siempre que existan comunicaciones de los datos necesarios; o bien en distintos *threads* de un monoprocesador. Este hecho viene indicado por el "orden" de los hiperarcos de los grafos. Según se indica en la sección 3.4.2.3, si dos hiperarcos tienen el mismo "color" y "orden", los cálculos que se realizan sobre los datos asociados con ellos se pueden hacer a la vez. En el GDM de la figura 5.19 vemos que existen hiperarcos con el mismo "orden" sobre los que se proyectan los elementos del vector y. La explicación anterior puede extenderse al caso de la sustitución regresiva.

La distribución de datos se va a realizar suponiendo la existencia de un multicomputador con np procesadores, de forma que cada uno de ellos almacena $\frac{n}{np}$ filas de la matriz A y de los vectores $b(y)$ e $y(x)$.

5.4.2.3 Comunicaciones necesarias

Partiendo de la distribución comentada, el procesador P^i necesita recibir datos previamente calculados del procesador P^j $(0 \leq j \leq i-1)$, para actualizar sus valores. Supongamos cinco procesadores diferentes, tal y como indica el $GDM(d_1)$ más uno que almacena al elemento y_0. En este caso se necesitan comunicaciones de datos del tipo *broadcast* para que todos los procesadores tengan el dato necesario, antes de su utilización. En la figura 5.21 se presenta la comunicación del elemento y_0. En general, los elementos y_i y $A_{i,j}$ se deben almacenar en el procesador P^i (distribución por filas). En las distintas iteraciones de j, el procesador propietario de y_j debe comunicarlo median-

te un *broadcast* al resto de procesadores, antes de que éstos actualicen sus datos. La explicación anterior puede extenderse al caso de la sustitución regresiva.

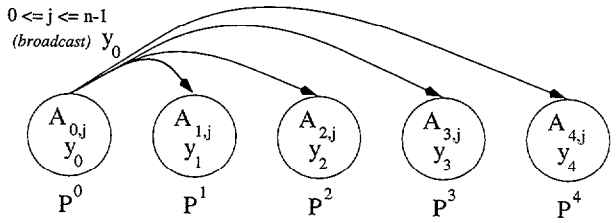


Figura 5.21: Ejemplo de movimiento de datos en la sustitución progresiva ($n = 5$)

5.5 Factorización LU dispersa

La resolución de sistemas de ecuaciones $A \times x = b$, donde A es una matriz dispersa y no simétrica de $n \times n$ elementos y b y x son vectores densos de n elementos, es un problema ampliamente utilizado en aplicaciones científicas. Entre estas aplicaciones se pueden destacar:

- El modelado físico de estructuras civiles y militares, semiconductores, recursos geológicos etc., en los que los resultados de los modelos matemáticos están formados por sistemas lineales dispersos con un gran número de ecuaciones y variables.
- Programación lineal y optimización en la que el cálculo de sistemas lineales dispersos es una parte importante.
- Indexación y recopilación de documentos en los que se utilizan matrices dispersas para encontrar la relación existente entre términos, similitud entre palabras, etc.

Cuando A es una matriz dispersa y no simétrica, la estructura de las matrices L y U dependen tanto de la forma dispersa de la matriz como de los valores numéricos de A. Así, la solución de un sistema disperso, basada en la factorización de la matriz A, se divide en cuatro pasos [32]: a) análisis de la estructura dispersa; b) factorización simbólica; c) factorización numérica y d) solución del sistema de ecuaciones. El primer paso consiste en encontrar el orden de permutación correcto para que se produzca o aparezcan el menor número de nuevos elementos distintos de 0, es decir, que el

rellenado de ceros (*fill-in*) sea el menor posible. Este, al ser un problema NP-completo, necesita ser resuelto mediante heurísticas con las cuales se logran buenos resultados [63] (ordenaciones de mínimo grado, en la que se elige el pivote de la diagonal principal en aquella fila que tiene menor número de elementos distintos de cero, heurísticas basadas en particionado de grafos o aproximaciones híbridas, etc.). En el segundo paso se realiza una factorización simbólica para determinar los elementos distintos de cero que aparecen en la factorización numérica. La aparición de nuevos elementos lleva consigo un mayor número de cálculos en punto flotante y necesita mayor cantidad de memoria para su almacenamiento. En el tercer paso se realiza el cálculo de la factorización numérica (cálculo de las matrices L y U) y en el último se calculan los sistemas de ecuaciones generados con el objetivo de encontrar el vector resultado x. En los casos en los que se deben considerar los valores numéricos para la elección del pivote, los tres primeros pasos se combinan dentro de una fase de análisis-factorización.

La factorización LU de una matriz dispersa se puede desarrollar siguiendo diferentes estrategias. Algunas de ellas se basan en la utilización de supernodos [28] o en aproximaciones frontales [27] que se caracterizan porque en ambas se realizan cálculos sobre porciones densas de matrices dispersas. Otras se basan en realizar modificaciones, para la iteración *k-ésima*, sobre la columna (fila) *k-ésima* utilizando las columnas (filas) previamente calculadas (*left-looking* LU o *fan-in*) o utilizando la columna (fila) *k-ésima*, una vez calculada, para actualizar las columnas (filas) posteriores (*right-looking* LU o *fan-out*) [5].

Los métodos de submatrices usan, por lo general, un pivote elegido siguiendo los criterios de Markowitz [68], en los cuales el pivote se elige en aquella porción de matriz aún no resuelta, para preservar la estabilidad numérica y la estructura dispersa. Los métodos basados en columnas (filas) utilizan el pivoteo parcial ordinario. El pivote se elige de la columna (fila) *k-ésima* de acuerdo sólo a consideraciones numéricas, aunque las columnas (filas) pueden ordenarse para preservar la estructura dispersa.

5.5.1 Análisis del algoritmo secuencial

De las aproximaciones anteriores, se ha paralelizado el método *right-looking* LU. Un algoritmo secuencial que utiliza esta aproximación es el siguiente:

```
do k=0,n-2,1
   pivot=k
   Búsqueda del pivote (k,piv)
   Intercambiar las filas (k,piv)
   do j=k,n-1,1        /*calcular la factorización*/
      U_{k,j} = A_{k,j}
   enddo
   do i=k+1,n-1,1
      L_{i,k} = L_{i,k}/A_{k,k}
      do j=k+1,n-1,1
         A_{i,j} = A_{i,j} - L_{i,k} × U_{k,j}
      enddo
   enddo
enddo
```

En este algoritmo vemos como se elige el pivote de la columna *k-ésima* (valor absoluto máximo de los elementos de esa columna) y a continuación, se intercambian las filas pivote y *k-ésima*. Una vez realizados estos pasos, se obtiene una nueva fila de la matriz U y una columna de la matriz L. Con estos datos se actualizan los valores de la submatriz interior $(A_{k+1..n-1 \times k+1..n-1})$. Estas operaciones se vuelven a realizar hasta que la submatriz interior sea de dimensión 1×1. En la figura 5.22 se observa un ejemplo de matriz dispersa y las dependencias en el EI, donde los arcos en las direcciones i y j representan los datos de las matrices U y L respectivamente, que se necesitan en una iteración dada, y los arcos en la dirección k representan los datos que se necesitan entre iteraciones.

En el algoritmo secuencial, para cada iteración de la variable k, se calcula el pivote de la columna *k-ésima*. A continuación se intercambia la fila k con la fila que contiene al pivote (fila pivote) y una vez terminado el intercambio se realiza el cálculo de los elementos de L y U en esa iteración y se recalculan los elementos de A, a ser utilizados en las siguientes iteraciones.

Si suponemos distintas matrices A, cada una de ellas presenta una representación de las dependencias en el EI diferente, debido a la dispersidad de la matriz y a que, por lo general, la fila k y pivote no coinciden para los distintos valores de k. Sin embargo, las

dependencias de datos existentes en la factorización LU siguen una misma estructura, es decir, en las direcciones i y j existen dependencias de flujo en una misma iteración y en la dirección k hay dependencias de flujo entre iteraciones [9]. Según esto, se puede analizar el GDM de la factorización LU densa y extender sus resultados a la factorización LU dispersa.

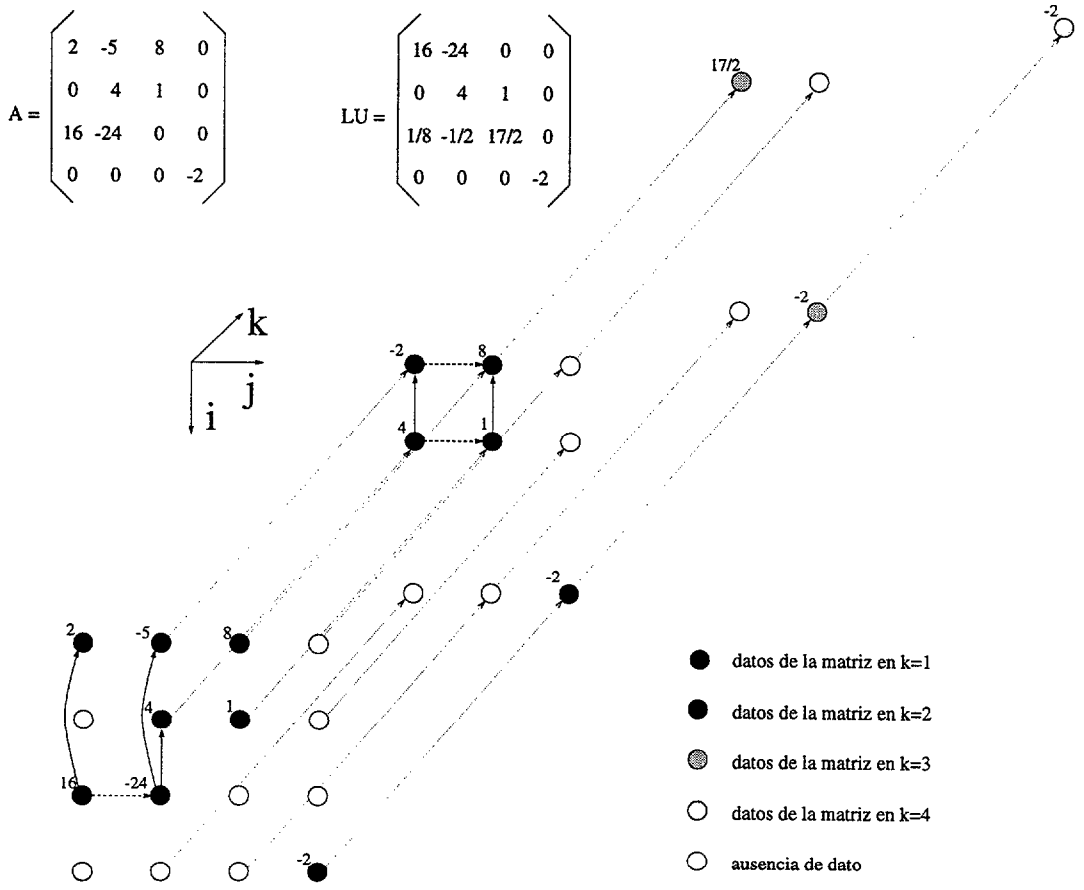


Figura 5.22: Ejemplo de matriz dispersa. Representación de las dependencias en el EI

5.5.2 Distribución inicial de datos

La distribución inicial de datos se hace en base al GDM siguiendo los mismos criterios que en la factorización LU densa. Sin embargo, en el caso de la LU dispersa, hay que analizar cómo se deben almacenar los datos, ya que el almacenamiento de este tipo de matrices necesita de estructuras complejas [5], [88]. La estructura más óptima debe ser aquella que permita un acceso rápido a los datos que almacena y que consiga

que el algoritmo paralelo diseñado para manejarla sea el más simple y eficiente. Sin embargo, estos dos objetivos son, por lo general, contrapuestos, ya que tenemos un acceso rápido a los datos cuando estos están almacenados en posiciones conocidas (estructuras estáticas), hecho no corriente si estamos tratando con un algoritmo paralelo que actualiza, no sólo el valor de los datos, sino también el número de estos. Ante este último caso, estos algoritmos paralelos tienen que utilizar estructuras dinámicas (no estáticas) para almacenar los datos, con lo que la velocidad del algoritmo decrece y por tanto, su eficiencia. La estructura elegida en nuestro caso, almacena la matriz en posiciones consecutivas de memoria (ver ejemplo en figura 5.23), evitando así almacenar la posición del siguiente elemento. Hay que tener en cuenta, que en la factorización LU pueden aparecer elementos en posiciones de la matriz que antes estaban vacías o pueden desaparecer elementos. Debido a esto, en cada iteración del algoritmo hay que recolocar la matriz en memoria.

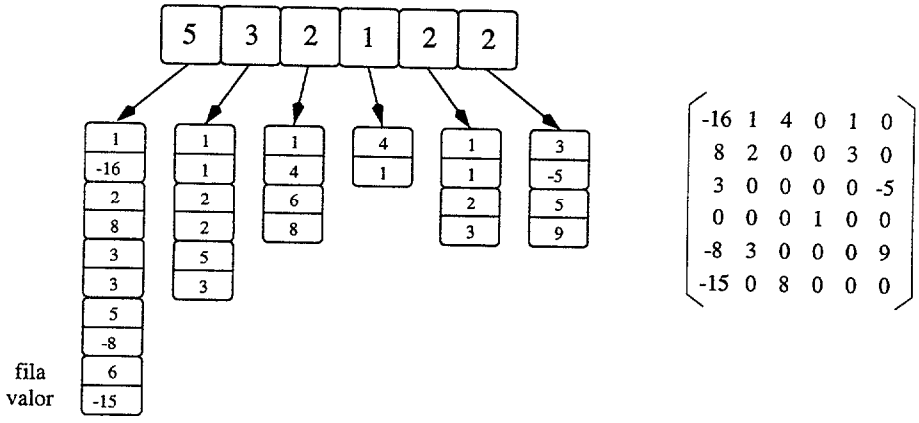


Figura 5.23: Ejemplo de estructura de almacenamiento

5.5.3 Comunicaciones necesarias

Para reducir el número de comunicaciones, se distribuyen los datos en bloques de columnas (según la matriz marcada en azul en la figura 5.16(b)), de forma que el procesador propietario de la columna k debe comunicarla al resto de procesadores, una vez actualizada. Así, el resto de procesadores dispone de los elementos $A_{i,k}, A_{k,k}$ necesarios según la distribución realizada. Aplicando esta distribución, el número de procesadores no es dependiente de la dimensión de la matriz.

Capítulo 6

Resultados experimentales y aplicaciones a problemas de ingeniería

6.1 Introducción

Este capítulo presenta los desarrollos de los problemas propuestos en capítulos anteriores, después de analizar los algoritmos secuenciales y generar los grafos de dependencias modificados, que establecen las distribuciones de datos. La mayoría de estos problemas se han resuelto sobre el multicomputador IBM-SP2, utilizando lenguaje de programación C y la biblioteca de paso de mensajes MPI. Cada aplicación consta de la generación de los archivos iniciales, la ejecución del programa paralelo y la ejecución del programa secuencial sobre uno de los procesadores del sistema y en las mismas condiciones.

Asimismo, se presentan dos desarrollos prácticos de Ingeniería de Telecomunicación: el cálculo del método de los momentos para analizar estructuras microtiras y el cálculo de la respuesta impulsional en un canal de infrarrojos.

6.2 Multiplicación matriz-vector

La distribución de datos de la matriz A se realiza por filas o bloques de filas. Si existen n filas en la matriz y n procesadores, cada uno de ellos actualiza una fila de esta, reali-

zando el producto de la matriz por el vector. Sin embargo, si el número de procesadores (np) es menor a n, la matriz se reparte en bloques de $p = \frac{n}{np}$ filas consecutivas. La misma distribución se realiza con respecto al vector b (bloques de p filas consecutivas). Partiendo de esta distribución de datos, en cada iteración, cada procesador del sistema computador debe calcular el producto de una submatriz A_x^z de dimensión $p \times n$ por un subvector b^z de dimensión $p \times 1$ y a la vez comunicar los datos para la siguiente iteración. El algoritmo paralelo para un procesador genérico, considerando np procesadores, se muestra en la tabla 6.1. En la figura 6.1 se muestran los resultados obtenidos en las simulaciones sobre un IBM-SP2 (para cinco -un procesador repartidor de datos y cuatro procesadores calculando el resultado- y diez procesadores -un repartidor y nueve calculando-), donde las comunicaciones son de tipo *broadcast.*

comunicar b^0
do z=1,np-1,1
  par
    calcular $c_i = c_i + A_{i,j} \times b^{z-1}$
    comunicar b^z
  endpar
enddo
$c_i = c_i + A_{i,j} \times b^{np-1}$

Tabla 6.1: Algoritmo paralelo de la multiplicación matriz-vector

Para que exista solapamiento de cálculos y comunicaciones, debe cumplirse la ecuación $\beta + L \times \tau_{com} \leq l \times \tau_{cal}$ (siendo L el número de elementos a comunicar y l el número de elementos a calcular). Así, el tiempo de cálculo medido en el IBM-SP2 (τ_{cal}) al realizar una operación de multiplicación más una de adición, es de aproximadamente 3 $\mu sg.$, y los valores de β y τ_{com} en función de L son [72]:

Tamaño del mensaje L	β	τ_{com}
$0 < L < 4Kbytes$	47	0.0254
$4Kbytes < L < 32Kbytes$	131	0.0195
$L > 32Kbytes$	390	0.0115

Si se realizan los cálculos del $p \times p$ elementos y las comunicaciones de p elementos en cada caso (tamaño máximo de elementos calcular/comunicar), se cumple la ecuación

anterior, y por tanto, el solapamiento es total.

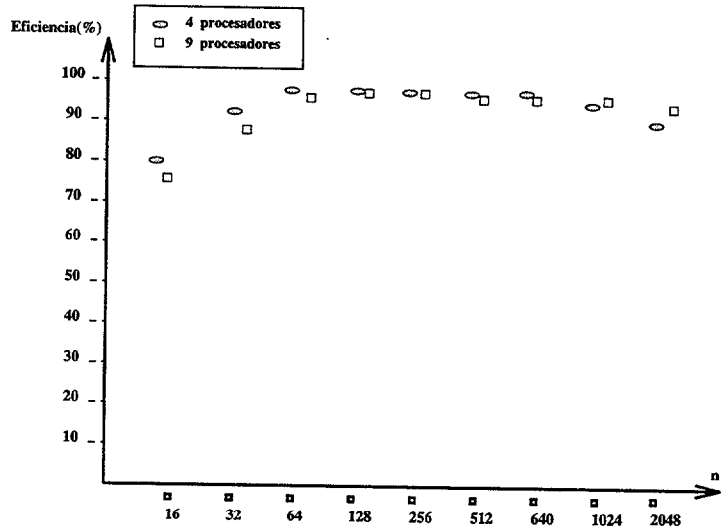


Figura 6.1: Multiplicación matriz-vector: resultados de la eficiencia

6.3 Multiplicación matriz-matriz

La multiplicación matriz-matriz se ha implementado realizando dos distribuciones iniciales de datos. En la primera distribución, la matriz $A(C)$ se distribuye en bloques de $p \times n$ filas consecutivas ($p = \frac{n}{np}$) y la matriz B en bloques de $n \times p$ columnas consecutivas (figura 5.8). En la segunda distribución, las matrices se dividen en submatrices de dimensión $p \times p$. Si suponemos una malla de procesadores de dimensión $np1 \times np2$, tal que $np = np1 \times np2$, al procesador (x, y) le corresponde las submatrices x, y (figura 5.9).

Distribuyendo los datos como se indica, se generan los algoritmos paralelos, para un procesador genérico, mostrados en las tablas 6.2 y 6.3. Estos algoritmos han sido implementados sobre un IBM-SP2, utilizando comunicaciones de tipo *broadcast* (primera distribución) y punto a punto (segunda distribución). En la figura 6.2 se muestran los resultados obtenidos en la simulación para la primera distribución de datos. En todos los casos existe un procesador repartidor de datos y $np-1$ procesadores de cálculo. En las figuras 6.3 y 6.4 se presentan los resultados para la segunda distribución de datos, utilizando una topología toroidal creada por la biblioteca MPI y suponiendo un toro

virtual, respectivamente. En [96] se presentan medidas similares sobre un multicomputador basado en *transputer* T800 [101].

```
comunicar B^0
do z=1,np-1,1
   par
      calcular C_{i,j} = C_{i,j} + A_{i,j} × B^{z-1}
      comunicar B^z
   endpar
enddo
calcular C_{i,j} = C_{i,j} + A_{i,j} × B^{np-1}
```

Tabla 6.2: Algoritmo paralelo de la multiplicación matriz-matriz: primera distribución

```
do z=0,np-2,1
   par
      enviar(A^z,←), enviar(B^z,↑), recibir(A^{z+1},→), recibir(B^{z+1},↓)
      calcular C^z = C^z + A^z × B^z
   endpar
enddo
calcular C^{np-1} = C^{np-1} + A^{np-1} × B^{np-1}
```

Tabla 6.3: Algoritmo paralelo de la multiplicación matriz-matriz: segunda distribución

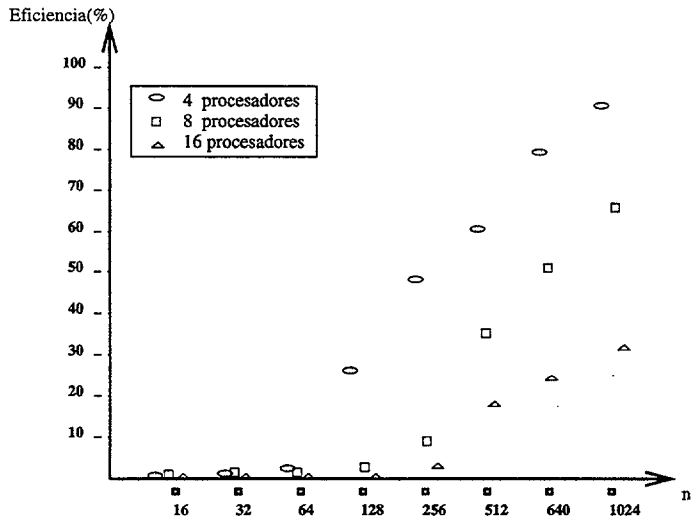


Figura 6.2: Multiplicación matriz-matriz: primera distribución

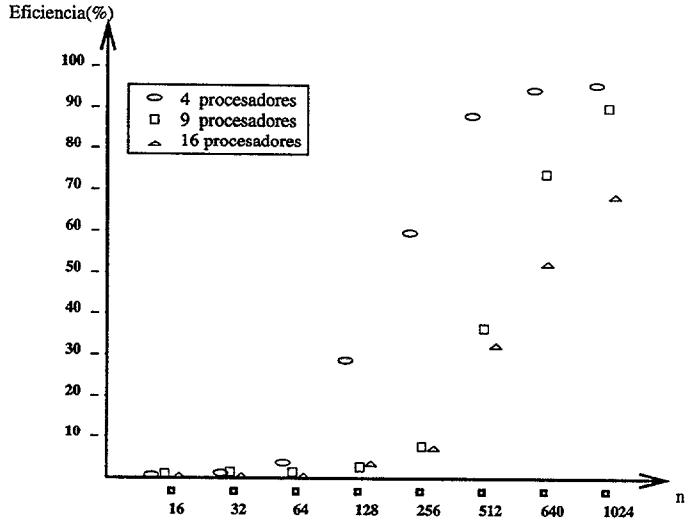


Figura 6.3: Multiplicación matriz-matriz: segunda distribución (topología MPI)

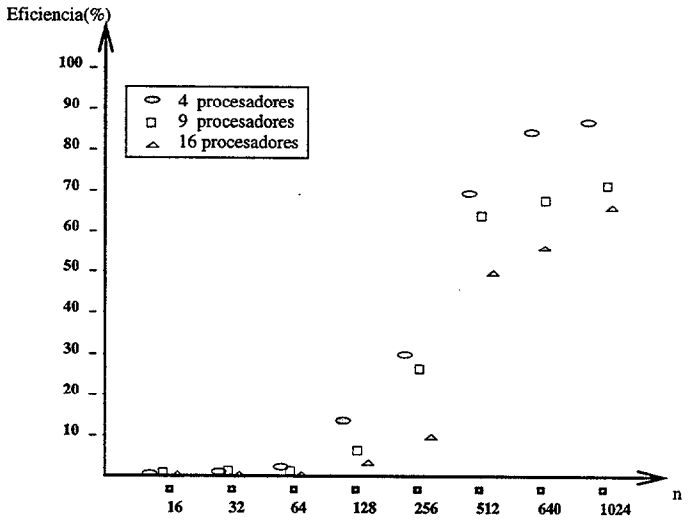


Figura 6.4: Multiplicación matriz-matriz: sgunda distribución (toro virtual)

La primera distribución (figura 6.2) presenta una eficiencia menor que la segunda distribución (figura 6.3) ya que los cálculos y las comunicaciones están mejor solapadas en esta última y los procesadores no tienen que permanecer inactivos esperando por los datos. Asimismo, utilizando las topologías de MPI para construir el toroide en vez de comunicar los datos con los cuatro vecinos sin topología existente, hace que la eficiencia mejore, ya que las topologías de MPI proveen un subespacio más seguro para las comunicaciones y permiten mayores generalidades. Si se realizan los cálculos de $p^2 \times n$ elementos en la primera distribución (p^3 elementos en la segunda distribución)

y las comunicaciones de $p \times n$ elementos ($p \times p$ en la segunda distribución) existe un solapamiento total. El número de elementos a comunicar para las dos distribuciones de datos es menor en el segundo caso, ya que se realizan cuatro comunicaciones en paralelo.

6.4 Factorización LU

En la factorización LU, la matriz A se distribuye en bloques de $\frac{n}{np}$ filas (columnas), que a su vez se dividen en grupos de tamaño g, siendo g divisor entero de $\frac{n}{np}$ (en caso contrario se añaden a la matriz original las filas y columnas necesarias para que lo sea). Los grupos de g filas (columnas) consecutivas son repartidas de forma cíclica y ordenada entre los procesadores del multicomputador. De esta manera el procesador genérico p (siendo $p = 0, ..np-1$) recibe la fila (columna) i que cumple la condición:

$$(i \text{ div } g) \bmod np = p \quad \forall i = 0..n-1$$

Partiendo de estas distrituciones se generan dos algoritmos paralelos. En la tabla 6.4 se muestra el algoritmo cuando la distribución de datos es por filas (el algoritmo paralelo con distribución por columnas es similar, pero comunicándose los subbloques L).

En la iteración k_1, el procesador propietario de la fila $A_{k_1,j}$ (o columna A_{i,k_1}) ha de comunicarla al resto de procesadores para que éstos puedan realizar sus cálculos. Debido a la distribución de datos realizada, se pueden agrupar los cálculos con lo que se evita el cómputo fila a fila (columna a columna) que provoca $\frac{n}{g}$ comunicaciones de tipo *broadcast*, en lugar de las n que hacen falta. Asimismo, con esta distribución de datos se consigue aumentar el paralelismo ya que el procesador propietario de un grupo de filas (columnas) puede comunicar este grupo precalculado y en paralelo actualizar el resto de los valores de A.

En las figuras 6.5 y 6.6 se pueden observar las eficiencias obtenidas en la factorización LU para las configuraciones probadas sobre un IBM-SP2 (un procesador repartidos y $np-1$ procesadores calculando), y distintos tamaños de matrices. En el peor caso

se realizan $(n - g) \times n$ cálculos y se comunican $g \times n$ elementos. Si existe un mayor número de procesadores, el número de comunicaciones aumenta con respecto al caso de un menor número de procesadores (al ser el *broadcast* una primitiva bloqueante, se sustituye por comunicaciones punto a punto no bloqueantes, con todos los procesadores). En las figuras 6.5 y 6.6 se observa que la eficiencia no alcanza la saturación para las dimensiones de matrices probadas. En [81] se presentan medidas similares sobre un multicomputador basado en *transputer* T800.

```
x=0
if z=0 then
    calcular bloque U^x
endif
comunicar bloque U^x
do k=0,n-1,g
    if ((k+g)/g mod np) = z then
        calcular L^x; calcular A^x; calcular U^x+1
        par
            comunicar bloque U^x+1
            calcular L; calcular A
        endpar
    else
        par
            comunicar bloque U^x+1
            calcular L; calcular A
        endpar
    endif
    x=x+1
enddo
```

Tabla 6.4: Algoritmo de la factorización LU con distribución por filas

6.4.1 Modificación: factorización con pivoteo parcial

El principal defecto de la factorización sin pivoteo es que sólamente puede ser aplicada a matrices estrictamente de diagonal dominante. En las aplicaciones científicas es muy

usual encontrar matrices con algún elemento de la diagonal principal igual a "cero", con lo que el algoritmo secuencial explicado en la sección anterior no puede ser aplicado. Así mismo, el cálculo de los factores L y U produce errores de redondeo que perjudican el cálculo de la resolución de los sistemas triangulares siguientes. Estos errores se agravan si los elementos $A_{k,k}$ son relativamente pequeños en comparación con el resto [42]. El proceso de factorización es equivalente al de la factorización sin pivoteo salvo que en cada iteración se localiza la fila (fila pivote) donde se encuentra el máximo valor absoluto de la columna (pivote) y se intercambia de posición con la fila que corresponde en una ejecución sin pivoteo (fila *k-ésima*).

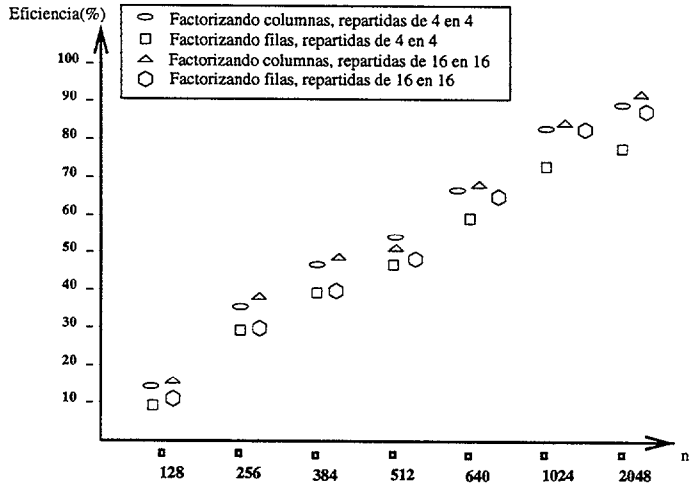


Figura 6.5: Factorización sin pivoteo (4 procesadores)

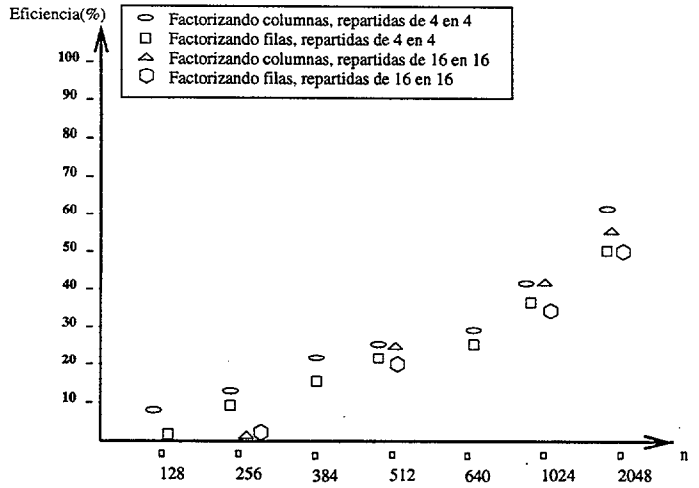


Figura 6.6: Factorización sin pivoteo (16 procesadores)

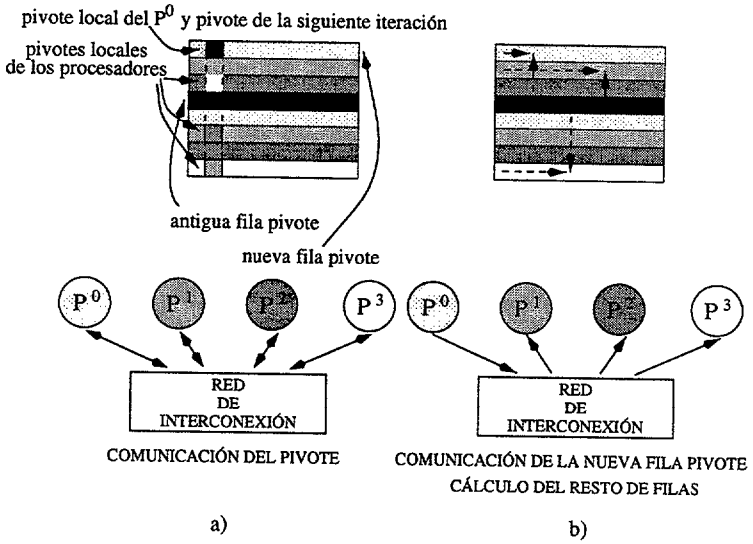


Figura 6.7: a) Comunicación del pivote. b) Solapamiento de cálculos y comunicaciones

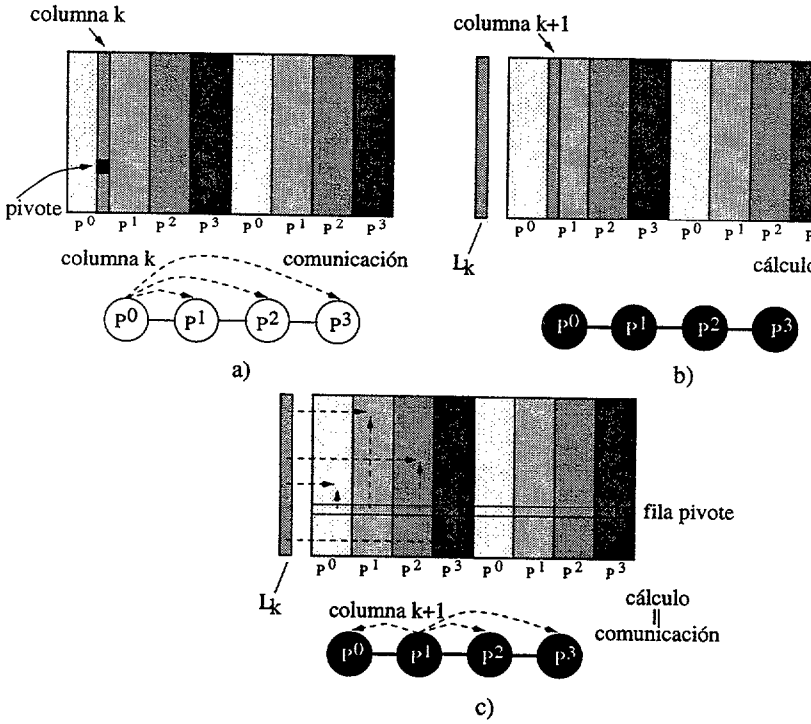


Figura 6.8: a) Cálculo del pivote y comunicación de la columna pivote. b) Cálculo de L_k. c) Comunicación solapada con el cálculo del resto A

El proceso de búsqueda del pivote, añadido al algoritmo secuencial, no puede ser paralelizado ya que sólamente hay un procesador propietario del pivote. Si existe una

distribución de datos por filas, los procesadores deben comunicarse hasta encontrar el procesador pivote (comunicaciones punto a punto). Este, una vez actualizada la fila pivote, debe comunicarla al resto de procesadores (figura 6.7) mediante comunicaciones de tipo *broadcast*. Con esta distribución se tiene el problema de que los cálculos no se pueden realizar en bloques de filas consecutivas, ya que la siguiente fila pivote no tiene porque ser la siguiente fila física. En caso de una distribución por columnas, el paso de búsqueda del pivote con comunicaciones se elimina, ya que el procesador propietario del pivote, posee también todos los elementos de la columna pivote y por tanto, no necesita comunicarse con el resto de procesadores para encontrarlo. Una vez establecido el pivote, el cálculo y las comunicaciones (de tipo *broadcast*) se realizan igual que en la factorización sin pivoteo (figura 6.8).

```
búsqueda (pivote,propietario) /*comunicaciones entre todos*/
if z=propietario then
    calcular fila U^0
endif
comunicar fila U^0
do k=0,n-1,g
    búsqueda (pivote,propietario) /*comunicaciones entre todos*/
    if z = propietario then
        calcular L^x; calcular A^x; calcular U^{x+1}
        par
            comunicar fila U^{x+1}
            calcular L; calcular A
        endpar
    else
        par
            comunicar fila U^{x+1}
            calcular L; calcular A
        endpar
    endif
    x=x+1
enddo
```

Tabla 6.5: Algoritmo de la factorización LU con pivoteo y distribución por filas

En la tabla 6.5 se presenta el algoritmo de la factorización LU con pivoteo y distribución por filas (para un procesador z) y en la tabla 6.6 se presenta el algoritmo de la factorización LU con pivoteo y distribución por columnas.

```
if z=0 then
    calcular bloque L^0
endif
comunicar bloque L^0 y el orden de pivoteo
do k=0,n-1,g
    if ((k+g)/g mod np) = z then
        se pivotan las filas según el orden recibido
        calcular U^x; calcular A^x; calcular L^x+1
        par
            comunicar bloque L^x+1 y el orden de pivoteo
            calcular U; calcular A
        endpar
    else
        par
            comunicar bloque L^x+1 y el orden de pivoteo
            calcular U; calcular A
        endpar
    endif
    x=x+1
enddo
```

Tabla 6.6: Algoritmo de la factorización LU con pivoteo y distribución por columnas

Obviamente la factorización sin pivoteo tiene mayores cotas de eficiencia en comparación con la factorización con pivoteo, pues en esta última se hacen necesarias más comunicaciones en cada iteración (caso especial con distribución por filas), sin embargo, no es tan aplicable a problemas reales como la segunda factorización.

Las figuras 6.9 y 6.10 representan los resultados de eficiencia para las mismas configuraciones de nodos y los mismos tamaños de matrices que en el caso sin pivoteo. Al igual que anteriormente, la eficiencia no alcanza la saturación para las dimensiones de matrices probadas.

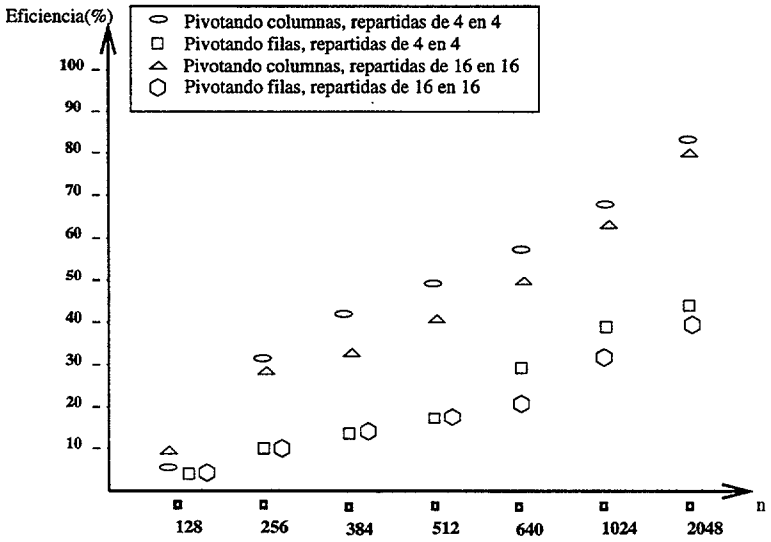


Figura 6.9: Factorización con pivoteo (4 procesadores)

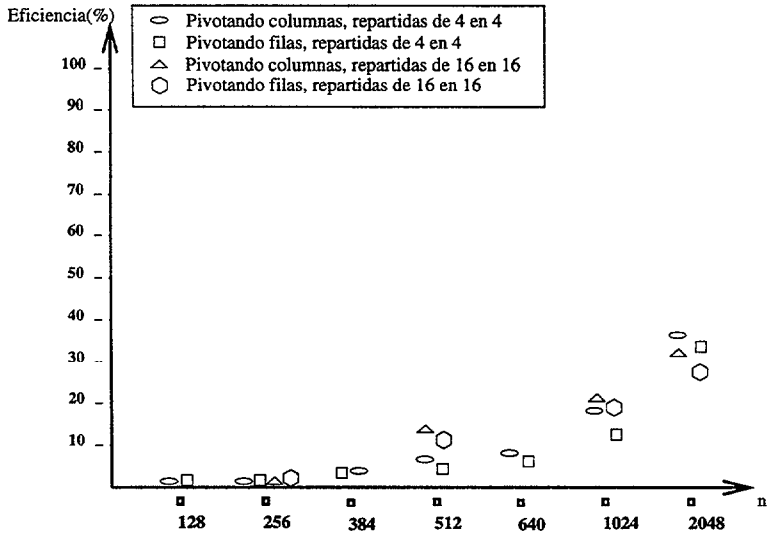


Figura 6.10: Factorización con pivoteo (16 procesadores)

6.5 Sistemas triangulares

La distribución propuesta para este ejemplo es una distribución por bloques de filas consecutivas. En la tabla 6.7 se presenta el algoritmo paralelo de la sustitución progresiva (el de la sustitución regresiva es similar). Los procesadores deben indexar submatrices A de dimensión $\frac{n}{np} \times n$ y subvectores b de dimensión $\frac{n}{np}$ en sus memorias locales, calculándose subvectores $y(x)$ de dimensión $\frac{n}{np}$.

```
if z=0 then
   calcular y_i = b_r/A_{i,i}   (i = 0..g − 1)
   calcular y_r(y_i)   (r = 1..g − 1) e (i = 0..r − 1)
   par
      comunicar y_r
      calcular y_i = b_i/A_{i,i}   (∀ sus i > 0)
   endpar
else
   par
      comunicar y_r
      calcular y_i = b_i/A_{i,i}   (∀ sus i > 0)
   endpar
endif
do i = 0, n − 1, g
   if (((i + g)/g)  mod np)= z then
      calcular y_r(y_j)   (r = i + g..i + 2 × g − 1) y (j = i..r − 1)
      par
         comunicar y_r
         calcular y_l(y_j)   (∀ sus l >(i + g + mg))   y   (j = i + g..i + 2 × g − 1)
      endpar
   else
      par
         comunicar y_r
         calcular y_l(y_j)   (∀ sus l >(i + g)) y (j = i..i + g − 1)
      endpar
   endif
enddo
```

Tabla 6.7: Algoritmo paralelo de la sustitución progresiva

Estos algoritmos se han probado sobre el multicomputador SN-1000 basado en *transputer* T800 y lenguaje OCCAM [47]. En esta configuración los *transputers* tienen 4Mbytes de memoria local y una frecuencia de reloj de 20MHz, y se han dispuesto siguiendo una topología en anillo. En las figuras 6.11 y 6.12 podemos ver los resultados experimentales (eficiencia *vs* n) para distintos valores de la variable g. Los datos de la

matriz A se encuentran inicialmente almacenados en la memoria del *host*, por lo que se deben comunicar a los *transputers* para que éstos realicen el cálculo de la factorización. En la medida de la eficiencia se ha tenido en cuenta el tiempo invertido en enviar estos datos (*host-transputers*) y el invertido en recibir los datos finales (*transputers-host*). Se puede observar el sentido ascendente de la curva de eficiencia, alcanzando la saturación alrededor del 70-75%.

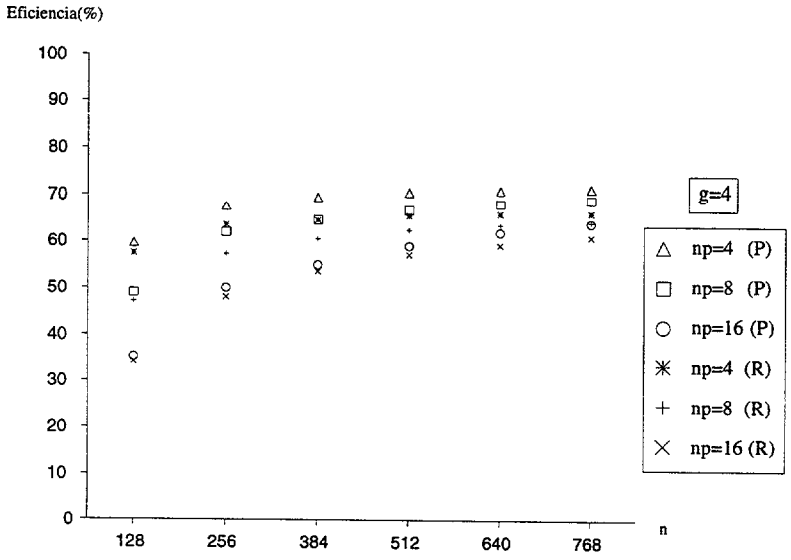


Figura 6.11: Eficiencia *vs* n del sistema triangular (P -progresiva-, R -regresiva-)

6.6 Factorización LU dispersa

La distribución propuesta es una distribución por columnas, para evitar las comunicaciones en la búsqueda del pivote. Así, en una iteración k se realiza la búsqueda de P pivotes, pertenecientes a las columnas $k..k+P$, por parte del procesador propietario de esas columnas. Una vez calculadas, este procesador las envía al resto de procesadores. Cuando los datos se reciben, todos los procesadores calculan las columnas $L_k..L_{k+P}$ de la matriz L y el propietario de las columnas $k+P+1..k+2\times P$ (próximas columnas pivote) actualiza dichas columnas. A continuación, y de forma solapada, todos los

procesadores comunican las P columnas $k+P+1..k+2\times P$ (desde el propietario al resto), y calculan el resto de elementos de A que poseen. Partiendo de esta explicación se genera el algoritmo paralelo de la tabla 6.8 (para un procesador genérico x), donde las comunicaciones son de tipo *broadcast.*

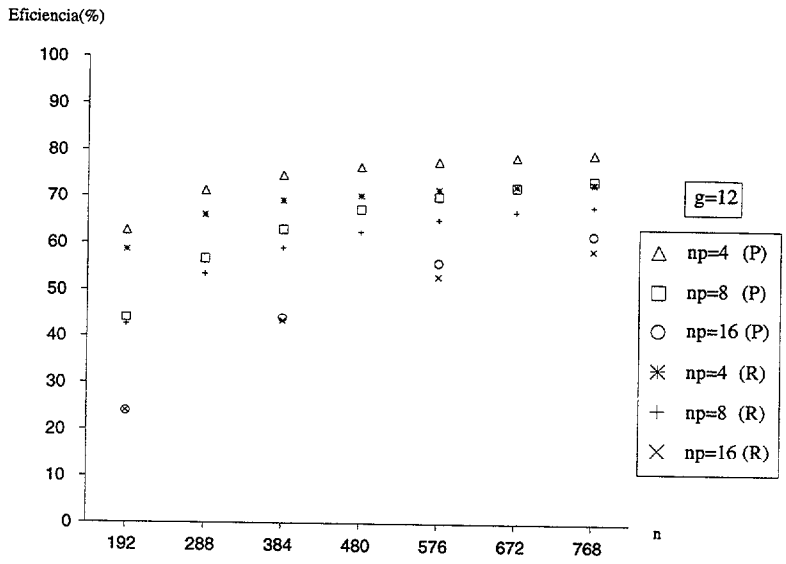


Figura 6.12: Eficiencia *vs* n del sistema triangular (P -progresiva-, R -regresiva-)

El algoritmo paralelo se implementó sobre un IBM-SP2, utilizando la opción del compilador *mpcc -03 -qarch=pwr2 name.c* (genera un eficiente código Power2). Las matrices sobre las que se realizó la factorización pertenecen a la colección Harwell-Boeing [71]. En la tabla 6.9 se presentan los resultados obtenidos [82]. Otros trabajos importantes sobre la factorización LU se pueden encontrar en [39] donde se desarrolla un nuevo código llamada S^* para máquinas distribuidas. Este código utiliza una factorización simbólica estática para predecir los nuevos ceros aparecidos en la factorización numérica y la técnica de supernodos (identifica los bloques densos dentro las matrices empleando así las rutinas BLAS-3). El código fue probado sobre un CRAY-T3D utilizando matrices no simétricas de la colección Harwell-Boeing [33]. En la misma línea de trabajo está la SuperLU que es analizada ampliamente en [63]. La SuperLU es un código paralelo para el cálculo de la LU de matrices dispersas no simétricas sobre multiprocesadores de memoria compartida. Basándose en la técnica de supernodos

evalúan el algoritmo paralelo generado sobre diferentes arquitecturas como pueden ser: MIPS R8000, DEC Alpha 21164, etc. Las mejoras obtenidas dependen en gran medida de las características de las matrices en cada caso (todas las matrices de prueba forma parte de la colección Harwell-Boeing). En [5] se presenta la factorización LU utilizando la técnica *rigth-looking* LU y estudiando dos posibles representaciones de datos: doble lista enlazada y vectores empaquetados. Se realiza un análisis para cada caso desarrollando los algoritmos en C sobre PVM para el CRAY T3D. Este algoritmo emplea las rutinas de la biblioteca MA48 [31] en la etapa de análisis, la cual se realiza separadamente de la de factorización. En las medidas se llega a la conclusión de que los vectores empaquetados presentan mejoras con respecto a las listas enlazadas ya que el uso de la jerarquía de memoria es más eficiente.

```
k=0
if x=0 then
   calcular bloque L_k
   comunicar bloque L_k
else
   comunicar bloque L_k
endif
do k=1,n/g,1
   if bloque k ∈ x then
      calcular bloque L_{k+1}
      par
         calcular bloque A_k
         comunicar bloque L_{k+1}
      endpar
   else
      par
         calcular bloque A_k
         comunicar bloque L_{k+1}
      endpar
   endif
enddo
```

Tabla 6.8: Algoritmo paralelo de la factorización LU dispersa

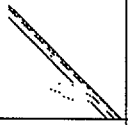
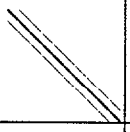
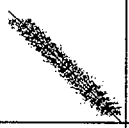
	Proc.	*G*	*P*	*Tiempo(sg.)*
$steam2_{600\times600}$ 	3	100	50	0.3088
	6	50	25	0.2266
	12	50	25	0.1978
$sherman2_{1080\times1080}$ 	3	180	90	4.35
	6	180	30	4.099
	12	45	45	2.6050
$nos5_{468\times468}$ 	3	156	39	0.0885
	6	78	78	0.08607
	12	39	13	0.10670
$sherman1_{1000\times1000}$ 	4	250	50	0.4229
	8	125	25	0.3808
	10	100	25	0.3649
$jpw991_{991\times991}$ 	4	124	31	1.3265
	8	124	31	1.1534
	12	84	21	1.0099
$mcca_{180\times180}$ 	3	60	30	0.054
	6	30	30	0.0537
	10	18	9	0.0580

Tabla 6.9: Resultados de la factorización LU dispersa

6.7 Aplicaciones a problemas de Ingeniería de Telecomunicación

En esta sección se estudia la paralelización del *método de los momentos (MoM)*, que permite calcular numéricamente la distribución de corriente en cualquier estructura

microtira y la respuesta impulsional de un canal de infrarrojos.

6.7.1 Método de los momentos

La aplicación del MoM, a la resolución de la *Ecuación Integral del Campo Eléctrico (EFIE)* sobre cualquier estructura microtira, genera una matriz Z de números complejos que forma parte de un sistema de ecuaciones lineales, donde el dato a calcular es la distribución de corrientes. En la ecuación 6.1 se muestra la ecuación funcional de la EFIE.

$$\bar{E}^i + jw\mu_0 L(\bar{J}_s) = 0 \tag{6.1}$$

siendo L un operador integro-diferencial diádico, $\bar{E}^i$ el campo incidente en la superficie y $\bar{J}_s$ la densidad de corriente superficial a calcular.

Para resolver esta ecuación se discretiza la estructura en M segmentos diferentes a los que se les denomina *parches* y se definen una serie de funciones *base* para cada uno de estos parches. Asimismo, se hace la suposición de que la distribución de la densidad de corriente superficial es una combinación lineal de funciones base bidimensionales, definidas en cada parche [4]. Una vez resuelta esta ecuación, la EFIE se transforma en un sistema de ecuaciones cuya notación matricial es:

$$\begin{pmatrix} Z^{ij}_{x,x} & Z^{ij}_{x,y} \\ Z^{ij}_{y,x} & Z^{ij}_{y,y} \end{pmatrix}_{2M\times 2M} \times \begin{pmatrix} I^i_x \\ I^i_y \end{pmatrix}_{2M\times 1} = \begin{pmatrix} V^j_x \\ V^j_y \end{pmatrix}_{2M\times 1} \qquad i,j = 1..M$$

El significado físico de $Z^{ij}_{x,x}$ y $Z^{ij}_{y,y}$ son las autoimpedancias debido a los acoplos de las componentes de las densidades de corriente iguales. $Z^{ij}_{x,y}$ y $Z^{ij}_{y,x}$ corresponde con los acoplos debido a componentes de corriente distintas y se les denomina acoplos mutuos. Cada uno de estos elementos está compuesto por integrales cuádruples tal y como indica la ecuación 6.2.

$$\int_{x_i-l_{1i}}^{x_i+l_{2i}} dx' \int_{x_j-l_{1j}}^{x_j+l_{2j}} dx' \int_{y_i-w_{1i}}^{y_i+w_{2i}} dy' \int_{y_j-w_{1j}}^{y_j+w_{2j}} F_i(x',y') F_j(x,y) G(x-x', y-y') dy \tag{6.2}$$

Para minimizar el tiempo de cómputo de estas integrales, se hace un cambio de dominio de forma que se convierten en una doble sumatoria de integrales dobles, con lo que se disminuye el tiempo de cálculo considerablemente [53].

6.7.1.1 Paralelización del método de los momentos

El MoM puede ser paralelizado en dos puntos: el cálculo de la matriz Z y la resolución del sistema de ecuaciones. En esta sección se va a analizar el cálculo de Z y la factorización LU implicada en la resolución de ecuaciones ($Z \times I = V$, donde Z y V son matrices complejas e I es un vector complejo).

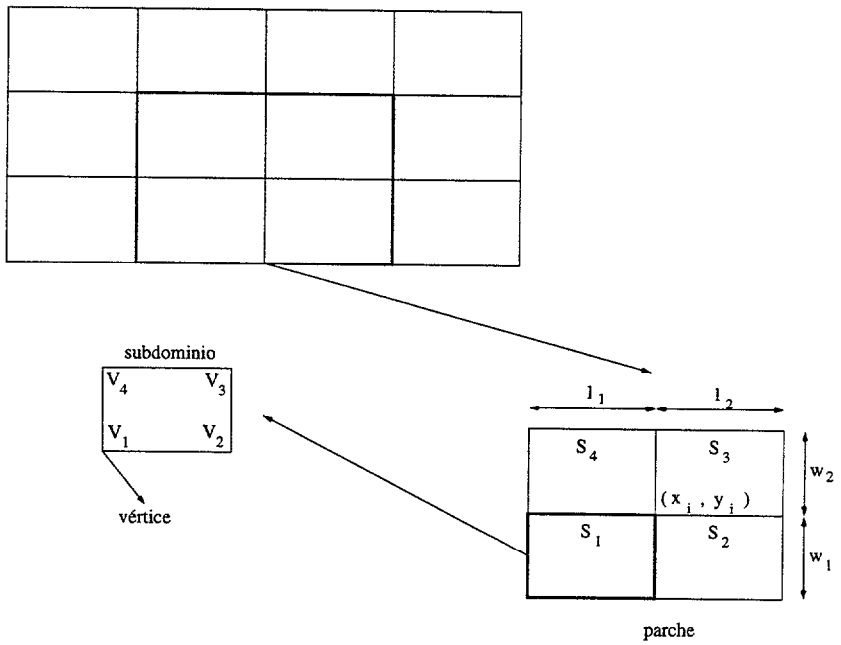


Figura 6.13: Parches, subdominios y vértices de una estructura

- Cálculo de la matriz Z. El cálculo de Z requiere el análisis de la estructura microtira. Esta estructura se divide en *parches* formados por cuatro *subdominios*, cada uno de los cuales tiene cuatro *vértices* (figura 6.13). En esta figura se observa una estructura microtira con seis parches de dimensión $(l_1 + l_2) \times (w_1 + w_2)$.

  La forma de esta estructura está almacenada en un archivo de geometría, que hay que recorrer para establecer los valores de las dimensiones y construir un vector, de dimensión igual al número de parches. Este vector se utiliza en la generación de los acoplos entre los distintos parches de la estructura. El cálculo de los acoplos incluye el cómputo del doble sumatorio y las integrales dobles comentadas previamente. Con esta información, se actualizan los valores de la matriz compleja Z, de forma que:

$$\begin{pmatrix} Z_{1,1} & Z_{1,2} & Z_{1,3} & \dots & Z_{1,N} \\ Z_{2,1} & Z_{2,2} & Z_{2,3} & \dots & Z_{2,N} \\ Z_{3,1} & Z_{3,2} & Z_{3,3} & \dots & Z_{3,N} \\ \dots & \dots & \dots & \dots & \dots \\ Z_{N,1} & Z_{N,2} & Z_{N,3} & \dots & Z_{N,N} \end{pmatrix}$$

donde el número de parches $M = \frac{N}{2}$. El cálculo de cada elemento de Z es independiente, siendo sólo función del vector de dimensiones (algoritmo 6.3).

```
do j=1,N/2,1
   do i=1,N/2,1
      call acoplo(acop,j,i,dimp(1,j),dimp(1,i),l0)
      Z(j,i)=(acop(1)+acop(2))/c4
      Z(j,i+N/2)=acop(3)/c4
      Z(j+N/2,i)=acop(4)/c4
      Z(j+N/2,i+N/2)=(acop(5)+acop(6))/c4
   enddo
enddo
```

(6.3)

En este algoritmo se puede observar que no existen dependencias en el cálculo de los elementos de Z ($c4$ es una constante), por tanto, éstos pueden ser calculados en diferentes procesadores. La distribución de datos para la matriz Z se realiza en función del cálculo posterior.

- Cálculo de la factorización LU. Según los $GDMs$ de la factorización LU (figura 5.10 a figura 5.14), la distribución de datos óptima para realizar la factorización de una matriz es la distribución por columnas. En este desarrollo se ha optado por una distribución cíclica, de forma que a cada procesador le corresponde $\frac{N}{np}$ columnas, siendo np el número de procesadores de la arquitectura elegida (figura 6.14).

  El cálculo de la factorización se hace, a su vez, dividiendo la matriz en bloques de $\frac{\frac{N}{2}}{np}$ columnas, de esta forma, se consigue que los procesadores estén ociosos menos tiempo.

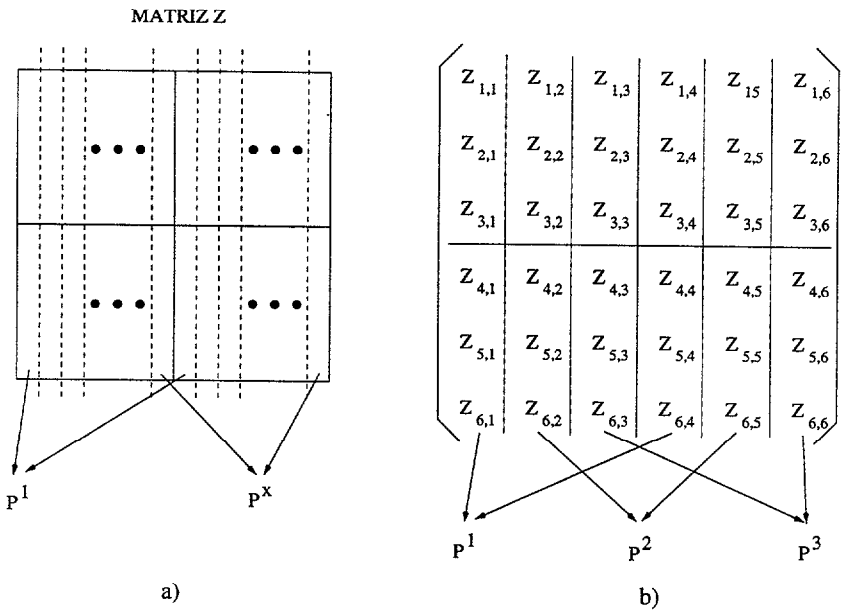


Figura 6.14: a) Distribución de la matriz Z. b) Ejemplo para $N = 6$

6.7.1.2 Desarrollo de la aplicación y resultados experimentales

El desarrollo práctico de este ejemplo se realizó utilizando los lenguajes de programación *C* y *Fortran*77. El núcleo de cálculo está implementado en *Fortran*77, ya que se han empleado rutinas complejas de cálculo de interpolación e integración desarrolladas previamente [54]. Estas rutinas acceden a los archivos de geometría que generan los vectores necesarios para el cómputo de las integrales.

Mediante el lenguaje de programación *C* se ha desarrollado el programa principal y el de cálculo de la factorización. Este programa llama a las rutinas *Fortran*, intercambiándose datos entre ellas. Asimismo, se ha utilizado la biblioteca de paso de mensajes *MPI* para establecer las comunicaciones necesarias en la paralelización de la aplicación.

Las medidas de tiempo se han realizado sobre tres arquitectura diferentes: SUN Ultra 5 (UltraSparc II 270MHz/64Mb), DualPentium (300MHz/128Mb) e IBM-SP2. Sobre estas tres máquinas se han computado los tiempos secuenciales y sobre la última se ha realizado la ejecución paralela. En la medida de los tiempos se ha tenido en cuenta el acceso a disco, para lectura de los archivos de geometrías y la generación de los archivos finales con los resultados de las corrientes y admitancias. Asimismo, se ha

utilizado doble precisión (tanto para números complejos como reales) en el cálculo de los elementos de la matriz Z, ya que los órdenes de magnitud de estos elementos oscilan entre los $10e^{-10}$ y los $10e^{4}$. En la tabla 6.10 se muestran las estructuras microtiras analizadas, en la que se indican sus características. Las formas de estas estructuras se pueden observar en las figuras 6.15, 6.16, 6.17 y 6.18. En la tabla 6.11 se pueden observar los resultados obtenidos en minutos [84].

Estructuras	Parches	Subdominios	Vértices	Excitaciones	Puertos	No Frecuencias
Dstub2	66	92	134	4	2	20
Híbrido	44	68	96	4	4	10
Stub2	25	40	64	2	2	81
Meandro	32	62	96	2	2	10

Tabla 6.10: Estructuras y características

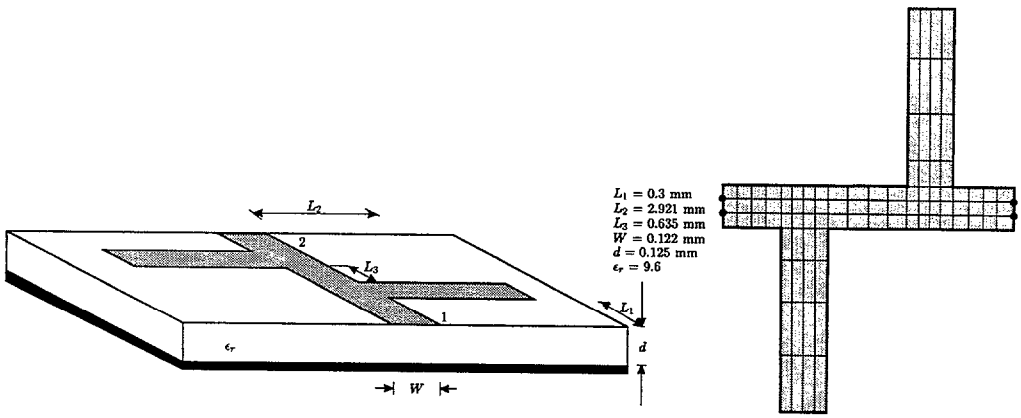


Figura 6.15: Estructura *Dstub2* y mallado realizado

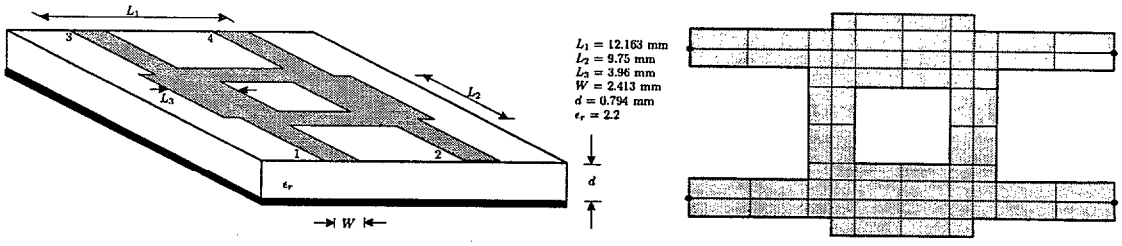


Figura 6.16: Estructura *Híbrido* y mallado realizado

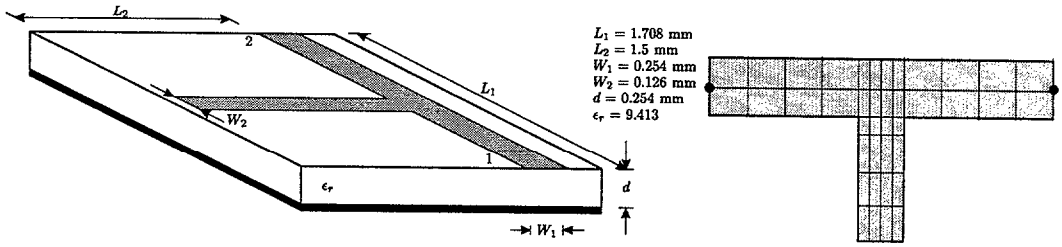


Figura 6.17: Estructura *Stub2* y mallado realizado

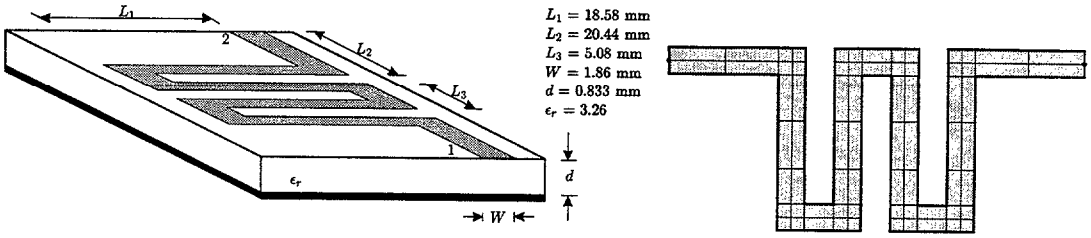


Figura 6.18: Estructura *Meandro* y mallado realizado

	Dstub2	*Híbrido*	*Stub2*	*Meandro*
SP-2 Paralelo	$\simeq 27$ (6 proc)	$\simeq 8$ (4 proc)	$\simeq 30$ (5 proc)	$\simeq 8$ (4 proc)
	$\simeq 25$ (11 proc)	$\simeq 6$ (11 proc)	$\simeq 9$ (25 proc)	$\simeq 5$ (8 proc)
	$\simeq 14$ (22 proc)	$\simeq 4$ (22 proc)		$\simeq 3$ (16 proc)
SP-2 (Secuencial)	$\simeq 131$	$\simeq 30$	$\simeq 75$	$\simeq 15$
DualPentium	$\simeq 170$	$\simeq 39$	$\simeq 82$	$\simeq 19$
Sun	$\simeq 200$	$\simeq 43$	$\simeq 97$	$\simeq 21$

Tabla 6.11: Tiempos de ejecución del MoM

6.7.2 Respuesta impulsional de un canal de infrarrojos

Supongamos un sistema real que consiste en una habitación vacía, en cuyo interior se encuentran uno o varios emisores de luz (LEDs) y uno o varios receptores. El problema que se presenta a continuación pretende obtener la respuesta al impulso de luz o a una señal generada por la(s) fuente(s) de luz. Esta respuesta es la distribución de potencia que llega a cada uno de los receptores en cada instante de tiempo.

Para caracterizar el problema, el emisor transmite potencia óptica en el instante de tiempo t_0 con un cierto diagrama de radiación, es decir, la potencia no se transmite por igual en todas las direcciones. Parte de esa potencia llega al receptor y otra parte alcanza las superficies reflectoras de la habitación. La potencia que llega a las superficies reflectoras rebota hacia otros lugares de la habitación, incluso hacia el receptor. Según esto, al receptor le llega potencia por dos caminos diferentes:

- A través de una línea directa hipotética trazada entre emisores y receptores.
- A través de un camino hipotético (emisor $\rightarrow$ pared1 $\rightarrow$ pared2 $\rightarrow$ ... $\rightarrow$ receptor), generado por rebotes en las superficies reflectoras (paredes de la habitación).

La distribución temporal de la potencia que llega al receptor está formada por un pico de potencia correspondiente al enlace directo entre emisor y receptor, seguido de una cola de potencia debida a los rebotes en las paredes de la habitación. La potencia en el receptor disminuye a medida que avanza el tiempo debido a los múltiples rebotes que van atenuando la señal.

Para considerar las reflexiones en las paredes de la habitación, éstas se dividen en pequeños elementos discretos a los que denominamos *celdas*. En la figura 6.19 se muestran las paredes de una habitación divida en celdas mediante el trazado de líneas rectas perpendiculares y paralelas a alguna de las aristas que las conforman.

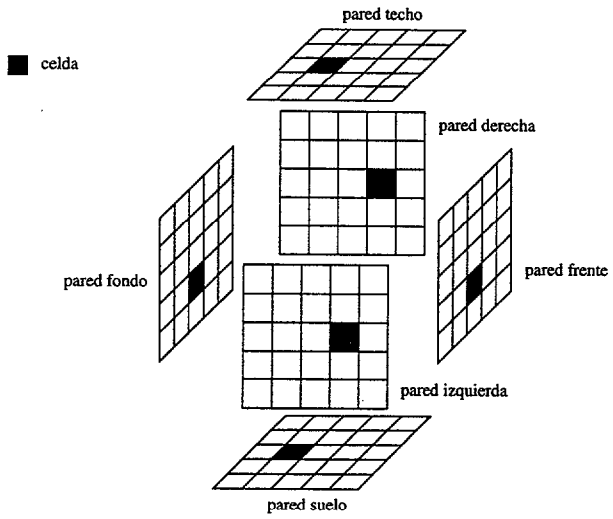


Figura 6.19: División en celdas de las paredes de una habitación

Una celda, con potencia distinta de cero, ilumina al resto de celdas que están en paredes diferentes a la suya. El tiempo que tarda en iluminar una celda emisora a una celda receptora, se calcula como:

$$t = \frac{d}{c} \tag{6.4}$$

donde c es la velocidad de la luz y d es la distancia que existe entre ambas celdas.

Dado que es necesario calcular todos los posibles casos de radiación desde una celda situada en una pared al resto de celdas que no están en su pared, entonces hay que obtener el tiempo máximo y mínimo en el que puede producir una radiación. Teniendo en cuenta la relación 6.4 y aplicándola para los valores extremos de la variable tiempo, se obtiene:

$$t_{max} = \frac{d_{max}}{c} \qquad t_{min} = \frac{d_{min}}{c}$$

donde t_{max} y t_{min} son los tiempos máximo y mínimo en los que cualquier dos celdas de la habitación se iluminan; y d_{max} y d_{min} corresponde con las distancias máxima y mínima respectivamente entre cualquier pareja de celdas de la habitación.

Estas distancias se muestran en la figura 6.20, donde d_{max} es la diagonal mayor de la habitación y d_{min} es la distancia entre dos celdas pertenecientes a esquinas enfrentadas diagonalmente.

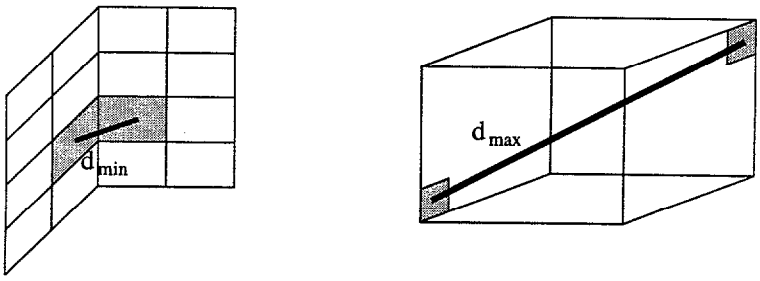


Figura 6.20: Distancia máxima y mínima entre dos celdas de una habitación

La potencia existente en una celda en el instante de tiempo t_1 depende de las potencias emitidas por otras celdas y emisores en instantes de tiempo anteriores a t_1. De igual manera, la potencia que llega al receptor en t_2 depende de la potencia emitida por las celdas y emisores en tiempos inferiores a t_2. Por tanto, debido a la dependencia temporal que existe en el cálculo de la potencia que llega a una celda o a un receptor, se emplea una simulación por tiempo en la solución del problema.

6.7.2.1 Paralelización del cálculo de la respuesta impulsional de un canal de infrarrojos

Partiendo de la explicación anterior, el núcleo del algoritmo secuencial que calcula la respuesta impulsional de un canal infrarrojos se presenta en 6.5.

```
do x=0,long_canal-1,1
   do pared_o=0,max_num_paredes,1
      do celda_o=0,TAM_PARED_pared_o,1
         if m_pot_pared > 0.0 then
            y=START_pared_o+celda_o
            do pared_d=0,pared_o-1,1
               do celda_d=0,TAM_PARED_pared_d,1
                  j=START_pared_d+celda_d
                  i=...
                  m_pot_i,j = m_pot_i,j + f(m_pot_x,y)
               enddo
            enddo
            do pared_d=pared_o+1,max_num_paredes,1
               do celda_d=0,TAM_PARED_pared_d,1
                  j=START_pared_d+celda_d
                  i=...
                  m_pot_i,j = m_pot_i,j + f(m_pot_x,y)
               enddo
            enddo
         endif
      enddo
   enddo
enddo
```
(6.5)

donde $START_l$ almacena la celda de comienzo de la pared l respecto de todas las celdas de una fila; y TAM_PARED_l almacena el número de celdas de la pared l.

Como se explicó anteriormente, una celda $(x = x_0, y = y_0)$ con potencia mayor que 0.0 perteneciente a la pared z, ilumina a cualquier otra celda (i, j) siempre que $0 \leq i < x$ o $x < i \leq long\_canal$ y $0 \leq j < START_z$ o $START_{z+1} \leq j \leq max\_num\_paredes$, es

decir una celda iluminada en un tiempo $x_0 = 0$ ilumina a cualquier otra que no esté en su pared, en algún tiempo x_1 posterior a x_0. Supongamos que todas las celdas que ilumina la celda $(x = 0, y = 0)$ son iluminadas en un tiempo $i = x_1 = 1$. En este caso, el GDM para la dependencia $m\_pot_{i,j} = f(m\_pot_{x,y})$ se presenta en la figura 6.21, donde todos los hiperarcos $A_{(1,j)}$ presentan el mismo "color" que el hiperarco $A_{(0,0)}$. Esto es así, porque los elementos $m\_pot_{1,j}$ dependen del elemento $m\_pot_{0,0}$.

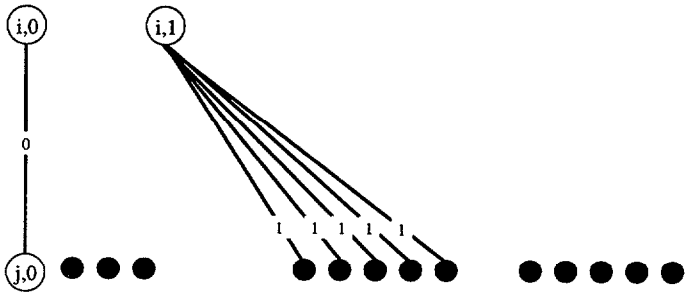


Figura 6.21: Ejemplo de GDM para el problema planteado

En la figura 6.21, se observa que todos los hiperarcos sobre los que se proyectan los elementos $m\_pot_{1,j}$ tienen el mismo "orden", por lo que estos datos pueden ser actualizados simultáneamente porque no dependen entre ellos (cada uno en un procesador diferente). A su vez, el "orden" de estos hiperarcos es inferior al del hiperarco sobre el que se proyecta el dato $m\_pot_{0,0}$. Según la definición 8, el dato $m\_pot_{0,0}$ debe ser comunicado mediante *broadcast* después de ser actualizado, a los procesadores que actualizan los elementos $m\_pot_{1,j}$ (si éstos se actualizan sobre más de un procesador).

Teniendo en cuenta el GDM generado y la explicación anterior, las n distintas celdas del problema se pueden distribuir en n procesadores, de forma que cada procesador debe enviar su celda actualizada al resto de procesadores para que estos actualicen la suya. En la práctica, se ha distribuido cada pared entre los $nproc$ procesadores ($nproc < n$ y $nproc < max\_num\_paredes$), de forma que cada procesador tiene $\frac{max\_num\_paredes}{nproc}$ celdas. Según esto, el algoritmo paralelo sigue el esquema de la tabla 2.1. De esta forma, se realiza un primer *broadcast* con los elementos de la primera fila de la matriz desde el P^0 al resto. A continuación todos los procesadores calculan con los elementos recibidos a la vez que el P^1 envía sus elementos de la primera fila al resto y así sucesivamente. Es decir, en la iteración k se calcula con los elementos de la fila $(k-1)^x$ del P^x a la vez que el P^y envía sus elementos (fila $(k-1)^y$). Al final debe hacerse un cálculo con los últimos datos recibidos.

Capítulo 7

Proyección hardware de algoritmos regulares

7.1 Introducción

En 1974 Leslie Lamport [60] presenta dos métodos para ejecutar en paralelo cálculos independientes (pertenecientes a un conjunto de bucles imbricados): el método del hiperplano de tiempo y el método de las coordenadas. El primer método pretende obtener un conjunto de bucles equivalentes a los de partida, utilizando transformaciones tal que los bucles transformados expresen qué iteraciones se pueden ejecutar en paralelo. En este trabajo se dan las primeras ideas intuitivas sobre la transformación de bucles, pero no se pretende hacer un cálculo profundo de las mismas. Estas ideas son aprovechadas por Bokari, quien en 1981 estudia la proyección de un conjunto de bucles imbricados en un conjunto de procesadores [12]. Esta proyección consiste en la asignación de diferentes cálculos a procesadores distintos, de forma que los cálculos independientes se llevan a cabo al mismo tiempo en procesadores separados. En [74] se presentan por primera vez las ideas formales para el diseño de algoritmos paralelos sistólicos, directamente en *hardware*. Este trabajo sentó las bases del diseño automático de *hardware* paralelo de propósito específico, realizado durante los años 80 y principios de los 90. En [77] se mejora esta teoría, para diseñar *hardware* paralelo, considerando un espacio de diseño con un número menor de restricciones. En [95] se estudia una metodología de

diseño de algoritmos paralelos que se implementan tanto en multiprocesadores *Transputer* [101] como en *hardware*, generalizando las técnicas de *slow-down* y *retiming* [61] y considerando la minimización de los ciclos de E/S.

En este capítulo nosotros pretendemos demostrar que los algoritmos paralelos obtenidos con nuestra metodología, a partir de algoritmos numéricos regulares o irregulares, pueden ser implementados en *hardware*. Este trabajo se planteó en colaboración con miembros del departamento de Ingeniería Electrónica y Automática de la U.L.P.G.C. En primer lugar, se debe aclarar que no es posible emplear las técnicas de diseño de algoritmos sistólicos porque nosotros solapamos los cálculos con las comunicaciones (en los algoritmos sistólicos no se tiene en cuenta el tiempo de comunicación). Además no pretendemos obtener algoritmos óptimos empleando esta técnica tal como se hizo en [104]. Por el contrario, sólo estamos interesados en comprobar que es posible diseñar *hardware* considerando el algoritmo obtenido a partir de nuestra metodología de distribución de datos, solapando cálculos y comunicaciones. Esto es, queremos comprobar que podemos considerar los parámetros de solapamiento teniendo en cuenta los tiempos de respuesta del *hardware* directamente.


```
do i=1,v_f,Step
   par hspace*.2cm
      calcular datos^(i-1)
      comunicar datos^i
   endpar
enddo
calcular datos^(v-f)
```

```
calcular datos^0
comunicar datos^0
do i=1,v_f,Step
   par
      calcular datos^(i-1)
      comunicar datos^i
   endpar
enddo
calcular datos^(v-f)
```

Tabla 7.1: Esquema de algoritmos paralelos

Para entender cómo se obtiene la descripción del *hardware* se ha de tener en cuenta el estudio del GDM y la distribución de datos y comunicaciones propuesta por éste. Cada "color" presente en el GDM corresponde con un proceso a realizar. El conjunto de

todos los procesos constituye el programa paralelo, que se puede ejecutar directamente en *hardware* (una parte del *hardware* realiza el proceso de cálculo y comunicación y otra parte, implementa la unidad de control). En base a este análisis se obtiene la descripción de las unidades de proceso y comunicación del sistema *hardware* final. En concreto, el esquema básico de cualquier algoritmo paralelo diseñado según nuestra metodología se encuentra en la tabla 7.1. En estos algoritmos, existe un primer paso de comunicación entre los EPs (que no siempre es necesario si los EPs pueden hacer un cálculo inicial), uno o más pasos de cálculo y comunicaciones solapadas y un último paso de cálculo. El *hardware* final debe pasar por estos distintos estados para implementar, de forma semánticamente correcta, las operaciones realizadas por el programa paralelo. Por último, hay que establecer la unidad de control que gestiona las señales de sistema desarrollado.

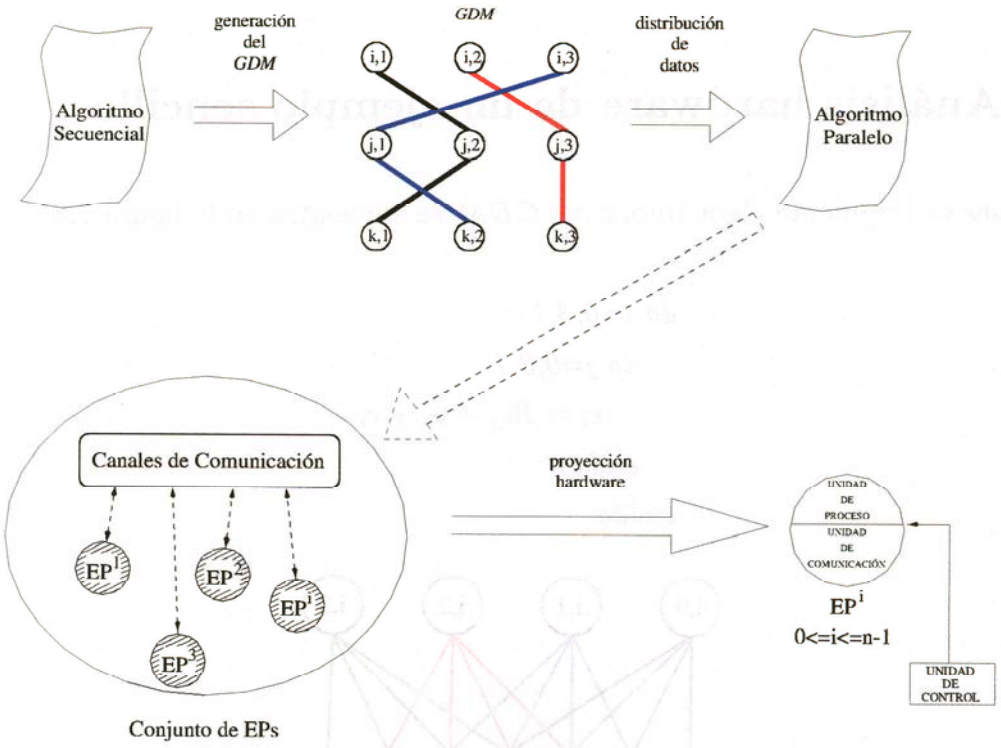


Figura 7.1: Esquema de la proyección de algoritmos paralelos

En la figura 7.1 se muestra gráficamente la idea utilizada para sintetizar *hardware* usando nuestra metodología. Para demostrar la generación de *hardware* se presentan tres ejemplos de complejidad creciente. En [35] se puede encontrar una descripción completa de la implementación del algoritmo de la multiplicación matriz-vector usando

el lenguaje Verilog y FPGAs [102]. El rendimiento de estos algoritmos es aceptable (figura 7.2), así como la ocupación del *hardware* [83].

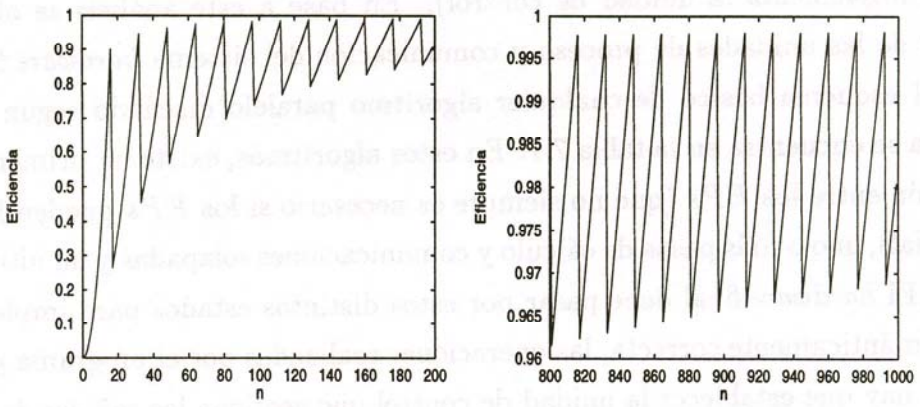


Figura 7.2: Medida de la eficiencia dependiente de la dimensión de la matriz (n)

7.2 Análisis hardware de un ejemplo sencillo

Supongamos el siguiente algoritmo, cuyo GDM se encuentra en la figura 7.3.

do i=0,3,1
  do j=0,3,1
    $c_i = B_{i,j} + a_j + c_i$
  enddo
enddo

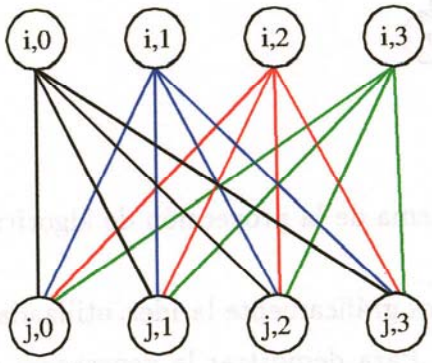


Figura 7.3: GDM del algoritmo anterior

Analizando el GDM del algoritmo observamos que existen cuatro procesos (en el GDM existen cuatro colores -pesos-), cada uno de los cuales se realiza en un elemento de proceso (EP) diferente. Según la distribución de datos indicada por el GDM , el EP^x $(0 \leq x \leq 3)$ almacena los datos: c_x, $B_{x,t}$ y a_t $(0 \leq t \leq 3)$. Según esta distribución, los EPs almacenan datos diferentes del vector c y de la matriz B, pero los datos del vector a se replican. Para evitar esta replicación se distribuye este vector entre los distintos EPs, tal que el EP^x almacena exclusivamente el elemento a_x, necesitándose comunicaciones entre ellos para realizar todos los cálculos. Con esta distribución inicial de datos y los requisitos de comunicaciones explicados, un proceso x tiene la siguiente estructura:

do t=0,2,1
  par
    calcular $c^x = B^{x,t} + a^t + c^x$
    comunicar a^t
  endpar
enddo
calcular $c^x = B^{x,3} + a^3 + c^x$

En este algoritmo se han empleado superíndices para referirnos a los bloques utilizados en cada caso. Así, en el caso del vector a, el bloque a^0 corresponde con a_0 en el EP^0, con a_1 en el EP^1, con a_2 en el EP^2 y con a_3 en el EP^3. La instrucción *comunicar* a^t significa que el EP^x comunica su elemento a^t al EP^y si $y = x - 1$ $(1 \leq x \leq nproc - 1)$ o si $y = nproc - 1$ $(x = 0)$; y el EP^x recibe el siguiente elemento a^{t+1} del EP^z si $z = x + 1$ $(0 \leq x < nproc - 1)$ o si $z = 0$ $(x = nproc - 1)$. En cualquier caso, $nproc$ es el número de EPs.

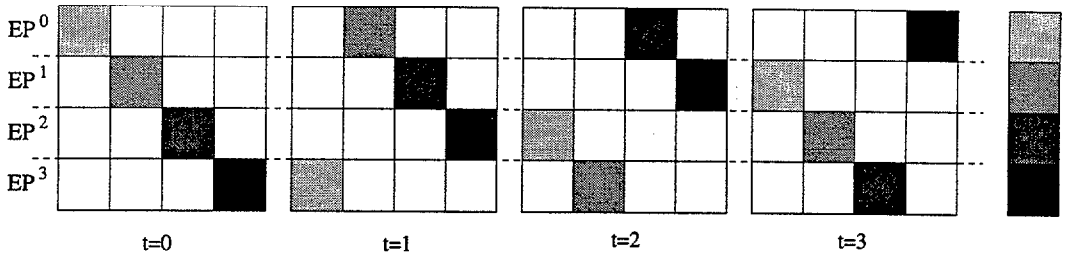


Figura 7.4: Datos utilizados en cada iteración t del algoritmo paralelo para la matriz y los vectores

En la figura 7.4 se muestra qué elementos de la matriz B y del vector a se utilizan en cada iteración de t (los datos del vector a tienen que ser comunicados entre iteraciones para que los cálculos se puedan realizar correctamente). Asimismo, se observa que no es necesario un primer paso de comunicación, ya que los EPs pueden hacer un cálculo inicial con los datos que tienen almacenados en sus memorias locales.

7.2.1 Unidad de control

Teniendo en cuenta el esquema de la tabla 7.1, el sistema *hardware* debe tener tres estados de funcionamiento (figura 7.5): E_Cm (estado de comunicación), E_Sp (estado de solapamiento de cálculos y comunicaciones) y E_Cl (estado de cálculo).

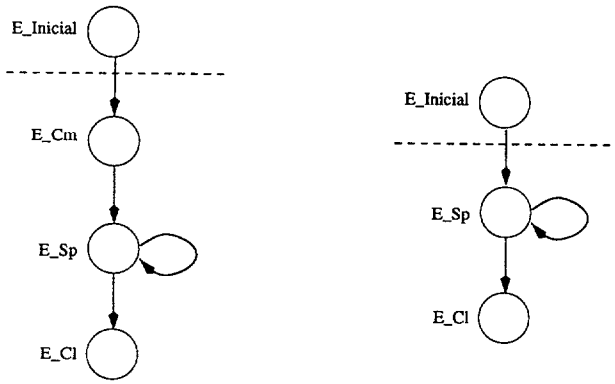


Figura 7.5: Posibles estados del sistema *hardware*

En el tiempo t_0, se realiza el estado E_Cm, que puede no ser necesario según lo explicado anteriormente; en el tiempo t_i $(0 < i < n)$ se procesa el estado E_Sp y en el tiempo t_n, el sistema se encuentra en el estado E_Cl. Estos tres estados están generados por una unidad de control que gobierna la ejecución del algoritmo paralelo en *hardware*. Asimismo, es necesario un nuevo estado, que llamaremos E_Inicial, en el cual se cargan los datos iniciales según el análisis realizado utilizando la metodología.

7.2.2 Unidad de proceso

El sistema *hardware* está formado por una unidad de proceso que se encarga de realizar las operaciones de cálculo del algoritmo paralelo y por tanto, depende de las características de este.

Inicialmente se puede suponer que existen tantas unidades como procesos paralelos (colores diferentes en el GDM). Así, para el ejemplo anterior se necesitan cuatro unidades de proceso que realizan la suma de los elementos de los vectores c y a y de la matriz B. En la figura 7.6 se muestra un esquema previo de las unidades de proceso para el ejemplo comentado, donde el multiplexor $MUXS$ premite la carga inicial de c.

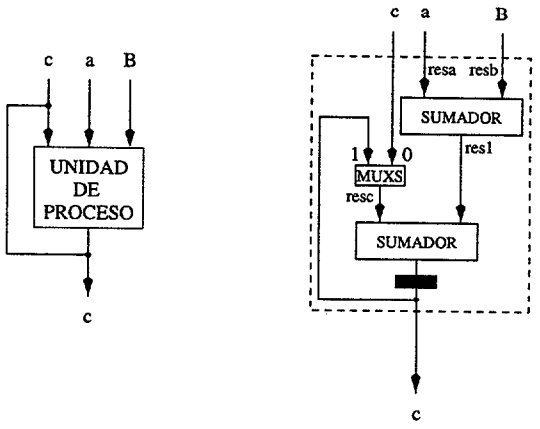


Figura 7.6: Unidad de proceso del ejemplo estudiado

7.2.3 Unidad de comunicación

Esta unidad es la encargada de recibir y enviar los datos para realizar el cálculo por lo que está fuertemente relacionada con la unidad de proceso, ya que suministra a esta última los datos necesarios en cada fase. Estos datos deben ser almacenados para su posterior tratamiento, por tanto, se deduce la necesidad de memorias utilizadas en ambas unidades.

La unidad de proceso junto con la unidad de comunicación forman el EP. El elemento de proceso dispone de memorias para almacenar los datos que se van a utilizar en los cálculos y los resultados parciales o finales obtenidos de este proceso. En la figura 7.7 se representa un EP que se comunica con el exterior a través de su unidad de comunicación. Los datos que se reciben se van almacenando en una memoria de entrada para su posterior utilización. A su vez, el EP puede enviar los datos que se encuentran en su memoria de salida, al exterior. En los algoritmos desarrollados siguiendo nuestra metodología, los datos de entrada a un EP son los datos de salida de otro EP, ya que corresponden con las comunicaciones solapadas que circulan a través

de los distintos EPs. Es sencillo deducir de forma automática, el tamaño de estas memorias, ya que corresponde con el tamaño del bloque de datos a transmitir.

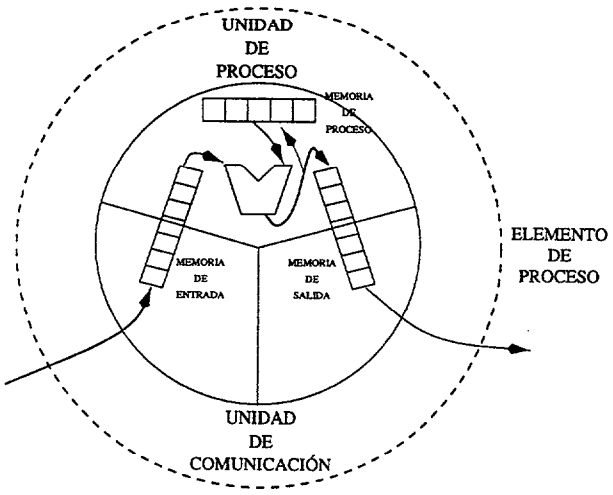


Figura 7.7: Elemento de proceso

Si analizamos el caso en estudio, vemos que en el estado E_Cm y en cada fase del estado E_Sp se comunica un bloque del vector a que corresponde con un único elemento, por tanto, podemos utilizar un registro (memoria de entrada/salida) para almacenar el dato de entrada que será utilizado en la siguiente iteración y que posteriormente será comunicado al EP vecino. Por otro lado, hay que almacenar los datos del vector c (un elemento por EP) y de la matriz B (cuatro elementos por EP) en memorias internas que corresponden con la memoria de proceso.

7.2.4 Esquema final

En la figura 7.8 se presenta un esquema de los EPs que realizan el cálculo del vector c mediante acumulaciones sucesivas. Los datos iniciales de cada EP son cargados en los registros Ra y Rc y en la memoria MB. El registro Rc y la memoria MB almacenan los datos a utilizar en todo el cálculo del algoritmo descrito, siguiendo el orden indicado en la figura 7.4, sin embargo, el registro Ra almacena datos diferentes para cada iteración en t del algoritmo. Asimismo, se observa la inclusión de multiplexores para seleccionar las señales oportunas en cada momento: el dato c de entrada al segundo sumador puede proceder del registro Rc o de la salida del sumador (c parcial). De igual forma, el dato a almacenar en Ra, puede proceder del exterior o bien del registro Ra vecino.

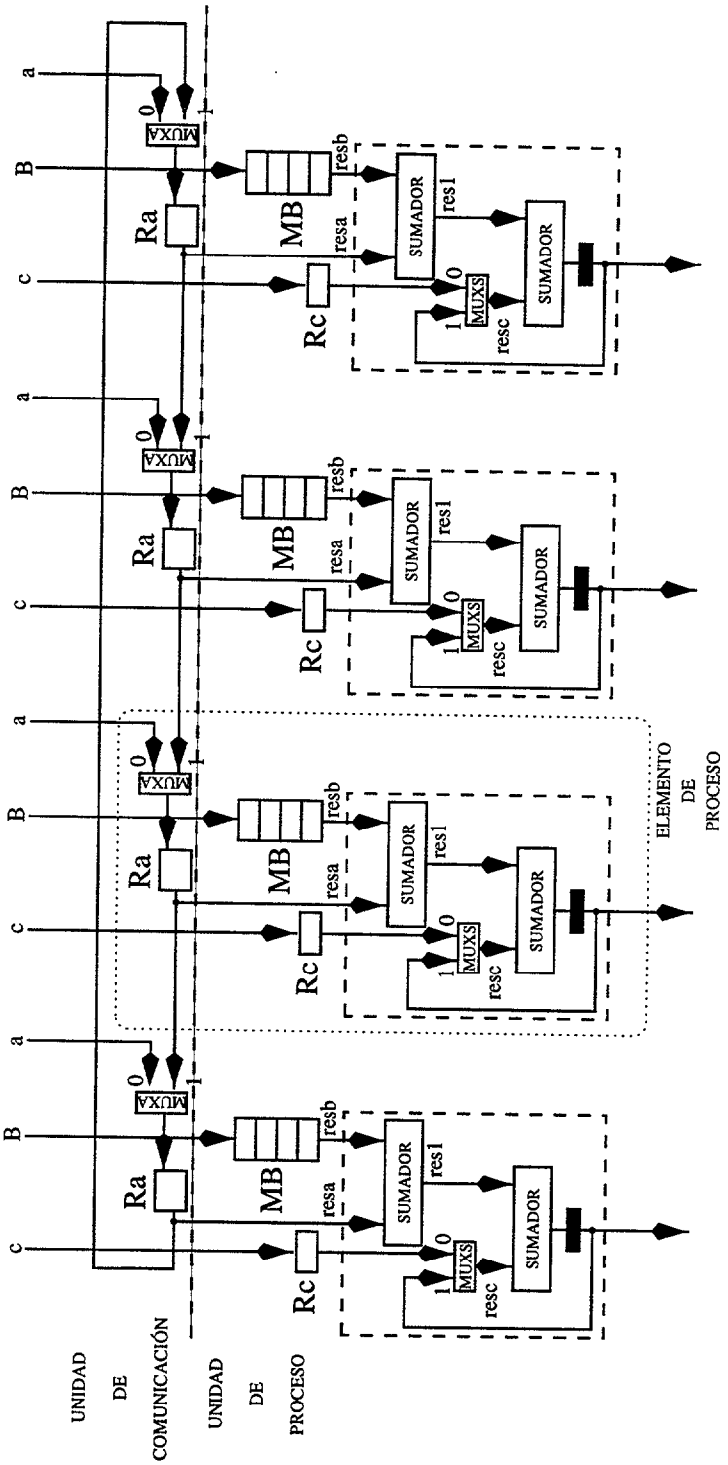


Figura 7.8: Sistema *hardware* del ejemplo estudiado

```
(*Estado E_Inicial*)
E_MUXA ← 0
do ep=0,3,1
   Ra ← MUXA
   Rc ← c
enddo
WR_MB ← SET
do ep=0,3,1
   do dir=0,3,1
      MB[dir] ← B
   enddo
enddo
WR_MB ← RESET
```

```
(*Estado E_Sp*)
E_MUXA ← 1
doall ep=0,3,1
   E_MUXS ← 0
   resc ← Rc
   E_MUXS ← 1
   do pos=0,2,1
      resb=MB[pos]
      resa=Ra
      res1=SUM1(resa,resb)
      resc=SUM2(res1,resc)
      Ra ← MUXA
   enddo
enddoall
```

```
(*Estado E_Cl*)
doall ep=0,3,1
   resb=MB[3]
   resa=Ra
   res1=SUM1(resa,resb)
   resc=SUM2(res1,resc)
enddoall
```

Tabla 7.2: Descripción RTL del primer ejemplo

Una vez establecidas las distintas unidades *hardware*, que ejecutan el proceso paralelo, hay que desarrollar la unidad de control que las gobierna. Así, la unidad de control, en el estado E_Inicial debe generar las señales necesarias para almacenar los datos de los vectores c y a y de la matriz B. Para ello, esta unidad tiene que direccionar los registros y memoria asociados a estos elementos en un orden determinado; y debe generar las señales de escritura necesarias. En el estado E_Sp, y para cada iteración t (que puede ser establecida por el reloj del sistema o por una señal obtenida a partir de este), se debe producir el desplazamiento de los datos almacenados en los registros Ra (que supone la fase de comunicación solapada) y el cálculo de la suma (fase de cálculo solapado). Antes de que comience la siguiente iteración, el resultado parcial c y el nuevo dato de a deben ser estables en los registros correspondientes (para el caso del vector c, en el registro de salida del sumador). En este estado, la unidad de control debe proveer las señales de lectura y el direccionamiento de la memoria. Por último, en el estado E_Cl, se realiza el mismo proceso que en el estado anterior, ya que aunque se desplaza el dato de a, no se consume tiempo adicional.

En la tabla 7.2 se presenta la descripción a *nivel de transferencia de registros (RTL)* de los tres estados de un posible autómata de control para este ejemplo. Las señales

E_{MUXS}, E_{MUXA}, WR_{MB} y el direccionamiento de la memoria MB es común para todos los EP. En la tabla 7.3 se muestran los estados de estas señales para el sistema *hardware* de la figura 7.8.

ciclo	E_{MUXS}	E_{MUXA}	WR_{RB}	*dirección MB*
1	0	0	1	0
2	0	0	1	1
3	0	0	1	2
4	0	0	1	3
5	0	1	0	0
6	1	1	0	1
7	1	1	0	2
8	1	X	0	3

Tabla 7.3: Estados de las señales del sistema *hardware* del primer ejemplo

7.2.4.1 Estudio de la eficiencia

En el periodo de operación solapada, la ruta crítica del sistema viene determinada por el camino a través de los sumadores, el registro de salida y el registro Ra, de tal forma que se puede decir que el *tiempo de operación* (t_{op}) corresponde con: $t_{op} = 2 \times t_{sum} + t_{ff}$, donde t_{sum} es el tiempo empleado en los sumadores y t_{ff} es el tiempo invertido en un *flip-flop*. En cualquier caso, t_{op} siempre será netamente superior al retardo de comunicación ya que el tiempo de comunicación (t_{com}) corresponde con t_{ff} ($t_{op} > t_{com}$). Asimismo, en el sistema *hardware* para realizar tanto una comunicación como un cálculo se necesita un ciclo de reloj, por tanto, el solapamiento es total. Por otro lado, si no tenemos en cuenta el tiempo que se necesita para realizar la carga inicial de datos, se puede constatar que la eficiencia del sistema es del 100%, puesto que es capaz de realizar la operación completa en 4 ciclos, con 4 EPs. Sin embargo, si contamos el tiempo de carga inicial, se necesitan 4 ciclos para almacenar a, c y B, con lo que la eficiencia decrece al 50%. En cualquier caso, y teniendo en cuenta que en cada ciclo de operación cada EP tan sólo necesita un elemento de la matriz B, este puede ser cargado en paralelo con la operación siguiente. Así, se reduce el tiempo de carga inicial a un ciclo, y por tanto la eficiencia aumenta al 80%. De forma genérica, para realizar

el cálculo con matrices de dimensión $n \times n$, y n EPs se tiene una eficiencia total de:

$$\eta = \frac{n_{optotal}}{nproc \times n_{ciclos}} \times 100 = \frac{n^2}{n \times (n+1)} \times 100 = \frac{n}{n+1} \times 100$$

7.2.5 Generalización del tamaño del problema

Si suponemos que la dimensión de la matriz y de los vectores es n, el GDM generado indica que se necesitan n procesos para resolver el problema, por tanto, el sistema *hardware* final debe disponer de n EPs. Sin embargo, existen problemas físicos que impiden que el número de EPs aumente indefinidamente. Debido a esto, en esta sección se plantea la solución del problema planteado utilizando el mismo número de EPs que anteriormente, aunque la dimensión de la matriz y de los vectores sea $n > 4$. Así, cada EP almacena $g = n/nproc$ filas, donde $nproc$ es el número de EPs utilizados. El proceso paralelo que realiza un EP^x es el siguiente:

do t=0,n-2,1
  par
    do fila=0,g-1,1
      calcular $c^{fila} = B^{fila,t} + a^t + c^{fila}$
    enddo
    comunicar a^t
  endpar
enddo

do fila=0,g-1,1
  calcular $c^{fila} = B^{fila,n-1} + a^{n-1} + c^{fila}$
enddo

donde a^t corresponde con el bloque que se utiliza en el proceso de la iteración t del algoritmo; y la instrucción *comunicar* a^t sigue las mismas pautas comentadas anteriormente.

Partiendo de esta nueva distribución de datos y del esquema paralelo, cada EP posee una unidad de proceso que consta de dos sumadores tal y como se presenta en la

figura 7.6. Asimismo, cada EP posee $\frac{n}{nproc}$ elementos de los vectores a y c y $\frac{n}{nproc} \times n$ elementos de la matriz B. Debido a esto, los datos de las tres estructuras deben almacenarse en memorias del tamaño indicado (memoria de entrada/salida y memoria de proceso). En la figura 7.9 se presenta un esquema del EP para la generalización del tamaño del problema.

Para este ejemplo, la unidad de control en el estado E_Inicial, debe generar las señales necesarias para almacenar los datos de los vectores c y a y de la matriz B (seleccionando los multiplexores asociados a Ma y Mc para que la entrada de datos sea desde el exterior). En el estado E_Sp, y para cada iteración t, se debe producir el desplazamiento de los datos almacenados en las memorias Ma (que supone la fase de comunicación solapada) y el cálculo de la suma de los subvectores y la submatriz (fase de cálculo solapado). Las memorias Ma se leen y escriben a la vez durante la fase de solapamiento. En la implementación física esta memoria puede ser llevada a cabo, bien por una memoria de doble puerto o bien por un registro de desplazamiento, de tal forma que los diferentes registros Ma de los EPs forman un único registro de desplazamiento distribuido entre ellos. Antes de que comience la siguiente iteración, el resultado parcial c y el nuevo dato de a deben ser estables en la memoria y registro correspondientes (para el caso del vector c, en el registro de salida del sumador). En este estado, la unidad de control debe proveer las señales de lectura y el direccionamiento de las memorias. Hay que tener en cuenta que en este caso, la memoria Mc debe ser escrita al final de cada proceso de cálculo parcial sobre una fila de la matriz B, es decir, después de utilizar un bloque del vector a y una fila de la matriz B (inicialmente la memoria es cargada del exterior y en cada subproceso es cargada con el dato de salida del último sumador). Finalmente, en el estado E_Cl, se realiza el mismo proceso que en el estado anterior, ya que aunque se desplazan los datos de a, no se consume tiempo adicional.

En la tabla 7.4 se presenta una descripción RTL de las tres fases de la unidad que genera las señales para el control del sistema que se especifica en esta sección. Los cálculos se realizan según indica la figura 7.4, aunque en este caso cada cuadrado de la figura representa una submatriz B y un subvector de a y c. Asimismo, los datos se almacenan en las memorias del sistema *hardware* siguiendo el orden de cálculos establecido (es decir, el mismo indicado por la figura 7.4).

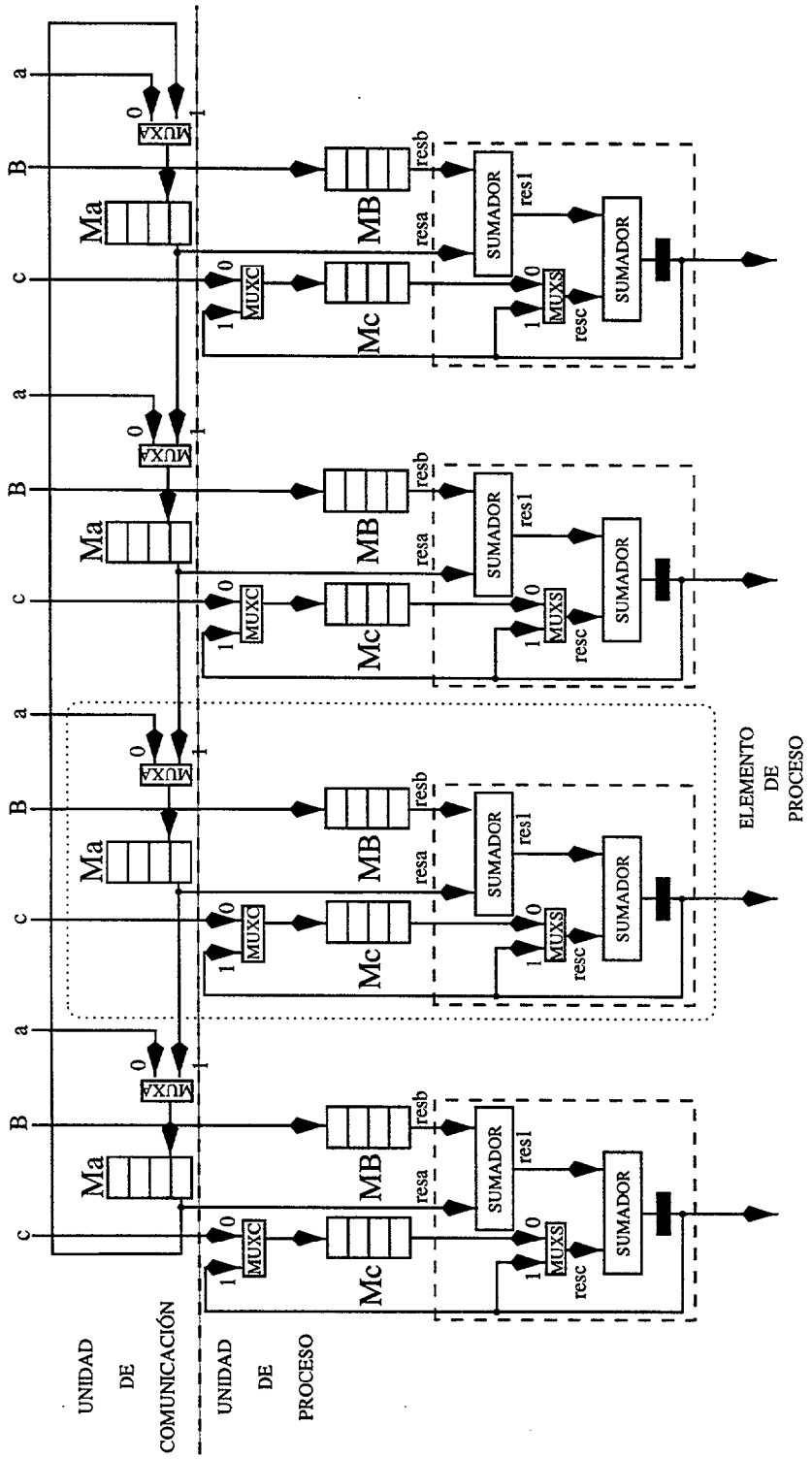


Figura 7.9: Sistema *hardware* al generalizar el tamaño del problema

```
(*Estado E_Inicial*)
E_MUXA ← 0;E_MUXC ← 0
WR_Ma ← SET;WR_Mc ← SET
do ep=0,3,1
   do dir=0,g-1,1
      Ma[dir] ← MUXA;Mc[dir] ← MUXC
   enddo
enddo
WR_Ma ← RESET;WR_Mc ← RESET
WR_MB ← SET
do ep=0,3,1
   do dir=0,(g× n)-1,1
      MB[dir]=B
   enddo
enddo
```

```
(*Estado E_Cl*)
doall ep=0,3,1
   do dir=0,g-1,1
      E_MUXS ← 0;resc=Mc[dir]
      E_MUXS ← 1;ele=dir×g
      do pos=ele,ele+g-1,1
         resb=MB[pos];resa=Ma[pos-ele]
         res1=SUM1(resa,resb)
         resc=SUM2(res1,resc)
      enddo
      WR_Mc ← SET;Mc[dir] ← resc
      WR_Mc ← RESET
   enddo
enddoall
```

```
(*Estado E_Sp*)
E_MUXA ← 1;E_MUXC ← 1
doall ep=0,3,1
   do t=0,nproc-2,1
      do dir=0,g-2,1
         E_MUXS ← 0;resc=Mc[dir]
         E_MUXS ← 1;ele=dir×g
         do pos=ele,ele+g-1,1
            resb=MB[pos];resa=Ma[pos-ele]
            res1=SUM1(resa,resb)
            resc=SUM2(res1,resc)
         enddo
         WR_Mc ← SET;Mc[dir] ← resc
         WR_Mc ← RESET
      enddo
      dir=g-1
      E_MUXS ← 0;resc=Mc[dir]
      E_MUXS ← 1;ele=dir×g
      do pos=ele,ele+g-1,1
         resb=MB[pos];resa=Ma[pos-ele]
         res1=SUM1(resa,resb)
         resc=SUM2(res1,resc)
         WR_Ma ← SET;Ma[dir] ← MUXA
         WR_Ma ← RESET
      enddo
      WR_Mc ← SET;Mc[dir] ← resc
      WR_Mc ← RESET
   enddo
enddoall
```

Tabla 7.4: Descripción RTL del segundo ejemplo (estados E_Inicial, E_Sp y E_Cl)

7.2.5.1 Estudio de la eficiencia

En la generalización del problema se sigue cumpliendo que:

$$t_{op} = 2 \times t_{sum} + t_{ff}$$

donde t_{sum} es el tiempo empleado en los sumadores y t_{ff} es el tiempo invertido en el *flip-flop* asociado a la salida del sumador. En este ejemplo, el tiempo de comunicación (t_{com}) corresponde con la comunicación de los elementos del vector a. En cualquier caso, t_{op} siempre será netamente superior al retardo de cada comunicación, ya que $t_{com} = t_{Ma}$. En este ejemplo, cada bloque de datos es procesado en $\frac{n^2}{nproc}$ ciclos, necesitándose $\frac{n}{nproc}$ ciclos para comunicar los elementos (solapamiento total). Partiendo de una solución en la que los datos iniciales están cargados en las memorias correspondientes, la eficiencia es del 100%. Si se tiene en cuenta el tiempo de carga inicial, hay que considerar que se pueden realizar los cálculos desde que el primer dato está cargado. Así, si se tiene en cuenta que cada EP debe realizar $\frac{n^2}{nproc}$ cálculos, el número de ciclos necesarios corresponde con $1 + \frac{n^2}{nproc}$, y la eficiencia se establece de la forma:

$$\eta = \frac{n_{optotal}}{nproc \times n_{ciclos}} \times 100 = \frac{n^2}{nproc \times (1 + \frac{n^2}{nproc})} \times 100 = \frac{n^2}{nproc + n^2} \times 100$$

Todos estos casos están basados en situaciones donde n es múltiplo entero de $nproc$. Si esta situación no fuera posible, es necesario completar con ceros las matrices y vectores, con lo que la eficiencia se reduce (el número de ciclos corresponde con: $1 + \frac{(\lceil n/nproc \rceil \times nproc)^2}{nproc} = 1 + \frac{m^2}{nproc}$). Así:

$$\eta = \frac{n_{optotal}}{nproc \times n_{ciclos}} \times 100 = \frac{n^2}{nproc + m^2} \times 100$$

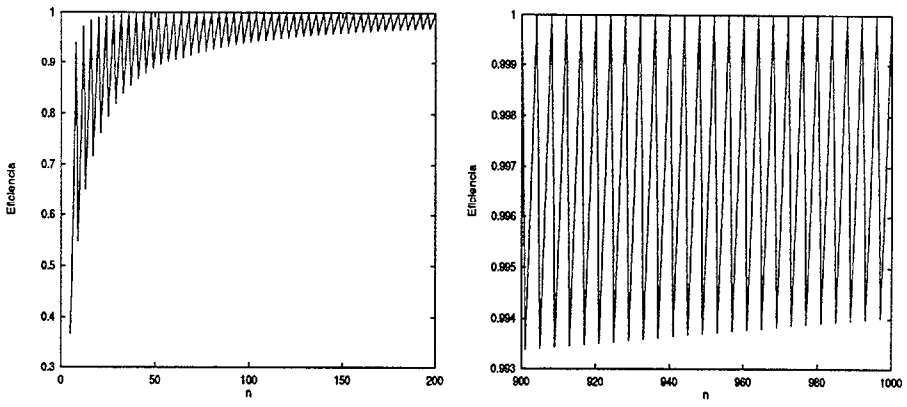


Figura 7.10: Medida de la eficiencia dependiente de la dimensión de la matriz (n)

Si $n \geq nproc$, el peor caso coincide con $n = nproc + 1$, de forma que si hay 16 procesadores, la eficiencia corresponde con: $\eta = \frac{17^2}{16+32^2} \times 100 = 27.78\%$. A medida que

crece n, la eficiencia aumenta puesto que se hace despreciable el relleno de ceros frente a la matriz total (figura 7.10).

7.3 Análisis hardware de un problema complejo

Supongamos el algoritmo secuencial de la factorización LU:

```
do k=0,n-2,1
   do i=k+1,n-1,1
      A_{i,k} = A_{i,k}/A_{k,k}
      do j=k+1,n-1,1
         A_{i,j} = A_{i,j} - A_{i,k} × A_{k,j}
      enddo
   enddo
enddo
```

Los *GDMs* de este algoritmo se encuentran en las figuras 5.10 a 5.14. Según la distribución inicial de datos, el proceso paralelo que realiza un EP genérico x, es el siguiente:

```
comunicar columna 0
do k=1,n-2,1
   if k=x then
      calcular columna k-1; calcular columna k
      par
         comunicar columna k
         calcular A^x (utilizando la columna k − 1)
      endpar
   else
      par
         comunicar columna k
         calcular A^x (utilizando la columna k − 1)
      endpar
   endif
enddo
```

if x=n-1
    calcular columna n-2; calcular A^x
endif

En este algoritmo, *calcular columna k* y *calcular* A^x corresponden a las sentencias exterior e interior del algoritmo secuencial respectivamente. Asimismo, *comunicar columna k* significa que el EP^x comunica la columna x al resto de EPs.

7.3.1 Unidad de control

Teniendo en cuenta el esquema de la tabla 7.1 y el algoritmo paralelo, el sistema *hardware* debe tener tres estados de funcionamiento: E_Cm, E_Sp y E_Cl (este último sólo lo realiza el EP propietario de la última columna). Asimismo, es necesario un estado E_Inicial en el cual se cargan los datos iniciales.

En el estado E_Cm, el EP^0 comunica al resto de EPs los datos de la columna 0. En el estado E_Sp, el EP propietario de la columna k calcula la columna $k-1$ (que le fue comunicada anteriormente), actualiza los elementos de la columna k en función de la columna anterior y comunica al resto de EPs la columna k; los EPs no propietarios de la columna k reciben esta columna en paralelo con el cálculo de la columna $k-1$ y el cálculo de sus propios elementos. Finalmente, en el estado E_Cl, el EP propietario de la columna $n-1$, actualiza la columna anterior y los elementos de su propia columna.

7.3.2 Elemento de proceso

El sistema *hardware* a desarrollar consta de cuatro EPs ya que el estudio final de la factorización LU (sección 5.4.1.3) indica que deben existir cuatro procesos a realizar en paralelo. Cada EP consta de una unidad de proceso y de una unidad de comunicación. La unidad de proceso está formada por: un divisor, un multiplicador y un restador (figura 7.11). El primero es utilizado para actualizar los elementos de la columna $k-1$ cuando se está en la iteración k del proceso paralelo. El multiplicador y el restador se utilizan para calcular los valores de la columna propiedad de cada EP.

Asimismo, cada EP posee inicialmente una columna formada por cuatro elementos

(el EP^j almacena los datos de la columna j -$A_{i,j}$ siendo $0 \leq i < n$-) que se deben almacenar en la memoria de proceso (M). Los cálculos a realizar sobre estos elementos utilizan los datos $A_{k,k}$ y $A_{k,j}$ como valores fijos (según el algoritmo secuencial), por lo que estos deben almacenarse en registros independientes (RAkk, RAkj). Por otro lado, en cada iteración del proceso paralelo se debe comunicar la columna *k-ésima*, desde el propietario al resto de EPs. La unidad de comunicación de cada EP debe almacenar estos valores de forma independiente, por lo que hay que utilizar una memoria de entrada/salida de datos (ML) con capacidad para una columna completa.

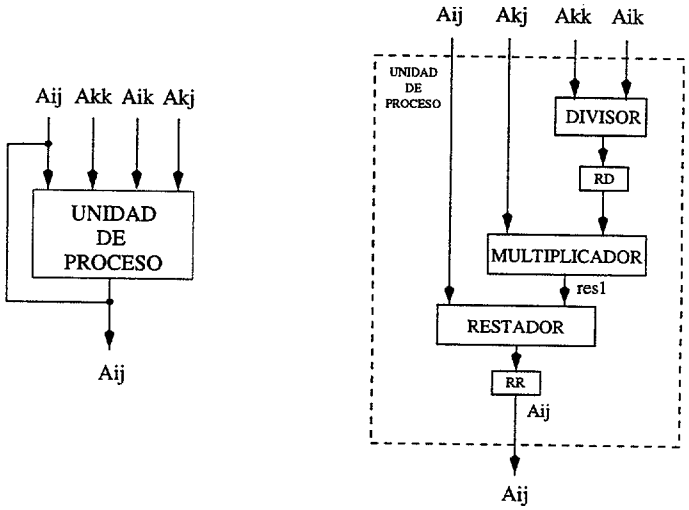


Figura 7.11: Unidad de proceso

7.3.3 Esquema final

El sistema completo se puede observar en la figura 7.12, en el que se han añadido los multiplexores y registros intermedios para seleccionar las señales de salida oportunas. Una vez establecidas las distintas unidades *hardware* que ejecutan el proceso paralelo, hay que desarrollar la unidad de control que las gobierna.

La unidad de control, en el estado E_Inicial debe generar las señales necesarias para almacenar los datos de la matriz A en las memorias de proceso de cada EP. Para ello, esta unidad tiene que direccionar las memorias asociadas a estos elementos, en un orden determinado; y debe generar las señales de escritura necesarias.

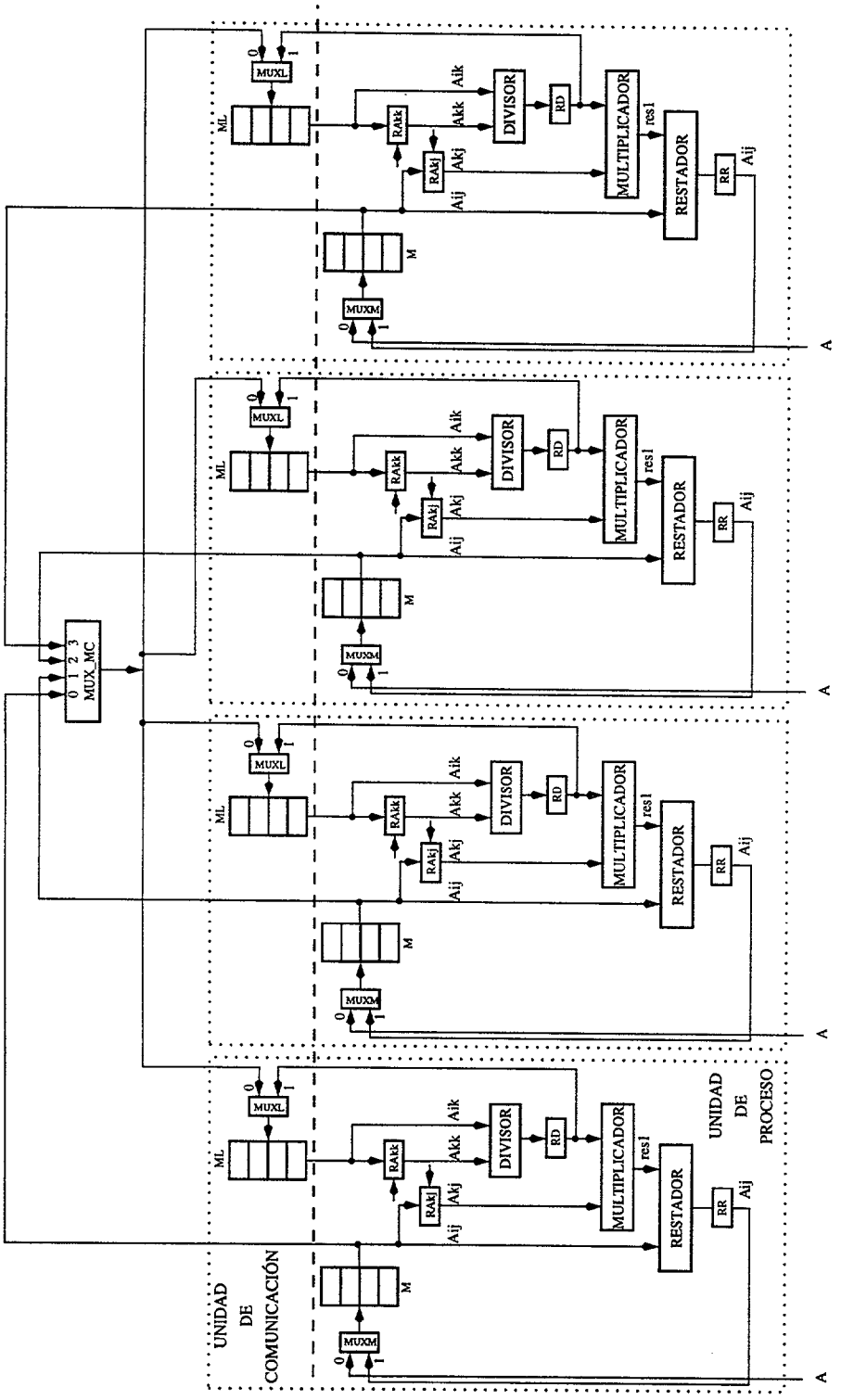


Figura 7.12: Sistema *hardware* de la factorización *LU*

Símbolo	*Explicación*
min	Bucle más interno que indexa al conjunto dimensionado.
n_b_θ	Número de iteraciones en las que a un bloque de elementos diferentes y consecutivos del conjunto dimensionado se les aplica el mismo vector de desplazamiento.
$n\_b_{\neq\theta}$	Número de vectores de desplazamiento diferentes que se pueden aplicar sobre un elemento del conjunto dimensionado.
n_d	Número de iteraciones en las que un mismo elemento del conjunto dimensionado es referenciado de forma consecutiva.
n_r	Número de bloques de bucles representativos que no indexan al conjunto dimensionado.
n_s	Número de iteraciones en las que un bloque de elementos del conjunto dimensionado es referenciado de forma no consecutiva.
n_sec	Número de iteraciones en las que una misma secuencia de n_θ vectores de desplazamiento se repite.
n_θ	Número de vectores de desplazamiento diferentes.
n_θ_i	Número de iteraciones en la que $desp_i$ se mantiene constante.
$orden$	Peso del hiperarco del GDM.
$\vec{p}$	Punto del $EI(I,n)$.
PS	Plano de superpuntos.
$S_1\delta S_2$	Dependencia de flujo de S_2 respecto de S_1.
s_max	Bucle más externo que asigna valor inicial a un desplazamiento.
SP_{f,c_f}	Agrupación de puntos del $EI(I,n-1)$.
V_{I_f,c_f}	Vértice del GDM correspondiente al valor c_f del bucle f.
$V_{\vec{p}}$	Hipervértice del GDM sobre el que se proyecta el punto $\vec{p}$ del $EI(I,n)$.
WR_X	Señal de escritura de la memoria X.
z_min	Bucle más interno que modifica el valor de un desplazamiento.

```
(*Estado E_Inicial*)
E_MUXM ← 0
WR_M ← SET
do ep=0,3,1
   do dir=0,3,1
      M[dir] ← MUXM
   enddo
enddo
E_MUXM ← 1
WR_M ← RESET
```

```
(*Estado E_Cm*)
WR_ML ← SET
E_MUXL ← 0
E_MUX_MC ← 0
doall ep=0,3,1
   do dir=0,3,1
      ML[dir] ← M[dir]
   enddo
enddoall
WR_ML ← RESET
```

```
(*Estado E_Sp*)
doall ep=0,3,1
   do k=1,2,1
      Akk ← ML[k-1];Akj ← M[k-1]
      do dir=k,3,1
         RD ← DIVI(ML[dir],Akk)
         E_MUXL ← 1;WR_ML ← SET
         ML[dir] ← RD;WR_ML ← RESET
         res1=MULTI(RD,Akj)
         RR ← REST(M[dir],res1)
         WR_M ← SET;M[dir] ← RR;WR_M ← RESET
      enddo
      WR_ML ← SET;E_MUXL ← 0;E_MUX_MC ← k
      do dir=0,3,1
         ML[dir] ← M[dir]
      enddo
      WR_ML ← RESET
   enddo
enddoall
```

```
(*Estado E_Cl*)
Akk ← ML[2];Akj ← M[2]
RD ← DIVI(ML[3],Akk)
E_MUXL ← 1;WR_ML ← SET
ML[3] ← RD;WR_ML ← RESET
res1=MULTI(RD,Akj)
RR ← REST(M[3],res1)
WR_M ← SET;M[3] ← RR;WR_M ← RESET
```

Tabla 7.5: Descripción RTL de la factorización LU

En el estado E_Cm, la unidad de control debe generar las instrucciones de escritura y direccionamiento para que el EP propietario de la columna 0 comunique estos datos al resto de EPs, (almacenándose esta columna en cada memoria ML). En el estado E_Sp, y para cada iteración k (establecida a partir del reloj del sistema), se debe

producir la comunicación de los datos desde el EP propietario de la columna k al resto (paso de comunicación solapada) y el cálculo de la columna propia de cada EP (fase de cálculo solapado). El solapamiento en este estado no es total, como ocurre en el ejemplo anterior, debido a las características de este problema. Antes de que comience la siguiente iteración, el resultado parcial $A_{i,j}$ y el nuevo dato $A_{i,k}$ deben estar estables en los registros correspondientes (RR y RD respectivamente). En este estado, la unidad de control debe proveer las señales de lectura, escritura y el direccionamiento de la memoria. Por último, en el estado E_Cl, se realiza el mismo proceso que en el estado anterior, ya que aunque únicamente sea el EP propietario de la última columna el que debe realizar el cálculo, no se consume tiempo adicional si el resto de EPs también calculan.

En la tabla 7.5 se presenta la descripción RTL de un posible autómata de control para este ejemplo. En la iteración k de este algoritmo, los datos de la factorización LU se almacenan en la matriz ML, por tanto, antes de introducir los datos de la iteración $k+1$ hay que comunicar los valores de la iteración k al exterior.

Las señales E_{MUXM}, WR_M, WR_{ML}, E_{MUXL}, E_{MUX_MC} y el direccionamiento de las memorias son comunes a todos los EPs (en la tabla 7.6 se presentan los diez primeros estados que pueden tomar estas señales).

ciclo	E_{MUXM}	WR_M	WR_{ML}	E_{MUXL}	E_{MUX_MC}	*dirección*
1	0	1	0	X	X	0
2	0	1	0	X	X	1
3	0	1	0	X	X	2
4	0	1	0	X	X	3
5	1	0	1	0	0	0
6	1	0	1	0	0	1
7	1	0	1	0	0	2
8	1	0	1	0	0	3
9	1	0	0	0	0	X
10	1	0	1	1	0	1

Tabla 7.6: Estado de los diez primeros ciclos de las señales del sistema *hardware* de la factorización LU

7.3.4 Generalización del tamaño del problema planteado

El estudio de la generalización del problema se hace para dos aproximaciones.

7.3.4.1 Primera aproximación

La primera solución al problema surge de la distribución de datos explicada en la sección 5.4.1.3. Al implementar en *hardware* este algoritmo, siendo $n > 4$, existen problema físicos que impiden que el número de EPs aumente indefinidamente. Debido a esto, se plantea una solución al problema utilizando 4 EPs ($nproc = 4$), aunque la dimensión de la matriz sea superior.

La distribución de datos en los EPs se realiza en bloques de dimensión $g = n/nproc$ de columnas no consecutivas. Cada EP posee una unidad de proceso que consta de un divisor, un multiplicador y un restador tal y como se presenta en la figura 7.11. Asimismo, cada EP posee una memoria de proceso (M) con capacidad para almacenar los elementos de su submatriz ($\frac{n}{nproc} \times n$) y una memoria de entrada/salida (ML) para almacenar los elementos a comunicar entre EPs (de dimensión n). En la tabla 7.7 se muestra un posible autómata de control para esta aproximación.

7.3.4.2 Segunda aproximación y estudio de la eficiencia

Analizando el proceso paralelo a realizar y el sistema *hardware* final (figura 7.12), se puede observar que los EPs almacenan los mismos datos en su memoria ML. Según esto, se puede mejorar el esquema inicial si se reduce el número de memorias de entrada/salida. Así, los elementos de la columna utilizada en una iteración dada (columna k) deben almacenarse en una variable que tiene que ser diferente a la que almacena la columna a comunicar (columna $k+1$), por lo que deben existir dos variables de tipo vector (variable de proceso y variable de comunicación) dentro del proceso paralelo. Estas dos variables, que almacenan la misma información en todos los EPs, se corresponden con dos memorias de dimensión n (ML0 y ML1) dentro del sistema *hardware*. Asimismo, los datos de la matriz A se distribuyen de forma cíclica en las memorias M, de forma que a cada EP le corresponden $\frac{n}{nproc}$ columnas de A.

```
(*Estado E_Inicial*)
E_MUXM ← 0
WR_M ← SET
do ep=0,3,1
   do dir=0,(g×n)-1,1
      M[dir] ← MUXM
   enddo
enddo
E_MUXM ← 1
WR_M ← RESET
```

```
(*Estado E_Cm*)
WR_ML ← SET
E_MUXL ← 0
E_MUX_MC ← 0
doall ep=0,3,1
   do dir=0,n-1,1
      ML[dir] ← M[dir]
   enddo
enddoall
WR_ML ← RESET
```

```
(*Estado E_Sp*)
doall ep=0,3,1
   do k=1,n-2,1
      Akk ← ML[k-1]
      do dir=k,n-1,1
         RD ← DIVI(ML[dir],Akk)
         E_MUXL ← 1;WR_ML ← SET
         ML[dir] ← RD;WR_ML ← RESET
         do col=0,g-1,1
            Akj ← M[col×n+k-1];res1=MULTI(RD,Akj)
            RR ← REST(M[col×n+dir],res1)
            WR_M ← SET;M[col×n+dir] ← RR
            WR_M ← RESET
         enddo
      enddo
      WR_ML ← SET;E_MUXL ← 0;E_MUX_MC ← k
      do dir=0,n-1,1
         ML[dir] ← M[dir+(k div nproc)×n]
      enddo
      WR_ML ← RESET
   enddo
enddoall
```

```
(*Estado E_Cl*)
Akk ← ML[n-2];RD ← DIVI(ML[n-1],Akk)
E_MUXL ← 1;WR_ML ← SET;ML[n-1] ← RD
WR_ML ← RESET
do col=0,g-1,1
   Akj ← M[col×n+n-2];res1=MULTI(RD,Akj)
   RR ← REST(M[col×n+n-1],res1)
   WR_M ← SET;M[col×n+n-1] ← RR;WR_M ← RESET
enddo
```

Tabla 7.7: Descripción RTL de la factorización LU (1^a aprox.)

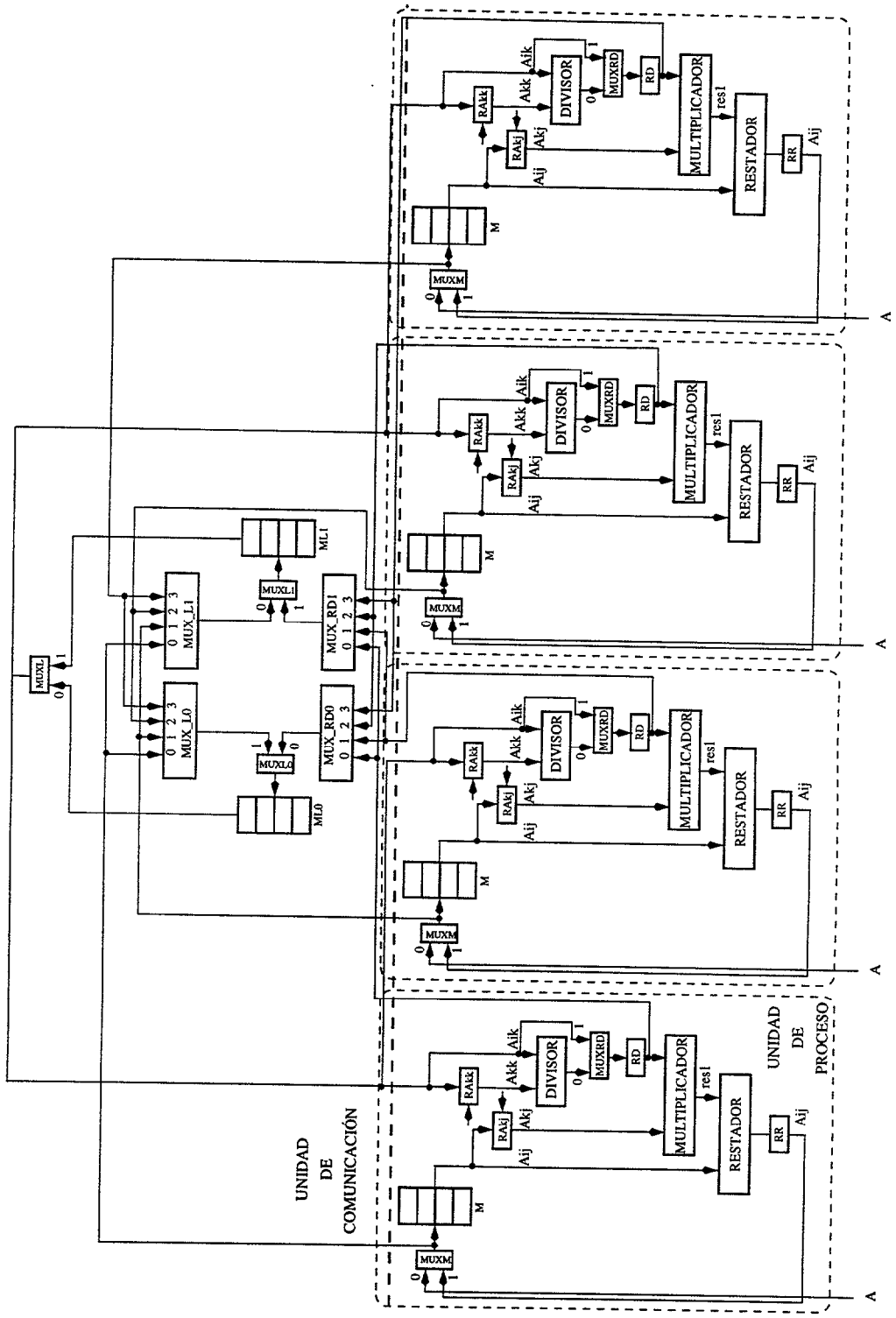


Figura 7.13: Esquema *hardware* de la factorización LU (2^a aprox.)

En la figura 7.13 se presenta esta modificación en la que se puede observar la inclusión de las memorias de entrada/salida. En una iteración k, los EPs están utilizando para el proceso la memoria M y una de las memorias de entrada/salida (ML0 o ML1). Esta memoria se carga con el resultado de la división a través de los multiplexores MUX_RD0 y MUX_RD1. La otra memoria de entrada/salida (ML1 o ML0) se utiliza para la comunicación, de forma que el EP que tiene que comunicar la columna k, modifica esta columna (en función de la $k-1$) antes de actualizar el resto de columnas que posee. Una vez que esta columna queda actualizada, el EP propietario la almacena en la memoria de entrada/salida correspondiente de forma que puede ser utilizada en la siguiente iteración (a través de los multiplexores MUX_L0 y MUX_L1).

El autómata de control que gobierna este sistema tiene tres fases diferentes dentro del estado E_Sp: cálculo de la columna $k-1$ y actualización de la columna k en función de la columna $k-1$ (fase 0); comunicación de la columna k que se hace en paralelo con el anterior (fase 1) y actualización del resto de columnas en función de la columna $k-1$ (fase 2).

En las tablas 7.8 y 7.9 se presenta un posible autómata de control que gobierna esta nueva aproximación (el estado E_Cl final se incluye en el estado E_Sp). En la tabla 7.10 se muestra el estado de algunas señales de escritura y de habilitación de los multiplexores en cada una de las fases mencionadas, para las cuatro primeras iteraciones del algoritmo paralelo.

(*Estado E_Inicial*)	(*Estado E_Cm*)
$E_{MUXM} \leftarrow 0$	$E_{MUXL0} \leftarrow 1$
$WR_M \leftarrow$ SET	$E_{MUX\_L0} \leftarrow 0$
do ep=0,3,1	$WR_{ML0} \leftarrow$ SET
do dir=0,(g×n)-1,1	doall ep=0,3,1
M[dir] ← MUXM	do dir=0,n-1,1
enddo	ML0[dir] ← M[dir]
enddo	enddo
$E_{MUXM} \leftarrow 1$	enddoall
$WR_M \leftarrow$ RESET	$WR_{ML0} \leftarrow$ RESET

Tabla 7.8: Descripción RTL de la factorización LU -2^a aprox.- (Parte I)

```
(*Estado E_Sp*)
doall ep=0,3,1
   do k=0,n-2,1
      E_MUX_L0 ← (k+1) mod nproc;E_MUX_L1 ← (k+1) mod nproc
      E_MUXL0 ← k mod 2;E_MUXL1 ← k mod 2
      E_MUX_RD0 ← k mod nproc;E_MUX_RD1 ← k mod nproc
      E_MUXL ← k mod 2;RAkk ← MUXL
      col=(k+1) div nproc;RAkj ← M[col×n+k];E_MUXRD ← 0
      do dir=k+1,n-1,1
         WR_ML0 ← (k+1) mod 2;WR_ML1 ← ¬((k+1) mod 2)
         RD ← DIVI(Aik,Akk);MUX_RD0 ← RD
         MUX_RD1 ← RD;res1=MULTI(RD,Akj)
         RR ← REST(M[col×n+dir],res1)
         WR_M ← SET;M[col×n+dir] ← RR;WR_M ← RESET
         WR_ML0 ← ¬((k+1) mod 2);WR_ML1 ← (k+1) mod 2
      enddo
      E_MUXRD ← 1
      do dir=k+1,n-1,1
         RD ← Aik
         do col=0,(k+1) div nproc-1,1
            RAkj ← M[col×n+k];res1=MULTI(RD,Akj)
            RR ← REST(M[col×n+dir],res1)
            WR_M ← SET;M[col×n+dir] ← RR;WR_M ← RESET
         enddo
         do col=(k+1) div nproc+1,1
            RAkj ← M[col×n+k];res1=MULTI(RD,Akj)
            RR ← REST(M[col×n+dir],res1)
            WR_M ← SET;M[col×n+dir] ← RR;WR_M ← RESET
         enddo
      enddo
   enddo
enddoall
```

Tabla 7.9: Descripción RTL de la factorización LU -2^a aprox.- (Parte II)

k	WR_{ML0}	E_{MUX_L0}	E_{MUXL0}	E_{MUX_RD0}	WR_{ML1}	E_{MUX_L1}	E_{MUXL1}	E_{MUX_RD1}	E_{MUXL}
1_{fase0}	1	X	0	0	0	X	X	X	0
1_{fase1}	0	X	X	X	1	1	0	X	0
1_{fase2}	0	X	X	X	0	X	X	X	0
2_{fase0}	0	X	X	X	1	X	1	1	1
2_{fase1}	1	2	1	X	0	X	X	X	1
2_{fase2}	0	X	X	X	0	X	X	X	1
3_{fase0}	1	X	0	2	0	X	X	X	0
3_{fase1}	0	X	X	X	1	3	0	X	0
3_{fase2}	0	X	X	X	0	X	X	X	0
4_{fase0}	0	X	X	X	1	X	1	3	1
4_{fase1}	1	4	1	X	0	X	X	X	1
4_{fase2}	0	X	X	X	0	X	X	X	1

Tabla 7.10: Estados de algunas señales del sistema *hardware* (2^a aprox.)

La eficiencia en este problema, podemos considerarla como la relación:

$$\eta = \frac{N_{ops}}{N_{opp} \times nproc}$$

donde N_{ops} es el número de operaciones realizadas en la implementacion secuencial. N_{opp} es el número de operaciones realizadas por cada EP, y $nproc$ es el número de EPs.

El número de operaciones puede ser considerado como el número de iteraciones totales de los bucles del algoritmo. No obstante, en este caso particular, fuera del bucle central se realiza una división, que puede ser costosa en tiempo y recursos. Por esta razón se ha considerado como una operación atómica. Asimismo, suponemos que puede ser realizada en una unidad de tiempo, equivalente al conjunto de resta y multiplicación. Con esta consideración, el número de ciclos para el algoritmo secuencial es:

$$N_{ops} = \sum_{k=0}^{n-2}(n-k) \times (n-k+1) = \frac{n^3 - n}{3}$$

De igual forma, en el desarrollo paralelo hay que contar el número de operaciones por ciclo, que corresponden con:

$$\begin{cases} E\_Inicial &= \frac{n^2}{nproc} \\ E\_Cm &= n \\ E\_Sp &= (n+1) + \left(2 + \frac{n}{nproc}\right) + \left(\frac{n^2-n}{2}\right) \end{cases}$$

En base a estos datos, se obtiene las curvas de eficiencia mostradas en la figura 7.14 (siendo $nproc = 4$). Se observa que el resultado tiende hacia un 66% de eficiencia, con un punto de inflexión sobre $n = 30$. Para valores altos de n, tiene un crecimiento asintótico hacia $\eta = 2/3$. En cualquier caso, el tiempo de comunicación de una columna de la matriz A, se puede solapar totalmente con los cálculos realizados, ya que se comunica un elemento en paralelo con el cálculo de una resta y una multiplicación (más una división en algunos casos).

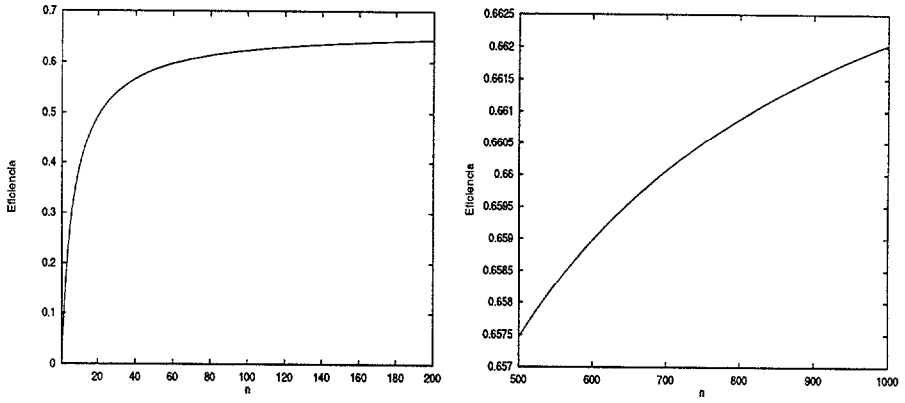


Figura 7.14: Medida de la eficiencia dependiente de la dimensión de la matriz (n)

7.4 Análisis hardware de un problema del álgebra dispersa

Los problemas resueltos en las secciones anteriores utilizan matrices densas, sin embargo, nuestra metodología se puede aplicar a problemas del álgebra dispersa. En este apartado se presenta el algoritmo secuencial de la multiplicación matriz-vector dispersa y su posterior análisis utilizando dicha metodología. Una vez generado el algoritmo paralelo se propone un sistema *hardware* que resuelve el problema planteado. Un posible algoritmo secuencial para el cálculo de la multiplicación matriz-vector (siendo la matriz dispersa y el vector denso) se presenta a continuación:

```
do j=0,n-1,1
   n_ele=Columna_j.n_ele
   A=Columna_j →punt
   do i=0,n_ele-1,1
      c_{A_i.fila} = c_{A_i.fila} + A_i.valor × b_j
   enddo
enddo
```

Cada columna de la matriz A está almacenada en un vector de dimensión igual al número de elementos de esa columna. Los datos que se guardan en cada entrada del vector son: el valor y la fila donde se encuentra el dato (en la figura 7.15 se muestra un ejemplo de matriz A y su almacenamiento). Asimismo, se supone que el vector resultado c almacena inicialmente el valor cero.

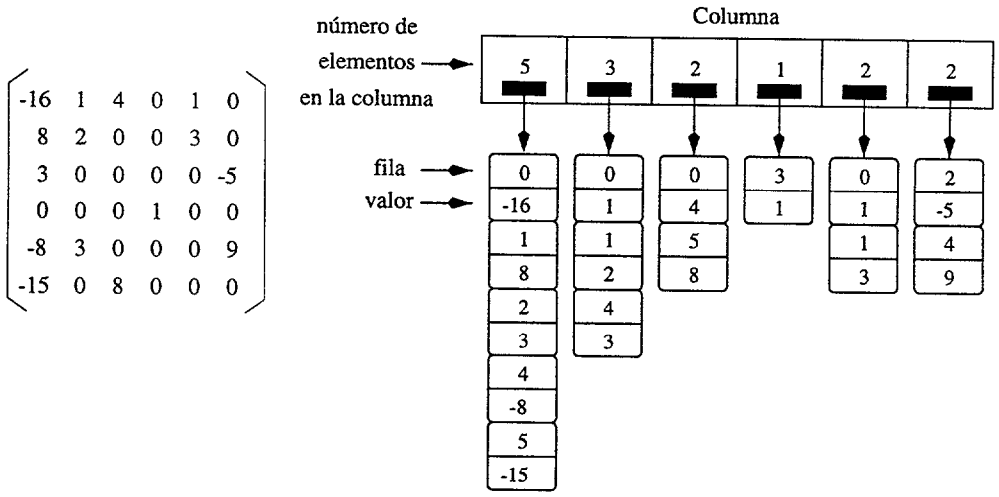


Figura 7.15: Ejemplo de matriz dispersa y su almacenamiento ($n = 6$)

Analizando el GDM del algoritmo anterior suponiendo que $n\_ele = n = 4$ para todas las columnas (figura 7.16) observamos que existe cuatro procesos que se pueden realizar en paralelo (en el GDM existen cuatro colores -pesos-) con una distribución inicial de datos tal que, el EP^x $(0 \leq x \leq 3)$ almacena los siguientes datos: c_x, $A_{x,t}$ y b_t $(0 \leq t \leq 3)$. Según la distribución de datos anterior, los EPs almacenan datos diferentes del vector c y de la matriz A, pero los datos del vector b se replican. Para evitar esta replicación se distribuye este vector entre los distintos EPs tal que el EP^x almacena exclusivamente el elemento b_x, necesitándose comunicaciones entre ellos para

realizar todos los cálculos.

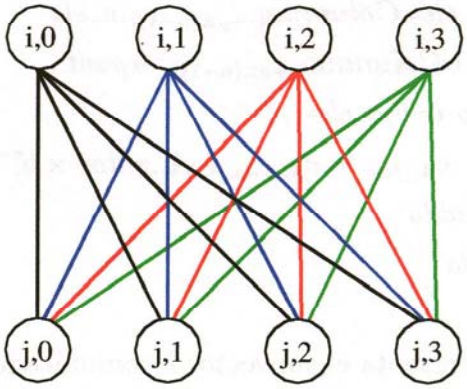


Figura 7.16: *GDM* del algoritmo de la multiplicación matriz-vector ($n = 4$)

Si generalizamos este problema para cualquier valor de n, los datos de los vectores y la matriz se deben distribuir en bloques de filas de tamaño $g = n/nproc$ (siendo $nproc$ el número de EPs). Por tanto, cada EP almacena g datos de los vectores c y b, y $\frac{n}{nproc} \times n$ elementos de la matriz A (si ésta fuera una matriz densa). Teniendo en cuenta que la matriz A es dispersa, en las memorias sólo almacenan los datos distintos de cero, utilizando una estructura de almacenamiento similar a la presentada en la figura 7.15.

Con esta distribución inicial de datos y los requisitos de comunicaciones explicados, el proceso paralelo a realizar por un EP genérico x es el siguiente:

```
do t=0,n-2,1
  par
    comunicar b^t
    do j=0,g-1,1
      n_ele=Columna_(j+g×t).n_ele
      A=Columna_(j+g×t) →punt
      do i=0,n_ele-1,1
        c_(A_i.fila) = c_(A_i.fila) + A_i.valor × b^t_j
      enddo
    enddo
  endpar
enddo
```

do j=0,g-1,1
  n_ele=Columna$_{(j+g\times(n-1))}$*.n_ele*
  A=Columna$_{(j+g\times(n-1))}$ $\rightarrow$*punt*
  do i=0,n_ele-1,1
    $c_{A_i.fila} = c_{A_i.fila} + A_i.valor \times b_j^{n-1}$
  enddo
enddo

En este algoritmo, b^t representa el subvector b comunicado en el instante t (para el EP^0 es el bloque 0, para el EP^1 es el 1 y así sucesivamente). En la siguiente iteración en t, el EP^0 comunicará el subvector 1, el EP^1 el subvector 2, etc., es decir, es una comunicación como la explicada en la sección 7.2.

7.4.1 Unidad de Control

Teniendo en cuenta el esquema del algoritmo paralelo del que se parte (tabla 7.1), el desarrollo *hardware* debe tener dos estados de funcionamiento: E_Sp y E_Cl. Asimismo, es necesario un estado E_Inicial en el cual se cargan los datos iniciales según el análisis realizado utilizando la metodología.

En el estado E_Sp, y en una iteración t, todos los EPs deben comunicar su subvector b a uno de sus vecinos y realizar un cálculo submatriz-subvector. Antes de que comience la iteración $t+1$, los nuevos elementos del vector b están almacenados en memoria dispuestos para su utilización en un nuevo producto submatriz-subvector. En el estado E_Cl, todos los EPs realizan el cálculo final utilizando los datos del subvector recibidos en la iteración $t = n-2$.

7.4.2 Elemento de proceso

El sistema *hardware* a desarrollar está formado por cuatro EPs, ya que por problemas físicos, si la matriz A es muy grande no se puede disponer de n elementos. Cada EP consta de la unidad de proceso y de la unidad de comunicación. La unidad de proceso está formada por: un multiplicador y un sumador (figura 7.17). El multiplicador se

utiliza para realizar los cálculos parciales que a continuación se añaden, mediante el sumador, al vector c.

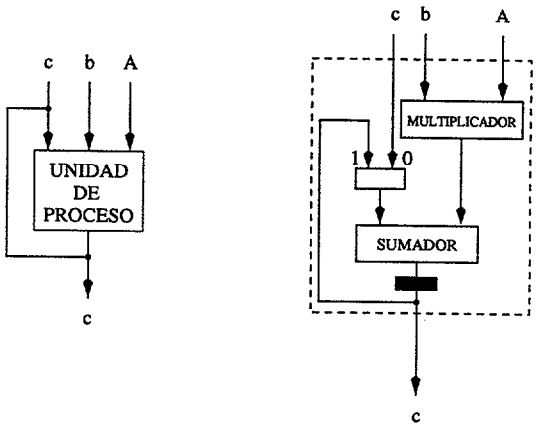


Figura 7.17: Unidad de proceso

La unidad de comunicación se encarga de realizar las comunicaciones de los elementos del subvector b entre EPs vecinos (almacenándolos en las memorias de entrada/salida). Por otro lado, hay que almacenar los datos del subvector c (g elementos por EP) y de la submatriz A en memorias internas que corresponden con la memoria de proceso. El almacenamiento de la submatriz A se debe realizar teniendo en cuenta que no sólo hay que guardar el dato, sino que además hay que guardar su fila y columna. Debido a esto, el almacenamiento de la submatriz va a consistir de tres memorias: una memoria (Mcol) de tamaño n para almacenar el número de elementos que posee el EP en cada columna y dos memorias (MA y Mfila) de tamaño $\frac{n}{nproc} \times n$ para almacenar los datos y las filas respectivamente (en la figura 7.18 se muestran las memorias comentadas para el EP^0, si la matriz es la de la figura 7.15).

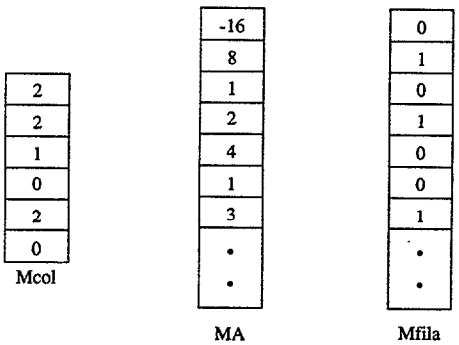


Figura 7.18: Ejemplo de datos almacenados en las memorias del EP^0

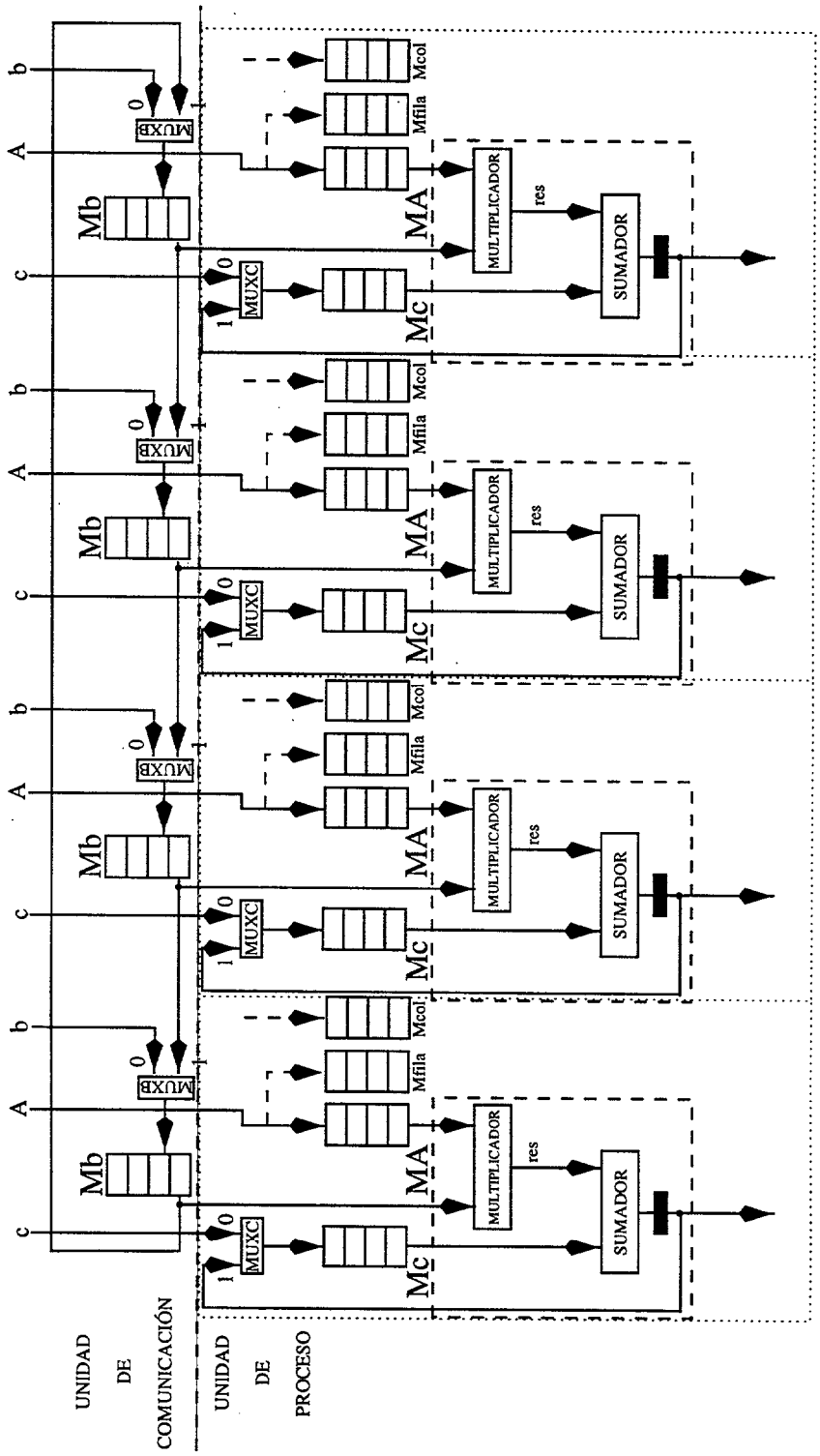


Figura 7.19: Sistema *hardware* de la multiplicación matriz-vector dispersa

```
(*Estado E_Inicial*)
E_MUXC ← 0;E_MUXB ← 0
WR_Mb ← SET
WR_Mc ← SET
WR_MA ← SET
WR_Mcol ← SET
WR_Mfila ← SET
do ep=0,3,1
   do dir=0,n-1,1
      Mc[dir] ← MUXC
      Mb[dir] ← MUXB
      Mcol[dir] ← ...
   enddo
   do dir=0,g×n-1,1
      MA[dir] ← ...
      Mfila[dir] ← ...
   enddo
enddo
E_MUXC ← 1
E_MUXB ← 1
WR_Mb ← RESET
WR_Mc ← RESET
WR_MA ← RESET
WR_Mcol ← RESET
WR_Mfila ← RESET
```

```
(*Estado E_Sp*)
doall ep=0,3,1
   inicial=0
   do t=0,n-2,1
      do j=0,g-1,1
         n_ele=Mcol[j+g×t]
         do dir=inicial,inicial+n_ele-1,1
            fila=Mfila[dir]
            res=MULTI(MA[dir],Mb[j])
            WR_Mc ← SET
            Mc[fila] ← SUM(Mc[fila],res)
            WR_Mc ← RESET
         enddo
         inicial=inicial+n_ele
         WR_Mb ← SET
         Mb[j] ← MUXB;WR_Mb ← RESET
      enddo
   enddo
enddoall
```

```
(*Estado E_Cl*)
doall ep=0,3,1
   do j=0,g-1,1
      n_ele=Mcol[j+g×(n-1)]
      do dir=inicial,inicial+n_ele-1,1
         fila=Mfila[dir]
         res=MULTI(MA[dir],Mb[j])
         WR_Mc ← SET
         Mc[fila] ← SUM(Mc[fila],res)
         WR_Mc ← RESET
      enddo
   enddo
enddoall
```

Tabla 7.11: Descripción RTL de la multiplicación matriz-vector dispersa

7.4.3 Esquema final

El sistema completo se puede observar en la figura 7.19, en el que se han añadido los multiplexores y registros intermedios para seleccionar los datos adecuados en cada estado.

Una vez establecidas las distintas unidades *hardware* que ejecutan el algoritmo paralelo, hay que desarrollar la unidad de control. En el estado E_Inicial se deben generar las señales necesarias para almacenar los datos de la matriz A (así como su fila y columnas) y de los vectores b y c en las memorias de proceso de cada EP. Para ello, esta unidad tiene que direccionar las memorias asociadas a estos elementos, en un orden determinado; y debe generar las señales de escritura necesarias. En el estado E_Sp, y para cada iteración t (que puede ser establecida por el reloj del sistema o por una señal obtenida a partir de este), se debe producir la comunicación de los datos de cada EP a uno de sus vecinos, lo que constituye el paso de comunicación solapada y el cálculo de la submatriz por el subvector en cada EP (fase de cálculo solapado). En este estado se produce un solapamiento total, ya que una vez ha finalizado el cálculo con un elemento del subvector b (almacenado en Mb), se sobreescribe el nuevo dato desde un vecino. En este estado, la unidad de control debe proveer las señales de lectura, escritura y el direccionamiento de las memorias. Por último, en el estado E_Cl se realiza el mismo proceso que en el estado anterior, exceptuando el paso de comunicación que ya no es necesario. En la tabla 7.11 se presenta la descripción RTL de los tres estados, de un posible autómata de control para este ejemplo.

Capítulo 8

Conclusiones y trabajo futuro

8.1 Conclusiones

Un objetivo fundamental de la paralelización de algoritmos secuenciales es la distribución eficiente de datos en los procesadores de la arquitectura paralela. Esta distribución define las operaciones a realizar por parte de los procesadores de la máquina paralela. Asimismo, la distribución de datos es fuertemente dependiente de las relaciones existentes entre los elementos del conjunto dimensionado a distribuir. Estas relaciones, también llamadas dependencias de datos, indican no sólo dónde almacenar los datos sino la necesidad de comunicaciones entre procesadores.

Ante este panorama, la existencia de herramientas de ayuda a la programación paralela, desde el punto de vista de la distribución de datos, no sólo es útil sino que es necesaria. Estas herramientas, más que hacer el trabajo de forma automática (independiente del programador), deben ofrecer alternativas o ideas de cómo hacer la distribución. Por otro lado, la ditribución automática es una tarea bastante compleja ya que ésta siempre depende del tipo de problema al que nos enfrentemos. Según esto, la experiencia adquirida por los programadores en el desarrollo de su trabajo es básica a la hora de distribuir los datos, si se quieren conseguir unos resultados que hagan que la utilización de arquitecturas paralelas merezca la pena.

El primer requisito que debe cumplir una herramienta de ayuda a la programación

es que sea gráfica. El programador necesita "ver" dónde deben ser utilizados los datos a lo largo del código. Desde nuestro punto de vista, el uso del "color" para indicar las relaciones entre los datos es fundamental. Si observamos un gráfico en el que ciertos datos están marcados con un mismo "color", intuitivamente podemos suponer que tienen algún tipo de relación. Otro requisito importante de estas herramientas es que sea aplicable a cualquier tipo de algoritmos. En este caso, cualquiera que sean las dependencias entre los datos deben ser representables de forma sencilla.

Con el objetivo de resolver el problema de la distribución de datos, de forma que el usuario programador pueda analizar visualmente dónde se deben almacenar inicialmente los datos y cómo deben ser comunicados posteriormente, se ha desarrollado una herramienta de distribución de datos y comunicaciones. Esta herramienta constituye el núcleo de la metodología que aplicamos a los algoritmos numéricos. La paralelización de estos algoritmos incluye, no sólo la distribución inicial de datos, sino que gracias a esta distribución nos ayuda a solapar las comunicaciones con los cálculos, ocultando así la latencia producida al comunicar.

La metodología desarrollada consta de tres pasos:

- *Análisis del algoritmo secuencial* construyendo el grafo de dependencias modificado (*GDM*).
- *Distribución inicial de datos* en base al GDM.
- *Establecimiento de las comunicaciones* (redistribución de datos) en base al GDM.

Las ventajas de nuestra metodología son:

1. Presenta una forma sencilla de descubrir el paralelismo de las aplicaciones numéricas tanto si existen dependencias uniformes como no uniformes. Para ello se construye un *grafo de dependencias* (*GDM*) que tiene las siguientes características:
   - Existe un grafo por cada dependencia existente en el código.
   - Es independiente del número de bucles que encierran a la dependencia.
   - Tiene forma planar, con tantas filas como bucles encierran a la dependencia y tantas columnas como posibles valores tengan los índices de estos bucles.

- Permite descubrir el máximo paralelismo de forma visual, asignando pesos a los arcos.
- Cada hiperarco (conjunto de arcos) del grafo presenta dos pesos: "color" y "orden". El "color" asocia a todos los datos que deben almacenarse en la misma memoria por ser dependientes entre ellos. El "orden" indica qué dato de un mismo "color" debe actualizarse antes que otro.
- Cada "color" corresponde con un procesador. Si en la arquitectura paralela existen menos procesadores que colores en el grafo, se debe redistribuir la información tratando de evitar las comunicaciones y asegurando el mejor balanceo posible.
- Los datos relacionados con arcos de distinto "color" se pueden actualizar en paralelo. Datos relacionados con arcos del mismo "color" se deben realizar secuencialmente según el "orden" del arco. En cualquier caso, si existen arcos con el mismo "color" y "orden", los datos asociados no dependen entre ellos (dependen de un tercer dato), por lo que pueden actualizarse a la vez.
- Representa cualquier tipo de relación entre los datos (dependencias uniformes y no uniformes).
- Representa las posibles comunicaciones de datos existentes en el código. La comunicación siempre se realiza desde el procesador que tiene el dato a modificar hacia el procesador que tiene los datos necesarios.
- No es una herramienta automática de distribución de datos, sino que es un herramienta de ayuda a la distribución de datos. Informa al programador cómo debe distribuir los datos para asegurar un número mínimo de comunicaciones. De esta forma, la distribución inicial se hace colocando aquellos elementos dependientes en la misma memoria. Si, posteriormente existe dependencia entre elementos que no están en la misma memoria, debe haber comunicación de datos. La herramienta indica de qué procesador debe partir el dato y a qué procesador debe llegar.

2. Permite ahorrar tiempo de diseño de los algoritmos paralelos. Por un lado, el programador no necesita utilizar herramientas como el espacio de iteraciones y

la representación de las dependencias sobre este espacio (representación limitada a un cierto número de bucles). El *GDM* informa directamente al programador de la distribución de datos que asegura paralelismo máximo. Por otro lado, los algoritmos paralelos diseñados tienen la forma indicada en la tabla 2.1. La etapa de cálculo en estos algoritmos es la que se realiza en el algoritmo secuencial pero sobre un subconjunto de datos (cada procesador trabaja con su subconjunto). La etapa de comunicación depende de las dependencias existentes en el algoritmo o bien de la redistribución hecha para evitar replicaciones de datos. En este último caso, el programador decide no utilizar la distribución de datos indicada por el *GDM*.

3. Permite el solapamiento de comunicaciones y cálculos. El programador puede ver dónde distribuir inicialmente los datos y cómo se van a comunicar entre los distintos procesadores. El programador sabe que la primera actualización de los datos no necesita comunicaciones. Según esto, antes de hacer las siguientes actualizaciones (que pueden significar comunicaciones según indique el *GDM*) se pueden comunicar los datos necesarios a la misma vez que se hace el primer cálculo. Estas comunicaciones pueden ser punto a punto o *broadcast.*

4. Permite emplear la metodología para diseñar algoritmos paralelos en *hardware* o bien implementarlos en una máquina paralela. Esto es, la metodología de especificación es independiente de la metodología de implementación.

5. Nosotros hemos aplicado nuestra metodología a los problemas típicos del álgebra lineal tanto densa como dispersa con el objetivo de demostrar que es aplicable a los *benchmarks* típicos y comparable a los resultados obtenidos por otros autores. Asimismo, hemos obtenido la descripción del *hardware* ad-hoc para FPGAs, mostrando unos resultados de velocidad y de ocupación del silicio bastante buenos.

   Por otro lado hemos aplicado la metodología a problemas de la Ingeniería de Telecomunicación como son:

   - Diseño de estructuras microtiras, mostrando una mejora significativa del rendimiento del algoritmo paralelo frente a la ejecución secuencial, lo cual ha sido de especial utilidad para el grupo de Ingeniería de Comunicaciones

de nuestra Universidad, ya que les ha permitido evaluar sus estudios sobre diseño de antenas, filtros, etc.

- Estudio y mejora del algoritmo de simulación de la respuesta impulsional de un sistema de transmisión infrarrojo difuso que ha sido de utilidad para nuestro propio grupo de investigación (proyecto de investigación TIC98-C02-02-1115) así como para el Grupo de Tecnología Fotónica de nuestra Universidad y el de la Universidad Politécnica de Madrid.

8.2 Trabajo futuro

Utilizando como núcleo la herramienta de distribución de datos y comunicaciones queremos dirigir nuestros futuros esfuerzos en los siguientes puntos:

- Agrupar la información obtenida para cada dependencia de forma que la herramienta pueda proveer al programador de una información completa de cómo se relacionan las distintas dependencias entre sí. Hasta ahora, el programador utiliza la distribución de datos indicada por el GDM, de la dependencia más restrictiva. Una mejora futura está en que la herramienta estudie los $GDMs$ generados descargando al programador de esta tarea.

- Análisis de la replicación de datos en memoria de forma automática. La herramienta debe redistribuir los datos para evitar esta replicación, pero debe decidir qué parámetro desea optimizar (comunicaciones o balanceo de la carga).

- Presentación del movimiento de los datos sobre el conjunto dimensionado respecto a todas las dependencias (no particular para cada dependencia como hasta ahora).

- Desarrollar una herramienta completa y simple utilizable a través de una interfaz WEB que permita desarrollar programas paralelos de forma cooperativa y que tenga todas las características anteriores.

- Aplicar nuestra metodología a LANs inalámbricas estudiando un modelo de comunicación más completo desarrollando el estudio de parámetros que puedan influir en el solapamiento de los cálculos y las comunicaciones. Además es interesante estudiar nuevos tipos de algoritmos paralelos.

Bibliografía

[1] www.ac.upc.es/recerca/CAP/gsomk

[2] Allen J. R., Kennedy K.: *Automatic Translations of Fortran Programs to Vector Form.* ACM Toplas. (1987) 9:491–542

[3] www.analog.com/support/standard_linear/selection_guides/switch_select.html

[4] Arabi R. T., Murphy A. T., Sarkar T. K.: *Electric Field Integral Equation Formulation for a Dynamic Analysis of Nonuniform Microstrip Multi-Conductor Transmision Lines.* IEEE Trans. on Microwave, Theory Tech., vol. 40. (1992) 1857–1869

[5] Asenjo R., Zapata E. L.: *Sparse LU Factorization on the Cray T3D.* Int'l. Symp. on High-Performance Computing an Networking (HPCN). Milan, Italy. Springer-Verlag, LNCS 919. (1995) 690–696

[6] Ayguadé E., García J., Gironésa M., Labarta J., Torres J. y Valero M.: *Detecting and Using Affinity in an Automatic Data Distribution Tool.* In Languages and Compilers for Parallel Computing. Springer-Verlag. (1995)

[7] Banarjee P., Chandy J. A., et al.: *The Paradigm Compiler for Distributed-Memory Multicomputers.* IEEE Computer, vol. 28, no. 10. (1995)

[8] Banerjee U.: *Unimodular Transformation for Double Loops.* Proceeding of Third Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, CA. (1990)

[9] Banerjee U.: *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers. (1988)

[10] Banerjee U.: *Loop Transformation for Restructuring Compilers. The Foundations.* Kluwer Academic Publishers. (1993)

[11] Benkner S., Chapman B. Zima H.: *Vienna Fortran 90.* Proceedings of the 1992 Scalable High Performance Computing Conference. (1992)

[12] Bokari S. H.: *On Mapping Problem.* IEEE Transactions on Computers, vol. 30, no. 3. (1981)

[13] Boudet V., Rastello F., Robert Y.: *Alignment and Distribution Is Not (Always) NP-Hard.* Technical Report N. 98-30. Laboratoire de l'Informatique du Parallélisme. École Normale Supérieure de Lyon. (1998)

[14] Bruening U., Giloi W. K., Schroeder-Preikschat W.: *Latency Hiding in Message-Passing Architectures.* Proceedings of the 8th International Parallel Processing Symposium. (1994)

[15] Bull J. M.: *A Hierarchical Classification of Overheads in Parallel Programs.* Proceedings of First IFIP TC10 Intl. Workshop on Software Engineering for Parallel and Distributed Systems. (1996)

[16] Calvin C., Colombet L., Michallon P.: *Overlapping Techniques of Communications.* The International Conference and Exhibition of High Performance Computing and Networking (HPCN EUROPE 1995). (1995)

[17] Cooper K. D., Hall M. W., Hood R. T., Kennedy K., et al.: *The ParaScope Parallel Programming Enviroment.* The Proceedings of the IEEE, vol. 81. (1993) 244–263

[18] Cornard M., Trystram D.: *Parallel Algorithms and Architectures.* International Thomson Computer Press. (1995)

[19] cray.com/products/systems/crayt3e

[20] cray.com/products/systems/crayt90

[21] www.crpc.rice.edu/HPFF/home.html

[22] www.cs.umd.edu/~keleher/dsm.html

[23] Culler D., Arpaci-Dusseau A., et al.: *Parallel Computing on the Berkeley NOW.* 9th Joint Symposium on Parallel Processing. (1997)

[24] Culler D., Karp R., Patterson D., Sahay A., et al.: *LogP: Towards a Realistic Model of Parallel Computation.* In Fourth ACMSIGPLAN Symposium on Principles and Practice of Parallel Programming. (1993)

[25] Cypher R., Leu E.: *The Semantics of Blocking and Nonblocking Send and Receive Primitives.* Proceedings 1994 International Parallel Processing Symposium. (1994) 729–735

[26] d'Auriol B. J.: *Expressing Parallel Program Using Geometric Representation: Case Studies.* Proceedings of the IASTED International Conference PDCS'99. Massachusetts, USA. (1999) 985–990

[27] Davis T. A., Duff I. S.: *An Unsymmetric-Pattern Multifrontal Method for Sparse LU Factorization.* RAL-93-036. Department for Computation and Information. Atlas Center. Rutherford Appleton Laboratory. (1993)

[28] Demmel J. W., Eisenstat S. C., Gilbert J. R., Li X. S., Liu J. W.: *A Supernodal Approach to Sparse Partial Pivoting.* Technical Report CSD-95-883. UC Berkeley. (1995)

[29] Díaz de Cerio L., González A., Valero-García M.: *Communication Pipelining in Hypercubes.* Technical Report UPC-DAC-1995-14. (1995)

[30] Dion M., Philippe J. L., Robert Y.: *Parallelizing Compilers: What Can Be Achieved?* High Performance Computing and Networking. Proceedings, vol. II: Networking and Tools. (1994)

[31] Duff I. S., Reid J. K.: *MA48, a Fortran Code for Direct Solution of Sparse Unsymmetric Linear Systems of Equations.* Tech. Report RAL-93-072, Rutherford Appleton Lab., U.K. (1993).

[32] Duff I. S.: *Sparse Numerical Linear Algebra: Direct Methods and Preconditioning.* RAL-TR-96-047. Department for Computation and Information. Atlas Center. Rutherford Appleton Laboratory. (1996)

[33] Duff I. S., Grimes R. G., Lewis J. G.: *Sparse Matrix Test Problems.* ACM Transactions on Mathematical Software, vol. 15, no. 1. (1989) 1–14

[34] www.epm.ornl.gov/pvm/EuroPVM97

[35] Estupiñán M.: *Desarrollo Hardware de un Algoritmo Paralelo para la Multiplicación Matriz-Vector Solapando Cálculos y Comunicaciones.* Proyecto Fin de Carrera, E.U.I.T. de Telecomunicación, U.L.P.G.C. (1998)

[36] www.eurotools.org/home.top.html

[37] Feautrier P.: *Dataflow Analysis of Array and Scalar References.* Int. Journal Parallel Programming. (1991) 23–51

[38] Fraigniaud P., Lazard E.: *Methods and Problems of Communication in Usual Networks.* Discrete Applied Mathematics 53. Elsevier. (1994) 79–133

[39] Fu C., Yang T.: *Efficient Sparse LU Factorization with Partial Pivoting on Distributed Memory Architectures.* Department of Computer Science. University of California at Santa Barbara. (1996)

[40] García J., Ayguadé E., Labarta J.: *Automatic Data Distribution for Massive Parallel Computers.* Technical Report, UPC-CEPBA-95-04. (1995)

[41] www.gigabit-ethernet.org/alliance

[42] Golub G. H., Van Loan C. F.: *Matrix Computations.* Second edition. The Johns Hopkins University Press. (1989)

[43] Gupta M.: *Automatic Data Partitioning on Distributed Memory Multicomputers.* PhD thesis. University of Illinois at Urbana-Champaign. (1992)

[44] Heath M. T., Malony A. D., Rover D. T.: *Parallel Performance Visualization: From Practice to Theory.* IEEE Parallel and Distributed Technology, vol. 3, no. 4. (1995)

[45] Heydemann M., Opatrny J., Sotteau D.: *Embeddings of Hypercubes and Grids into de Bruijn Graph.* Journal of Parallel and Distributed Computing, vol. 23. (1994)

[46] High Performance Fortran Forum, *High Performance Fortran Language Specification. Version 1.0.* Scientific Programming. (1993)

[47] Hoare C. A. R.: *Occam 2. Reference Manual.* INMOS Limited. Prentice Hall International Series in Computer Science. (1988)

[48] Hwang K.: *Advanced Computer Architecture. Parallelism, Scalability, Programmability.* McGraw-Hill Inc. (1993) 650–652.

[49] ibm.tc.cornell.edu/ibm/pps/sp2/sp2.html

[50] www.intel.com/celeron/index.htm

[51] www.internet2.edu

[52] www.iol.unh.edu/training/fddi/htmls/index.html

[53] Jiménez E., Cabrera F., Cuevas J. G.: *Análisis de Onda Completa por el Método de los Momentos de Estructuras Microtiras Eléctricamente Grandes.* Proc. X Symposium Nacional URSI. (1995) 679–682

[54] Jiménez E., Cabrera F., Cuevas J. G.: *Sommerfeld: A Fortran Library for Computing Sommerfeld Integrals.* IEEE Ap-s International Symposium and URSI Radio Science Meeting. (1996)

[55] Kelly W., Pugh W.: *A Unifying Framework for Iteration Reordering Transformation.* Department of Computer Science, University of Maryland. (1995)

[56] Kelly W., Pugh W.: *Minimizing Communication while Preserving Parallelism.* Proceedings of the 10^{th} ACM International Conference on Supercomputing. ACM Press. (1996)

[57] Kulkarni D., Stumm M.: *Loop and Data Transformations: A Tutorial.* CSRI Tech. Report 337, University of Toronto. (1993)

[58] Kulkarni D., Stumm M.: *CDA Loop Transformations.* Languages, Compilers and Run-Time Systems for Scalable Computers. Kluwer Academic Pyblishers. (1996) 29–42

[59] Kumar K. G., Kulkarni D., Basu A.: *Generalized Unimodular Loop Transformation for Distributed Memory Multiprocessors.* Proceeding of International Conference of Parallel Processing. (1991)

[60] Lamport L.: *The Parallel Execution of Do Loops.* Communications of the ACM, vol. 17, no. 2. (1974)

[61] Leiserson C. E., Saxe J. B.: *Optimizing Syncronous Systems.* Proceedings of the 22nd Annual Symposium on Foundations of Computer Science. (1981) 23–36

[62] Li J., Chen M.: *The Data Alignment Phase in Compiling Programs for Distributed Memory Machines.* Journal Parallel Distributed Computing. (1991) 213–221

[63] Li X. S.: *Sparse Gaussian Elimination on High Performance Computers.* PhD Thesis. University of California at Berkeley. (1996)

[64] Lim A., Cheong G., Lam M.: *An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication.* Proceeding of Intl. Conference on Supercomputing. (1999)

[65] Ma E., Tao L.: *Embedding among Meshes and Tori.* Journal Parallel and Distributed Computing, vol. 18. (1993) 44–55

[66] Macías E. M., Suárez A., Ojeda-Guerra C. N.: *Parallelization of the Sequential Simulation of Infrared Channels: New Results.* Proceedings of the Tenth IASTED International Conference on Parallel and Distributed Computing and Systems. Las Vegas, Nevada, USA. (1998)

[67] MacLaren J. M., Bull J. M.: *Lessons Learned when Comparing Shared Memory and Message Passing Codes on Three Modern Parallel Architectures.* Proceedings of Conference on High-Performance Computing and Networking (HPCN Europe), vol. 1401 of LNCS, Springer-Verlag. (1998) 337–346

[68] Markowitz H. M.: *The Elimination Form of the Inverse and its Application to Linear Programming.* Management Science, 3. (1957) 255–269

[69] Martin R., Vahdat A., Culler D., Anderson T.: *Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture.* Proceeding of the 24th International Symposium on Computer Architecture. (1997)

[70] www.math.fau.edu/locke/graphthe.htm

[71] math.nist.gov/MatrixMarket

[72] Miguel J., Arruabarrena A., Beivide R., Gregorio J. A.: *Assessing the Performance of the New IBM SP2 Communication Subsystem.* IEEE Parallel and Distributed Technology. Systems and Applications, vol. 4, no. 4. (1996) 12–22

[73] Mohapatra P.: *Wormhole Routing Techniques for Directly Connected Multicomputer Systems.* ACM Computing Surveys, vol. 30, no. 3. (1998) 374–410

[74] Moldovan D. I.: *The Design of Algorithms for VLSI Systolic Arrays.* Proceedings of the IEEE, vol.1, no. 1. (1983)

[75] Mukherjee N., Gurd J. R.: *A Comparative Analysis of Four Parallelisation Schemes.* Proceedings of the Intl. Conference on Supercomputing. (1999)

[76] Mukherjee N.: *On the Effectiveness of Feed-back Guided Parallelization.* PhD Thesis. Department of Computer Science, University of Manchester. (1999)

[77] Navarro J. J., Llabería J. M., Valero M.: *Partitioning: An Essential Step in Mapping Algorithms into Systolic Array Processors.* IEEE Computer, vol. 20, no. 7. (1987) 77–89

[78] www.nec.com

[79] netlib.org/mpi

[80] O'Boyle M., Bull J.: *Expert Programmer versus Parallelising Compiler: A Comparative Study of Two Approaches for Distributed Shared Memory.* Scientific Programming, vol. 5, no. 1. (1996)

[81] Ojeda-Guerra C. N., Suárez A.: *Solving Linear Systems of Equations Overlapping Communications and Computations in Torus Networks.* Proceedings of the 5th Euromicro Workshop on Parallel and Distributed Processing. London, U.K. (1997)

[82] Ojeda-Guerra C. N., Macías E., Suárez A.: *Sparse LU Factorization with Partial Pivoting Overlapping Communicacions and Computations on SP-2 Multicomputer.* EuroPVMMPI'98. LNCS Springer-Verlag. Liverpool, UK. (1998)

[83] Ojeda-Guerra C. N., Esper-Chaín R., Estupiñan M., Macías E. M., Suárez A.: *Hardware Mapping of a Parallel Algorithm for Matrix-Vector Multiplication Overlapping Communications and Computations.* FPL'98. LNCS 1482, Springer Verlag. Tallin, Estonia. (1998)

[84] Ojeda-Guerra C. N., Cabrera F., Jiménez E., Macías E., Suárez A.: *Parallelization of the Moment Method in the Computation of Microstrip Structures.* Proceedings of the 5th Joint Conference on Information Sciences. Atlantic City, USA. (2000)

[85] www.openmp.org

[86] www.csrd.uiuc.edu/parafrase2

[87] Petersen P. M., Padua D. A.: *Static and Dynamic Evaluation of Data Dependence Analysis Techniques.* IEEE Transaction on Distributed Systems, vol. 7, no. 11. (1996)

[88] Ponnusamy R., Hwang Y., Das R., Saltz J.: *Supporting Irregular Distributions Using Data-Parallel Languages.* IEEE Parallel and Distributed Technology, vol. 3, no. 1. (1995)

[89] Quinn M. J.: *Parallel Computing. Theory and Practice.* McGraw-Hill International Editions. (1994)

[90] Riley G., Bull J., Gurd J.: *Performance Improvement Through Overhead Analysis: A Case Study in Molecular Dynamics.* Proceedings of the Intl. Conference on Supercomputing. (1997)

[91] Saltz J., Ponnusamy R., et al.: *A Manual for the CHAOS Runtime Library.* University of Maryland: Department of Computer Science and UMIACS Technical Reports CS-TR-3437 and UMIACS-TR-95-34. (1995)

[92] Severance C.: *IEEE 802.11: Wireless is Coming Home.* IEEE Computer, vol. 32, no. 11. (1999)

[93] www.sgi.com

[94] Steen A.: *Overview of Recent Supercomputers.* Publication of the NCF. Utrecht University. (1999)

[95] Suárez A.: *Ejecución Entrelazada de las Particiones en Algoritmos Sistólicos Particionados.* Tesis Doctoral. Departamento de Arquitectura de Computadores, Universidad Politécnica de Cataluña. (1993)

[96] Suárez A., Ojeda-Guerra C. N.: *Overlapping Computations and Communications in Torus Networks.* Proceedings of the 4th Euromicro Workshop on Parallel and Distributed Processing. Braga, Portugal. (1996)

[97] suif.stanford.edu

[98] sunsite.auc.dk/ldp/HOWTO/Parallel-Processing-HOWTO-3.html

[99] www.techservers.hp.com

[100] Tourino J., Doallo R., Asenjo R., Plata O. y Zapata E.: *Analyzing Data Structures for Parallel Sparse Direct Solvers: Pivoting and Fill-in.* Sixth Workshop on Compilers for Parallel Computers. (1996) 151–168

[101] *Transputer. Occam 2 Toolset. User Manual.* INMOS Limited. (1989)

[102] Trimberger S. N.: *Field Programmable Gate Array Technology.* Kluwer Academic Publishers. (1994)

[103] www-ugrad.cs.colorado.edu/~csci4576/VectorArch/VectorArch.html

[104] Valero M.: *Adaptación Automática de Algoritmos Sistólicos al Hardware.* Tesis Doctoral. Departamento de Arquitectura de Computadores, Universidad Politécnica de Cataluña. (1989)

[105] www.w3.org

[106] Wakatani A., Wolfe M.: *Effectiveness of Message Strip-Mining for Regular and Irregular Communication.* PDCS. (1994)

[107] Watt A.: *Programming Language. Concepts and Paradigms.* C.A.R. Hoare Serie Editor. Prentice Hall, I.S.B.N. 0-13-728866-2. (1990)

[108] Woodward P.: *Perspectives on Supercomputing: Three Decades of Change.* IEEE Computer, vol. 29, no. 10. (1996)

[109] Wolfe M.: *Optimizing Supercompilers for Supercomputing.* MIT press. (1990)

[110] Xu Z., Hwang K.: *Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2.* IEEE Parallel and Distributed Technology, vol. 4, no. 1. (1996)

Glosario

Símbolo	*Explicación*
$A_{I_f,(p_f,p_{f+1})}$	Arco del GDM que une los vértices V_{I_f,p_f} y $V_{I_{f+1},p_{f+1}}$.
$A_{\vec{p}}$	Hiperarco del GDM sobre el que se proyecta el punto $\vec{p}$ del $EI(I,n)$.
$A_{\vec{p}}^{color,orden}$	Hiperarco del GDM, con pesos *color* y *orden*, sobre el que se proyecta el punto $\vec{p}$ del $EI(I,n)$.
color	Peso del hiperarco del GDM.
d	Número de elementos del conjunto dimensionado entre dos secuencias iguales de vectores de desplazamiento.
desp	Vector de desplazamientos.
E_{MUX}	*Enable* del multiplexor indicado.
$EI(I,n)$ o EI	Espacio de iteraciones n-dimensional.
final	Vector de valores finales.
GDM	Grafo de dependencias modificado.
i^p	Coordenadas del conjunto dimensionado M correspondiente a la iteración I^p.
$I = (I_1, I_2, ..., I_n)^t$	Vector de índices del $EI(I,n)$.
$I' = (I_1, I_2, ..., I_f = c_f, ..., I_n)^t$	Vector de índices del $EI(I', n-1)$.
$I^p = (I_1 = p_1, I_2 = p_2, ..., I_n = p_n)$	Iteración del algoritmo correspondiente al punto $\vec{p}$ del $EI(I,n)$.
inc	Vector de incrementos.
inicial	Vector de valores iniciales.
$M(i^p)$	Elemento del conjunto dimensionado M correspondiente a las coordenadas i^p.
max	Bucle más externo que indexa al conjunto dimensionado.