E.T.S.I. INFORMÁTICA

UNIVERSIDAD DE MÁLAGA

Un Lenguaje de Coordinación para la Resolución de Problemas Basados en Descomposición de Dominios

Tesis Doctoral

Presentada por:

D. Enrique Soler Castillo

Dirigida por:

Dr. José María Troya Linero

Catedrático de Universidad del Área de Lenguajes y Sistemas Informáticos.

Dr. Juan Ignacio Ramos Sobrados

Catedrático de Universidad del Área de Ciencias de la Computación e Inteligencia Artificial.

Málaga, septiembre de 2001

D. José María Troya Linero, Catedrático de Universidad del Área de Lenguajes y Sistemas Informáticos de la E.T.S. Ingeniería Informática y D. Juan Ignacio Ramos Sobrados, Catedrático de Universidad del Área de Ciencias de la Computación e Inteligencia Artificial de la E.T.S. Ingenieros Industriales de la Universidad de Málaga

Certifican

Que D. Enrique Soler Castillo, Licenciado en Informática, ha realizado en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, bajo nuestra dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulada

"Un Lenguaje de Coordinación para la Resolución de Problemas Basados en Descomposición de Dominios".

Revisado el presente trabajo, estimamos que puede ser presentado al tribunal que ha de juzgarlo, y autorizamos la presentación de esta Tesis Doctoral en la Universidad de Málaga.

Málaga, 4 de mayo de 2001.

Fdo. José María Troya Linero

Catedrático de Universidad del Área de Lenguajes y Sistemas Informáticos.

Fdo. Juan Ignacio Ramos Sobrados

Catedrático de Universidad del Área de Ciencias de la Computación e Inteligencia Artificial.

Agradecimientos

En primer lugar quisiera mostrar mi agradecimiento a los directores de este trabajo. A José María Troya por el tiempo que me ha dedicado y por la confianza que ha depositado en mí desde el momento que ingresé en el departamento como miembro del proyecto PACOTE. A Juan Ignacio Ramos quiero agradecerle el haberme guiado en la parte numérica y su paciencia al explicarme con igual entusiasmo desde conceptos matemáticos simples hasta las partes más complejas de los fenómenos físicos que hemos tratado.

También quiero mostrar mi gratitud a Manuel Díaz, por sus valiosas ideas y sugerencias en la definición de los objetivos y a Bartolomé Rubio por su incondicional apoyo en la definición del lenguaje desarrollado y en la redacción de los artículos que hemos publicado.

Por otro lado, también quiero agradecer el desinteresado apoyo mostrado por otros miembros del departamento como son Francisco Villatoro y Carmen María García en los momentos en los que necesité de sus conocimientos y experiencia.

Finalmente quiero dedicar este trabajo a mi familia, tanto a mis padres, como a mi mujer María José y a mi hijo Enrique por el tiempo que he dejado de estar con ellos durante la realización de este trabajo.

Índice

Prólogo

La cada vez mayor disponibilidad de ordenadores paralelos de altas prestaciones, así como de redes de estaciones de trabajo, está provocando un gran esfuerzo de investigación tanto en el desarrollo de métodos numéricos para resolver ecuaciones en derivadas parciales (*Partial Differential Equations* o PDEs) como en el desarrollo de lenguajes paralelos que saquen rendimiento a este tipo de ordenadores y que permitan abstraer al programador de unas arquitecturas hardware cada vez más complejas.

Los métodos de descomposición de dominios son métodos generales y flexibles para resolver sistemas de ecuaciones lineales y no lineales que surgen de la discretización de PDEs.

Elegir un lenguaje de programación adecuado para obtener una solución eficiente de este tipo de problemas es complicado, sobre todo si se pretende obtener una solución transportable y eficiente, capaz de sacar partido a las arquitecturas paralelas, en las cuales el programador tiene que coordinar distintos niveles de paralelismo y de localidad de los datos. Los lenguajes que permiten el paralelismo de tareas son más adecuados para programar la comunicación y sincronización entre los procesos que resuelven cada uno de los subdominios en que se divide el problema global. Por otro lado, el paralelismo de datos es más apropiado para la resolución de cada uno de los subdominios, ya que se pueden obtener soluciones eficientes sin (relativamente) un gran esfuerzo de programación.

El propósito de esta tesis es presentar un lenguaje de coordinación transportable y eficiente para la resolución de problemas numéricos estructurados por bloques. Los aspectos relativos a la comunicación y sincronización entre los distintos subdominios que forman el dominio global de la aplicación se separan de la codificación del método numérico, incrementándose así la reusabilidad de ambas partes, la de coordinación y la computacional. Puesto que la causa más importante de comunicación entre los

subdominios son las fronteras entre estos, se ha denominado a este lenguaje **BCL** (*Border-based Coordination Language*).

Este lenguaje también permite la integración del paralelismo de datos y tareas, de modo que en la parte de coordinación se describen, junto con los distintos dominios, las distribuciones de estos dominios entre los distintos procesadores pertenecientes a cada tarea que los va a resolver. Estas tareas son codificadas con un lenguaje que permite el paralelismo de datos, HPF (*High Performance Fortran*), de forma que se incrementa notablemente la eficiencia y la escalabilidad de la solución.

Aunque BCL está diseñado para la solución de problemas de descomposición de dominios y estructurados por bloques, otros problemas científicos que se pueden beneficiar de la integración del paralelismo de datos y tareas y cuyo patrón de comunicación está basado en el intercambio de matrices también se pueden plantear con BCL. Sin embargo, este tipo de problemas puede ser expresado mejor mediante el uso de patrones o esqueletos. Por esta razón, se han desarrollado una serie de construcciones de alto nivel que permiten la definición de forma declarativa de un conjunto de tareas HPF que trabajan de forma coordinada. Además, se ofrece la posibilidad de utilizar plantillas de implementación para facilitar la programación de la parte de computación de la aplicación. A este conjunto de construcciones se le ha dado el nombre de **DIP** (*Domain Interaction Patterns*) para diferenciarlo del resto de BCL.

La eficiencia y expresividad del lenguaje se demuestra mediante la codificación de diversas aplicaciones científicas como la resolución de la ecuación de Laplace en dos dimensiones, la transformada rápida de Fourier en dos y tres dimensiones y la resolución de ecuaciones de reacción-difusión en dominios bidimensionales irregulares. Mención especial hay que dedicar a esta última ya que, utilizando BCL, se han hecho gran cantidad de pruebas para obtener el comportamiento de distintos métodos de descomposición de dominios con un sistema de dos ecuaciones no lineales de reacción-difusión. Se han utilizado dominios irregulares con esquinas reentrantes y se ha enfocado el problema desde dos puntos de vista distintos, el de la exactitud de los resultados y el de la eficiencia de la solución.

Capítulo 1. Introducción

Se podría llegar a pensar que a medida que los ordenadores se hagan más potentes, conseguirán, algún día, ser lo suficientemente rápidos como para satisfacer la creciente demanda de potencia de cálculo. Sin embargo, la historia nos revela que cuando una tecnología satisface las necesidades de aplicaciones conocidas, surgen otras aplicaciones (o las mismas con mayor número de puntos y matrices más grandes) como consecuencia de esa tecnología, demandando nuevos avances [Foster 95]. Así, por ejemplo, un informe preparado por el gobierno británico a finales de los años cuarenta concluyó que las necesidades de Gran Bretaña serían cubiertas con dos o tres ordenadores. Los autores de este informe sólo pensaban en el desarrollo de tablas de balística. Igualmente, las perspectivas iniciales de la compañía Cray Research Inc. predecían un mercado de unos diez superordenadores. En estas predicciones solamente se tuvo en cuenta la demanda de computación de un limitado número de disciplinas.

Tradicionalmente, la demanda a más alto nivel de recursos computacionales ha sido motivada por simulaciones numéricas de sistemas complejos, como por ejemplo, predicciones meteorológicas, modelado global del clima, dispositivos mecánicos, aeronáutica, procesos industriales, reacciones químicas, etc. Actualmente, en cambio, el factor más importante que dirige el desarrollo de ordenadores cada vez más rápidos viene determinado por las nuevas aplicaciones comerciales, las cuales requieren ordenadores capaces de manejar gran cantidad de datos de manera sofisticada. Sin embargo, las aplicaciones científicas tradicionales siguen siendo importantes consumidores de la tecnología de computación paralela. A medida que los resultados de las investigaciones teóricas llevan a problemas no lineales y a medida que la experimentación resulta más costosa o incluso impracticable, las necesidades de estudios mediante simulación computacional se hacen más importantes, aumentando así la necesidad de cálculo.

Tal demanda de cálculo sólo puede ser satisfecha mediante los ordenadores paralelos y distribuidos. Éstos se pueden definir como un conjunto de procesadores capaces de trabajar cooperativamente para resolver un problema computacional. Esta definición es lo suficientemente amplia como para abarcar desde los grandes superordenadores que disponen de cientos o incluso miles de procesadores hasta las redes de estaciones de trabajo y sistemas empotrados.

Pero al mismo tiempo que estos sistemas paralelos y distribuidos constituyen un buen soporte para este tipo de aplicaciones, presentan nuevos retos para la tecnología del software. Casi de forma general, se puede decir que los grandes avances tecnológicos en el campo del desarrollo de soportes informáticos físicos se ven de algún modo frenados a la hora de poder utilizarse de una manera inmediata y satisfactoria debido, en gran parte, al no tan notable avance paralelo en el campo del desarrollo del soporte informático lógico. Aunque siempre ha sido un punto fundamental el disponer de una tecnología software adecuada, esta necesidad se transforma en exigencia si se desea aprovechar la potencia de los nuevos equipos informáticos para dar soporte a las grandes y complejas aplicaciones actuales.

Algunas de las razones que explican las dificultades del desarrollo del software paralelo son las siguientes:

- El pensamiento humano consciente nos parece ser secuencial (aunque algunos aspectos de la percepción se desarrollan de forma paralela) con lo cual, hay alguna tendencia a considerar el software como algo secuencial.

- Los fabricantes de ordenadores paralelos han dirigido su mercado hacia la computación de alto rendimiento científica y numérica en lugar del mercado comercial, mucho mayor y más efectivo. El mercado de la computación de alto rendimiento ha sido siempre pequeño y ha tendido hacia aplicaciones militares. Esto hace que los ordenadores paralelos sean caros puesto que se venden pocos.

- El tiempo de ejecución de un programa secuencial varía en no más de un factor de una constante cuando se migra de un sistema uniprocesador a otro. Esto no es así en computación paralela, en donde los tiempos de ejecución pueden cambiar en un orden de magnitud cuando se cambia de arquitectura. La naturaleza no

local de un programa paralelo requiere la interacción con la estructura de comunicación, y el coste de cada comunicación depende en gran medida de cómo estén dispuestos tanto el programa como la interconexión. La portabilidad es, por tanto, un factor mucho más serio que en programación secuencial.

- El hecho de que la programación paralela sea complicada y la gran cantidad de programas secuenciales disponibles hacen que en ocasiones se prefiera un programa secuencial a tener que desarrollar uno paralelo.

Los esfuerzos que se han hecho en desarrollar lenguajes paralelos se han dirigido tanto desde arriba, es decir, desde la elegancia de la abstracción teórica, como desde abajo, sacando partido directamente a la tecnología. Los modelos muy abstractos ocultan incluso la presencia de paralelismo a nivel software. Tales modelos hacen que la programación sea fácil y transportable pero la eficiencia es, generalmente, difícil de conseguir. En el otro extremo, los modelos de bajo nivel hacen que todos los detalles de la programación paralela se tengan que hacer explícitamente, de forma que la programación se hace muy complicada y muy poco transportable, pero, generalmente, más eficiente. Sin embargo, el mayor progreso obtenido hasta ahora y los resultados más esperanzadores se basan en dirigir el desarrollo desde el medio, atacando el problema desde el nivel del modelo que sirve como interfaz entre los aspectos del software y el hardware. Una clasificación muy interesante y detallada de modelos de paralelismo atendiendo a su nivel de abstracción se puede encontrar en [Skillicorn y Talia 98].

Otra clasificación similar es la que se puede encontrar en [Pelagatti 98]. De esta forma, en el nivel más alto de abstracción se tienen los **modelos de paralelismo implícitos** en los que el programador no tiene que preocuparse del paralelismo, utilizando un lenguaje secuencial (que puede ser tanto imperativo como declarativo). La clave de estos modelos es que no se requiere ningún esfuerzo por parte del programador para desarrollar o manejar el paralelismo. El compilador y su soporte de tiempo de ejecución se han desarrollado para detectar el paralelismo de manera transparente. Una de las ventajas de estos modelos es que los códigos antiguos pueden ser paralelizados sin grandes esfuerzos por parte del programador. En estos sistemas, la detección del

paralelismo se basa en el descubrimiento de las dependencias entre las sentencias del programa.

El problema de estos modelos surge a la hora de descubrir cuándo dos sentencias se refieren a la misma dirección de memoria y, por tanto, deben ser ejecutadas secuencialmente. Desafortunadamente, no es factible resolver este problema directamente, ni siquiera para expresiones lineales de índices de arrays, ya que encontrar estas dependencias es un problema NP-completo. Lo que hacen los compiladores es intentar detectar qué iteraciones de bucles son independientes, mediante la realización de varias comprobaciones con las expresiones de los índices.

Otras aproximaciones intentan hacer reorganizaciones del código. Sin embargo, no existe ninguna evidencia de que se pueda obtener un buen programa paralelo partiendo de un programa secuencial que resuelva el mismo problema. Esto es debido a que un buen algoritmo paralelo, para un problema dado, puede ser completamente distinto del obtenido de cualquier transformación de incluso el mejor programa secuencial.

Esta es la razón por la cual, por el momento, el programador debe tener en cuenta, de algún modo, el paralelismo incluso usando un modelo implícito, lo cual nos acerca a los modelos explícitos.

Dentro de los **modelos de paralelismo explícitos** se encuentran aquellos *completamente abstractos*, en los cuales el programador desarrolla un algoritmo paralelo de acuerdo a un modelo de computación muy abstracto, dejando muchos aspectos del paralelismo al sistema. Los lenguajes de paralelismo de datos entrarían dentro de estos modelos. De esta forma, para la paralelización de código secuencial, se han desarrollado diversos dialectos de Fortran (Vienna Fortran [Zima y otros 92], Fortran 90 [Adams y otros 92] y HPF [Koelbel y otros 94]) que proporcionan construcciones que permiten al programador expresar el paralelismo explícitamente.

Los modelos *explícitos parcialmente abstractos* son aquellos en los que el programador se puede abstraer de detalles de la arquitectura, pero ha de especificar tanto la comunicación y sincronización entre procesos como la tarea a desarrollar por cada uno de ellos. Típicamente, las instrucciones no paralelas son tomadas de algún

lenguaje secuencial. Algunos ejemplos dignos de resaltar son Fortran M [Foster y Chandy 92], basado en el paso de mensajes, Linda [Gelernter 85], que se comentará más adelante, y CC++ [Chandy y Kesselman 93] que extiende C++ con un espacio de direcciones compartido. Otro enfoque que ha tenido una gran aceptación es el del desarrollo de bibliotecas que facilitan primitivas de interacción que pueden ser llamadas por lenguajes secuenciales. Este el caso, entre otros, de PVM [Geist y otros 94] y MPI [MPIF 95] que permiten desarrollar aplicaciones eficientes, incluso para la paralelización de códigos complejos preexistentes [Díaz, Llopis, Pastrana, Rus y Soler 96].

Por último, habría que mencionar los modelos explícitos *dependientes de la máquina*, en los cuales el programador tiene el control total sobre la asignación de procesos a procesadores y el flujo de control. Típicamente, estos modelos proporcionan el paralelismo mediante una biblioteca específica para la máquina, utilizable desde C o Fortran, como puede ser el MPP System para Cray T3D. Sin embargo, existen ejemplos dependientes de la máquina que no se basan en bibliotecas sino que son lenguajes paralelos como puede ser Occam [INMOS 94] para los sistemas basados en Transputers [May y otros 86].

Una clasificación distinta es la que tradicionalmente se ha hecho de los lenguajes según el tipo de problemas a los que están orientados. Este enfoque, que se viene utilizando desde los primeros lenguajes secuenciales (como Fortran para cálculo científico y COBOL para programas de gestión), se puede realizar también para los lenguajes paralelos que no son de propósito general, sino que están orientados hacia una clase específica de problemas [Pelagatti 98]. Esta especialización permite que se pueda simplificar el modelo de paralelismo, limitándolo a unas cuantas estructuras y facilitando así la tarea del programador.

En nuestra aproximación nos hemos planteado como objetivo la definición de un lenguaje de coordinación que ayude a desarrollar programas paralelos para codificar aplicaciones científicas de forma sencilla, transportable y eficiente. Nos hemos centrado especialmente en aquellas en las que la comunicación se limita a intercambio de (sub)matrices como son los problemas de descomposición de dominios. El aprendizaje de este lenguaje no debe ser un gran problema para el programador, sino que debe

abstraer los aspectos de comunicación y sincronización de forma amigable, de modo que la paralelización no le distraiga de la codificación de los métodos numéricos empleados para resolver el problema.

A continuación, se describen los métodos de descomposición de dominios a cuya resolución paralela va destinada la mayor parte de este trabajo. Como es bien sabido, el lenguaje más utilizado para la implementación de la solución de este tipo de problemas ha sido principalmente Fortran. Es por ello, que se le ha dedicado un apartado a los lenguajes que han surgido de la evolución de Fortran hacia el paralelismo, Fortran 90 y HPF, y que sirven de lenguaje base para nuestra aproximación. Ésta se basa en sacar provecho de la integración del paralelismo de datos y tareas (apartado 1.3) mediante los lenguajes de coordinación y la programación paralela estructurada (apartados 1.4 y 1.5, respectivamente). En el apartado 1.6, introducimos brevemente el lenguaje desarrollado y lo comparamos con el trabajo relacionado. Para concluir la introducción, se presenta el esquema del resto de la memoria.

1.1 Métodos de Descomposición de Dominios.

La mayoría de los problemas encontrados en física e ingeniería se caracterizan por tener dominios irregulares, interacciones entre sistemas distintos, fenómenos físicos diferentes, distintas fases, etc. Para obtener una solución precisa de estos problemas se deben tener en cuenta distintos aspectos:

- La geometría de cada dominio.
- La física de los distintos fenómenos que tienen que ser modelados.
- Los métodos numéricos a emplear para resolver cada dominio de forma que, por ejemplo, gradientes pronunciados, capas internas o en la frontera sean resueltos de manera apropiada.
- Las condiciones a imponer en las fronteras entre los dominios.

- Si se quiere sacar partido a los ordenadores paralelos y su enorme potencial para la resolución numérica de problemas físicos y de ingeniería, se debe tener en cuenta la arquitectura de estos, lo que puede conllevar importantes cambios en la solución implementada.

Los métodos de descomposición de dominios son métodos generales y flexibles para la resolución de ecuaciones algebraicas lineales y no lineales que surgen de la discretización de ecuaciones en derivadas parciales y que utilizan propiedades de estas ecuaciones para obtener soluciones eficientes [Smith y otros 96]. Actualmente se está realizando un gran esfuerzo de investigación en estos métodos aunque su origen proviene del siglo XIX [Schwarz 90].

Hay que tener en cuenta que el término descomposición de dominios tiene significados distintos para los especialistas en la disciplina de las PDEs:

- En computación paralela, generalmente significa el proceso de distribuir los datos de un modelo computacional entre los procesadores de un ordenador de memoria distribuida. En este contexto, descomposición de dominios se refiere a las técnicas para descomponer una estructura de datos y que puede ser independiente de los métodos numéricos.

- En análisis asintótico, significa la separación del dominio físico en regiones que pueden ser modeladas mediante ecuaciones o fenómenos físicos diferentes, con las interfaces entre las regiones manejadas por diversas condiciones (por ejemplo, continuidad de las funciones). En este contexto, descomposición de dominios se refiere a la determinación de qué ecuaciones resolver.

- En los métodos de precondicionador, descomposición de dominios se refiere al proceso de subdividir la solución de un gran sistema de ecuaciones lineales en problemas más pequeños, cuyas soluciones pueden ser utilizadas para producir un precondicionador para el sistema de ecuaciones que resulta de discretizar la PDE en el dominio completo. En este contexto, descomposición de dominios se refiere sólo al método de resolución para el sistema de ecuaciones algebraicas que surgen de la discretización.

Independientemente de esta distinción, los métodos de descomposición de dominios pueden clasificarse en aquellos que utilizan superposición de dominios (también denominados métodos de Schwarz) y aquellos que utilizan dominios sin superposición (métodos de Schur).

Los problemas que se tratan en esta tesis pertenecen a la primera y segunda categoría antes mencionadas (computación paralela y análisis asintótico) y se han considerado principalmente problemas con superposición de dominios. En [Ramos y Soler 01] se realiza un estudio de distintos métodos para la resolución de un sistema de dos ecuaciones no lineales de reacción difusión en dos dimensiones. Para este problema se emplean dos tipos de dominios, por un lado, dominios regulares y, por el otro, dominios irregulares que se pueden descomponer en bloques regulares. Se estudian métodos con superposición y sin ella, aplicando, además, distintas condiciones en la frontera entre los dominios para cada uno de los métodos (condiciones de Dirichlet, Neumann, Robin y combinaciones de ellas). El resultado de aplicar estos métodos, así como la resolución del mismo problema mediante otros métodos sin descomposición de dominios y la comparación entre ellos se encuentra en [Ramos, Soler y Troya 98].

El desarrollo del software paralelo para tales aplicaciones es una tarea difícil debido a la complejidad de la solución aplicada a cada dominio junto con el método de descomposición empleado, especialmente cuando el producto resultante tiene que ser transportable y eficiente y, por tanto, capaz de aprovechar las ventajas de una gran diversidad de arquitecturas paralelas. En [Drashansky, Joshi y Rice, 95] se presenta **SciAgents**, una herramienta diseñada para la resolución de problemas mediante descomposición de dominios. Con esta aproximación, el programador construye su aplicación mediante el enlace de las soluciones de diversas PDE en distintos dominios mediante agentes. Éstos vienen a ser distintos procesos (uno para cada subdominio y uno para cada interfaz entre éstos) que se pueden definir y coordinar mediante una interfaz gráfica. Cada dominio se puede resolver mediante un paquete estándar (PELLPACK [Houstis y otros 89]) pensado para la resolución de PDE elípticas en paralelo.

SciAgents no es un lenguaje de programación sino una herramienta gráfica. El programador tendrá que escribir el código para resolver cada dominio y la interfaz entre éstos mediante un lenguaje de programación.

El lenguaje más utilizado en programación científica es sin duda Fortran. Éste ha sido modificado recientemente para adaptarlo a las nuevas metodologías de programación y a los nuevos ordenadores paralelos, surgiendo así dos nuevos estándares, **Fortran 90** [Adams y otros 92][Metcalf y Reid 90] y **HPF** [Koelbel y otros 94] cuyas características generales se comentan en el apartado siguiente.

1.2 Fortran 90 y HPF.

En muchos aspectos, Fortran 90 es una modernización (con compatibilidad hacia atrás) del gran estándar de programación científica Fortran 77 [ANSI 78]. Sus dos mayores aportaciones con respecto a su predecesor son el manejo de arrays y los tipos abstractos de datos. El objetivo de su definición fue el de "modernizar Fortran de manera que pueda continuar su larga historia de lenguaje de programación científico y de ingeniería". Un objetivo secundario fue el de proporcionar características de los lenguajes modernos para permitir que los programadores abandonen el uso de formas obsoletas y no deseables de Fortran 77. Aunque proporciona nuevas estructuras, no se ignoraron los requisitos de los códigos "heredados". No hubiese tenido tanta aceptación si los miles de códigos ya escritos en Fortran 77 hubiesen tenido que ser modificados para adaptarlos al nuevo estándar. Por tanto, Fortran 90 incluye todas las características de Fortran 77, permitiendo, por un lado, la reutilización del software ya escrito y, por otro, la aplicación de la programación estructurada y de los tipos abstractos de datos para la realización de software nuevo.

Además de todas las características de Fortran 77, se le han añadido extensiones significativas, algunas de las cuales (como la sintaxis de manejo de arrays) facilitan al compilador determinar las operaciones que se pueden realizar concurrentemente. Una lista detallada y comentada de todas las nuevas características de Fortran 90 se puede encontrar en [Press y otros 96].

Incluso antes de que se aprobara formalmente el estándar de Fortran 90, ya se escuchaban voces que pedían nuevas extensiones. En particular, aquellas que permitieran desarrollar aplicaciones transportables y eficientes en la nueva generación de ordenadores paralelos. Tomando como base a Fortran 90, principalmente por su sintaxis de arrays, y utilizando conceptos de dialectos como Vienna Fortran [Chapman y

otros 92] y Fortran D [Hiranandani y otros 91], se publicó en 1993 el estándar de HPF [HPFF 93] que desde entonces ha sido ampliamente implementado.

Este lenguaje ha emergido con fuerza como el nuevo estándar de lenguaje de paralelismo de datos de alto nivel. Los lenguajes con paralelismo de datos facilitan la programación porque las operaciones que requieren bucles en lenguajes de más bajo nivel pueden ser escritas como una sola instrucción que, además, es más reveladora para el compilador ya que no tiene que inferir el patrón que el programador trataba de escribir.

Los objetivos bajo los cuales se ha desarrollado HPF son los siguientes:

- Proporcionar programación paralela de datos (una sola hebra de control, espacio de nombres global y computación paralela poco sincronizada).
- Portabilidad entre distintas arquitecturas.
- Alto rendimiento en ordenadores paralelos (con costes de acceso a memoria no uniformes).
- Uso de Fortran estándar como base.
- Inter-operabilidad con otros lenguajes (por ejemplo, C).

HPF consiste básicamente en instrucciones que se introducen en el programa en forma de directivas al compilador a modo de comentarios para indicarle la manera en que se ha de realizar la distribución de los datos e identificar los bucles que pueden ser paralelizados. La gran ventaja de HPF es que es el compilador el que decide cuándo son necesarios los pasos de mensajes entre los distintos procesadores. El programador se limita a elegir la distribución de datos y a “ayudar” a la paralelización con el uso de las directivas oportunas.

La distribución de datos se realiza mediante las siguientes directivas:

- `!HPF$ DISTRIBUTE` que describe cómo cada dimensión de un array es dividida en trozos y distribuida entre procesadores de una forma regular. La distribución

de cada dimensión de un array puede ser del tipo `BLOCK`, `CYCLIC` o `*`. Así, por ejemplo la siguiente declaración:

```
REAL A(100,100)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK)
```

indica que tanto la primera como la segunda dimensión están distribuidas. En el caso de disponer de 4 procesadores, uno de ellos tomaría la sección `A(1:50,1:50)`, otro tomaría `A(51:100,1:50)`, etc.

La directiva:

```
!HPF$ DISTRIBUTE A(CYCLIC,*)
```

indica que cada procesador recibe una de cada 4 filas de `A`. Es decir, un procesador tomaría `A(1,1:100)`, `A(5,1:100)`, etc., otro procesador tomaría `A(2,1:100)`, `A(6,1:100)`, etc.

- `!HPF$ ALIGN` permite indicar que determinados índices de dos arrays deben alinearse juntos, es decir, pertenecer al mismo procesador. Así, por ejemplo:

```
!HPF$ ALIGN X(I) WITH Y(2*I-1)
```

indica que el elemento i-ésimo de `X` se debe almacenar en el mismo procesador que el elemento de `Y` cuyo índice cumpla la expresión `2*I-1`, o dicho de otro modo, los elementos de `X` se deben alinear con los impares de `Y`.

El paralelismo de datos se realiza mediante instrucciones que el ordenador puede realizar para distintos índices de un array. Estas instrucciones son:

- Instrucciones de manejo de arrays. Así, la instrucción `A = B**2 + 4` puede ser realizada concurrentemente si `A` y `B` son arrays distribuidos de la misma forma. Si no fuese así, el compilador se encargaría de generar los mensajes necesarios entre los distintos procesadores.

- La nueva instrucción `FORALL` extiende Fortran 90 para mejorar las operaciones de manejo de arrays. Así, por ejemplo:

```
FORALL (i=2:N-1) A(i,i)= A(i-1,i-1)+ A(i,i) + A(i+1, i+1)
```

realiza una suma a lo largo de la diagonal de `A` en paralelo (la parte izquierda de una asignación dentro de una sentencia `FORALL` no se actualiza hasta que no se hayan evaluado todos los valores del lado derecho de la expresión).

- Las funciones `PURE` indican que no se realiza ningún efecto lateral dentro de ellas, con lo cual pueden ser llamadas concurrentemente para distintos índices de un array distribuido.

- La directiva `INDEPENDENT` indica al compilador que el bucle que viene a continuación se puede realizar en paralelo.

1.3 Integración del Paralelismo de Datos y Tareas.

La desventaja de utilizar un lenguaje como HPF es que el usuario está limitado por el modelo de paralelismo soportado por el lenguaje. Es ampliamente aceptado que ciertas aplicaciones no pueden ser implementadas eficientemente siguiendo un paradigma puro de paralelismo de datos. Por ejemplo:

- Encauzamiento de tareas con paralelismo de datos (*pipelines of data parallel task*) [Dinda y otros 94]. Este tipo de computación es una estructura común en el procesado de imágenes y de señales, en visión por ordenador, etc. En estos sistemas, se trabaja generalmente con conjuntos de datos pequeños con lo que el paralelismo de datos es limitado.

- Códigos multibloque y de descomposición de dominios con mallas irregulares [Agrawal y otros 95] y aplicaciones irregulares [Chassin y otros 00]. Por un lado, el paralelismo de tareas es más apropiado para la descomposición de dominios y la comunicación entre los distintos procesos encargados de resolver cada dominio (normalmente heterogéneos). Por otro lado, la solución de cada dominio puede ser más fácil usando un lenguaje de paralelismo de datos (por ejemplo, HPF) con lo que se pueden obtener programas muy eficientes sin, relativamente, mucho esfuerzo.

- Problemas de optimización multidisciplinar, como el modelado global del clima o el diseño de aviones [Chapman y otros 97] explotan el paralelismo de tareas

para coordinar la ejecución de varios módulos independientes, los cuales pueden ser programados mediante paralelismo de datos.

Para estas aplicaciones, en lugar de tener un único programa con paralelismo de datos, es más apropiado subdividir la computación entre distintas tareas (cada una de las cuales puede sacar ventaja del paralelismo de datos) de forma que las tareas se ejecuten concurrentemente y cooperen, explotándose así ambos tipos de paralelismo.

La integración del paralelismo de datos y tareas es actualmente un área activa de investigación. Existen al menos tres ventajas para la integración del paralelismo de datos y tareas en un solo lenguaje:

a) Generalidad. Dado el gran número de modelos y lenguajes paralelos disponibles [Skillicorn y Talia 98], tener un único lenguaje capaz de codificar una amplia variedad de aplicaciones paralelas es ciertamente atractivo.

b) Incrementar la escalabilidad explotando ambas formas de paralelismo. En muchas aplicaciones, los tamaños de datos son fijos y no pueden ser incrementados fácilmente, limitándose, por tanto, la cantidad de paralelismo de datos que puede ser explotado.

c) Coordinación de aplicaciones multidisciplinarias como las mencionadas anteriormente. Así, por ejemplo, en el diseño de aviones se tienen que coordinar modelos de disciplinas distintas como la aerodinámica, propulsión, análisis de estructuras, etc. Generalmente, cada modelo se codifica mediante paralelismo de datos.

Sin embargo, integrar las dos formas de paralelismo de una manera clara y dentro de un modelo de programación coherente es difícil. En general, las aproximaciones basadas en el análisis en tiempo de compilación, están limitadas desde el punto de vista de las formas del paralelismo de tareas que pueden soportar, mientras que aquellas que se basan en el sistema en tiempo de ejecución (*runtime system*) requieren que el programador tenga que manejar el paralelismo de tareas a un nivel más bajo que el paralelismo de datos.

En [Bal y Haines 98] se muestran algunas de estas aproximaciones y las dificultades que presenta la implementación eficiente de cada una de ellas. Integrar las dos formas de paralelismo no es una tarea fácil, puesto que existen muchas diferencias entre ellas. Una diferencia fundamental es la forma en que se estructura un programa. En los lenguajes con paralelismo de datos, existe un único programa que se ejecuta en todas las máquinas resultando un modelo SPMD (*Single Program, Multiple Data*). Por el contrario, el paralelismo de tareas consiste en varios tipos de procesos que se ejecutan de forma bastante independiente unos de otros. Estos programas se escriben en el que se denomina estilo MIMD (*Multiple Instructions, Multiple Data*). Debido a estas diferencias, la mayoría de las aproximaciones son más cercanas al estilo SPMD o al MIMD, haciendo restricciones en las construcciones que se pueden realizar.

Otra diferencia importante es la organización del espacio de direcciones. En los lenguajes con paralelismo de datos se ofrece un espacio de arrays global que es compartido entre los distintos procesos. De esta forma, en entornos distribuidos, el compilador es el que se encarga de generar código para la transferencia de datos, que de esta forma aparece transparente al programador. Los lenguajes con paralelismo de tareas generalmente ofrecen un espacio de direcciones separado, con lo cual, es el programador el que tiene que introducir explícitamente las sentencias de envío y recepción de datos.

La organización del espacio de direcciones obliga a que los compiladores de los lenguajes con paralelismo de datos dependan de análisis extensos del programa para, por ejemplo, realizar planificación de comunicaciones o agrupar múltiples envíos en un solo mensaje. Consecuentemente, estos lenguajes se diseñan generalmente para permitir que el análisis se realice de forma estática en tiempo de compilación. Los compiladores de lenguajes con paralelismo de tareas son mucho menos “heroicos”. De hecho, muchos de estos lenguajes usan la tecnología de los compiladores tradicionales. Como estos lenguajes no necesitan tanto del compilador, imponen generalmente menos restricciones que los lenguajes con paralelismo de datos. Típicamente, permiten de forma dinámica tanto la creación de procesos como la asignación de estos a procesadores, lo cual no suele estar permitido en el otro modelo.

A pesar de las dificultades de la integración, se han realizado algunas aproximaciones de las cuales comentamos las que han tenido más trascendencia, o bien, las que tienen alguna relación con el trabajo que se ha realizado en la elaboración de esta tesis.

1.3.1 Orca.

El lenguaje de paralelismo de tareas Orca ha sido extendido para el manejo de paralelismo de datos [Hassen y Bal 96][Hassen y otros 98]. Orca permite la creación dinámica de procesos y su asignación a procesadores. La comunicación no se basa en paso de mensajes sino en objetos compartidos, que son instancias de tipos abstractos de datos (TADs). Los procesos se comunican aplicando las operaciones de los TADs definidos por el usuario en los objetos compartidos.

En la extensión de Orca, el paralelismo de datos se expresa mediante objetos partidos que son objetos que contienen arrays que pueden ser distribuidos entre varios procesadores. Una operación de paralelismo de datos en un objeto partido se realiza en paralelo por esos procesadores. La distribución se expresa por el usuario invocando primitivas de tiempo de ejecución que especifican el conjunto de procesadores a utilizar y el dueño de cada elemento. Esta distribución puede cambiarse en tiempo de ejecución.

El modelo de Orca es, sin embargo, más apropiado para el paralelismo de tareas ya que impone restricciones al paralelismo de datos. El modelo no soporta bien las operaciones que utilizan paralelismo de datos en varios arrays porque estas operaciones siempre se aplican a un único objeto. Es posible almacenar varios arrays en un único objeto, pero entonces, éstos tendrán la misma distribución, lo cual no siempre es aconsejable como, por ejemplo, en la multiplicación de matrices.

1.3.2 HPF/MPI.

En [Foster y otros 96][Foster y otros 97] se describe la integración de la biblioteca de paso de mensajes MPI [MPIF 95][Gropp y otros 94] dentro de un lenguaje de paralelismo de datos, HPF. De esta forma, se permite la llamada a subrutinas MPI desde programas HPF. Como las directivas de HPF se escriben como comentarios, si uno ignora estas directivas, el código se puede interpretar como si implementara un conjunto de tareas secuenciales que se comunican mediante el uso de funciones MPI.

En los artículos publicados se observa cómo esta integración no es trivial. El problema surge cuando dos tareas HPF quieren enviarse un array que está distribuido de forma distinta en el receptor y emisor. Se presentan distintas posibilidades de solución, haciendo referencia a algoritmos para la comunicación en paralelo de datos entre distintos procesadores.

La implementación de HPF/MPI se ha hecho sobre un compilador comercial de HPF (Portland Group) y se han tenido que hacer algunas modificaciones al código fuente del compilador. No se ha integrado todo el estándar MPI sino sólo las funciones más utilizadas.

Este trabajo proviene de uno anterior [Foster y otros 94] en el que se integraba HPF con una extensión de Fortran llamada Fortran M [Foster y Chandy 92]. Este lenguaje provee a Fortran de canales con tipo punto a punto y de mecanismos para manejar el paralelismo de tareas. Parece ser que la integración del paralelismo de datos y tareas con Fortran M y HPF no llegó a implementarse puesto que los resultados ofrecidos son teóricos.

1.3.3 KeLP-HPF.

Se trata de una herramienta software basada en C++ para implementar métodos multinivel y de malla adaptable (es decir, creada para resolver problemas científicos y, en concreto, PDEs). Este trabajo es una ampliación de KeLP [Fink y Baden 98] para la integración del paralelismo de datos y tareas.

KeLP es una biblioteca específica que soporta estructuras de datos irregulares y dinámicas. Las abstracciones de KeLP representan descomposiciones de datos y patrones de comunicación. Las aplicaciones pueden ser escritas en cualquier lenguaje.

Como modelo de programación, se puede ver como paralelismo de datos de grano grueso. Proporciona al programador la ilusión de un único espacio de índices global y una única hebra de control.

El código de una aplicación es escrito típicamente en dos niveles:

- Código KELP de alto nivel que maneja las estructuras de datos y el paralelismo.
- Código de bajo nivel escrito en C, C++ o Fortran para implementar el cálculo numérico.

Para integrar el paralelismo de datos con el de tareas, se ha desarrollado KeLP-HPF [Merlin y otros 98]. De esta forma, se puede mejorar el rendimiento del paralelismo de datos en un sistema irregular estructurado por bloques, ya que se está sacando partido a dos niveles de paralelismo: entre bloques y dentro de ellos. KeLP se encarga de manejar la distribución de datos y las comunicaciones entre bloques y de invocar a HPF concurrentemente en cada bloque.

KeLP no implica extensiones al lenguaje HPF (recuérdese que se trata de código C++). Sin embargo, para su implementación, es necesario modificar tanto el compilador de HPF utilizado (SHPF [Merlin y otros 96], un compilador de dominio público que está basado en Fortran 90 y MPI para realizar el paso de mensajes) como su sistema en tiempo de ejecución. Este sistema se ha modificado para permitir la ejecución de código HPF en subconjuntos de procesadores definidos dinámicamente (lo cual no forma parte del estándar HPF).

La comunicación de las distintas tareas se hace siempre desde KeLP, no existiendo la posibilidad de modificar el código HPF (teóricamente original) con instrucciones de KeLP. El paso de información a estas subrutinas HPF se hace en la cabecera de los subprogramas.

Al igual que KeLP, su extensión, KeLP-HPF, no consiste en un lenguaje de coordinación sino en una herramienta específica que permite la integración de distintos niveles de paralelismo. En el lenguaje KeLP se crean distintas tareas HPF pero la comunicación entre éstas se define en el propio lenguaje KeLP (fuera de las rutinas HPF).

1.3.4 HPF 2.0.

Las carencias que HPF poseía en cuanto al manejo de estructuras de datos irregulares y a la integración del paralelismo de datos y tareas hicieron surgir la nueva versión de este lenguaje, denominado HPF 2.0 [HPFF 97].

Su definición se divide en dos partes: el lenguaje HPF 2.0 en sí y una serie de extensiones aprobadas. La especificación del lenguaje incluye características básicas que se esperaba que fuesen implementadas en los primeros compiladores de HPF. Las extensiones aprobadas incluyen características para resolver problemas específicos como, por ejemplo, la solución eficiente de problemas irregulares. Entre estas cabe destacar:

- Distribución de arrays en subconjuntos de procesadores.

- Nuevas distribuciones de datos. Además de las ya utilizadas `BLOCK` y `CYCLIC`, se añaden dos nuevas formas de distribución: `GEN_BLOCK` e `INDIRECT`.

- Paralelismo de datos y tareas. Se han aprobado tres nuevas extensiones para sacar partido a ambos tipos de paralelismo. Las tres están asociadas al concepto de "conjunto de procesadores activos" que permite restringir en algún momento el conjunto de procesadores que ejecutarán una o varias instrucciones.

  1. La directiva `ON` permite especificar en qué procesador se ejecutarán las instrucciones siguientes.
  2. La cláusula `RESIDENT` permite dar información al compilador de que el dato indicado se encuentra en el conjunto de procesadores activos. De esta forma, se intenta evitar la realización de comunicaciones innecesarias.
  3. Para permitir la ejecución de códigos diferentes en procesadores distintos se ha añadido la estructura `TASK_REGION`. Mediante el uso de la estructura `ON`,

`END ON` se pueden definir distintas tareas a ejecutar dentro de la región `TASK_REGION`, `END TASK_REGION`.

Este modelo de integración es una evolución del modelo del compilador Fx [Gross y otros 94][Subhlok y Yang 97], el cual, al igual que HPF 2.0, no permite que las distintas tareas se comuniquen entre sí. De esta forma, estas comunicaciones hay que hacerlas con asignaciones entre variables distribuidas entre los distintos procesadores en los que se ejecutan las tareas. Además, estas asignaciones hay que hacerlas fuera del subprograma que realice la tarea, en el mismo nivel en el que se llama a éstas. Esto limita bastante el número de aplicaciones que se pueden programar de forma sencilla y clara.

1.3.5 Adaptor.

El compilador de HPF de libre disposición Adaptor [Brandes 99a] dispone en su versión 7.0 de una biblioteca para la comunicación entre tareas HPF [Brandes 99b] (que como se ha visto anteriormente no se corresponde al estándar de HPF 2.0). Esta biblioteca se denomina `hpf_task_library` y permite la comunicación de una variable entre dos tareas, independientemente de su distribución. Además, cuando esta comunicación se tiene que realizar repetidas veces, se ofrecen unas primitivas para que el intercambio de la información sobre la distribución de la variable entre tareas sólo se tenga que realizar la primera vez.

Las tareas se pueden crear de dos formas diferentes. La primera de ellas consiste en la llamada a distintas subrutinas dentro de la estructura `TASK_REGION` de HPF 2.0. De esta manera, unos procesadores dentro del programa ejecutan una subrutina y otros procesadores, otras. La segunda forma consiste en tener programas HPF independientes que se ejecutan a la vez (MIMD) y que con ayuda de esta biblioteca se pueden comunicar.

1.4 Modelos y Lenguajes de Coordinación.

Los lenguajes de coordinación son una clase de lenguajes de programación paralela que ofrecen una solución al problema del manejo de la interacción entre distintos programas concurrentes. El propósito de un modelo de coordinación y su lenguaje asociado es proporcionar una integración entre un número significativo de componentes, posiblemente heterogéneos, de manera que el colectivo forme una única aplicación que pueda ejecutarse sobre sistemas paralelos y distribuidos con las ventajas que ello supone.

Diversos modelos y lenguajes interesantes han sido propuestos y aplicados a la paralelización de programas secuenciales intensivos en el campo de: la simulación de fluidos, dinámica de sistemas, composición de cadenas DNA, síntesis molecular, simulación paralela y distribuida, gráficos por ordenador, etc.

Parece necesario pues, describir más en detalle en qué consiste el paradigma de la coordinación describiendo algunos de los lenguajes más significativos para luego ver algunos de estos lenguajes utilizados para la integración del paralelismo de datos y tareas.

1.4.1 Paradigma de la Coordinación.

El concepto de coordinación no es exclusivo de la computación sino que se puede aplicar a gran cantidad de disciplinas. De esta forma, se pueden tener gran cantidad de definiciones, desde una tan simple y general como “coordinación es el manejo de dependencias entre entidades”, hasta definiciones más elaboradas, como “coordinación es el proceso de información adicional llevada a cabo cuando múltiples actores conectados entre sí persiguen unos determinados objetivos que un único actor, persiguiendo los mismos objetivos, no realizaría” [Malone y Crowston 94].

En el área de la computación, probablemente la definición más aceptada de coordinación es la dada en [Carriero y Gelernter 90]: “coordinación es el proceso de construir programas ‘pegando’ diferentes piezas activas”. Cada una de estas piezas activas es un proceso, una tarea, una hebra o cualquier otro elemento de ejecución

independiente (concurrente y de forma asíncrona) del resto. “Pegar” diferentes piezas activas significa reunirlas en un conjunto de forma que éste constituya el programa. El mecanismo de unión debe permitir la comunicación y sincronización entre las distintas piezas exactamente en la forma que éstas necesitan. Un modelo de coordinación es el “pegamento” que une las actividades separadas para formar el conjunto.

Un *modelo de coordinación* se puede ver como una terna (E,M,L), donde E representa las entidades a coordinar, M el medio utilizado para coordinarlas y L las “leyes” de coordinación que regulan las acciones llevadas a cabo por dichas entidades, es decir, el marco de trabajo que sigue el modelo [Wegner 96][Ciancarini 97]. Un *lenguaje de coordinación* es el conjunto de elementos lingüísticos necesarios para la integración del modelo de coordinación correspondiente en un determinado lenguaje de programación (el cual contiene el modelo de computación y se denomina comúnmente lenguaje base o lenguaje *“host”*), con el objeto de ofrecerle facilidades para la comunicación, sincronización, creación y terminación de actividades de cómputo.

Existen diversas publicaciones [Ciancarini y Hankin 96][Ciancarini 97] [Papadopoulos y Arbab 98] en las que se recoge el estado del arte en este paradigma de la coordinación. En poco más de una década han aparecido numerosos modelos y lenguajes de coordinación. Está fuera del alcance de este trabajo hacer una relación de todos ellos por lo que nos limitamos a comentar los siguientes: Linda [Gelernter 85], MANIFOLD [Arbab y otros 93] y TCM [Rubio 98]

Linda puede considerarse históricamente como el primer miembro de la familia de lenguajes de coordinación, proporcionando una forma simple y elegante de separar los aspectos de computación y coordinación. Está basado en el paradigma de la comunicación generativa: si dos procesos desean intercambiar un dato, denominado *tupla* (secuencia de valores con tipo), el emisor lo sitúa en algún espacio de datos compartido (conocido como *espacio de tuplas*) de donde lo recogerá el receptor. Este paradigma desacopla los procesos tanto desde el punto de vista del espacio como del tiempo, en el sentido de que ningún proceso necesita conocer la identidad de los demás, ni es necesario que todos los procesos involucrados en la computación estén presentes al mismo tiempo. En Linda se distinguen dos tipos de tuplas: las tuplas pasivas que

contienen sólo datos y las activas que representan procesos a ejecutar y que tras su ejecución se convierten en tuplas pasivas.

El modelo de Linda puede integrarse en casi cualquier lenguaje o paradigma de programación, imperativo, declarativo, orientado a objetos, etc. De hecho, ha sido integrado en numerosos lenguajes tales como C, Modula, Pascal, Ada, Prolog, Lisp y Eiffel. Aunque, por un lado, el modelo de Linda es atractivo y sencillo, por otro, esta sencillez hace que su implementación no lo sea tanto, sobre todo cuando se trata de entornos distribuidos, existiendo también algunas implementaciones en las que no se ofrecen todas las primitivas.

Con la aparición de Linda toma fuerza el concepto de modelos de coordinación y surge el de lenguajes de coordinación. Linda ha inspirado la creación de otros lenguajes similares, algunos de los cuales son extensiones directas de su modelo básico, mientras que otros difieren significativamente de él.

MANIFOLD es uno de los lenguajes más representativos de la familia de los modelos y lenguajes dirigidos por control, es decir, aquellos en los cuales el estado de la computación en un instante dado está únicamente definido en términos de los patrones de coordinación a los que están sujetos los procesos involucrados en la computación[1].

En realidad, este lenguaje está basado en el modelo de coordinación **IWIM** (*Idealized Worker Idealized Manager*) [Arbab 96]. Los conceptos básicos de este modelo son los procesos, eventos, puertos y canales. Un proceso es una caja negra con unos puertos de conexión unidireccionales a través de los cuales intercambia información con los demás procesos de su entorno. Las interconexiones entre puertos se realiza a través de canales punto a punto. Se permite la comunicación tanto síncrona como asíncrona.

[1] En contraste con los lenguajes de coordinación dirigidos por datos en los que el estado de la computación en un instante dado se define en términos de, por un lado, los valores de los datos que están siendo enviados y recibidos por los componentes del sistema y, por otro, de la configuración de estos componentes coordinados [Papadopoulos y Arbab 98].

Un proceso no necesita conocer la identidad de los procesos con los que intercambia información. Un proceso IWIM puede ser un "trabajador" o un proceso "coordinador". La responsabilidad del primero es realizar una tarea (computacional). Un proceso trabajador no es responsable de la comunicación necesaria para que él pueda obtener los datos de entrada que necesita, ni del mecanismo de comunicación necesario para enviar a su entorno los resultados que él produce en sus puertos de salida. El único proceso, relacionado con su comunicación, que lleva a cabo es una mera lectura de sus puertos de entrada y una escritura en sus puertos de salida, además de una posible elevación de eventos. Es siempre responsabilidad de un proceso coordinador realizar las tareas necesarias para la coordinación entre los procesos trabajadores: creación dinámica de procesos, diseminación y reacción ante ocurrencias de eventos, creación y destrucción dinámica de canales de conexión entre los puertos de los procesos coordinados. En realidad, en este modelo, ningún proceso es responsable de su propia comunicación con otros procesos. En una aplicación siempre existirá una capa en el nivel inferior de procesos trabajadores, denominados atómicos. La aplicación se construye mediante una jerarquía dinámica de procesos (trabajadores y coordinadores) encima de esta capa. Sin tener en cuenta los procesos trabajadores atómicos, la distinción entre trabajador y coordinador es subjetiva: un proceso coordinador *c* que coordina la comunicación de un determinado número de trabajadores, puede ser también considerado como trabajador por otro coordinador responsable de coordinar la comunicación de *c* con otros procesos.

MANIFOLD es una versión concreta del modelo IWIM. Todas las comunicaciones son asíncronas y no existe esa relajación en la subjetividad mencionada sobre la distinción entre procesos trabajadores y coordinadores. En este lenguaje, la separación entre coordinación y computación es más fuerte; existen trabajadores atómicos que llevan a cabo una mera tarea computacional, comunicándose con el exterior en la forma comentada anteriormente, que pueden estar codificados en cualquier lenguaje de programación convencional, aumentado con ciertas primitivas de comunicación; y, por otro lado, están los procesos coordinadores, también llamados "*manifolds*", codificados en el lenguaje de coordinación MANIFOLD, al que no se le dota de ninguna capacidad computacional usual de otros lenguajes de programación.

MANIFOLD ha sido usado con éxito para convertir código secuencial Fortran en una aplicación paralela/distribuida [Everaars y otros 96] así como para resolver otros problemas científicos [Everaars y Arbab 96][Everaars y Koren 98].

Finalmente, hay que destacar **TCM** (*Tuple Channel coordination Model*) como parte del trabajo de investigación realizado dentro del grupo GISUM de la Universidad de Málaga. Este lenguaje de coordinación está basado en la utilización de canales de comunicación multipunto, denominados Canales de Tuplas, los cuales permiten la comunicación de estructuras complejas, denominadas tuplas, entre múltiples productores y consumidores. TCM es una evolución de un modelo previo que utilizaba canales lógicos [Díaz, Rubio y Troya 96], los cuales fueron inicialmente propuestos en la definición de **DRL** [Díaz, Rubio y Troya 94][Díaz, Rubio y Troya 97], un lenguaje lógico concurrente para programación distribuida y de tiempo real.

Una de las principales aportaciones de TCM es la posibilidad de su integración con paradigmas de programación distintos. De hecho, en [Díaz, Rubio y Troya 98] se muestra la integración con los lenguajes C, Haskell y Smalltalk como representantes de los paradigmas imperativo, declarativo y orientado a objetos. De una de sus posibles ampliaciones, la de utilizarla para comunicar tareas con paralelismo de datos, surgió la idea de BCL.

1.4.2 Integración del Paralelismo de Datos y Tareas Mediante Coordinación.

El uso de modelos y lenguajes de coordinación está demostrando ser una buena alternativa para la integración del paralelismo de datos y tareas. De esta forma, se dispone de un mecanismo de alto nivel para soportar distintas estructuras de paralelismo de tareas de manera clara y elegante.

Entre las aproximaciones más sobresalientes en este campo cabe destacar **Opus** [Chapman y otros 97], **Task Graphs** [Rauber y Rünger 99] y ***COLT*$_{\mathbf{HPF}}$** [Orlando y Perego 99].

Opus. Muchas aplicaciones científicas y de ingeniería son heterogéneas y multidisciplinarias, no adecuándose bien al paradigma de programación del paralelismo

de datos. Una forma de solucionar este problema es mediante Opus, un lenguaje de coordinación basado en la orientación a objetos y cuyo elemento principal es lo que se denominan SDA o *"ShareD Abstractions"*. Las SDA pueden ser servidores de cómputos, o bien, depósitos de datos compartidos entre distintas tareas. Este lenguaje se utiliza sobre HPF, es decir, en un programa HPF se declaran las SDA y se utilizan siguiendo una sintaxis del estilo de Fortran 90, pudiéndose integrar, de esta forma, el paralelismo de tareas con el paralelismo de datos de HPF.

Un tipo SDA especifica una estructura de objeto que contiene datos junto con los métodos que manipulan esos datos. Una SDA es generada creando una instancia de un tipo SDA. La creación de una SDA implica asignación de recursos en los cuales se ejecutará la SDA, la asignación de las estructuras de datos en memoria y cualquier inicialización que sea necesaria para establecer un estado inicial bien definido.

Un método de una SDA puede ser llamado de forma síncrona o asíncrona. En el segundo caso se puede asociar la ejecución del método a un evento que puede ser utilizado para interrogaciones de estado y para sincronización. Un método puede tener también asociada una cláusula condicional.

Cada SDA está asociada a una única tarea SDA que es el lugar de toda la actividad de control asociada a la SDA. La ejecución de un programa Opus puede ser vista como un sistema de tareas SDA en las cuales una tarea ejecuta un método de su SDA en respuesta a una demanda de otra SDA.

El modelo **Task Graphs** proporciona un lenguaje de especificación y un lenguaje de coordinación. En el primero, el programador define todo el nivel de paralelismo posible en su aplicación. En el segundo, se determina cómo el paralelismo potencial es explotado en una implementación específica. Los programas de especificación dependen sólo del algoritmo mientras que los programas de coordinación pueden ser diferentes para máquinas distintas, pudiéndose obtener, así, un mejor rendimiento.

El proceso de derivación completo en el cual un programa de especificación expresando posibles órdenes de ejecución entre módulos y describiendo el grado de paralelismo disponible (tanto de tareas como de datos) es transformado en un programa

de coordinación, se realiza mediante pasos bien definidos, por lo que puede ser automatizado. Para ayudar en este paso, se presenta un modelo de coste que permite decidir qué grado de paralelismo explotar para una arquitectura hardware concreta. El lenguaje de coordinación resultante es una descripción completa de un programa paralelo que puede ser fácilmente traducido a un programa con paso de mensajes.

Esta propuesta es más una aproximación de especificaciones que de programación. El programador es responsable de especificar el grado máximo de paralelismo disponible pero la decisión final sobre si el paralelismo de tareas disponible será explotado o no y cómo los procesadores serán repartidos en grupos es tomada por el compilador. No está basado en HPF y el programa paralelo final es escrito en C con MPI.

***COLT*$_{\mathbf{HPF}}$** es una capa de coordinación/comunicación transportable entre tareas HPF. Proporciona mecanismos adecuados para lanzar distintas tareas con paralelismo de datos en grupos disjuntos de procesadores, así como primitivas optimizadas para la comunicación entre tareas. Esta comunicación se realiza mediante canales con tipo, a través de los cuales se pueden intercambiar datos distribuidos entre los procesadores de cada tarea, de acuerdo con las primitivas que en este sentido tiene HPF.

La primera implementación de *COLT*$_{\mathrm{HPF}}$ se realizó usando MPI y el compilador de HPF de dominio público ADAPTOR en su versión 5.0 [Brandes 97]. Ello implicó la modificación del sistema de tiempo de ejecución del compilador para permitir la ejecución de tareas HPF simultáneas que intercambien datos [Brandes 99b]. En su segunda implementación [Orlando y otros 00a], se utilizó PVM para permitir la creación dinámica de tareas y el uso de compiladores comerciales como el de Portland Group.

Ya en la presentación de este lenguaje, se menciona que *COLT*$_{\mathrm{HPF}}$ es de demasiado bajo nivel y que es posible utilizar un lenguaje de más alto nivel basado en patrones para ocultar muchos de los detalles necesarios cuando se trabaja con tareas y canales.

1.5 Programación Paralela Estructurada. Patrones o Esqueletos.

Una forma de inferir la estructura que se va a usar para ejecutar un programa abstracto es exigir que el programa abstracto esté basado en unidades fundamentales o componentes cuyas implementaciones están predefinidas. En otras palabras, los programas se construyen mediante la conexión de bloques ya hechos. Esta aproximación tiene las siguientes ventajas:

- Los bloques elevan el nivel de abstracción porque son las unidades fundamentales con las que trabajan los programadores, escondiendo así una cantidad importante de complejidad interna.

- Los bloques pueden ser internamente paralelos pero ser compuestos de forma secuencial, en cuyo caso los programadores no necesitan tener en cuenta el hecho de que, en realidad, están haciendo programación paralela.

- Para arquitecturas hardware distintas, sólo se tiene que cambiar la implementación de cada bloque. Además, esta implementación puede estar realizada por especialistas en paralelismo de forma que se le puede dedicar tiempo y dinero para que sean eficientes.

En el contexto de la programación paralela tales bloques reciben el nombre de esqueletos o patrones (*skeletons* o *patterns*) [Cole 89] y son la base de un número importante de modelos. Cole propuso una "*skeletal machine*" compuesta por 4 esqueletos candidatos: *Fixed Degree Divide & Conquer*, *Iterative Combination*, *Cluster* y *Task Queue*. De esta forma, el programador tiene que resolver su problema eligiendo entre uno de estos cuatros esqueletos. Sin embargo, estos esqueletos son demasiado complejos, no ocultan todos los detalles de la implementación y no son lo suficientemente generales.

La propuesta de Darlington [Darlington y otros 93] es de más alto nivel y estrictamente declarativa, abstrayendo totalmente la implementación. **SCL** [Darlington y otros 95] suministra un conjunto más rico de esqueletos y permite la composición de

éstos. Sin embargo, la transformación a un programa ejecutable es costosa y necesita, de alguna forma, la ayuda del programador.

Otras propuestas más actuales son: ***P*^{3}*L*** [Bacci y otros 95], **Activity Graphs** [Cole y Zavanella 00] y **taskHPF** [Orlando y otros 00b][Ciarpaglini y otros 00].

P*^{3}*L (*Pisa Parallel Programming Language*) usa un conjunto de esqueletos algorítmicos que capturan paradigmas de programación paralela comunes tales como *pipelines, worker farms* y *reducciones*, así como la composición estructurada de estos. Además, por cada esqueleto se presentan distintas plantillas de implementación para distintas arquitecturas hardware. También presenta un modelo de costes (tiempo de ejecución) de modo que el programador puede predecir el coste computacional de su aplicación antes de ejecutarla. El lenguaje base en el que se sustenta es C++.

Los **Activity Graphs** presentan una capa intermedia para el proceso de compilación de un lenguaje basado en esqueletos, haciendo hincapié en la semántica. Esta capa es independiente del lenguaje de esqueletos que se esté empleando y sirve para proporcionar una semántica operacional a los lenguajes basados en esqueletos. En [Cole y Zavanella 00] se presentan los mecanismos para pasar de un lenguaje de esqueletos a los Activity Graphs correspondientes y cómo, a partir de estos, generar un programa basado en MPI.

taskHPF es un lenguaje de coordinación de alto nivel para definir los patrones de interacción entre tareas HPF de forma declarativa. Las aplicaciones consideradas son estructuradas como agrupaciones de módulos HPF independientes, los cuales interaccionan de acuerdo a patrones estáticos y previsibles. taskHPF proporciona un patrón *pipeline* y directivas que ayudan al programador a balancear las etapas del encauzamiento: la directiva `ON PROCESSORS` fija el número de procesadores asignados a una tarea HPF y la directiva `REPLICATE` puede ser usada para replicar etapas que no son escalables. Los patrones pueden ser combinados para crear estructuras complejas de forma declarativa.

En realidad, taskHPF es la aplicación de P^3L para el paralelismo de datos y tareas y su implementación se realiza mediante la creación de plantillas que contienen

las llamadas para la comunicación. Esta comunicación se realiza mediante la creación de los canales y las primitivas de $COLT_{\mathrm{HPF}}$.

1.6 Nuestra Aproximación, BCL.

La computación paralela tiene unos 20 años de antigüedad. En este tiempo, se han podido resolver problemas complejos con un gran rendimiento en áreas tradicionales como la ingeniería y la ciencia y en nuevas aplicaciones como la inteligencia artificial o las finanzas. Sin embargo, a pesar de algunos éxitos y de un comienzo esperanzador, la computación paralela no ha llegado a ser muy importante en la informática, y los ordenadores paralelos vendidos sólo representan un pequeño porcentaje del total.

Esto es debido, entre otras razones, a que la teoría en el desarrollo de la programación paralela ha ido por detrás de la tecnología hardware, como se mencionó anteriormente. Hay que tener en cuenta, además, que la ejecución de un programa paralelo es algo extremadamente complejo. Para poner un ejemplo, considérese un programa ejecutándose en una máquina con 100 procesadores (grande pero no inusual hoy día). Existen 100 hebras de control activas en cada momento. Cada una de ellas se puede comunicar con cualquiera de las otras y esta comunicación puede ser asíncrona o puede necesitar una sincronización con la hebra destino. Por tanto, hay 100^2 posibles interacciones en progreso en un instante dado. El estado de un programa como este es demasiado grande.

El objetivo de nuestra aproximación es el de proporcionar un modelo muy sencillo de programación paralela, de modo que el programador de aplicaciones científicas se pueda centrar en la solución de su problema, pudiendo abstraerse de la complejidad que supone la ejecución paralela de su código.

Con este propósito es con el que se ha desarrollado **BCL** (*Border-based Coordination Language*) [Díaz, Rubio, Soler y Troya 99][Díaz, Rubio, Soler y Troya 00a], un lenguaje de coordinación para la solución numérica de problemas basados en la descomposición de dominios, especialmente aquellos con superficie irregular que pueden ser descompuestos en subdominios formados por bloques regulares. Usando este

lenguaje, los aspectos de coordinación de una aplicación se pueden separar claramente de los métodos numéricos, viéndose incrementada la posibilidad de reutilizar tanto la parte de coordinación como el código numérico.

En la parte de la coordinación se definen los diferentes bloques que forman el dominio global del problema y las distintas fronteras entre esos bloques. Además, la forma en la que esas fronteras serán actualizadas también se especifica en la parte de coordinación, la cual se escribe completamente en BCL (que tiene una sintaxis basada en Fortran 90/HPF). La parte computacional puede ser escrita en Fortran 77, 90 o HPF junto con unas pocas extensiones BCL. De esta forma, se consiguen dos objetivos: el programador de aplicaciones científicas no necesita usar varios lenguajes, sino que puede escribir ambos, la parte de coordinación y la de programación, con el mismo tipo de lenguaje (excepto por algunos aspectos de coordinación), de manera fácil y clara. Por otro lado, se consigue la integración de paralelismo de datos y tareas, explotándose así dos niveles de paralelismo: entre subdominios y dentro de ellos [Díaz, Rubio, Soler y Troya 00b].

Aunque BCL se ha utilizado satisfactoriamente para la resolución de problemas de descomposición de dominios y multibloque, otras aplicaciones que sacan rendimiento a la integración del paralelismo de datos y tareas, pueden ser expresadas más fácilmente mediante el uso de patrones o esqueletos. Este hecho hizo surgir a **DIP** [Díaz, Rubio, Soler y Troya 01a], un conjunto de construcciones de más alto nivel para expresar el paralelismo de tareas entre una colección de tareas HPF con paralelismo de datos, las cuales interaccionan mediante patrones estáticos y predecibles.

Usando las características de DIP, una aplicación se organiza como una combinación de patrones comunes, tales como `multiblock` y `pipe`. Los patrones especifican la interacción entre los dominios de la aplicación junto con la asignación a procesadores y la distribución de datos. Por otro lado, el uso de dominios, que son regiones junto con alguna información adicional, como son las fronteras, hace que sea adecuado para la solución de problemas numéricos, especialmente aquellos con una superficie irregular que puede ser estructurada en bloques regulares. Además, otros tipos de problemas con un patrón de comunicación basado en el intercambio de (sub)matrices (*FFT2D*, *Convolución*, *Narrowband Tracking Radar*, etc.) pueden ser

definidos y resueltos de forma fácil y clara. El uso de dominios evita que algunos aspectos computacionales de la aplicación, tales como los tipos de los datos, tengan que aparecer en el nivel de coordinación. De esta forma se incrementa la reusabilidad del patrón. Por otro lado, la codificación de la parte de cómputo se facilita mediante el uso de las plantillas de implementación que permiten al programador utilizar un nivel de abstracción más alto para desarrollar sus aplicaciones, mejorándose también la posibilidad de reutilizar el código computacional.

En cuanto al trabajo relacionado comentado previamente, la única herramienta diseñada expresamente para sacar partido a la descomposición de dominios es SciAgents. Sin embargo, no se trata de un lenguaje, sino de una herramienta gráfica diseñada para problemas de descomposición de dominios en PDE elípticas. BCL permite afrontar otros tipos de problemas, tanto parabólicos como aquellos cuyo patrón de comunicación se base en el intercambio de (sub)matrices. Aunque SciAgents permite varios niveles de paralelismo, su objetivo no es la eficiencia, sino el desarrollo rápido de aplicaciones con múltiples dominios. Por esta razón, no se optimiza la comunicación entre los distintos agentes que resuelven cada dominio.

Con BCL, al tratarse de un lenguaje de coordinación, se puede separar claramente la parte de coordinación de la de cómputo, al contrario que ocurre en otras propuestas como Orca, HPF/MPI o Adaptor (en cuanto a la integración del paralelismo de datos y tareas se refiere). Los dos últimos son de más bajo nivel puesto que obligan a la inclusión explícita de las instrucciones de paso de mensajes entre las tareas. Si, además, se pretende una implementación eficiente, es necesario utilizar instrucciones adicionales para el intercambio de los descriptores de distribución de los arrays entre las tareas HPF.

KeLP-HPF, a pesar de no ser un lenguaje de coordinación, permite separar la parte de definición de las estructuras de datos y el paralelismo del código encargado de implementar el cálculo numérico. Sin embargo, si se quiere sacar partido a la integración del paralelismo de datos y tareas, se obliga al usuario a tener en mente, no sólo la aplicación y el paralelismo, sino también dos lenguajes de programación HPF y C++ que, además, pertenecen a dos paradigmas distintos, el imperativo y el orientado a objetos. Asimismo, la comunicación entre las tareas se debe realizar siempre con

instrucciones KeLP y pasando los datos a las tareas HPF mediante la cabecera de las subrutinas.

En cuanto a los lenguajes de coordinación que permiten la integración del paralelismo de datos y tareas, Opus está pensado para problemas heterogéneos y multidisciplinares mientras que el nuestro está orientado a problemas numéricos donde se coordinan un conjunto de tareas HPF estáticamente predefinidas. En Task Graphs, el programador es el responsable de definir los niveles de paralelismo posibles pero la decisión final la toma el sistema. No está basado en HPF sino que el resultado final es C con llamadas a MPI. En $COLT_{\mathrm{HPF}}$, el esquema de comunicación entre tareas no puede ser establecido en tiempo de compilación y, además, la implementación de este sistema conlleva cambios en el compilador de HPF en el que se basa.

De los sistemas basados en el uso de patrones, el único que hasta la fecha permite la integración del paralelismo de datos y tareas es taskHPF. Si bien las construcciones de BCL englobadas bajo el nombre de DIP tienen ciertas semejanzas con taskHPF, en nuestro sistema se trabaja con dominios en lugar de canales con tipo, lo que permite no tener que especificar los tipos de los datos a nivel de coordinación. Además, el esquema de comunicación se establece en tiempo de compilación (lo que evita tener que hacerlo por cada envío) y nuestro compilador es capaz de obtener más información de los patrones (por ejemplo el número de instrucciones de envío/recepción necesarias) lo que hace que los programas resultantes sean más flexibles.

1.7 Estructura de la Memoria.

La memoria se ha organizado en siete capítulos de los que el primero constituye esta introducción.

En el siguiente capítulo se describe el lenguaje de coordinación BCL. En primer lugar se presenta el lenguaje con el esquema típico de un problema resuelto con BCL. A continuación, se describen las características más importantes del lenguaje, diferenciando entre las primitivas que se incorporan para los procesos coordinadores y las primitivas que utilizan los procesos trabajadores. Seguidamente se presenta un ejemplo sencillo y comentado donde sólo se utilizan esos aspectos básicos del lenguaje.

A continuación se profundiza en otros aspectos que se han incorporado para aumentar su expresividad. Por último, se presentan otros ejemplos para explicar de forma más clara cada una de las características del lenguaje.

En el capítulo 3 se muestra la utilización de BCL para la integración del paralelismo de datos y tareas. Para ello, se explica primero la problemática de esta integración y se comentan los inconvenientes de otras aproximaciones. A continuación, se explican las nuevas primitivas con las que se ha extendido BCL para afrontar la integración y se muestran ejemplos.

El cuarto capítulo explica las características con las que se ha extendido BCL para el uso de patrones o esqueletos, lo que hemos denominado DIP. En primer lugar, se describen los patrones introducidos para mejorar la expresividad de la parte de coordinación de una aplicación y la posibilidad de utilizar plantillas de implementación para facilitar la programación de la parte computacional. A continuación se presentan algunos ejemplos para ilustrar el modelo.

Los detalles de la implementación de BCL en lo que se refiere a la integración del paralelismo de datos y tareas y a su ampliación para el manejo de patrones y plantillas se muestran en el capítulo 5. En primer lugar se explican distintas alternativas para su implementación y se justifica la elegida.

En el capítulo 6 se presentan distintos problemas resueltos mediante BCL (con o sin su extensión DIP) y se muestra la eficiencia de la implementación. En uno de los apartados se estudia en profundidad un problema numérico que ha sido resuelto mediante distintas técnicas de descomposición de dominios, tanto desde el punto de vista de la precisión de los resultados como de su eficiencia. El esquema de resolución de este problema se presenta en el anexo.

El séptimo y último capítulo muestra las conclusiones obtenidas tras la realización del trabajo presentado en esta tesis, se enumeran sus contribuciones y se señalan posibles líneas de investigación futuras.

Capítulo 2. BCL. Un lenguaje de coordinación basado en fronteras

Nuestra aproximación está pensada para ser utilizada por una clase de usuarios (tales como ingenieros, matemáticos y físicos), que aún sabiendo programar, no están acostumbrados a utilizar un lenguaje de programación paralelo de alto nivel que, además, no siempre proporciona la eficiencia deseada para un tipo de aplicaciones que requieren gran cantidad de recursos computacionales. Hay que tener en cuenta que utilizar un modelo de programación de bajo nivel, como puede ser el uso de bibliotecas estándar como MPI o PVM, hace que el desarrollo de software sea complicado y tendente a la aparición de errores, puesto que el programador tiene que tener en mente, no sólo el problema que está resolviendo y el lenguaje con el que lo está programando sino también, el modelo de paralelismo, el paso de mensajes, la sincronización entre procesos, etc.

BCL es un lenguaje de coordinación ideado para definir de una forma fácil la solución numérica de problemas científicos, especialmente aquellos basados en la descomposición de dominios. Este tipo de problemas suele ser heterogéneo, lo cual implica una complejidad añadida a la resolución del problema. La heterogeneidad a la que nos referimos puede ser de cuatro tipos:

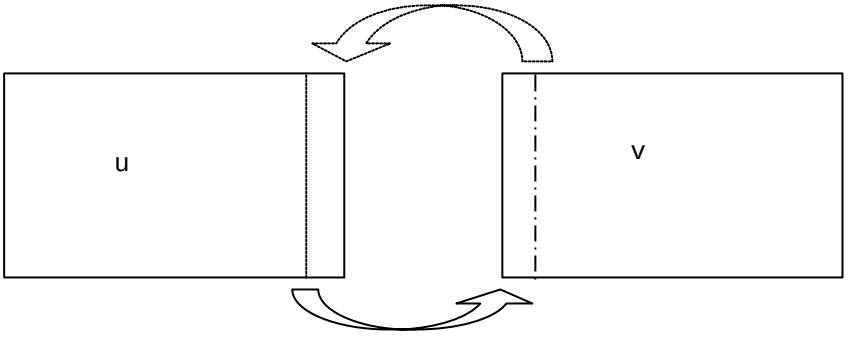
- Física: Los problemas a resolver siguen procesos físicos distintos.
- Algorítmica: La forma de resolver cada parte del problema es distinta.
- Geométrica: El dominio de la aplicación puede ser irregular.
- Computacional: La arquitectura de los ordenadores que resuelven cada parte del problema puede ser distinta.

Con este lenguaje se intenta ofrecer un modelo de programación que resulte fácil de aprender y que se pueda utilizar cómodamente tanto para la definición y resolución de este tipo de aplicaciones, como para proporcionar un modelo de paralelismo sencillo que puede ser de gran utilidad a la clase de usuarios mencionados anteriormente.

Los objetivos marcados a la hora de desarrollar BCL son, por un lado, buscar un lenguaje que sea fácil tanto de aprender como de utilizar y, por otro, que su implementación se pueda realizar de forma eficiente. Para cumplir el primer objetivo, se han separado los aspectos de sincronización y comunicación entre los distintos procesos de la parte de cómputo, de modo que, cuando se está definiendo la primera, el programador no tenga que tener en cuenta todos los detalles de la segunda y viceversa. Para lograr una implementación eficiente, se saca partido a la integración del paralelismo de datos y tareas. En este capítulo nos centraremos en describir el lenguaje sin la integración de ambos tipos de paralelismo ya que esto se realiza en el capítulo siguiente.

BCL no es un lenguaje de propósito general, sino que está pensado para problemas numéricos donde la comunicación se reduce al intercambio de (sub)matrices, como, por ejemplo, problemas multibloque y de descomposición de dominios. En el apartado 1.1 se hizo una introducción de los métodos de descomposición de dominios, puesto que éstos son los tipos de problemas a los que está orientado BCL. A continuación, se muestra el esquema general de un problema de este tipo (*Figura 1*). En este ejemplo, se han definido dos dominios `u` y `v`, cuyos puntos interiores se resuelven de forma separada. Una vez resueltos estos valores de ambos dominios, es necesaria una comunicación de los valores calculados para la frontera. Cuando estos llegan, se aplican condiciones a las fronteras y se comprueba, mediante alguna técnica, la convergencia del método. Este proceso se repite hasta que el error en las fronteras cumpla algún criterio, como, por ejemplo, ser menor que un valor predeterminado.

Puesto que BCL está pensado para resolver este tipo de problemas, en un programa BCL habrá un proceso *coordinador* y uno o varios procesos *trabajadores*. En el proceso coordinador, se describe, por un lado, la geometría del dominio de la aplicación, y por otro, los aspectos de comunicación y sincronización entre los distintos procesos que resuelvan la aplicación. Para ello, se definen los subdominios en los cuales




```
Para un paso de tiempo:
REPETIR
      Resolver u
      Resolver v
      Intercambio de fronteras
      Resolver fronteras
HASTA que converja.
```

Figura 1. Esquema típico de resolución de un problema de descomposición de dominios.

se ha dividido el dominio general de la aplicación y el criterio de convergencia. La definición de los dominios incluye las fronteras y las funciones para actualizar estas fronteras. El proceso coordinador también es responsable de la creación de los procesos que resolverán los subdominios (procesos trabajadores).

De esta forma, se consigue que la codificación de los procesos trabajadores sea independiente del resto de subdominios de la aplicación. Al igual que ocurre con el modelo IWIM, los procesos trabajadores no necesitan saber con qué procesos tienen que comunicarse, sino que esto se establece en el coordinador, aumentándose así la posibilidad de reutilizar la parte de cómputo. Por otro lado, el proceso coordinador puede estar definiendo una geometría muy compleja, la cual puede ser reutilizada para resolver otro problema numérico con la misma geometría.

Los procesos trabajadores pueden ser implementados mediante programas distintos o mediante el mismo programa (MIMD o SPMD). En el primer caso, al escribirlos usando BCL, el programador se puede abstraer de los procesos a los que tiene que enviar o de los que tiene que recibir información y de los datos a intercambiar con el resto de los procesos. Al usar el mismo programa para trabajadores distintos, se potencia la reutilización de código, que al recibir como parámetro distintos dominios con sus respectivas fronteras, puede servir para resolver distintos subdominios del problema global. Además, el proceso coordinador puede pasar a los procesos

trabajadores un número indeterminado de argumentos, incluso funciones, de modo que, de esta forma, se puede parametrizar el proceso trabajador con el tipo de ecuaciones a resolver, la inicialización de cada subdominio, las ecuaciones a imponer en las fronteras, etc.

El hecho de que la sintaxis de BCL esté basada en Fortran 90 y HPF produce tres efectos:

- Puesto que Fortran es el lenguaje más extendido entre la clase de usuarios a la que se dirige este trabajo, tanto su aprendizaje como su utilización será más fácil.

- Ambas partes, la de coordinación y la computacional, pueden ser escritas en el mismo lenguaje y, por tanto, el programador de la aplicación no necesita aprender varios lenguajes para describir las partes del problema (al contrario que ocurre en otras aproximaciones [Fink y Baden 98]).

- Al utilizar HPF en la parte computacional, se puede sacar provecho de la integración del paralelismo de tareas entre los diferentes subdominios y al paralelismo de datos aplicado para la solución de cada subdominio. Debido a que la comunicación entre las diferentes tareas está limitada a aquellas especificadas en el proceso coordinador, se evitan los inconvenientes de otras aproximaciones, consiguiéndose así una implementación eficiente.

2.1 Esquema de un Programa BCL.

La *Figura 2* muestra el esquema típico de un programa en BCL. El proceso coordinador será el cuerpo principal del programa, *Figura 2a*. Este proceso es el responsable de la definición de los dominios del problema, las fronteras que existen entre ellos y las variables que se necesitan para el test de convergencia. También se encargará de crear los procesos trabajadores que realicen los cálculos asociados a cada dominio.

El esquema típico de un proceso trabajador es el de la *Figura 2b.* Los procesos trabajadores son declarados como una subrutina y reciben como argumentos, entre

```
program nombre_programa
declaración de DOMINIOS
declaración de CONVERGENCIA
…
definiciones de DOMINIOS
definiciones de FRONTERAS
…
CREACION de PROCESOS
end
```

a)

```
subroutine nombre_subrutina(. . .)
declaración de DOMINIOS
declaración de CONVERGENCIA
declaración de GRID

inicialización GRID
do while .not. converge
   …
   PUT_BORDERS
   …
   GET_BORDERS
   …
   computación local
   test de CONVERGENCIA
enddo
…
end subroutine
```

b)

Figura 2. Esquema típico de un programa BCL. a) El Proceso coordinador. b) El proceso trabajador.

otros, los dominios y las variables de convergencia definidas en el proceso coordinador. Las variables declaradas con el atributo `GRID` se utilizan para almacenar los datos correspondientes a cada uno de los dominios. Los cálculos locales son llevados a cabo mediante sentencias estándares Fortran90/HPF mientras que la comunicación y la sincronización entre los procesos trabajadores se realizan mediante tres primitivas sencillas pero potentes que introduce el lenguaje: `PUT_BORDERS`, `GET_BORDERS` y `CONVERGE`.

2.2 El Núcleo Básico de BCL.

Como puede verse en la *Figura 2*, algunas de las características del lenguaje se utilizan exclusivamente en los procesos coordinadores, como, por ejemplo, la definición de fronteras entre dominios. Otros aspectos se utilizan sólo en los procesos trabajadores, como es el uso de las variables con atributo `GRID`, mientras que otras características son comunes a ambos tipos de procesos. A continuación, se describen las partes básicas del lenguaje, haciendo distinción entre el proceso coordinador y los trabajadores. De esta forma, el lector se puede hacer una idea general del modelo sin entrar en otros detalles que se describen en el apartado de Aspectos Adicionales del Lenguaje. Aunque éstos

aspectos son importantes, puesto que incrementan la expresividad de BCL, su estudio se puede retrasar para comprender mejor la parte esencial del lenguaje.

2.2.1 El Proceso Coordinador.

El proceso coordinador se encarga principalmente de la declaración de los dominios sobre los que trabajará cada proceso trabajador y de la definición de las fronteras entre ellos. Para facilitar el trabajo con dominios se ha introducido el tipo de datos:

- `DOMAINdD`. Donde `d` es la dimensión del problema con el que se está trabajando y está comprendida entre 1 y 4. Una variable de este tipo consiste en $2 \cdot d$ números enteros y representa un dominio en el plano, es decir, un subconjunto de Z^d. Se ha preferido no limitar el valor de `d` a 3 porque en algunas aplicaciones (como aquellas en las que hay que almacenar varios valores por punto) puede ser interesante trabajar con una dimensión más. La declaración de las variables se realiza como en el ejemplo:

  ```
  DOMAIN2D u
  ```

  donde se declara la variable de tipo dominio de dos dimensiones denominada `u`.

  La asignación de valores a una variable de este tipo se puede realizar siguiendo el estilo de Fortran 90 como en el ejemplo que sigue. Los valores asignados corresponden a los puntos cartesianos necesarios para definir el dominio. Un ejemplo de asignación tendría la forma:

  ```
  u=(/1,1,Nx,Ny/)
  ```

  Esta instrucción asigna a la variable `u` la región del plano que se extiende desde el punto `(1,1)` hasta el punto `(Nx,Ny)`. Los cuatro números que definen `u` deben ser variables o constantes de tipo `INTEGER`. Un dominio representa una región junto con otra información que es de interés al proceso trabajador y que se define en el coordinador. Tanto el proceso coordinador como los procesos trabajadores pueden definir variables del tipo `DOMAIN`. Sin embargo, sus valores se asignan en el proceso coordinador y los trabajadores los utilizan para

consultarlos. La información que se guarda en una variable dominio consiste en la región y las fronteras del dominio[2].

- En problemas de descomposición de dominios, los procesos trabajadores han de comunicarse para comprobar si se ha alcanzado la convergencia del método. Para facilitar la comunicación y la sincronización, se ha introducido un nuevo tipo de dato, `CONVERGENCE`. El proceso coordinador declara variables de este tipo, pasándolas como parámetro a los distintos trabajadores. Los trabajadores que compartan una variable de este tipo podrán comunicarse de forma elegante para hacer un test de convergencia. El número de procesos que tomarán parte en el test es indicado opcionalmente mediante la cláusula `OF`. Por ejemplo:

```
CONVERGENCE c OF num
```

  Donde `num` es una expresión evaluable en tiempo de compilación. Sin embargo, si `num` no es especificado con la cláusula `OF`, será obligatorio asignar un valor al campo `NUM_PROCESSORS` correspondiente a la variable declarada del tipo `CONVERGENCE`.

```
c%NUM_PROCESSORS = n * p
```

  que obviamente se realiza en tiempo de ejecución.

- La creación de los procesos trabajadores se hará mediante la instrucción `CREATE`. La asignación de procesos a los procesadores se hará de forma transparente desde el punto de vista del programador,

```
CREATE  processName (u,c,. . .)
```

  donde `processName` es el nombre del segmento de código que deberá ser ejecutado como un nuevo proceso de manera asíncrona. De esta forma, varios procesos pueden ser ejecutados en paralelo. Las variables `u` y `c` son de tipo `DOMAIN` y `CONVERGENCE`, respectivamente. Aunque no es obligatorio que un proceso trabajador tome como parámetro una variable de tipo `DOMAIN`, sí es lo habitual, puesto que es la forma (junto con las variables `CONVERGENCE`) de

[2] En una variable de tipo dominio también se almacena otra información como se verá en los apartados de Aspectos adicionales y en el de Implementación.

comunicarse con el resto de los procesos. Un trabajador puede recibir como argumento varios dominios, pero un dominio no puede ser pasado a más de un trabajador.

El proceso `processName` puede recibir, además, un número arbitrario de argumentos adicionales de cualquier tipo que la aplicación necesite. También se pueden pasar como argumentos subrutinas y funciones declaradas externas. Esto es especialmente útil cuando la función principal de varios trabajadores es la misma y sólo una parte específica es diferente. Como ejemplo, cuando las condiciones iniciales impuestas para cada dominio son diferentes, pero el resto del problema es resuelto con el mismo algoritmo, la subrutina de inicialización puede ser pasada como un argumento.

- En el proceso coordinador, también se definen las diferentes fronteras entre los dominios declarados. Esto se hace mediante el conector `<-` que afecta a dos o más dominios (en general, a una región de cada uno de ellos). Por ejemplo:

  ```
  u(Nx,1, Nx,Ny) <- v(2,1, 2,Ny)
  ```

  indica que la región de `u` delimitada por los puntos `(Nx,1)` a `(Nx,Ny)` será actualizada por los valores correspondientes a la región de `v` delimitada por los puntos `(2,1)` y `(2,Ny)`.

  Los tamaños de las regiones de los dominios especificados a ambos lados del operador `<-` deben ser iguales[3] aunque no su forma. Sin embargo, sí se permite aplicar una función al lado derecho del operador donde estén implicados varios dominios (o una región de ellos).

Desde ahora llamaremos *frontera* a la conexión resultante de la aplicación del operador `<-`. Se denominará *frontera de entrada* con respecto a una variable de tipo dominio a la conexión en la cual la variable está a la izquierda del operador. De forma similar, se denominará *frontera de salida* si la variable dominio está a la derecha del operador y no a la izquierda. Hay que tener en cuenta que una variable de tipo dominio

[3] En un trabajo futuro el modelo será extendido para soportar mallas no estructuradas.

se puede encontrar a ambos lados del conector, si sus valores son actualizados usando una función que tome como argumento, tanto sus propios valores como valores de otro dominio. Un dominio puede tener varias fronteras asociadas (de entrada y/o de salida).

2.2.2 Procesos Trabajadores.

La parte de cómputo de la aplicación la realizan los procesos que han sido creados desde el coordinador. Para facilitar la codificación de estos, se introduce un atributo, denominado `GRID`, y dos nuevas instrucciones de forma que el programador no tenga que descentrarse de su problema para describir la comunicación y sincronización.

- El atributo `GRID` se usa para declarar un array de datos que también contiene un dominio asociado (junto con sus fronteras). Desde el punto de vista sintáctico, una variable que se declara con el atributo `GRID` será considerada como un registro con dos campos. Uno de ellos, llamado `DOMAIN`, será el dominio asociado a ese `GRID`. El otro, será denotado como `DATA` y contendrá los valores correspondientes a los puntos de ese dominio, equivaliendo, por tanto, a un array dinámico[4] de Fortran 90. Como en el caso de un tipo `DOMAIN`, la dimensión estará comprendida entre 1 y 4. Por lo tanto, el ejemplo:

  ```
  REAL, GRID2D :: g
  ```

  declara una variable que contiene un dominio, `g%DOMAIN`, y un array de números reales, `g%DATA`, que será creado dinámicamente cuando un valor sea asignado al campo del dominio. Una variable de tipo `GRID` puede ser asignada a otra variable del mismo tipo si tienen el mismo tamaño de dominio o si la variable asignada no tiene definido un dominio todavía. En este caso, se ejecutarán los siguientes pasos automáticamente:

  1. Se copia el campo `DOMAIN` junto con la información relativa a las fronteras asociadas a ese dominio.

[4] Esto es una extensión del lenguaje puesto que un array dinámico no puede ser un campo de un registro en Fortran 90 estándar.

2. Se crea dinámicamente el campo `DATA` de la variable receptora con espacio suficiente para almacenar los datos correspondientes a ese dominio.

3. Por último, se copia el dato almacenado en el campo `DATA`.

Nótese que el tipo de la variable puede ser cualquiera, incluyendo tipos definidos por el usuario. De esta forma, se puede incrementar la reusabilidad del proceso coordinador. Como ejemplo, se puede pensar en un proceso coordinador que representa una geometría compleja de un dominio. Puede ser que se necesiten distintas aplicaciones donde los procesos físicos a resolver por los trabajadores sean muy distintos, necesitándose, en algunas aplicaciones, varias variables por punto. Basta con cambiar el tipo base del `GRID` en los procesos trabajadores sin que ello afecte a los dominios definidos en el coordinador.

- Los datos correspondientes a un proceso y que necesita otro son enviados por la siguiente instrucción:

```
PUT_BORDERS (g)
```

donde `g` es una variable con el atributo `GRID`. Esta instrucción causa que aquellos datos correspondientes a `g` que están dentro de la región establecida como frontera de salida en el dominio `g%DOMAIN` sean mandados al proceso que contiene un `GRID` con un dominio asociado que tiene la misma frontera definida como de entrada.

Esta es una operación asíncrona; por tanto, el proceso en ejecución no necesita esperar que la comunicación finalice, sino que continúa su ejecución.

- Con el propósito de recibir los datos necesarios para actualizar las fronteras de entrada asociadas al dominio correspondiente a una variable `g` con el atributo `GRID`, se introduce la instrucción:

```
GET_BORDERS (g)
```

El proceso que llama a esta instrucción suspenderá su ejecución hasta que se reciban los datos que necesitan todas las fronteras de entrada asociadas a `g%DOMAIN`. Una vez recibidos, los valores afectados en la definición de fronteras

son actualizados. Si se ha definido una función al lado derecho del operador `<-`, entonces se llamará a esta función para actualizar los datos.

Ambas instrucciones pueden ser agrupadas en una sola instrucción `UPDATE_BORDERS(g)`. Sin embargo, en algunas aplicaciones es mejor separar el envío de la recepción de forma que el tiempo empleado en la comunicación pueda ser usado para realizar cálculos (solapamiento de comunicación y cálculo). Esta separación es la clave de la eficiencia de algunos algoritmos.

- La comunicación necesaria para determinar si el criterio de convergencia de un método ha sido alcanzado, es llevada a cabo por la instrucción:

  ```
  CONVERGE (c, variable, NombreProceso)
  ```

  donde `c` es una variable de tipo `CONVERGENCE`, `variable` es una variable escalar de cualquier tipo y `NombreProceso` es el nombre de una subrutina. Esta instrucción produce una reducción (en el sentido de paralelismo de datos[5]) del valor escalar usado como segundo argumento. La cabecera de la subrutina será como sigue:

  ```
  subroutine NombreProceso (resultado, variablearray, tam)
  ```

  La instrucción `CONVERGE` manda los valores de `variable` al resto de los procesos que comparten `c` y recibe de ellos el valor que cada uno tiene para `variable`. Esos valores son asignados entonces a los componentes de `variablearray`. El número de procesos implicados en la convergencia (definidos para la variable con el tipo `CONVERGENCE`, en la cláusula `OF` del coordinador) es pasado al argumento `tam`. Entonces, se llama a la subrutina `NombreProceso` y el valor obtenido para `resultado` es asignado a `variable`[6].

[5] Una variable escalar en un programa SPMD tiene el mismo valor en cada procesador. Cuando esto no ocurre, puede ser llevada a cabo una reducción, de forma que una función (máximo, media) pueda ser aplicada a los diferentes valores de esa variable. El valor calculado es pasado a cada procesador de forma que cada uno tenga el mismo valor.

[6] La función es ejecutada de forma redundante por todos los procesadores. Esto se hace así para evitar la comunicación final del resultado.

2.2.3 Un ejemplo simple.

A continuación se muestra un ejemplo sencillo que sólo utiliza los aspectos del lenguaje comentados hasta ahora.

El `Programa 1` muestra el proceso coordinador para un problema regular que resuelve la ecuación de Laplace en dos dimensiones:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad \text{en } \Omega \qquad \textbf{Ecuación 1.}$$

donde *u* es una función real, Ω es el dominio, un subconjunto de $\mathbf{R}^2$ y se han impuesto condiciones de frontera de Dirichlet en $\partial\Omega$, la frontera de Ω, es decir:

$$u = g \quad \text{en} \quad \partial\Omega \qquad \textbf{Ecuación 2.}$$

Para su resolución se ha utilizado el método de diferencias finitas de Jacobi con 5 puntos (*Figura 3*). Aunque éste no es un buen método de resolución del problema, su sencillez permite describir el lenguaje sin tener que entrar en los detalles de un método más complejo.

El problema se resuelve con dos dominios como se puede ver en la *Figura 4*. El primer dominio, denominado `u`, se extiende desde la columna `1` a la columna `Nx-1`. La columna `Nx` no pertenece a este dominio, con lo cual, no debe ser calculada junto con el resto del dominio `u`, sino que se resuelve junto con `v`, el cual pasará la información

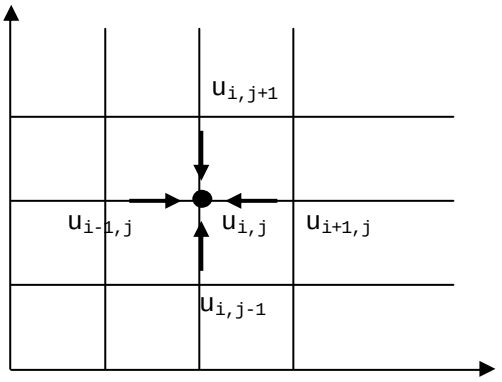


Figura 3. El método de Jacobi con 5 puntos. Cada valor se actualiza con el de sus vecinos.

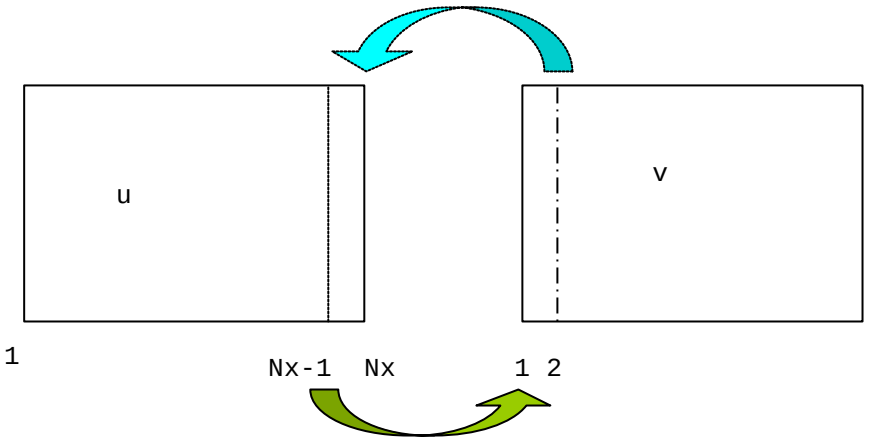


Figura 4. Representación de un dominio regular con dos subdominios.

calculada en la columna `2` a `u`. Simétricamente, los datos para la columna 1 no se calculan junto con el resto de `v` sino que se toman los calculados junto con `u` en la columna `Nx-1`. Nótese que en la figura, los valores a calcular se representan por las líneas (y no por el espacio entre ellas). Las columnas `Nx` de `u` y `1` de `v` son *puntos externos* (en alguna literatura se denominan *ghost points* o *shadow edges*) y se utilizan para almacenar datos que son necesarios para los cálculos locales, pero que pertenecen a otros procesos. Usando estos puntos, se disminuye la necesidad de comunicación, incrementándose la eficiencia de la aplicación[7].

La línea 2 del `Programa 1` se usa para declarar dos variables de tipo `DOMAIN2D`, las cuales representan dominios bidimensionales. Estas variables toman sus valores en las líneas 4 y 5. En este caso, toman los mismos valores, por lo que representan rectángulos que cubren la región que abarca desde el punto `(1,1)` hasta el `(Nx,Ny)`. Ambos, `Nx` y `Ny` (cuyas declaraciones han sido omitidas) pueden ser variables o constantes de tipo `INTEGER`. De hecho, las declaraciones 4 y 5 son simples asignaciones de cuatro valores enteros en tiempo de ejecución. No es necesario que los dominios sean geométricamente contiguos ni disjuntos, ya que las fronteras han sido definidas

[7] El uso de esta técnica tiene dos ventajas. Por un lado, los datos que se envían se almacenan localmente de modo que se evita una nueva comunicación si se tienen que usar varias veces. Por otro lado, el envío de los datos se hace de una sola vez en lugar de mandarlos uno a uno, disminuyéndose así el tiempo empleado en la comunicación.

explícitamente. También se pueden obtener fronteras implícitamente como se verá más adelante.

```
 1) program ejemplo1
 2) DOMAIN2D u , v
 3) CONVERGENCE c OF 2
 4) u = (/ 1,1, Nx,Ny /)
 5) v = (/ 1,1, Nx,Ny /)
 6) u (Nx,1, Nx,Ny) <- v ( 2,1, 2,Ny )
 7) v (1,1, 1,Ny ) <- u ( Nx – 1,1, Nx – 1,Ny )
 8) CREATE solve ( u , c )
 9) CREATE solve ( v , c )
10) end
```

Programa 1. El proceso coordinador para un dominio regular.

El enlace entre dominios se efectúa mediante la definición de sus fronteras usando el operador `<-`. Como se puede observar en el Programa 1, la definición de fronteras en la línea 6, causa que los datos de la columna `2` del dominio `v` actualicen la columna `Nx` de `u`. Análogamente, la declaración de la línea 7 causa que los valores de la columna `Nx – 1` de `u` actualicen la columna `1` de `v`. En el ejemplo, `u` tiene dos fronteras asociadas, una de ellas es una frontera de entrada (definida en la línea 6) y la otra es de salida (definida en la línea 7).

La línea 3 declara un tipo de variable `CONVERGENCE`, que es pasada como argumento al proceso trabajador creado por el coordinador. La cláusula `OF 2` indica el número de procesos que tomarán parte en el criterio de convergencia. Sin embargo, cuando el proceso trabajador declara el parámetro formal, la cláusula `OF` no se especifica, ya que este proceso no necesita saber cuántos procesos están implicados en la convergencia del método. De esta forma, se incrementa la reusabilidad de los trabajadores (los aspectos de coordinación se especifican en el proceso coordinador).

Las líneas 8 y 9 son las que crean los procesos trabajadores. Cada proceso recibe un tipo de variable `DOMAIN` y otro de tipo `CONVERGENCE`. Hay que destacar el hecho de que las variables del proceso coordinador `u` y `v` sean declaradas de tipo `DOMAIN`, en vez de `GRID`. Esto se hace así para evitar tener que comunicar una gran cantidad de información entre el proceso coordinador y cada uno de los trabajadores. De esta forma, solamente se necesita una pequeña cantidad de información para comunicar un `DOMAIN`

(cuatro números enteros, en el caso bidimensional, junto con las fronteras asociadas al dominio).

La línea 10 causa la finalización del proceso coordinador. El programa BCL terminará su ejecución cuando todos los procesos trabajadores finalicen.

El código del proceso trabajador se muestra en el Programa 2. La línea 1 es la cabecera de la subrutina llamada desde el coordinador. Las líneas 2 y 3 declaran los argumentos formales u y v, que son pasados desde el coordinador.

```
 1)subroutine solve (u, c)
 2)DOMAIN2D u
 3)CONVERGENCE c
 4)double precision, GRID2D:: g, g_old
 5)g%DOMAIN = u
 6)call initGrid (g)
 7)do i=1, niters
 8)    PUT_BORDERS (g)
 9)    GET_BORDERS (g)
10)    g_old = g
11)    call computeLocal (g, g_old)
12)    error = computeNorm (g, g_old)
13)    CONVERGE (c, error, maxim)
14)    Print *, “Max norm: ”, error
15)enddo
16)end subroutine solve
```

Programa 2. El proceso trabajador para el ejemplo de Jacobi.

El atributo GRID aparece en la línea 4. Cuando se le asigna un dominio a una variable con el atributo GRID, línea 5, se crea un array dinámico capaz de almacenar los datos para el dominio que está siendo asignado. El tipo de componente de este array es el especificado en la declaración de la variable (línea 4). En este caso, el tipo es double precision. La inicialización de g%DATA se efectúa en la subrutina llamada en la línea 6.

Las líneas 8 y 9 son las primeras que conllevan comunicación. La instrucción PUT_BORDERS en la línea 8 causa que los datos de g%DATA correspondientes a los valores de g%DOMAIN con una frontera de salida asociada (ver instrucciones 6 y 7 en el Programa 1) sean enviados a los procesos que contienen un GRID cuyo dominio tenga la misma frontera asociada como de entrada. En la línea 9, la declaración GET_BORDERS detiene la ejecución del proceso hasta que los datos necesarios para actualizar los valores con una

frontera de entrada asociada hayan llegado. Esos datos se almacenan entonces en las posiciones correspondientes de `g%DATA`.

La instrucción de la línea 10 produce la asignación de dos variables con el atributo `GRID`. Puesto que la primera vez que se ejecute esta instrucción `g_old` no tiene definido su dominio aún, se llevarán a cabo los tres pasos comentados anteriormente (es decir, copia del dominio, creación de la variable dinámica y copia de los valores de la variable). En las iteraciones sucesivas, sólo se realizará la copia de los valores del campo `g%DATA` a `g_old%DATA`.

La computación local se realiza mediante las llamadas a las subrutinas de las líneas 11 y 12 mientras que el método de convergencia usado se comprueba en la línea 13. La instrucción `CONVERGE` causa una comunicación entre los dos procesos que comparten la variable `c`, declarada como `CONVERGENCE`. El valor comunicado en este caso es el valor de la variable `error`. El valor máximo (calculado por la función `maxim`) obtenido por cada proceso es asignado a la variable `error` una vez que la ejecución de `CONVERGE` haya finalizado.

Una de las ventajas de BCL es la reusabilidad de los distintos procesos. Para ilustrarlo con un ejemplo, considérese ahora un problema irregular como el de la *Figura 5*. En él intervienen 3 dominios en lugar de 2, definiendo un dominio global irregular. Este problema puede ser resuelto utilizando el proceso coordinador mostrado en el Programa 3.

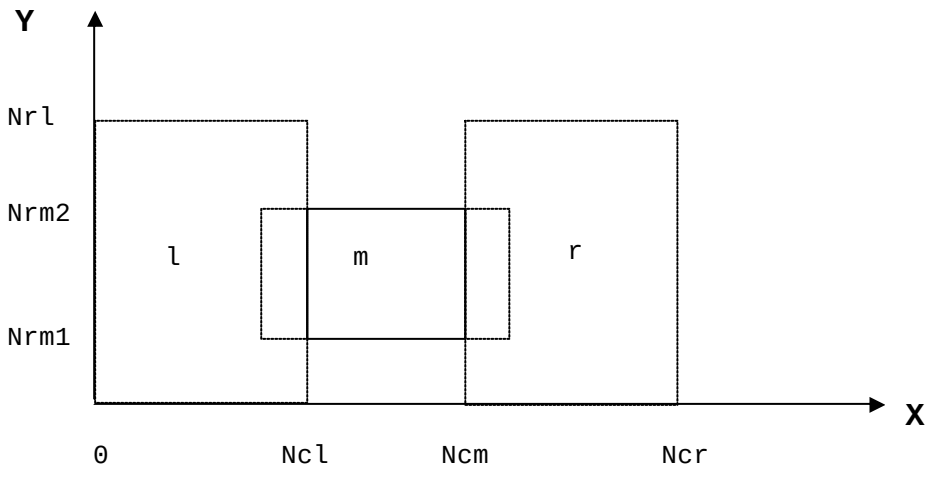


Figura 5. Un dominio irregular.

```
 1)  program example2
 2)  DOMAIN2D l, m, r
 3)  CONVERGENCE c OF 3
 4)  l = (/0,0, Ncl, Nrl /)
 5)  m = (/Ncl-1,Nrm1, Ncm+1, Nrm2 /)
 6)  r = (/Ncm,0, Ncr, Nrl /)
 7)  l(Ncl,Nrm1 ,Ncl,Nrm2) <- m(Ncl,Nrm1 ,Ncl,Nrm2)
 8)  m(Ncl-1,Nrm1 ,Ncl-1,Nrm2) <- l(Ncl-1,Nrm1 ,Ncl-1,Nrm2)
 9)  m(Ncm+1,Nrm1, Ncm+1,Nrm2) <- r(Ncm+1,Nrm1, Ncm+1,Nrm2)
10)  r(Ncm,Nrm1, Ncm,Nrm2) <- m(Ncm,Nrm1, Ncm,Nrm2)
11)  CREATE solve ( l, c )
12)  CREATE solve ( m, c )
13)  CREATE solve ( r, c )
14)  end
```

Programa 3. Proceso coordinador para un problema irregular.

En éste se definen las coordenadas de los tres dominios en que se descompone la aplicación y las fronteras entre ellos. Puesto que las condiciones que se imponen en las fronteras son las mismas que las impuestas en los límites de cada dominio, el proceso `solve` que se utilizó en el Programa 2, se puede reutilizar sin ninguna modificación.

En este ejemplo, al contrario que en el del Programa 1, la definición de los dominios se realiza teniendo en cuenta, no sólo sus tamaños, sino también su posición en el plano. Hacerlo de esta forma revela la posibilidad de obtener automáticamente cierta información de forma implícita, como son las fronteras entre dominios. Esta posibilidad, entre otras, se comenta en el apartado siguiente.

2.3 Aspectos Adicionales del Lenguaje.

Además de las características antes mencionadas, algunos otros aspectos se han añadido a BCL para incrementar la expresividad del lenguaje. De esta forma, se facilita tanto la codificación de la solución de este tipo de problemas como la comprensión y modificación de los programas ya escritos.

2.3.1 Tipos de datos para la manipulación de regiones.

Además de los tipos de datos vistos hasta ahora, también se incluyen otros tipos de datos para la manipulación de regiones del plano:

- `POINTdD`. Consiste en una tupla con `d` elementos, representando un punto en un sistema de coordenadas enteras de dimensión `d`. De esta forma se puede declarar, por ejemplo:

  ```
  POINT2D p
  ```

  donde `p` representa un punto en el espacio bidimensional. Para acceder a cada uno de los `d` números enteros que forman el punto se utiliza la notación de array, es decir, `p(i)` será la `i`-ésima coordenada de `p`. El sistema de coordenadas es entero, es decir, $p \in Z^d$, o dicho de otro modo, el tipo de `p(i)` es `integer`. La dimensionalidad se ha limitado a 4.

- `REGIONdD`. Representa una región rectangular, es decir, un subconjunto del espacio Z^d. Una región es identificada unívocamente mediante dos puntos o bien mediante $2 \cdot d$ números enteros. Es decir, para el caso de tres dimensiones, se puede definir la variable `R` de la siguiente forma:

  ```
  REGION3D R
  ```

  y asignarle luego un valor mediante dos puntos `R%p1` y `R%p2` o mediante una instrucción del tipo `R= (/1,1,1,3,5,4/)`. Hay que tener en cuenta que el primer punto se entiende que será el más cercano al origen de coordenadas mientras que el segundo será el más lejano (*Figura 6*). Para el caso bidimensional el primero será la esquina inferior izquierda y el segundo la superior derecha. De esta forma, cada coordenada del primer punto ha de ser inferior a la del segundo, es decir, si $\exists i,\ 1 \le i \le d \mid$ `R%p1(i) > R%p2(i)` la región representa un conjunto vacío.

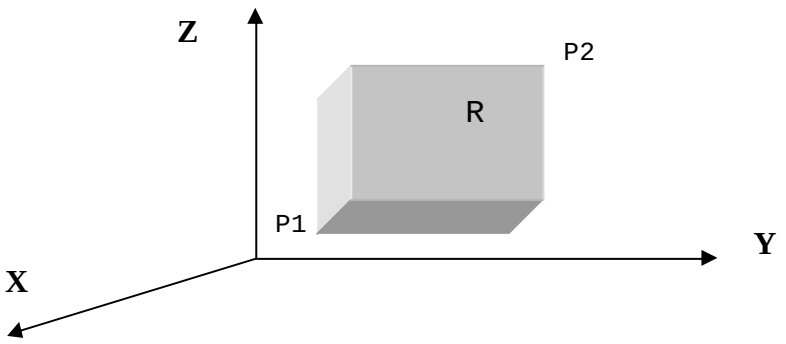


Figura 6. Un ejemplo de región tridimensional

Tanto las variables de tipo `REGION` como las de tipo `DOMAIN` podrán ser utilizadas para indexar una variable de tipo `GRID`. De esta forma, si `r` es una variable de tipo `REGION` con el valor `(/1,2,4,5/)` y `g` es una de tipo `GRID`, la instrucción

```
g( r )= 0.0
```

sería equivalente a `g%DATA(1:4, 2:5) = 0.0`.

Igualmente, las variables de tipo `REGION` se pueden utilizar para indexar variables de tipo `DOMAIN`. Esto es particularmente útil a la hora de definir las fronteras entre dominios. Así, por ejemplo, si `izq` y `der` son dos variables de tipo `REGION`, puede ser cómodo definir fronteras de la siguiente forma:

```
u ( izq ) <- v ( der)
```

2.3.2 Manipulación de regiones y dominios.

Para la manipulación de regiones y dominios se introducen varias subrutinas y funciones que se agrupan con nombres según su funcionalidad. Mediante la utilización de interfaces genéricas se puede llamar a distintos procedimientos con el mismo nombre. De esta forma, el compilador determina el procedimiento a llamar en función del número y tipo de los parámetros.

Entre estas se encuentran las de modificación de los valores de regiones para hacer crecer o disminuir el área de una región y para desplazarla (*Figura 7*).

- La función `GROW` toma dos argumentos, una variable de tipo `DOMAIN` o `REGION` y un número entero. Devuelve una variable que tiene un incremento en todas las direcciones en el valor indicado como segundo parámetro. Por ejemplo, cuando ejecutamos:

  ```
  u = ( / 2 , 2 , Nx , Ny / )
  v = GROW ( u , 1 )
  ```

  la variable `v` toma el valor `( 1 , 1 , Nx+1 , Ny+1 )`.

- La función `SHIFT` toma como argumento una región o dominio seguido de tantos números enteros como la dimensión del primer argumento. El tipo del valor devuelto es el del primer argumento y su valor es el resultado del desplazamiento de la región en los valores indicados. Así, si se ejecutan las instrucciones:

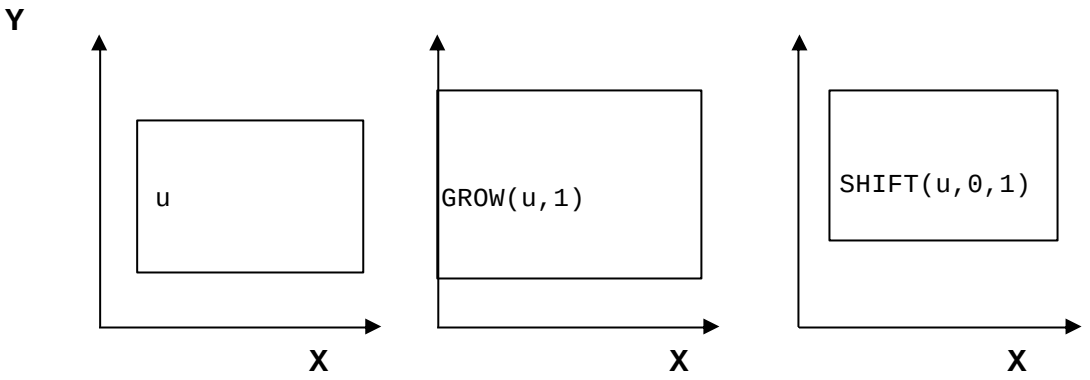


Figura 7. Resultado de las funciones `GROW` *y* `SHIFT`.

```
u = ( / 2 , 2 , Nx , Ny / )
v = SHIFT ( u, 0 , 1 )
```

`v` tomaría el valor `( 2 , 3 , Nx, Ny+1 )`.

- El procedimiento `INTERSECTION` toma dos argumentos que pueden ser de tipo `DOMAIN` o `REGION`.

  Si las variables son de tipo `REGION`, `INTERSECTION` actúa como una función que devuelve otra región con la intersección entre los dos parámetros:

  ```
  u = INTERSECTION(v,w)
  ```

  Si las variables son de tipo `DOMAIN`, se comporta como una instrucción en lugar de cómo una función:

  ```
  INTERSECTION(u,v)
  ```

  El procedimiento no hace nada si los dominios no comparten ninguna región. Si hay una región en común, las fronteras son asociadas automáticamente a los dominios, de tal manera que los puntos exteriores de la región en común serán los datos a actualizar. De esta forma, las instrucciones de 7 a 10 del Programa 3 se podrían sustituir por:

  ```
  INTERSECTION (u, v)
  INTERSECTION (v, w)
  ```

2.3.3 Mejoras en la definición de fronteras.

- Cuando se definen los dominios de una aplicación utilizando no sólo sus tamaños sino también sus coordenadas en el plano, como en el caso del Programa 3, la definición de las fronteras conlleva generalmente que la región afectada sea la misma a la izquierda y a la derecha del operador. Para evitar la duplicidad y una posible incoherencia (causada, por ejemplo, por un error tipográfico) se puede usar la expresión "_" para sustituir la región de los dominios de la parte derecha del operador. Así, la expresión:

  ```
  u(Ncu,Nrv1 ,Ncu,Nrv2) <- w(Ncu,Nrv1 ,Ncu,Nrv2)
  ```

  puede ser sustituida por:

  ```
  u(Ncu,Nrv1 ,Ncu,Nrv2) <- w(_)
  ```

- Para resolver algunos tipos de problemas es mejor usar varios tipos de fronteras que son comunicadas en diferentes fases del algoritmo. De esta forma, una definición de frontera puede ser etiquetada con un número que indica la clase de conexión, pudiéndose así, distinguir entre fronteras (o agruparlas usando el mismo número). Esto se realiza escribiendo la etiqueta justo detrás del conector de frontera y entre paréntesis, por ejemplo `<-(1)`. Así, cuando en un proceso trabajador se utilice `PUT_BORDERS` y `GET_BORDERS`, se puede utilizar opcionalmente un segundo argumento, un número entero que representa el tipo de frontera que se desea enviar o recibir. La instrucción sólo afecta entonces a aquellas fronteras etiquetadas con ese número. Las fronteras que no tienen etiqueta se suponen que son de tipo 0. De esta forma, si hay fronteras etiquetadas y `PUT_BORDERS` (o `GET_BORDERS`) no reciben un segundo parámetro, las fronteras son enviadas (o recibidas) en el orden marcado por el número de etiqueta, teniendo en cuenta que las fronteras sin etiqueta son las primeras en ser comunicadas.

  El uso de etiquetas es especialmente útil en la definición de problemas con un patrón de comunicación más complejo que los vistos hasta ahora. Para poner un ejemplo, hemos considerado uno de los problemas que pone en libre disposición el programa NAS (*Numerical Aerodynamic Simulation*) del centro de investigación NASA Ames (California) para la evaluación de ordenadores

paralelos (*NAS Parallel Benchmark* [Bailey y otros 94]). El problema denominado NPB-MG resuelve la ecuación de Poisson's en 3D usando un *multigrid* de ciclo en V. Este algoritmo se puede paralelizar usando el método denominado *red-black ordering* (véase [Freeman y Phillips 92]). En el caso tridimensional, cada punto necesita valores de sus 27 vecinos, es decir, se utilizan los valores de las esquinas. La *Figura 8* muestra un esquema simplificado para dos dimensiones del patrón de comunicación. Se han considerado 4 dominios que en dos pasos pueden intercambiar los valores necesarios para el algoritmo *red-black*.

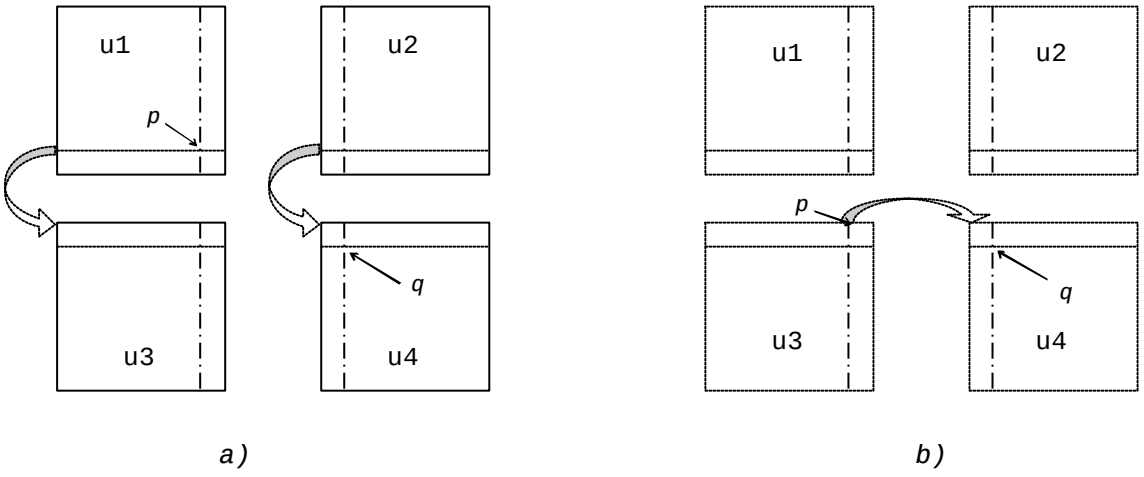


Figura 8. Comunicación de datos en dos etapas. a) Primera etapa con comunicación vertical. b) Segunda etapa donde se realiza comunicación horizontal. Sólo se muestran aquellas que afectan al cálculo del punto q. El resto de comunicaciones son simétricas a las mostradas.

En la figura se puede observar que el punto denominado *q* necesita el valor calculado para *p*. Esto se puede hacer utilizando la etiqueta 1 para las fronteras verticales y la etiqueta 2 para las horizontales:

```
u1 (1,1, 1,N) <-(1) u3 (1,N-1,N,N-1)
u3 (N,1, N,N) <-(1) u1 (1,2,N,2)
u2 (1,1, 1,N) <-(1) u4 (1,N-1,N,N-1)
u4 (N,1, N,N) <-(1) u2 (1,2,N,2)

u2 (1,1,1,N) <-(2) u1 (N-1,1,N-1,N)
u1 (N,1,N,N) <-(2) u2 (2,1,2,N)
u4 (1,1,1,N) <-(2) u3 (N-1,1,N-1,N)
u3 (N,1,N,N) <-(2) u4 (2,1,2,N)
```

De esta forma los procesos trabajadores sólo tendrán que hacer la llamada a las rutinas de comunicación de fronteras:

```
PUT_BORDERS(g, 1)
GET_BORDERS(g, 1)
PUT_BORDERS(g, 2)
GET_BORDERS(g, 2)
```

2.3.4 Creación automática de dominios y fronteras.

- El procedimiento `DECOMPOSE` toma una variable de tipo `DOMAIN`, un array de dominios y tantos números enteros como la dimensión del dominio. Los valores de los dominios del array se calculan como el resultado de la división del dominio en subdominios (tantos como los indicados para cada dimensión). También hay un argumento opcional que indica el solapamiento deseado entre los subdominios resultantes. Esta clase de descomposición se realiza para dominios regulares cuando el principal objetivo es obtener ventaja del paralelismo, ya que cada subdominio es resuelto en un procesador diferente.

  Así, por ejemplo, si se tienen las siguientes declaraciones de un dominio `u` y de un array de dominios `v`:

  ```
  DOMAIN2D u, v(3,3)
  ```

  se podría utilizar este procedimiento de la siguiente forma:

  ```
  call DECOMPOSE2D (u, v, 3, 3, OVERLAP = 1)
  ```

  lo cual daría lugar a una descomposición como la de la *Figura 9*. De esta forma, no sólo se asignan los valores a los elementos de `v` sino que también se crean los *puntos externos* que contendrán los datos que cada dominio necesita de sus vecinos. Además, se definen automáticamente las fronteras de modo que las llamadas a `PUT_BORDERS`, `GET_BORDERS` produzcan la comunicación necesaria. En la figura, las flechas muestran sólo algunas de estas fronteras. La definición de éstas se realiza teniendo en cuanta una actualización de 5 puntos (*Figura 3*) para el caso bidimensional.

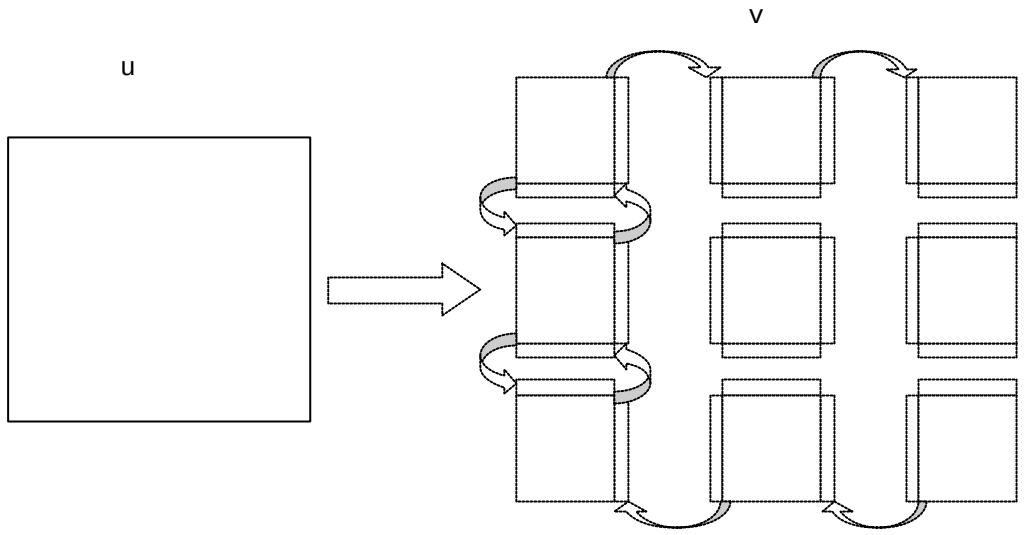


Figura 9. Uso del procedimiento `DECOMPOSE` *para la definición automática de dominios y fronteras. Las líneas punteadas representan los puntos externos incluidos automáticamente. Las flechas representan algunas de las fronteras definidas.*

2.3.5 Funciones de frontera.

La parte de la izquierda de un operador de conexión `<-` puede ser una expresión en la cual estén involucrados varios dominios. Así, por ejemplo, una frontera se puede definir de la forma:

```
u(1,1, 1,N) <- alfa * u(_) + (1-alfa) * v(_)
```

Este tipo de expresiones son bastante frecuentes cuando se desea actualizar una frontera con relajación (dependiente del parámetro `alfa`) de los valores calculados en el otro dominio. También se pueden especificar funciones definidas por el usuario que serán llamadas justo después de que los datos necesarios para actualizar una frontera se hayan recibido y antes de actualizar el `GRID` correspondiente. De esta forma, se pueden describir las ecuaciones necesarias en los problemas de descomposición de dominios en donde se necesiten cambiar las ecuaciones que se imponen en las fronteras (con respecto a las que se imponen en los límites del dominio).

2.3.6 Reutilización de rutinas en Fortran.

- Para incrementar las posibilidades de reutilizar código escrito previamente, se ha introducido una “macro” (en el sentido de C) llamada `ARGUMENTS`. Cuando la llamada a una subrutina usa una variable declarada de tipo `DOMAIN` como argumento, esta macro puede ser usada para expandir los valores que forman el

dominio. El resultado serán los números enteros de los puntos cartesianos que delimitan el dominio separados por comas. Estos valores pueden ser declarados como argumentos enteros en una subrutina de Fortran.

- Algunos de los algoritmos usados en la resolución de PDE usan ecuaciones sobre las fronteras entre subdominios que son diferentes de las empleadas sobre los límites del dominio global de la aplicación. En estos casos, si se desea reutilizar el código de un trabajador para implementar otros trabajadores, es necesario definir varias primitivas que extraen información relacionada con las fronteras asociadas a una variable dominio. Así, se han definido funciones que devuelven el número de fronteras de entrada o salida de un dominio, la región que ocupa cada frontera en cada trabajador, el tipo de frontera, etc. Sin embargo, para ilustrar claramente el modelo, los ejemplos mostrados en la próxima sección imponen la misma condición sobre las fronteras entre subdominios y sobre la frontera del dominio global.

2.4 Ejemplo de Programación.

A continuación se muestra un ejemplo que ilustra algunos de los aspectos comentados en el apartado anterior. Se trata del mismo problema de la ecuación de Laplace, pero que en este caso se resuelve sobre un dominio irregular en forma de T y que ha sido descompuesto en 3 dominios regulares denominados `l`,`r`, y `m` (*Figura 10*).

2.4.1 Proceso Coordinador.

El proceso coordinador es el mostrado en el Programa 4. En éste se definen los dominios de la aplicación (línea 2) y se asignan los valores cartesianos a sus correspondientes regiones (líneas 4 a 6). En este caso, la definición de los dominios se realiza teniendo en cuenta no sólo sus tamaños, sino también su distribución en el plano. En esta definición existe superposición de dominios, no sólo para contener los *puntos externos* mencionados anteriormente sino que también existen regiones que pertenecen a más de un dominio.

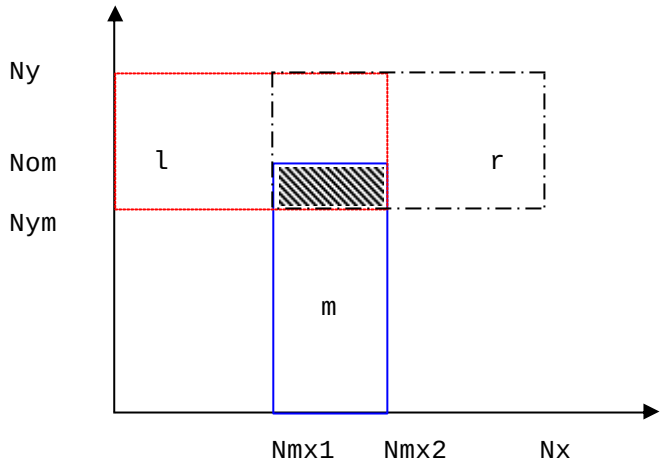


Figura 10. Un dominio irregular en forma de T.

Este hecho implica que la definición de las fronteras se tenga que hacer de manera que algunas de ellas se actualicen como la relajación de los valores calculados por cada uno de los procesos. Así, por ejemplo, la zona sombreada en la *Figura 10* debe ser actualizada mediante la media de los valores calculados para `l`, `m` y `r`. La línea 10 indica que la correspondiente llamada a `GET_BORDERS` por parte del proceso que calcule el dominio `l`, espere a que lleguen los valores de `r` y `m` y, a continuación, calcule la relajación de estos valores junto con los que posea para el dominio `l`.

```
 1)  program forma_de_t
 2)  DOMAIN2D r, l ,m
 3)  CONVERGENCE c OF 3
 4)  l = (/0,Nym, Nmx2, Ny /)
 5)  r = (/Nmx1,Nym, Nx, Ny /)
 6)  m = (/Nmx1,0, Nmx2, Nom /)
 7)  l(Nmx2, Nym + 1, Nmx2, Ny –1) <- r(_)
 8)  l(Nmx1+1,Nym,Nmx2-1,Nym) <- m(_)
 9)  l(Nmx1+1,Nom,Nxm2-1,Ny-1) <- (l(_) + r(_)) / 2.0
10)  l(Nmx1+1,Nym+1,Nmx2-1,Nom-1) <- (l(_) + r(_) + m(_) ) / 3.0
11)  r(Nmx1, Nym + 1, Nmx1, Ny –1) <- l(_)
12)  r(Nmx1+1,Nym,Nmx2-1,Nym) <- m(_)
13)  r(Nmx1+1,Nom,Nxm2-1,Ny-1) <- (l(_) + r(_)) / 2.0
14)  r(Nmx1+1,Nym+1,Nmx2-1,Nom-1) <- (l(_) + r(_) + m(_) ) / 3.0
15)  m(Nmx1, Nym + 1, Nmx1, Nom) <- l(_)
16)  m(Nmx2, Nym + 1, Nmx2, Nom) <- r(_)
17)  m(Nmx1+1,Nom,Nxm2-1,Nom) <- (l(_) + r(_)) / 2.0
18)  m(Nmx1+1,Nym+1,Nmx2-1,Nom-1) <- (l(_) + r(_) + m(_) ) / 3.0
19)  CREATE solve ( l, c )
20)  CREATE solve ( r, c )
21)  CREATE solve ( m, c )
22)  end
```

Programa 4. Proceso coordinador para el dominio irregular en forma de T.

Esta información es almacenada por las variables declaradas de tipo dominio. De esta forma, cuando los procesos trabajadores llaman a la instrucción `GET_BORDERS`, se reciben los datos de cada proceso y se realiza la función antes de actualizar los valores del `GRID` correspondiente.

En la definición de las fronteras (líneas 7 a 18) hay que tener en cuenta que se están imponiendo condiciones de Dirichlet, por lo que los puntos exteriores de un dominio no son resueltos con el resto del dominio en el método numérico. Esto hace que en las fronteras, los puntos más exteriores de un dominio se tengan que actualizar con los interiores de los dominios que los contengan.

2.4.2 Procesos Trabajadores.

Puesto que se han impuesto las mismas condiciones tanto en los límites del dominio como en las fronteras entre estos, al igual que en el caso del Programa 1, el proceso trabajador es el mismo que el del problema regular (mostrado en el Programa 2), lo cual nos da idea de las posibilidades de reutilización que ofrece el modelo.

A continuación, se muestran las rutinas llamadas en el programa `solve`, y que no fueron comentadas anteriormente para hacer hincapié en los aspectos principales del lenguaje. Estas rutinas son: `initGrid`, `computeLocal`, `computeNorm` y `maxim`.

Su codificación se podría haber hecho usando Fortran 90 estándar sin la sintaxis adicional de BCL de forma que estas rutinas podrían ser fácilmente reutilizadas si hubieran sido escritas previamente. Hay que hacer constar que éste es uno de los objetivos de BCL, la reutilización de código Fortran preexistente, uno de los aspectos principales de cualquier ampliación de Fortran que pretenda ser aceptada por los usuarios de este lenguaje.

Sin embargo, aunque esta reutilización es posible en nuestra propuesta, puede ser útil emplear el concepto de dominio para aumentar la claridad de estas subrutinas. Esto tiene la contrapartida de tener que modificarlas, incluyendo dentro del código los tipos introducidos por BCL: `GRID`, `DOMAIN`, etc. Ofrecer ambas posibilidades puede ser la mejor solución.

A continuación se muestra el código de estas subrutinas comentando los aspectos más relevantes de las mismas.

La primera subrutina llamada por el cuerpo principal del proceso coordinador es la subrutina `initGrid` (Programa 5). Como puede observarse en la línea 4, se puede asignar un valor al campo `DATA` del `GRID g` usando una única instrucción siguiendo la sintaxis de Fortran 90 para la manipulación de arrays.

```
1) subroutine initGrid ( g )
2) double precision, GRID2D :: g
3) REGION2D interior
4) g = 1.0
5) interior = GROW (g%DOMAIN, -1)
6) g (interior) = 0.0
7) end subroutine initGrid
```

Programa 5. Subrutina de inicialización del `GRID`.

La variable `interior` (de tipo `REGION`) obtendrá en la línea 5 la región del dominio de `g` pero reducida en 1. Aunque `g%DOMAIN` es de tipo `DOMAIN` y no `REGION`, se permite este tipo de instrucciones, en el cual a una región se le asigna un dominio (la región de ese dominio) o al revés, es decir, a un dominio se le asigna una región.

La instrucción 6 contiene un array (el campo `DATA` del `GRID`) que es indexado con una región, de forma que sólo los valores de esa zona recibirán la expresión indicada (valor real 0.0 en este caso). De esta forma, en cooperación con la línea 4 se asigna el valor 1.0 a los límites del dominio y 0.0 a la parte interior.

La línea 3 del Programa 6 utiliza la macro `ARGUMENTS`. Como se comentó anteriormente, esta macro se usa cuando se llama a una subrutina escrita en Fortran que no se desea modificar con sentencias BCL. De esta forma, esta rutina actúa de interfaz entre código en el cual se utilizan sentencias con los tipos introducidos por el lenguaje y código Fortran estándar como el del Programa 7.

```
1)  subroutine computeLocal (g, g_old)
2)  double precision, GRID2D :: g, g_old
3)  call j5relax (g%DATA, g_old%DATA, ARGUMENTS (g%DOMAIN))
4)  end subroutine computeLocal
```

Programa 6. Computación local.

La subrutina mostrada en el Programa 7 (en este caso muy simple) es la encargada de la resolución de la ecuación, y por lo tanto es la única con cierto peso computacional. Está completamente escrita en Fortran 90, así que puede ser reutilizada por otras aplicaciones.

La línea 1 es la cabecera de la subrutina. En ella se declaran los parámetros formales correspondientes a los argumentos de la línea 3 del Programa 6. Las variables a y a_old corresponden a g%DATA y g_old%DATA respectivamente y son declaradas de tipo double precision porque éste es el tipo base de g. Los cuatro últimos argumentos corresponden a aquellos expandidos por la macro ARGUMENTS.

```
 1)  subroutine j5relax (a, a_old, ul0, uh0, ul1, uh1)
 2)  integer ul0, uh0, ul1, uh1
 3)  double precision,dimension (ul0:uh0, ul1:uh1) :: a, a_old
 4)  integer i,j
 5)  do j = ul1 + 1, uh1 – 1
 6)      do i = ul0 + 1, uh0 – 1
 7)         a(i,j) = 1.0 / 4.0 * (a_old (i – 1, j) *   &
 8)            a_old (i + 1, j) + a_old (i, j – 1) + a _old(i, j + 1))
 9)      enddo
10)  enddo
11)  end subroutine j5relax
```

Programa 7. El método de Jacobi en Fortran 90 estándar.

La subrutina computeNorm (Programa 8) también usa la indexación del campo DATA de un GRID mediante una REGION. Calcula la norma de la diferencia entre el valor anterior y el actual. Este es el error local que ha de ser comunicado al resto de los procesos.

```
1)  double precision function computeNorm (g , g_old)
2)  double precision GRID2D :: g , g_old
3)  double precision r
4)  REGION2D interior
5)  interior = GROW (g%DOMAIN, - 1)
6)  r = MAXVAL (ABS(g (interior) – g_old (interior)))
7)  computeNorm = r
8)  end function computeNorm
```

Programa 8. Cálculo de la norma de la diferencia entre los valores anteriores y actuales.

Por último, el Programa 9 muestra la función pasada a la instrucción `CONVERGE` en la línea 13 del Programa 2. El argumento `length` recibe el número de procesadores definido en la cláusula `OF` de la declaración de la variable de tipo `CONVERGENCE` del programa coordinador (2 para el Programa 1 y 3 para los ejemplos mostrados en Programa 3 y Programa 4). En este caso se está calculando el máximo de los errores locales, con lo cual, por medio de la instrucción `CONVERGE`, se realiza una reducción (en el sentido del paralelismo) de los errores calculados de forma local por cada proceso trabajador. Esta reducción, con las comunicaciones que conlleva, se realiza de forma implícita.

```
1) subroutine maxim (result, error, length)
2) integer lenght, i
3) double precision result, error (length)
4) result = error (1)
5) do i = 2, lenght
6)      if (result < error ( i ) )  result = error ( i )
7) enddo
8) end subroutine maxim
```

Programa 9. Cálculo del máximo error entre procesos.

2.5 Conclusiones.

En este capítulo se ha descrito el lenguaje de coordinación BCL a excepción de los mecanismos que este lenguaje ofrece para la integración del paralelismo de datos y tareas que serán tratados en el capítulo siguiente.

Dada la complejidad de algunas aplicaciones de descomposición de dominios, y de los distintos aspectos que hay que tener en cuenta cuando se programa la solución de este tipo de problemas, se ofrece un modelo de paralelismo sencillo que permita la definición por separado de las distintas partes (o dominios) que se han de resolver. BCL permite, por un lado, programar de forma independiente los distintos problemas, de manera que el programador pueda abstraerse de los aspectos de coordinación y comunicación. Estos aspectos se declaran en el proceso coordinador que actúa como el "pegamento" que une las distintas partes del problema.

Por otro lado, mediante una serie de aspectos adicionales, se pueden reutilizar los procesos trabajadores para resolver problemas irregulares como se ha visto con un ejemplo. Los procesos trabajadores se pueden programar directamente en Fortran, lo que permite la reutilización de la gran cantidad de software disponible en este lenguaje, o bien, utilizar algunos aspectos de BCL para mejorar la expresividad de la solución. La parte de coordinación en la cual se describe la geometría del problema también puede ser reutilizada para distintos problemas que se quieran resolver en un dominio de geometría compleja, puesto que el domino es independiente del tipo de datos y del resto de la computación. Hay que tener en cuenta que desde el punto de vista de un científico, el aspecto más importante de la reutilización del código no es el tiempo ahorrado sino la fiabilidad ganada al usar un código en el que se confía [Dubois 99].

El hecho de que la sintaxis de BCL esté basada en Fortran permite que un programador de aplicaciones científicas no tenga que aprender otro lenguaje completo (sino sólo las extensiones de BCL) para la definición de tanto la parte de coordinación como la de cómputo y obtener así, un programa paralelo. Además, mediante el uso del lenguaje estándar para el paralelismo de datos, HPF, se pueden integrar de forma elegante ambas formas de paralelismo, como se describe en el capítulo 3.

Capítulo 3. Integración del paralelismo de datos y tareas usando BCL

HPF proporciona un modelo simple de programación de alto nivel para la implementación de aplicaciones paralelas de datos regulares en arquitecturas de memoria compartida y distribuida. Para tales aplicaciones, el uso de este lenguaje hace más fácil el desarrollo y mantenimiento de los programas, proporcionando, además, un alto rendimiento. Sin embargo, no es indicado para aplicaciones paralelas de datos irregulares estructurados en bloques como los expuestos aquí. En estos casos es preferible la integración del paralelismo de datos y tareas.

Como se comentó en la introducción, existen diversos factores que hacen difícil esta integración. Aunque existen diferentes aproximaciones, con problemáticas distintas, piénsese, por ejemplo, en los modelos que permiten la ejecución concurrente de tareas con paralelismo de datos. El problema que surge en estos casos es el de la comunicación de una variable entre tareas. Puesto que ésta puede estar distribuida de forma distinta en la tarea receptora y la emisora, cada procesador ha de determinar qué porción ha de enviar a qué procesador de la otra tarea.

En la *Figura 11* se representa este problema con un ejemplo. La primera tarea posee 4 procesadores denominados `P1` a `P4` y la variable está distribuida por filas, es decir, cada fila no pertenece a un solo procesador, sino que está distribuida entre ellos, mientras que cada columna reside en el mismo procesador. Esto se indica en HPF mediante la distribución `(*,BLOCK)`. La segunda tarea consta de dos procesadores `P5` y `P6` y recibe la variable con distribución `(BLOCK,*)`. Esto conlleva que cada procesador de la primera tarea tenga que enviar la parte que él posee de la variable a los procesadores en la segunda tarea. Igualmente, cada procesador de la segunda tarea tiene que recibir su parte de variable de distintos procesadores de la primera tarea. El

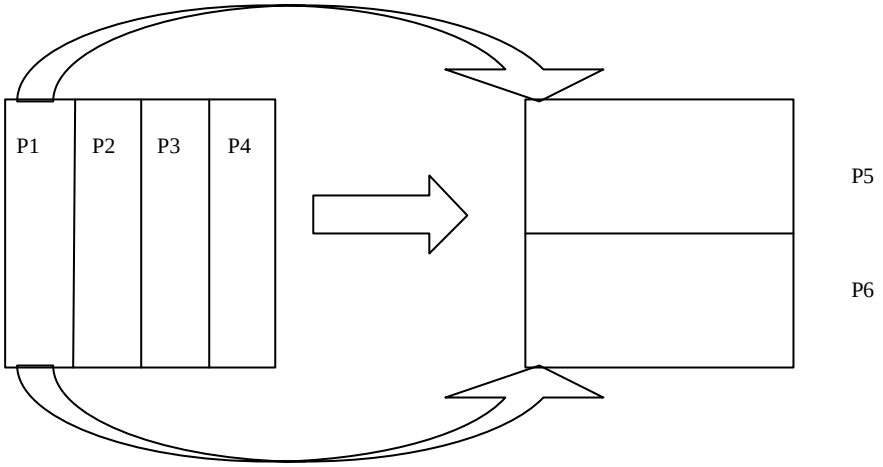


Figura 11. Comunicación de una variable con distintas distribuciones. Las flechas indican que el procesador `P1` *ha de mandar su porción de variable a cada uno de los procesadores que contienen la variable recibida.*

problema es que cada procesador de la primera tarea no sabe nada sobre la distribución del dato en la segunda tarea y viceversa

Para solucionar esto, algunos modelos (como HPF 2.0) simplemente no permiten la comunicación de datos entre tareas. Esto limita bastante el problema, con lo cual se tienen que buscar soluciones artificiales y poco claras a determinadas aplicaciones en las que ese patrón de comunicación sería el más natural.

Otros modelos, como HPF/MPI o $COLT_{HPF}$, permiten este tipo de comunicación sacrificando la eficiencia. Para ello se sigue el algoritmo de intercambio de información sobre la distribución de los datos, denominado *pitfalls* [Ramaswamy y Banerjee 95] y cuyos pasos se muestran en la *Figura 12*.

Aunque está más allá del propósito de este trabajo una explicación detallada de este algoritmo, sí merece la pena ver los problemas que plantea.

1. En el primer paso, es necesario llamar a rutinas intrínsecas de HPF para averiguar la distribución de la variable a mandar/recibir.

2. A continuación, se hacen llamadas extrínsecas, en las cuales el programa se comporta de modo local, es decir, cada procesador se comporta como si tuviera

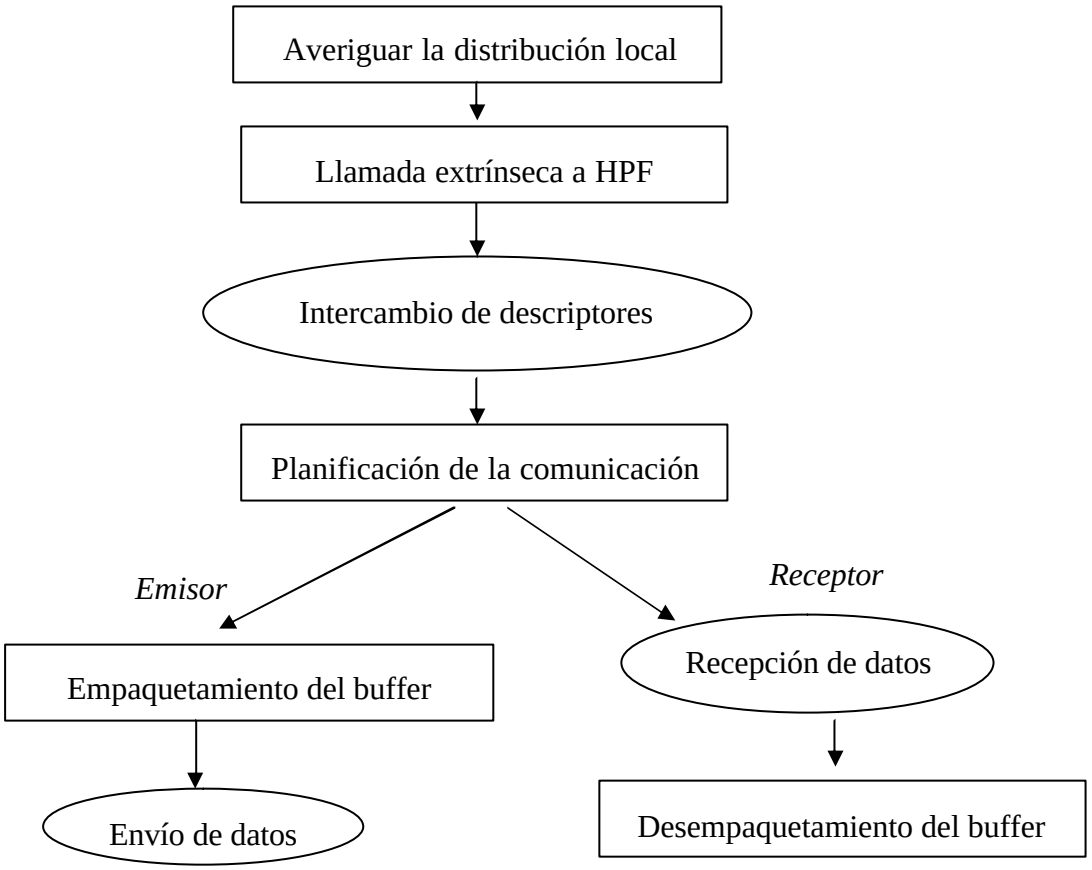


Figura 12. Pasos a ejecutar en la comunicación de datos distribuidos dinámicamente entre tareas. Los círculos indican los pasos que conllevan comunicación.

una hebra de control distinta, con lo que se pueden realizar operaciones distintas en cada procesador.

3. Los procesadores que van a enviar el dato intercambian información con los que lo tienen que recibir.

4. La información obtenida en el paso anterior se utiliza para planificar las subsecciones de cada array que tienen que ser enviadas a cada procesador.

5. Los procesadores que van a enviar los datos empaquetan la información mientras que los que la van a recibir, suspenden la ejecución esperando la recepción.

6. Finalmente se envían los datos. Una vez recibidos, se han de desempaquetar.

La comunicación requiere pues, una serie de pasos y de comunicaciones adicionales que sobrecargan mucho el proceso y, además, el sistema en tiempo de ejecución del compilador de HPF utilizado ha de ser modificado.

3.1 Integración con BCL.

Una de las posibles soluciones a la integración del paralelismo de datos y tareas es usar un lenguaje diferente de HPF tanto para definir los distintos bloques o subdominios de la aplicación como para controlar las diferentes tareas que resuelven cada subdominio. De esta forma, HPF se puede emplear para la paralelización interior de cada subdominio, con lo cual, se puede sacar partido a dos niveles de paralelismo: entre dominios y dentro de ellos.

Hacer una extensión de HPF para este propósito implica permitir la ejecución de varias subrutinas de HPF en paralelo (varias tareas paralelas que son ejecutadas en paralelo). Esto no está permitido en la mayoría de compiladores HPF (ya que no permiten la ejecución de subrutinas en subconjuntos de procesadores[8]).

En nuestro modelo, el coordinador es un programa secuencial que expande diferentes tareas HPF, definiendo la distribución (en el sentido de HPF) de cada subdominio dentro de cada tarea y la comunicación entre estas tareas. El hecho de que la distribución de los datos en cada tarea sea conocida por el coordinador, permite que éste transmita esta información a las tareas trabajadoras. De esta forma, cada procesador de éstas, puede conocer, no sólo la distribución de sus datos, sino también la distribución de los datos en aquellas tareas con las cuales tenga que comunicarse para enviar o recibir fronteras. Esta es la clave de la eficiencia del modelo.

Puesto que BCL proporciona un modelo de paralelismo de tareas, se han introducido tres nuevas características con el propósito de incorporar el paralelismo de datos. Todas ellas son usadas por el programa coordinador.

---

8 Aunque esto es una de las “extensiones aprobadas” de HPF 2.0.

- La declaración de los procesadores del sistema se hace de forma similar a como se hace en HPF. Así la declaración:

  `PROCESSORS p ( 4 , 4 )`

  indica una matriz cuadrada de 16 procesadores.

  `PROCESSORS p (N)`

  indica `N` procesadores conectados linealmente. `N` debe ser una expresión evaluable en tiempo de compilación. Al contrario que en HPF, cuando dos o más variables del tipo `PROCESSORS` son declaradas en el mismo programa, se entiende que se refieren a diferentes subconjuntos de procesadores.

- La instrucción `DISTRIBUTE` se aplica a variables de tipo `DOMAIN`. Esta instrucción no lleva a cabo la distribución por sí misma sino que indica al sistema la futura distribución del `GRID` que tiene asociado el dominio especificado. Las distribuciones posibles se corresponden a las de HPF estándar.

- La cláusula `ON` se ha añadido a la instrucción `CREATE` para indicar los procesadores HPF (definidos previamente con el tipo `PROCESSORS`) que ejecutarán las tareas indicadas.

3.2 El Ejemplo de Jacobi con Paralelismo de Datos y Tareas.

La *Figura 13* muestra un ejemplo de un dominio que ha sido descompuesto en dos subdominios, cada uno resuelto por una tarea HPF, la primera con 16 procesadores

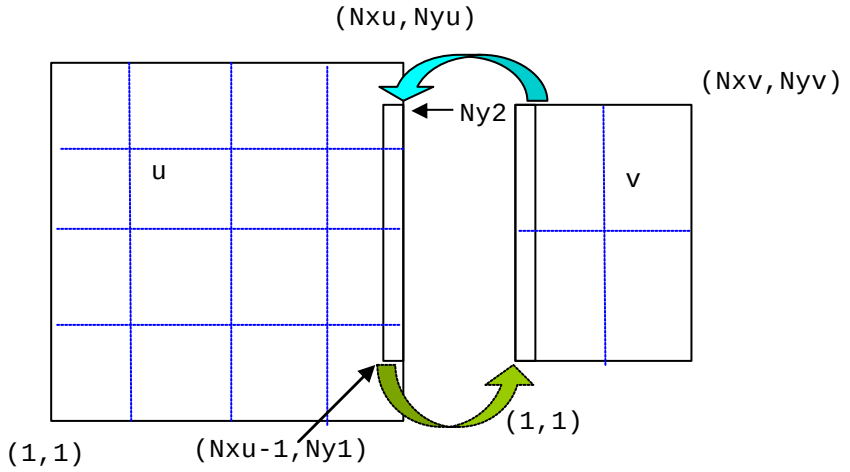


Figura 13. Comunicación de fronteras entre dos dominios con paralelismo de datos.

y la segunda con 4. Las líneas de puntos representan la distribución HPF. Nótese que la frontera entre ambos subdominios está distribuida entre varios procesadores HPF.

El Programa 10 es el código del proceso coordinador para este ejemplo. En éste, el número de tareas es 2 (línea 3) porque el número de subdominios es 2, aunque el número de procesadores HPF sea 20. Se definen dos subconjuntos de procesadores en la línea 4: `p1` y `p2`. El `GRID` asociado con el dominio `u` será distribuido en bloques de `4 x 4` (línea 5), por tanto, será distribuido por filas y por columnas (`BLOCK,BLOCK`). El `GRID` con el dominio `v` asociado también será distribuido por filas y columnas.

```
 1) program example
 2) DOMAIN2D u, v
 3) CONVERGENCE c OF 2
 4) PROCESSORS p1( 4,4 ) , p2 ( 2,2 )
 5) DISTRIBUTE u (BLOCK , BLOCK) ONTO p1
 6) DISTRIBUTE v (BLOCK , BLOCK) ONTO p2
 7) u = (/1 , 1 , Nxu, Nyu /)
 8) v = (/1 , 1 , Nxv, Nyv /)
 9) u (Nxu,Ny1, Nxu,Ny2 ) <- v (2,1, 2,Nyv)
10) v (1,1, 1,Nyv ) <- u (Nxu-1,Ny1, Nxu-1, Ny2)
11) CREATE solve ( u , c ) ON p1
12) CREATE solve ( v , c ) ON p2
13) end
```

Programa 10. El programa coordinador con integración de paralelismo de datos y tareas.

La información relativa a la distribución es usada por el sistema para optimizar las comunicaciones entre tareas HPF cuando se llaman a las instrucciones `PUT_BORDERS` y `GET_BORDERS`, de modo que cada procesador sabe qué parte de su `GRID` ha de enviar/ recibir de cada procesador de la tarea con la que se tiene que comunicar.

En las líneas 11 y 12 se llama al proceso `solve` (Programa 11). En este caso, puesto que la distribución es la misma, el proceso trabajador puede ser el mismo para ambas tareas (incluso teniendo distinto número de procesadores). La única modificación con respecto al Programa 2 es la distribución de los datos que ha de ser declarada en la línea 5. Al resto de las rutinas sólo tendrán que hacerse las modificaciones típicas de todo programa en Fortran que se quiere pasar a HPF (distribución de los datos, paralelización de los bucles, etc.).

```
 1)subroutine solve (u, c)
 2)DOMAIN2D u
 3)CONVERGENCE c
 4)double precision, GRID2D:: g, g_old
 5)!hpf$ DISTRIBUTE (BLOCK,BLOCK) :: g,g_old
 6)g%DOMAIN = u
 7)call initGrid (g)
 8)do i=1, niters
 9)     PUT_BORDERS (g)
10)     GET_BORDERS (g)
11)     g_old = g
12)     call computeLocal (g, g_old)
13)     error = computeNorm (g, g_old)
14)     CONVERGE (c, error, maxim)
15)     Print *, “Max norm: ”, error
16)enddo
17)end subroutine solve
```

Programa 11. Proceso trabajador para las dos tareas HPF.

Aunque en el ejemplo anterior, la distribución de ambos dominios es la misma, existen casos en los cuales utilizar distintas distribuciones puede ser más eficiente, incluso si el cómputo a realizar es el mismo. Este es el caso de la *Figura 14* en la que la distribución del segundo dominio es distinta a la del primero (sacando ventaja de sus respectivas geometrías, de modo que se disminuye, de esta forma, la cantidad de datos a comunicar entre los procesadores de las tareas HPF). El proceso trabajador ya no podrá ser el mismo para ambos dominios, sino que se tendrá que llamar a dos subrutinas distintas, por ejemplo `solve1` y `solve2` (Programa 12).

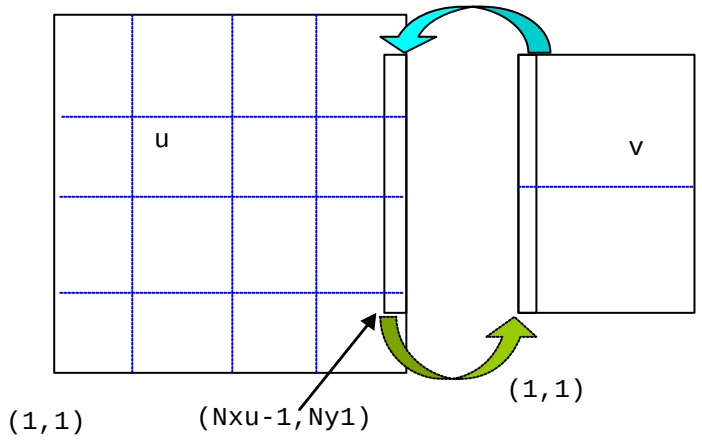


Figura 14. Dos dominios con distribuciones distintas. El dominio u *tiene distribución* `(BLOCK,BLOCK)` *mientras que el* v *la tiene* `(BLOCK,*)`

```
1)  subroutine solve1 ( u , c )
2)  DOMAIN2D u
3)  CONVERGENCE c
4)  GRID2D g, g_old
5)  !hpf$ DISTRIBUTE (BLOCK,BLOCK) :: g, g_old
6)  call solve ( u , g , g_old , c )
7)  end subroutine solve1

1)  subroutine solve2 ( u , c )
2)  DOMAIN2D u
3)  CONVERGENCE C
4)  GRID2D g, g_old
5)  !hpf$ DISTRIBUTE (BLOCK,*) :: g%DATA, g_old%DATA
6)  call solve ( u , g , g_old , c )
7)  end subroutine solve2
```

Programa 12. Las rutinas necesarias para la distribución distinta de los datos.

La razón de que se necesiten dos subrutinas distintas es que la distribución HPF debe conocerse en el momento de la compilación. Así, las subrutinas `solve1` y `solve2` sólo se diferencian en la distribución del `GRID` en la línea 5 del Programa 12. Sin embargo, la subrutina llamada en la línea 6 puede ser la misma (nueva versión de `solve`) ya que un argumento formal en HPF puede heredar su distribución.

```
 1)  subroutine solve ( u, g, g_old , c )
 2)  DOMAIN2D u
 3)  CONVEGENCE c
 4)  GRID2D g, g_old
 5)  !hpf$ inherit :: g, g_old
 6)  g%DOMAIN = u
 7)  call initGrid ( g )
 8)  do i = 1, niters
 9)        PUT_BORDERS ( g )
10)        GET_BORDERS ( g )
11)        g_old = g
12)        call computeLocal ( g , g_old)
13)        error = computeNorm ( g , g_old )
14)        CONVERGE ( c , error , maxim )
15)        Print * , “Max norm: ”, error
16)  enddo
17)  end subroutine solve
```

Programa 13. Subrutina en HPF para el proceso trabajador.

Volviendo al proceso coordinador del Programa 10, sólo se han añadido tres nuevas instrucciones (líneas 4 a 6) y solamente en el caso de distribuciones distintas, se tienen que modificar otras dos (líneas 11 y 12) con respecto al programa coordinador

del Programa 2. Por otro lado, sólo la distribución del `GRID` necesita ser incluida en la parte computacional (Programa 13) con respecto al proceso trabajador del primer ejemplo.

3.3 Transformada Rápida de Fourier en dos Dimensiones.

La transformada de Fourier es una aplicación ampliamente utilizada en gran cantidad de aplicaciones. Para hacernos una idea de la diversidad de campos en los que se utiliza citaremos como ejemplo: ingeniería biomédica, reconocimiento de imágenes, análisis financiero, análisis mecánicos, radar, comunicaciones, procesamiento del habla y de señales, sistemas no lineales, análisis geofísicos, simulación, síntesis de música, etc.

Dado un vector x de N elementos, la transformada de Fourier se obtiene aplicando:

$$X(n) = \sum_{k=0}^{N-1} x(k)\, e^{-i 2 pk/N} \qquad n = 0,1,...N-1,\ i = \sqrt{-1}$$ **Ecuación 3.**

Una descripción muy completa de la transformada de Fourier, su interpretación y los algoritmos para calcularla se puede encontrar en [Briham 88]. Uno de estos algoritmos, FFT (*Fast Fourier Transform*) fue desarrollado por John Tukey y James Cooley en 1965 [Cooley y Tukey 65]. De esta forma, se consigue que la complejidad del algoritmo se reduzca de $O(N^2)$ a O(N log N). Esto permitió la resolución del algoritmo en ordenadores "más modestos" y, por tanto, la generalización de su uso.

La transformada rápida de Fourier en dos dimensiones (FFT2D) consiste en la aplicación de la transformada rápida de Fourier en una dimensión sobre las columnas de una matriz de dos dimensiones para luego volver a calcularla sobre las filas de la matriz resultante. Un ejemplo de su aplicación es su uso en el procesamiento de señales. En la *Figura 15* se muestra el algoritmo de captura y procesamiento de una señal. El sensor genera una matriz de `N` x `N` datos a la cual hay que calcularle la transformada de Fourier.

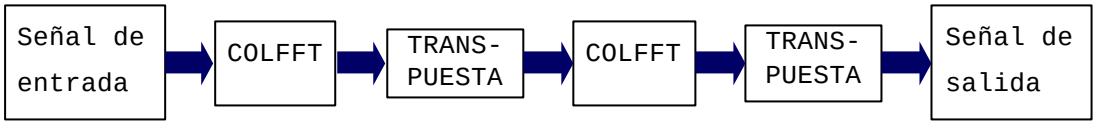


Figura 15. Esquema de resolución del algoritmo FFT2D utilizando HPF.

Una posible implementación en HPF (Programa 14) consistiría en la ejecución en paralelo de la transformada en cada una de las N columnas en P procesadores, realizar la transpuesta sobre la matriz resultante y volver a realizar la transformada para cada una de las columnas. La aplicación de la transformada a cada una de las columnas es independiente y, por tanto, se puede realizar sin ninguna comunicación.

```
 1) program fft2d_hpf
 2) !hpf$ processors pr(P)
 3) complex a(N,N)
 4) !hpf$ distribute a (*,BLOCK)
 5) do i=1, n_imagenes
 6)      call read(a)
 7)      call colfft (a,N)
 8)      a = transpuesta (a)
 9)      call colfft (a,N)
10)      a = transpuesta(a)
11)      call write (a)
12) enddo
13) end
```

Programa 14. Implementación del algoritmo FFT2D con HPF.

Dado que FFT2D escala con $N^2 \log N$ mientras que la comunicación debida a la transpuesta escala sólo como el máximo de (N^2,P^2), el algoritmo con paralelismo de datos es eficiente únicamente cuando N es mucho mayor que P. Sin embargo, los sistemas de procesado de señal deben ejecutar rápidamente el algoritmo sobre un caudal de datos de entrada relativamente pequeño (generalmente de 256 x 256 o incluso menores). En estas circunstancias, una implementación alternativa siguiendo una estructura de ejecución encauzada es más eficiente. En la *Figura 16* se muestra el esquema de la implementación del algoritmo siguiendo esta estructura. La primera tarea consta de P/2 procesadores y se encarga de obtener las señales y de realizar la transformada por columnas. Una vez terminada, manda sus datos a la segunda tarea

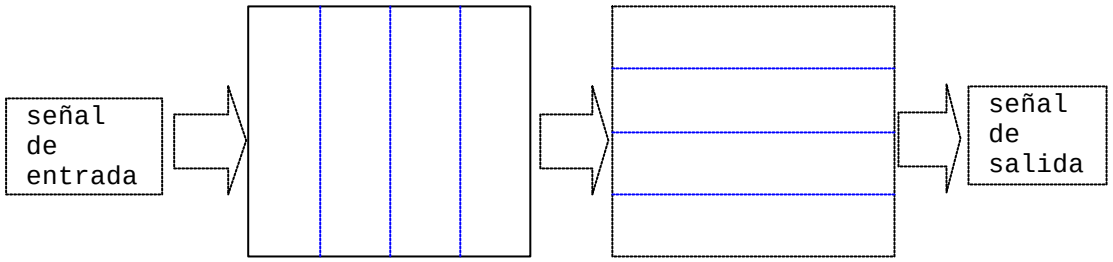


Figura 16. Distribución datos para la ejecución encauzada del problema FFT2D

(también con `P/2` procesadores). La redistribución se realiza en el envío. Como cada tarea tiene `P/2` procesadores, se necesitan $P^2/4$ mensajes.

Por el contrario, la versión con paralelismo de datos requiere una comunicación todos a todos, con lo cual, son necesarios `P(P-1)` mensajes.

Además de la ventaja en el número de mensajes, la segunda versión permite el solapamiento de cálculo y comunicación. La *Figura 17* muestra (de manera esquemática) el modo en que se ejecutan ambos algoritmos para 5 arrays de señales y `P=` 8 procesadores. Cuando sólo hay paralelismo de datos, los `P` procesadores calculan la transformada en las columnas del array para después realizar la transpuesta y volver a realizar la transformada, realizando de nuevo la transpuesta para obtener el valor resultante. En la figura se ha etiquetado `C1` al recuadro que resuelve el primer array (primero por columnas y después de nuevo por columnas tras hacer la transpuesta).

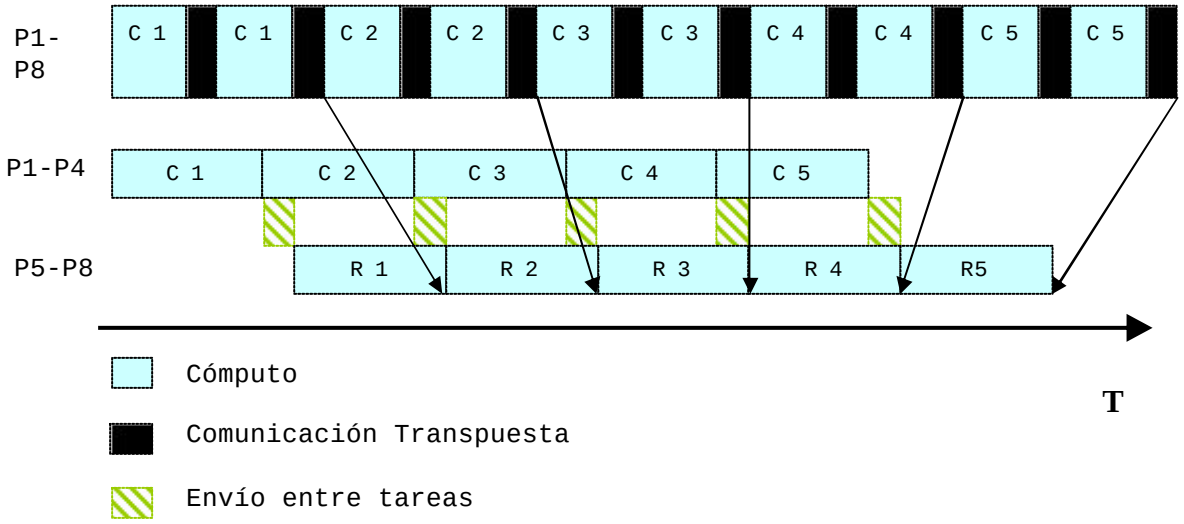


Figura 17. Comparación entre la ejecución del algoritmo FFT2D con paralelismo de datos (arriba) y con integración de paralelismo de datos y tareas (abajo).

Utilizando el esquema de encauzamiento, la transformada sobre las columnas del array se realiza sólo por `P/2` procesadores. Como este proceso no requiere comunicación, suponemos que el tiempo que tardará será el doble del empleado por el primer algoritmo. Una vez calculada la transformada por columnas, se envían los datos a la otra tarea, pero en este caso, la primera tarea no tiene que esperar a que lleguen sino que ya puede comenzar a procesar el segundo array de datos (esta operación es la que aparece como `C2`). La segunda tarea (formada por los procesadores `P5` a `P8`) comenzará a ejecutar la transformada por filas del primer array (`R1`) en cuanto lleguen los datos de la primera tarea. Sin embargo, para los siguientes arrays, apenas tiene que esperar, puesto que los datos han sido enviados antes.

En la figura, las flechas parten del instante en que se termina el proceso sobre cada array en el primer algoritmo y terminan en el instante correspondiente para el segundo algoritmo. Como puede observarse, bastan unos pocos arrays para que el segundo algoritmo sea más eficiente que el primero. Esto sería en un caso ideal, puesto que, obviamente, el envío de datos siempre consume algo de tiempo, especialmente si el tamaño de datos a enviar es grande y se llega a llenar el buffer de salida. En el capítulo 6 se ofrecen resultados experimentales para este problema.

3.3.1 Implementación con BCL.

El algoritmo que integra el paralelismo de datos y tareas puede ser codificado de forma sencilla utilizando BCL. El `Programa 15` muestra el proceso coordinador. En él, se definen los dominios `a` y `b` y los procesadores correspondientes a cada una de las tareas. La distribución del primer dominio se define en la línea 4 mientras que la del segundo se realiza en la línea 5.

La comunicación futura entre ambas tareas se declara en la línea 8. En ésta se define una frontera que (siguiendo la notación de Fortran 90 para matrices) afecta a todo el dominio, puesto que no se ha especificado ninguna región como índice de las variables. Las instrucciones 9 y 10 son las que crean las dos etapas del encauzamiento, recibiendo cada una un dominio. La primera etapa recibe el dominio `a`, cuyos datos serán enviados a la segunda tarea.

```
 1)  program fft2d
 2)  DOMAIN2D  a,b
 3)  PROCESSORS p1(P/2), p2 (P/2)
 4)  DISTRIBUTE a(*,BLOCK) ONTO p1
 5)  DISTRIBUTE b(BLOCK,*) ONTO p2
 6)  a= (/1,1,N,N/)
 7)  b= (/1,1,N,N/)
 8)  b <- a
 9)  CREATE etapa1 (a) ON p1
10)  CREATE etapa2 (b) ON p2
11)  end
```

Programa 15. Proceso coordinador para el problema FFT2D.

El código para la primera etapa se muestra en el Programa 16. La subrutina recibe una variable de tipo dominio y declara un `GRID` cuyo tipo base es `complex`. El campo `DATA` del `GRID` se crea cuando se ejecuta la instrucción de la línea 5. Después de leerse el primer array de entrada, se procede a realizar las transformadas por columnas. En el código se muestran las instrucciones necesarias para realizar el proceso `colfft`. Una vez que éste concluye, se envían los datos mediante la instrucción `PUT_BORDERS`.

```
 1) subroutine etapa1(d)
 2) DOMAIN2D  d
 3) complex, GRID2D :: a
 4) !hpf$ DISTRIBUTE a(*,BLOCK)
 5) a%DOMAIN = d
 6) do i=1,n_imagenes
 7)   call read_stream (a%DATA)
 8)   !hpf$ independent
 9)   do icol = 1,N
10)     call fft1D(a%DATA(:,icol))
11)   enddo
12)   PUT_BORDERS (a)
13) enddo
14) end
```

colfft

Programa 16. Primera etapa del algoritmo encauzado para FFT2D.

La segunda etapa se muestra en el Programa 17. Después de recibir los datos, línea 7, se realizan las transformadas unidimensionales por filas en paralelo. Una vez terminan, se escriben los resultados.

```
 1)  subroutine etapa2(d)
 2)  DOMAIN2D d
 3)  complex, GRID2D :: b
 4)  !hpf$ DISTRIBUTE b(BLOCK,*)
 5)  b%DOMAIN = d
 6)  do i=1,n_imagenes
 7)    GET_BORDERS (b)
 8)    !hpf$ independent                          rowfft
 9)    do ifila = 1,N
10)      call fft1D(b%DATA(ifila,:))
11)    enddo
12)    call write_stream (b%DATA)
13)  enddo
14)  end
```

Programa 17. Segunda etapa del algoritmo encauzado para FFT2D

3.4 Transformada de Fourier para un Problema de Difusión.

Como ya se ha comentado, el programa NAS (*Numerical Aerodynamic Simulation*) de NASA Ames pone a libre disposición una serie de aplicaciones para la evaluación de ordenadores paralelos, el denominado NPB (NAS Parallel Benchmark [Bailey y otros 94][NAS 00]). Una de las aplicaciones es la transformada de Fourier (NPB-FT) utilizada para resolver un problema de difusión en tres dimensiones. La ecuación del problema es:

$$\frac{\partial u}{\partial t} = a\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) \qquad \textbf{Ecuación 4.}$$

donde la incógnita es $u(x,y,z,t)$ y a es una constante. Después de aplicar la transformada de Fourier a ambos lados de la ecuación, se obtiene:

$$f(k_x,k_y,k_z,t) = e^{-ak^2t}v(k_x,k_y,k_z,t=0) \qquad \textbf{Ecuación 5.}$$

donde $v(k_x, k_y, k_z, t=0)$ es la transformada de Fourier de $u(k_x, k_y, k_z, t=0)$, k_x, k_y, k_z. son los números de onda en las direcciones X,Y,Z respectivamente y $k^2 = k_x^2 + k_y^2 + k_z^2$.

El banco de pruebas FT usa la Ecuación 5 para calcular la solución de la ecuación de difusión para un cierto número de pasos de tiempo. La estructura de la

aplicación es la que se muestra en la *Figura 18*, siendo `U`, `V` y `W` arrays tridimensionales de números complejos de doble precisión.

```
program NPB_FT
Fijar condiciones iniciales de U
V= FFT-3D (U)
do t = 1, T
    Wt = f(V,t)
    Ut = Inversa FFT3-D (Wt)
    Comprobar resultados de Ut
enddo
end
```

Figura 18. Estructura de resolución de la aplicación NPB-FT.

El NPB-FT versión 2.1, como muchos otros códigos FFT en 3D, usa una descomposición de datos en bloques en una dimensión. En la versión 2.3 ya se puede elegir entre una descomposición en una o dos dimensiones. Con el objeto de facilitar su compresión, aquí se explica únicamente la descomposición unidimensional (*Figura 19*). El array 3D, `U`, se distribuye en bloques a lo largo de su eje Z. Cada nodo posee una porción de datos conteniendo las dimensiones X e Y sin partir. Para realizar la FFT 3D, primero se realizan FFT de una dimensión a lo largo del eje X, después a lo largo del eje Y y, por último, a lo largo del eje Z. Las FFT a lo largo de los ejes X e Y se pueden realizar sin comunicación entre nodos, dada la descomposición que se ha hecho. Entonces se realiza una transpuesta global de la matriz, permutando los ejes de modo que el eje Z se intercambia con el X. En el array transpuesto, el eje Z ya no está distribuido, con lo cual, el conjunto final de FFT 1D no necesita comunicación.

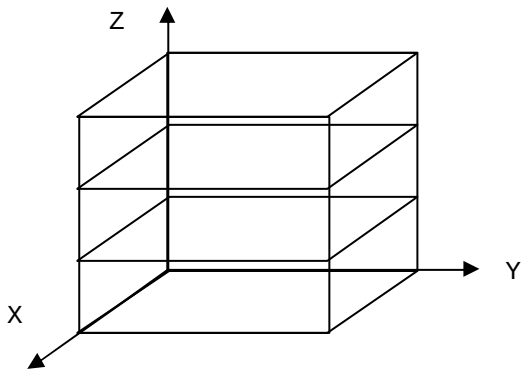


Figura 19. Descomposición del array tridimensional a lo largo del eje Z.

La transformada inversa se calcula con un algoritmo similar. Para realizar la FFT 3D inversa en el array `w` se siguen los siguientes pasos:

1. Realizar un conjunto de FFT 1D inversas a lo largo de la primera dimensión de `w` (correspondiente a la tercera dimensión de la matriz original `u`).

2. Realizar la transpuesta de `w`.

3. Realizar un conjunto de FFT 1D a lo largo de la primera dimensión de `w`.

4. Realizar un conjunto de FFT 1D a lo largo de la segunda dimensión de `w`.

Como ocurría en el caso del problema FFT 2D, el cálculo de la transpuesta es muy costoso, puesto que hay que realizar un gran número de comunicaciones. Para aliviar esta sobrecarga, en [Agarwal y otros 94] se presenta una versión reestructurada de la resolución de la aplicación para superponer explícitamente comunicación y computación. Su algoritmo hace un encauzamiento de la FFT 3D inversa con el resto de la computación realizada dentro del bucle de escalones de tiempo. Además, se realiza un encauzamiento dentro de la FFT 3D inversa. Sin embargo, en el artículo no se hace mención a los resultados puesto que su sistema no permite la superposición del cálculo y de la comunicación. En la tesis de Fink [Fink 98] se explica e implementa, utilizando el sistema KeLP, una versión más simple de este algoritmo en la que sólo se hace un encauzamiento.

3.4.1 Implementación con BCL.

Este esquema de resolución es fácilmente codificable en BCL. El Programa 18 muestra el programa coordinador para las dos etapas de la *Figura 20*. El cálculo de la FFT3D no se ha dividido, puesto que sólo se ejecuta una vez. Sin embargo, la FFT3D inversa se ejecuta para un número de iteraciones (tantas como escalones de tiempo se quieran calcular) por lo que se ha descompuesto su cálculo en dos etapas (de manera similar a como se hacía en el caso del problema FFT2D).

Como se puede observar, el proceso coordinador de este esquema de resolución es muy parecido al del Programa 15, cambiando únicamente la dimensionalidad del problema. En este caso, también se definen dos dominios pero ahora son tridimensionales. Se declara su distribución y, en la línea 8, se define la frontera entre ellos. En las líneas 9 y 10 se crean las dos etapas que se muestran en la *Figura 20*.

```
 1)  program NPB_FT
 2)  DOMAIN3D  a,b
 3)  PROCESSORS p1(P/2), p2 (P/2)
 4)  DISTRIBUTE a(*,BLOCK,*) ONTO p1
 5)  DISTRIBUTE b(*,*,BLOCK) ONTO p2
 6)  a= (/1,1,1,N,N,N/)
 7)  b= (/1,1,1,N,N,N/)
 8)  b <- a
 9)  CREATE etapa1 (a) ON p1
10)  CREATE etapa2 (b) ON p2
11)  end
```

Programa 18. Proceso coordinador para el problema NPB-FT.

El código necesario para la primera etapa se muestra en el Programa 19. Algunos de los detalles de la implementación se han ocultado para facilitar su comprensión. En concreto, se han omitido algunos cálculos previos que permiten realizar la transformada de forma más eficiente.

En general, la etapa declara los tres GRIDs que serán asociados al dominio pasado como argumento (d). El GRID u es el que contiene las condiciones iniciales y es el que está distribuido en su tercera dimensión (*Figura 19*). Tras calcular las condiciones iniciales y obtener su transformada, se almacena en v cuya distribución es distinta, puesto que cambia al realizarse la transformada (como se explicó anteriormente). En la línea 11 comienza el bucle en escalones de tiempo. Para cada

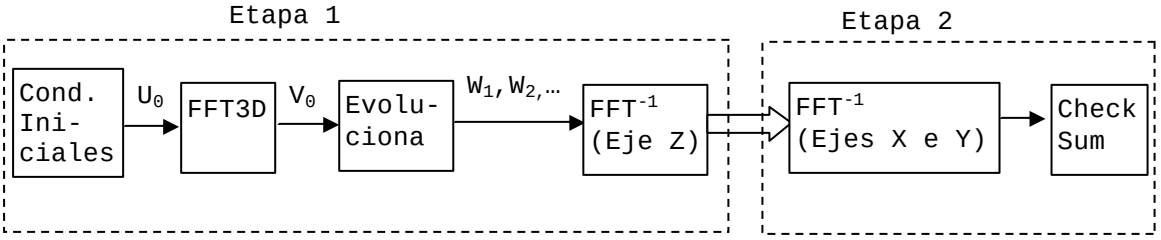


Figura 20. Esquema de resolución de NPB-FT con un encauzamiento de dos etapas.

iteración se genera un valor de `w`, que es al que hay que realizar la transformada inversa. La transformada sobre el eje Z se realiza en la línea 13 y se envían los datos obtenidos a la etapa siguiente. Al mismo tiempo (mientras se realiza la comunicación) ya se pueden ir generando los valores para el siguiente paso de tiempo.

```
 1) subroutine etapa1(d)
 2) DOMAIN3D  d
 3) complex, GRID3D :: u,v,w
 4) !hpf$ DISTRIBUTE u(*,*,BLOCK)
 5) !hpf$ DISTRIBUTE (*,BLOCK,*) :: v,w
 6) u%DOMAIN = d
 7) v%DOMAIN = d
 8) w%DOMAIN = d
 9) call cond_iniciales (u)
10) call fft3d (u,v)
11) do iter=1,T
12)   call evoluciona (v,w,iter)
13)   call fft3d_inversa_ejeZ(w)
14)   PUT_BORDERS (w)
15) enddo
16) end
```

Programa 19. Primera etapa del problema NPB-FT.

La segunda etapa se muestra en el Programa 20. Después de definir la variable de tipo `GRID` que se necesita, `u`, se entra en el bucle de escalones de tiempo en el que simplemente hay que recoger los datos de la etapa anterior y terminar de calcular la transformada inversa mediante el cálculo de las transformadas unidimensionales a lo largo del eje X (que se corresponde con el Y en el array original), la transpuesta del `GRID` y las transformadas de nuevo a lo largo del eje X. Por último, se comprueban los resultados.

```
 1) subroutine etapa2(d)
 2) DOMAIN3D  d
 3) complex, GRID3D :: u
 4) !hpf$ DISTRIBUTE (*,*,BLOCK):: u
 5) u%DOMAIN = d
 6) do iter=1,T
 7)   GET_BORDERS (u)
 8)   call fft3d_inversa_ejesXY(u)
 9)   call check_sum(u)
10) enddo
11) end
```

Programa 20. Segunda etapa del problema NPB-FT.

En el capítulo 6 se muestra una comparación entre los tiempos de ejecución de las soluciones con HPF y BCL.

3.4.2 Resolución con replicación de etapas.

Una descripción de la implementación de NPB utilizando HPF se puede encontrar en [Frumkin y otros 98], aunque en esta versión no se hace ningún tipo de encauzamiento. A pesar de que se menciona su existencia, ni en este artículo ni en ninguna de las otras dos versiones referenciadas (ni siquiera en la versión oficial paralela del NPB utilizando MPI [NAS 00]), se saca partido a otro nivel de paralelismo posible. Se trata del bucle en escalones de tiempo que es totalmente paralelizable ya que el cálculo de W_t no depende de W_{t-1}. Para sacar provecho de este nivel de paralelismo, en la *Figura 21* se muestra un esquema de resolución en el que se replica R veces el

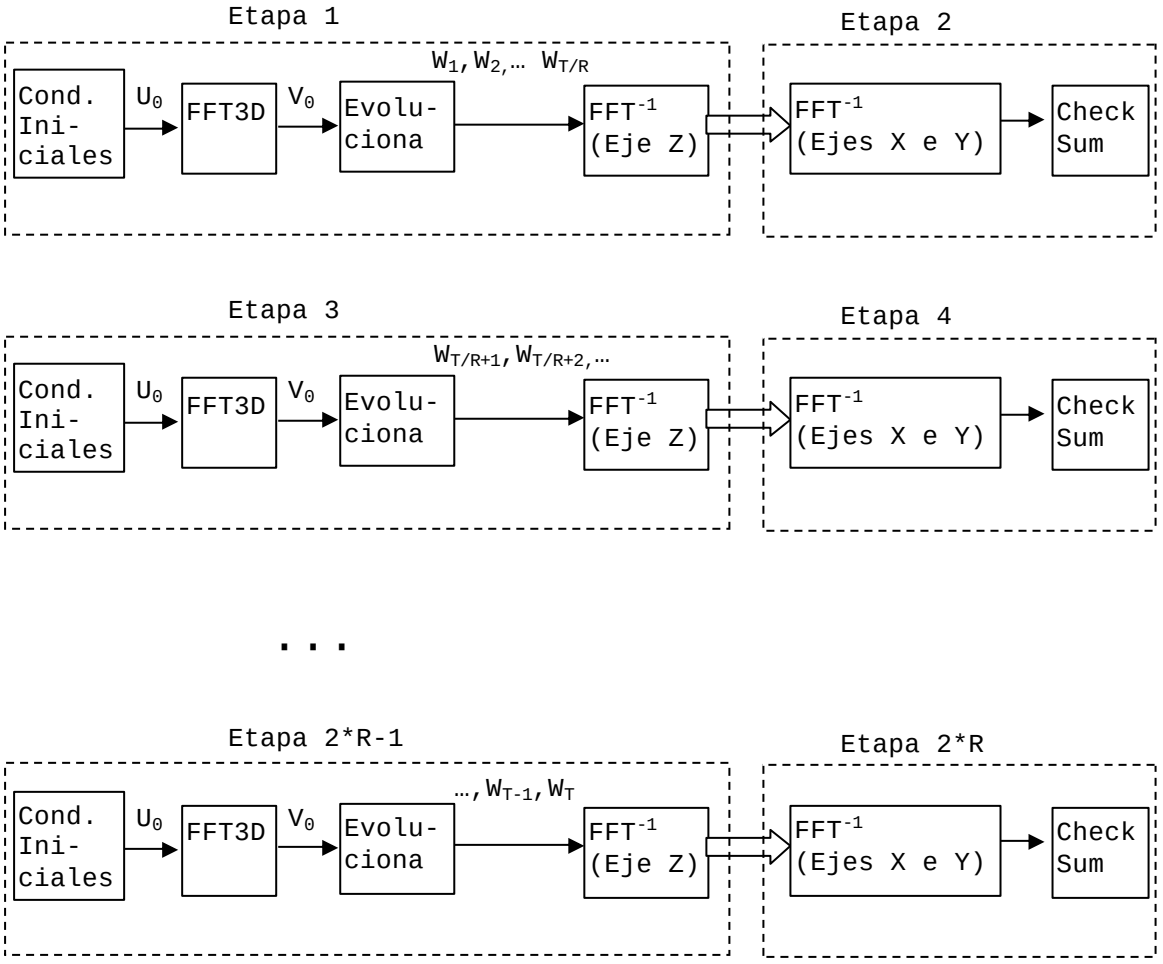


Figura 21. Esquema de resolución de NPB-FT sacando partido al nivel de paralelismo más externo con R *réplicas, es decir,* $2 \cdot R$ *etapas***.**

proceso de la *Figura 20*. De esta forma se tienen `2·R` etapas. En la primera réplica se obtienen los valores de `W` desde `t=1` hasta `T/R`. En la segunda réplica se calculan los calores de `W` desde `t=T/R+1` hasta `2T/R` y así sucesivamente.

Para implementar este esquema de ejecución con BCL bastaría con hacer algunas modificaciones en el proceso coordinador y unas modificaciones mínimas a los procesos trabajadores.

Como puede verse en el proceso coordinador mostrado en el Programa 21, en lugar de tener dos dominios, tendremos dos arrays de dominios cada uno de tamaño `R` (línea 2). La declaración de la línea 3 se realiza para distribuir equitativamente los `P` procesadores entre cada una de las dos etapas de las `R` réplicas. La distribución de los datos en el primer y segundo array de dominios se realiza según las instrucciones 4 y 5 respectivamente. La declaración de los tamaños de los dominios se realiza en el bucle que aparece en las líneas 6 a 9 mientras que las fronteras se declaran en las líneas 10 a 12. Para crear los procesos, se obtienen primero las iteraciones que ha de realizar cada uno (calculando el primer y último valor del bucle mediante los valores enteros `first_it` y `last_it`) y se pasa como argumento a cada tarea el dominio que le corresponde y número de veces que se ha de iterar.

```
 1)  program NPB_FT_2
 2)  DOMAIN3D  a(R),b(R)
 3)  PROCESSORS p1(2,P/(2*R),R)
 4)  DISTRIBUTE (*,BLOCK,*):: a(:) ONTO p1(1, P/(2*R),R)
 5)  DISTRIBUTE (*,*,BLOCK):: b(:) ONTO p1(2, P/(2*R),R)
 6)  do i=1, R
 7)     a(i) = (/1,1,1,N,N,N/)
 8)     b(i) = (/1,1,1,N,N,N/)
 9)  enddo
10)  do i=1,R
11)     b(i) <- a(i)
12)  enddo
13)  do i=1,R
14)    first_it = T/R*(i-1)+1
15)    last_it = T/R*i
16)    CREATE etapa1 (a(i),first_it,last_it) ON p1(1,:,i)
17)    CREATE etapa2 (b(i),first_it,last_it) ON p1(2,:,i)
18)  enddo
19)  end
```

Programa 21. Proceso coordinador para el problema NPB-FT con `P` procesadores y `R` replicas.

Las modificaciones a realizar en los procesos trabajadores consistirían simplemente en declarar como argumento formal las variables que indican la primera y última iteración (línea 1 del Programa 19 y del Programa 20) y modificar el bucle de pasos de tiempo para que se ejecute en los valores indicados (líneas 11 y 6, respectivamente). Ninguna otra instrucción tiene que ser modificada. Puesto que la versión completa de la parte de cómputo del programa consta de más de 1.000 líneas de código, la modificación de dos líneas del programa principal no parece ser de gran relevancia.

3.4.3 Replicación de la transformada inversa.

Como puede observarse en la *Figura 21*, el esquema propuesto tiene el inconveniente de replicar la parte de inicialización del programa (cálculo de los valores iniciales y de su transformada). Esto puede ser significativo si el número de pasos de

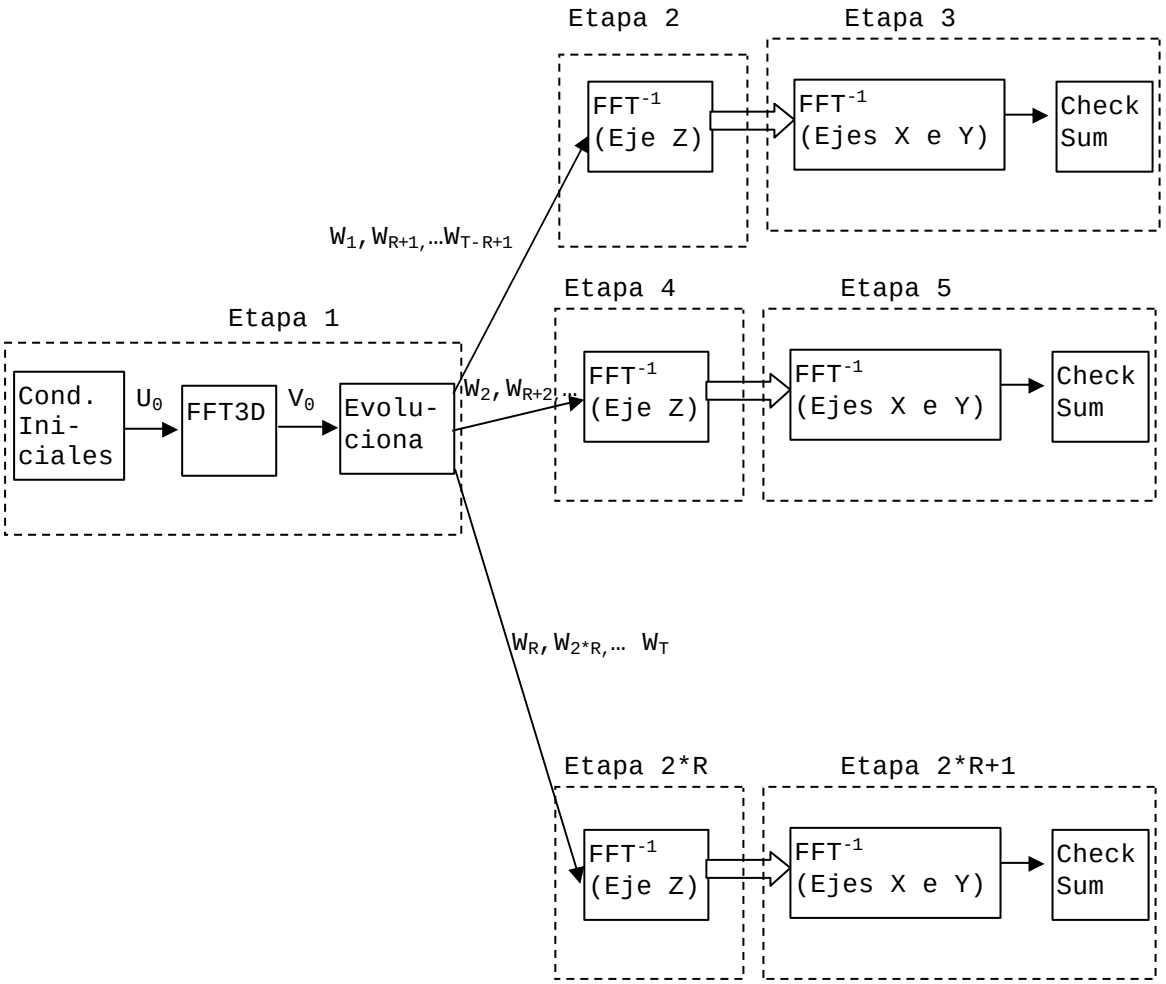


Figura 22. Esquema de resolución de NPB-FT replicando únicamente el cálculo de la transformada inversa.

tiempo no es muy grande, como ocurre con los tamaños pequeños de ejecución del NPB. Para aliviar esta réplica no deseable se puede optar, por ejemplo, por una estructura de cómputo parecida a la que se muestra en la *Figura 22*. En este caso, se tiene una primera etapa encargada de realizar los cómputos iniciales y de ejecutar cada una de las llamadas a la subrutina `evoluciona`, generando así, los distintos valores de la matriz `W`. Estos valores son enviados de manera cíclica a cada una de las `R` réplicas. De esta forma, la primera réplica recibirá los valores W_1, W_{R+1}, etc. Cada réplica está a su vez compuesta por dos etapas, la primera realiza la primera parte de la transformada inversa y la segunda la termina.

El proceso coordinador para esta estructura de cómputo es el mostrado en el Programa 22. Ahora tenemos un tercer dominio, `c`, que es el que se utiliza para la primera etapa del encauzamiento. Los arrays de dominios se declaran igual que en el caso anterior.

```
 1)  program NPB_FT_3
 2)  DOMAIN3D  a(R),b(R),c
 3)  PROCESSORS p1(P1), p2(P2), p3(P3)
 4)  DISTRIBUTE (*,BLOCK,*):: a(:) ONTO p1
 5)  DISTRIBUTE (*,*,BLOCK):: b(:) ONTO p2
 6)  DISTRIBUTE c(*,BLOCK,*) ONTO p3
 7)  do i=1, R
 8)     a(i) = (/1,1,1,N,N,N/)
 9)     b(i) = (/1,1,1,N,N,N/)
10)  enddo
11)  c = (/1,1,1,N,N,N/)
12)  do i=1,R
13)     a(i) <-(i) c
14)     b(i) <- a(i)
15)  enddo
16)  CREATE etapa1 (c,R) ON p3
17)  do i=1,R
18)    CREATE etapa_par (a(i),T/R) ON p1
19)    CREATE etapa_impar (b(i),T/R) ON p2
20)  enddo
21)  end
```

Programa 22. Proceso coordinador para el problema NPB-FT donde solamente se replica el cómputo de la transformada inversa.

Hay que mencionar el uso que se hace en la línea 13 de este programa de la etiqueta de frontera que aparece entre paréntesis detrás del operador de conexión. Puesto que un dominio puede tener varias fronteras, se está creando una frontera entre `c`

y cada dominio del array `a`. Sin embargo, para poder realizar la comunicación a cada etapa, simplemente hay que etiquetar cada frontera y así poder elegir a qué etapa enviar el dato.

En la línea 16 se crea la primera etapa, a la cual sólo hay que comunicarle el dominio y el número de réplicas que se desean. El paso de este valor, en realidad se podría eliminar, si `R` se definiese como una constante. En las líneas 17 a 20 se crean tanto las etapas pares como las etapas impares (las numeradas en la *Figura 22* como `3`, `5`, ... `2·R+1`). A cada etapa se le pasa como argumento, el dominio correspondiente y el número de iteraciones que ha de realizar.

La primera etapa está implementada en el Programa 23. Lo único que se ha modificado respecto al Programa 19 es la declaración como argumento formal del número de réplicas, la eliminación del cálculo de la primera parte de la transformada inversa y la etiqueta que acompaña al `GRID` en la llamada a la instrucción `PUT_BORDERS` (línea 13). De esta forma se consigue que cada envío se realice a una etapa distinta de manera cíclica (por simplicidad se asume que `T` es un múltiplo de `R`).

```
 1) subroutine etapa1(d,R)
 2) DOMAIN3D  d
 3) complex, GRID3D :: u,v,w
 4) !hpf$ DISTRIBUTE u(*,*,BLOCK)
 5) !hpf$ DISTRIBUTE (*,BLOCK,*) :: v,w
 6) u%DOMAIN = d
 7) v%DOMAIN = d
 8) w%DOMAIN = d
 9) call cond_iniciales (u)
10) call fft3d (u,v)
11) do iter=1,T
12)   call evoluciona (v,w,iter)
13)   PUT_BORDERS (w, mod (iter, R)+1)
14) enddo
15) end
```

Programa 23. Primera etapa del problema NPB-FT donde sólo se replica la transformada inversa.

Las etapas pares se codifican según se muestra en el Programa 24. Esta etapa es la típica de una estructura encauzada, en donde se reciben unos valores de una etapa anterior, se hace un cómputo y se envían los resultados a la etapa siguiente. Debido a las facilidades del lenguaje, no es necesario especificar de qué etapas, procesos ni

procesadores hay que recibir ni enviar información, puesto que esto se ha descrito en el coordinador. Nótese que al no especificarse ninguna etiqueta en la instrucción `GET_BORDERS` (línea 7) y según la definición de esta instrucción, se recibirá cualquiera de los tipos de frontera (recuérdese que la etapa `i`-ésima tendrá definida una frontera de tipo `i`, línea 13 del Programa 22). Finalmente, en la línea 9 se mandan los datos a la etapa siguiente de la misma réplica. El número de veces que se ha de ejecutar el bucle es pasado como argumento desde el coordinador.

```
 1) subroutine etapa_par(d, num_it)
 2) DOMAIN3D  d
 3) complex, GRID3D :: w
 4) !hpf$ DISTRIBUTE (*,BLOCK,*) :: w
 5) w%DOMAIN = d
 6) do iter=1,num_it
 7)   GET_BORDERS (w)
 8)   call fft3d_inversa_ejeZ(w)
 9)   PUT_BORDERS (w)
10) enddo
11) end
```

Programa 24. Etapa par del problema NPB-FT donde sólo se replica la transformada inversa.

La segunda etapa de cada una de las réplicas se escribe como se muestra en el Programa 25. También recibe como argumento el número de iteraciones que se tienen que realizar (aunque el coordinador podría haberle pasado el primer y último valor a ejecutar, pudiéndose entonces reutilizar el programa de la versión anterior). Por cada iteración se toman los valores de la etapa anterior, se termina de calcular la transformada inversa y se comprueban los resultados.

```
 1) subroutine etapa_impar(d,num_it)
 2) DOMAIN3D  d
 3) complex, GRID3D :: u
 4) !hpf$ DISTRIBUTE (*,*,BLOCK):: u
 5) u%DOMAIN = d
 6) do iter=1,num_it
 7)   GET_BORDERS (u)
 8)   call fft3d_inversa_ejesXY(u)
 9)   call check_sum(u)
10) enddo
11) end
```

Programa 25. Etapa impar del problema NPB-FT donde sólo se replica la transformada inversa.

3.5 Conclusiones.

La integración del paralelismo de datos y tareas es un campo de investigación abierto en el que se están haciendo esfuerzos en distintas direcciones. El uso de lenguajes de coordinación presenta la ventaja de poder describir de forma separada las distintas tareas con paralelismo de datos y la manera en que estas interactúan.

El uso del tipo dominio y la declaración de su distribución en el proceso coordinador permite averiguar en tiempo de compilación los datos que cada procesador de una tarea HPF tiene que enviar a cada procesador del resto de las tareas.

Mediante una serie de ejemplos se ha mostrado como, usando BCL, es relativamente sencillo describir esquemas de computación que integran tanto el paralelismo de datos y tareas como la replicación de tareas para sacar provecho a distintos niveles de paralelismo.

A la hora de escribir cada una de las tareas, el programador no tiene que pensar en todos los aspectos de la comunicación y sincronización. Esto se consigue gracias al modelo de paralelismo que ofrecen, por un lado HPF y por el otro BCL. De esta forma, sólo ha de tener en cuenta en qué momento necesita actualizar las fronteras de cada dominio, especialmente en problemas de descomposición de dominios, puesto que es para estos problemas para los que fue diseñado nuestro lenguaje. Para otros problemas en los que la comunicación entre tareas se basa igualmente en el envío de (sub)matrices, BCL también ofrece un modelo elegante de integración.

Además de estas ventajas y como se muestra en el capítulo de resultados, BCL ofrece soluciones más eficientes que HPF cuando se saca partido al paralelismo de datos y tareas.

Como se verá en el capítulo 4, tanto la parte de coordinación como la parte de cómputo se pueden describir de forma aún más expresiva mediante aspectos de más alto nivel, basados en el uso de patrones y esqueletos para la parte de coordinación y en el uso de plantillas para la de cómputo.

Capítulo 4. Uso de patrones para la integración del paralelismo de datos y tareas

La razón de ser de la programación paralela es la eficiencia (aunque para algunos autores la ventaja del paralelismo es el poder afrontar problemas más complejos, al disponer de más recursos computacionales). Por esta razón, es comprensible la tendencia que ha habido hasta ahora en la disciplina de la programación paralela a ignorar los aspectos de alto nivel relativos a la programación y diseño de lenguajes. Actualmente se están haciendo esfuerzos en esta dirección, en el sentido de aplicar a la programación paralela las disciplinas que en su momento supusieron la introducción de la programación estructurada en la programación secuencial.

La justificación de estos esfuerzos viene dada por la experiencia, la cual nos enseña que los programas paralelos raramente consisten en procesos aleatorios que se comunican e interaccionan de forma impredecible sino que, por el contrario, estas interacciones suelen estar bien estructuradas y se ajustan a unos pocos patrones más o menos regulares.

En [Pelagatti 98] se propone una metodología para el desarrollo de software paralelo que consiste en fijar unos cuantos patrones para definir el paralelismo (los constructores del paralelismo). El uso de estos constructores es la filosofía de la programación paralela estructurada, en la cual los programadores puede expresar la estructura paralela de un algoritmo mediante la composición jerárquica de los paradigmas básicos introducidos (y sólo de estos), de la misma forma que en la programación secuencial estructurada solamente se puede expresar la estructura de control mediante un limitado número de sentencias (`if`, `for`, etc.).

Siguiendo esta filosofía es como se ha realizado la extensión de BCL para facilitar la construcción estructurada de programas paralelos para el tipo de problemas que estamos tratando. Al conjunto de estas nuevas construcciones se le ha denominado DIP.

A continuación se describen las características de DIP, es decir, por un lado, los patrones de comunicación y sincronización, y, por otro, las plantillas de implementación que facilitan la codificación de las tareas computacionales. Seguidamente se muestran varios ejemplos que sirven para ilustrar el modelo. Un ejemplo más elaborado se encuentra en el anexo.

4.1 Patrones y Plantillas.

DIP proporciona una serie de características de alto nivel a BCL para permitir la definición de una red de tareas HPF que cooperan, donde cada tarea se asigna a un subconjunto disjunto de procesadores. Las tareas interaccionan siguiendo patrones estáticos y predecibles que pueden ser compuestos utilizando estructuras predefinidas llamadas *patrones o esqueletos*, de forma declarativa.

Se han propuesto dos patrones en DIP. El patrón `MULTIBLOCK` está enfocado a la solución de problemas multibloque y de descomposición de dominios, los cuales forman una clase importante de problemas en el área de la computación de alto rendimiento (*high performance computing*). El otro patrón suministrado por DIP es el denominado `PIPE`, que se introduce para facilitar la implementación de estructuras de cómputo en forma de encauzamiento de tareas.

DIP también se basa en el uso de dominios, es decir, regiones junto con alguna información sobre la interacción entre estos, que permite la coordinación eficiente entre tareas.

Para facilitar la implementación de las tareas computacionales, el programador también puede usar un nivel de abstracción más alto para manejar los aspectos de comunicación y sincronización mediante el uso de plantillas de implementación. Inicialmente hemos establecido un conjunto de plantillas para resolver el tipo de

problemas con los que estamos tratando. Sin embargo, nuestra aproximación permite al programador definir nuevas plantillas, ampliándose así, el abanico de posibilidades de aplicación de nuestro lenguaje.

4.1.1 El Patrón MULTIBLOCK.

El patrón `MULTIBLOCK` está pensado para la resolución de problemas multibloque y de descomposición de dominios de manera fácil, elegante y declarativa. La *Figura 23* muestra el esquema general de un patrón de este tipo. En la primera línea se escribe el nombre del patrón junto con la definición de los dominios que se vayan a emplear.

```
MULTIBLOCK nombre_patrón definición de dominios
  Tarea_1 (dominio_1:(distribución datos)) procesadores
  Tarea_2 (dominio_2:(distribución datos)) procesadores
  ....
  Tarea_m (dominio_n:(distribución datos)) procesadores
WITH BORDERS
  Definición de Fronteras
END
```

Figura 23. El patrón MULTIBLOCK.

Una definición de dominio se realiza mediante la asignación de sus puntos cartesianos, es decir, se establece la región del dominio. Por ejemplo, para el caso bidimensional, la expresión `u/1,1,Nx,Ny/` asigna al dominio `u` la región rectangular del plano que se extiende desde el punto `(1,1)` al `(Nx,Ny)`.

A continuación se especifican las distintas tareas que habrán de crearse. Cada una de ellas se corresponderá con el nombre de una subrutina escrita en HPF, a la cual se pasará como argumento el nombre de uno o varios dominios junto con su distribución. En la llamada a la tarea, también se especifica la disposición de los procesadores que la ejecutan. Los tipos de distribución se corresponden a los de HPF. Esta declaración no realiza ninguna distribución sino que indica la futura distribución de los datos asociados al dominio especificado. El conocimiento de la distribución en el nivel de la coordinación es la clave para una implementación eficiente de la comunicación entre las tareas HPF.

Las fronteras existentes entre los dominios especificados se definen en la sección `WITH BORDERS`. Las fronteras se definen igual que en el capítulo 2, permitiéndose

también aplicar una función al lado derecho del operador, `<-`, en el que estén implicados más de un dominio. Igualmente, es posible el etiquetado de fronteras para discriminar su comunicación o agruparlas, como se explicó anteriormente.

Una tarea conoce así la distribución de sus dominios y la distribución de los dominios que tengan alguna frontera en común con ella. De esta forma, se puede deducir qué parte de la frontera tiene que ser enviada a qué procesador de otra tarea. Esto se realiza en tiempo de compilación.

4.1.2 El Patrón PIPE.

Este patrón encauza secuencias de tareas HPF. La *Figura 24* muestra la estructura general de un encauzamiento con `n` etapas correspondiente al patrón PIPE mostrado en la *Figura 25*.

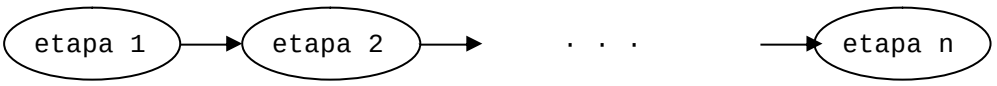


Figura 24. Estructura de un encauzamiento con n etapas.

```
PIPE nombre_patrón dominios
  etapa 1
  etapa 2
  ....
  etapa n
END
```

Figura 25. El patrón PIPE.

Cada etapa del encauzamiento consume y produce un flujo de datos, excepto la primera y última etapa que sólo produce o consume, respectivamente. El flujo de datos está compuesto por un número determinado o indeterminado de elementos. Puesto que nuestra aproximación está basada en el uso de dominios, el tipo de datos de los canales de entrada/salida que conecta a cada par de tareas, es decir, el tipo de los elementos del flujo, no tiene que ser especificado en el patrón, lo cual mejora la reusabilidad de la

parte de coordinación de la aplicación, al contrario de lo que ocurre en otros modelos [Ciarpaglini y otros 00].

Una etapa del encauzamiento puede ser una de las siguientes:

- Una llamada a una tarea, con una forma similar a la llamada a una tarea del patrón `MULTIBLOCK`.

- Una llamada a un encauzamiento, es decir, el nombre de un patrón `PIPE` anidado y declarado previamente, junto con los dominios que necesita.

- Una directiva `REPLICATE`, la cual se utiliza para replicar etapas que no son escalables. De esta forma, se puede mejorar la eficiencia del encauzamiento ya que diferentes elementos del flujo de datos se pueden calcular en paralelo en conjuntos distintos de procesadores [Gross y otros 94]. Cada una de las réplicas irá tomando datos del flujo, pudiéndose aplicar diferentes métodos, tales como Round-Robin o por demanda, es decir, la primera réplica que consume un dato, pedirá otro a su predecesor. En nuestra aproximación, lo hemos limitado a Round-Robin[9]. La *Figura 26* muestra un esquema de computación con 6 etapas en la cual se ha replicado la quinta.

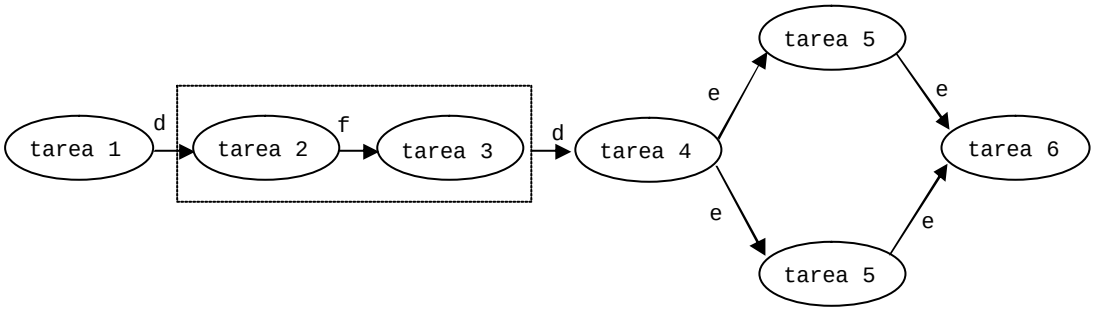


Figura 26. Estructura PIPELINE con 6 etapas y replicación de la quinta.

[9] Aunque la obtención de datos de la etapa anterior por demanda puede ser en algunas ocasiones más eficiente que la implementación cíclica, la complejidad que introduce no parece que sea compensada por ese aumento de eficiencia.

El patrón correspondiente a la estructura de cómputo de la *Figura 26* se muestra en el Programa 26. En la línea 5, comienza la declaración del PIPE más externo, es decir, el principal. A continuación del nombre dado al patrón debe aparecer cada uno de los dominios que van a ser utilizados junto con su definición (es decir, la región o tamaño que ocupan). En el ejemplo se definen dos dominios de tamaños distintos, `d` y `e`.

```
 1) PIPE pipe_anidado f/1,1,N,N/
 2)   tarea2 (f:(*,BLOCK)) ON PROCS (4)
 3)   tarea3 (f:(BLOCK,*)) ON PROCS (4)
 4) END
 5) PIPE main_PIPE d/1,1,N,N/, e/1,1,M,M/
 6)   tarea1 (d:(*,BLOCK)) ON PROCS(2)
 7)   pipe_anidado(d)
 8)   tarea4 (d:(BLOCK,*),e:(BLOCK,BLOCK)) ON PROCS (4)
 9)   REPLICATE (2) tarea5 (e:(BLOCK,*)) ON PROCS(2)
10)   tarea6 (e:(BLOCK,*)) ON PROCS (2)
11) END
```

Programa 26. Un ejemplo de uso del patrón PIPE.

Es muy importante resaltar que, al contrario de lo que ocurre en el patrón `MULTIBLOCK`, la comunicación entre las etapas del patrón no se declara mediante la definición de fronteras, sino que, en este caso, se realiza pasando como parámetro el mismo dominio a dos etapas del encauzamiento.

Cuando se pasa un dominio a una etapa se debe declarar la distribución de éste dentro de la etapa. Así, en el ejemplo anterior, la `tarea4` declara la distribución del dominio `e` como `(BLOCK,BLOCK)`, mientras que en la `tarea5` se declara `e` como `(BLOCK,*)`.

La etapa donde aparece un dominio por primera vez es considerada como la que genera los datos asociados a ese dominio. Si otra etapa recibe el mismo dominio, quiere decir que recibirá los datos generados por la etapa anterior. Si el dominio aparece en varias etapas del encauzamiento, quiere decir que las etapas intermedias (es decir, aquellas distintas de la primera y la última), además de recibir los datos, también los envían (seguramente tras realizar algún cómputo) a la etapa siguiente. En el ejemplo, la `tarea1`, genera los datos asociados al dominio `d` que son recibidos y enviados a las etapas siguientes por las tareas `tarea2` y `tarea3`. La `tarea4`, consume los datos y envía

a la etapa siguiente los datos asociados al dominio `e`, que son procesados por las réplicas de la `tarea5` y consumidos por la `tarea6`.

Los patrones `PIPE` se pueden anidar para incrementar su reutilización. De esta forma, se pueden utilizar estructuras `PIPE` previamente declaradas. En el ejemplo mostrado, se tiene una estructura denominada `pipe_anidado` que toma un dominio `f` y es procesado primero por la `tarea2` para pasárselo luego a la `tarea3`. Cuando en la línea 7 se llama a la estructura anidada, se debe pasar un dominio que tenga la misma dimensión y tamaño al declarado en la cabecera de la estructura anidada (línea 1).

En el ejemplo mostrado, la `tarea5` se ha replicado de manera que dos instancias de esta tarea se ejecuten en dos procesadores cada una. Cuando la `tarea4` envía un elemento asociado al dominio `e`, sólo una instancia de la `tarea5` la recibirá en orden cíclico (Round-Robin).

4.1.3 Plantillas de Implementación para las Tareas Computacionales.

A la hora de escribir las tareas computacionales, el programador puede elegir entre dos alternativas:

La primera de ellas consiste en usar código BCL como el mostrado en el capítulo anterior para los procesos trabajadores. Éstos recibirán como parámetro los dominios especificados en cada etapa del patrón `PIPE` o `MULTIBLOCK`. En este último caso, no es necesario, en principio, el uso de variables de tipo `CONVERGENCE` ya que se supone que el test de convergencia se realiza entre todas las tareas que intervienen en la resolución del problema. La instrucción `CONVERGE` se puede sustituir por otra denominada `REDUCE` en la cual se suprime la variable de tipo convergencia, realizando, por tanto, la reducción de la variable escalar pasada como primer parámetro sobre todas las tareas activas. Las instrucciones `PUT_BORDERS, GET_BORDERS` y `UPDATE_BORDERS` se utilizan para la comunicación entre tareas. El compilador utilizará entonces los patrones simplemente para crear el proceso coordinador.

La segunda posibilidad es la utilización de las plantillas de implementación. De esta forma, el programador utiliza un nivel de abstracción más alto para manejar los

aspectos de comunicación y sincronización, pudiéndose omitir, de esta manera, otros aspectos de bajo nivel.

Una plantilla es un esqueleto de una tarea HPF que coopera con otras tareas de acuerdo con un patrón de interacción predefinido. Para obtener la implementación real de las tareas del usuario, las plantillas deben ser *instanciadas* mediante la inclusión de código proporcionado por el usuario. Las plantillas se *parametrizan* mediante argumentos para recibir este código. Además de los tipos predefinidos de Fortran, se han establecido dos nuevos tipos de argumentos: `HPF_DECL` y `HPF_CODE` para la declaración de variables y código HPF, respectivamente. El compilador del sistema se encarga de crear automáticamente las instancias. Los tipos de los argumentos de las plantillas ayudan al compilador a detectar posibles errores en la especificación del usuario. De esta forma, se pueden identificar errores dentro de la especificación para que el usuario los pueda corregir en dicha especificación, en lugar de detectarlas en la compilación de la instancia, la cual, al ser parcialmente generada por nuestro compilador, contendrá código más complejo y, por tanto, más difícil de corregir.

A continuación se describen las distintas plantillas que se han establecido. El usuario puede añadir nuevas plantillas que sean apropiadas a la clase de aplicaciones con las que esté trabajando.

4.1.3.1 Plantillas Multiblock.

Se han definido dos plantillas de implementación para generar tareas HPF para el patrón `MULTIBLOCK`. Estas son las plantillas `Elliptic` y `Parabolic`. Ambas se ajustan al problema de la resolución de PDEs mediante métodos de diferencias finitas usando descomposición de dominios. Las ecuaciones elípticas, tales como la de Laplace o la de Poisson, están asociadas con problemas estacionarios o de equilibrio. Por otro lado, algunos de los problemas en los que está involucrado el tiempo como variable independiente derivan en ecuaciones parabólicas. Obviamente, estas plantillas no son las únicas que pueden ser definidas para estos tipos de problemas, de modo que se pueden establecer plantillas más complejas que tengan en cuenta otras formas de resolver el problema como, por ejemplo, multigrid, red-black ordering, etc.

La *Figura 27* muestra la especificación DIP de un patrón `MULTIBLOCK` junto con una tarea "llamada" desde éste que resuelve una ecuación parabólica. El usuario tiene que escribir tanto la especificación del patrón como la especificación de cada una de las tareas computacionales. Para realizar esto último, se escribe el nombre de la plantilla

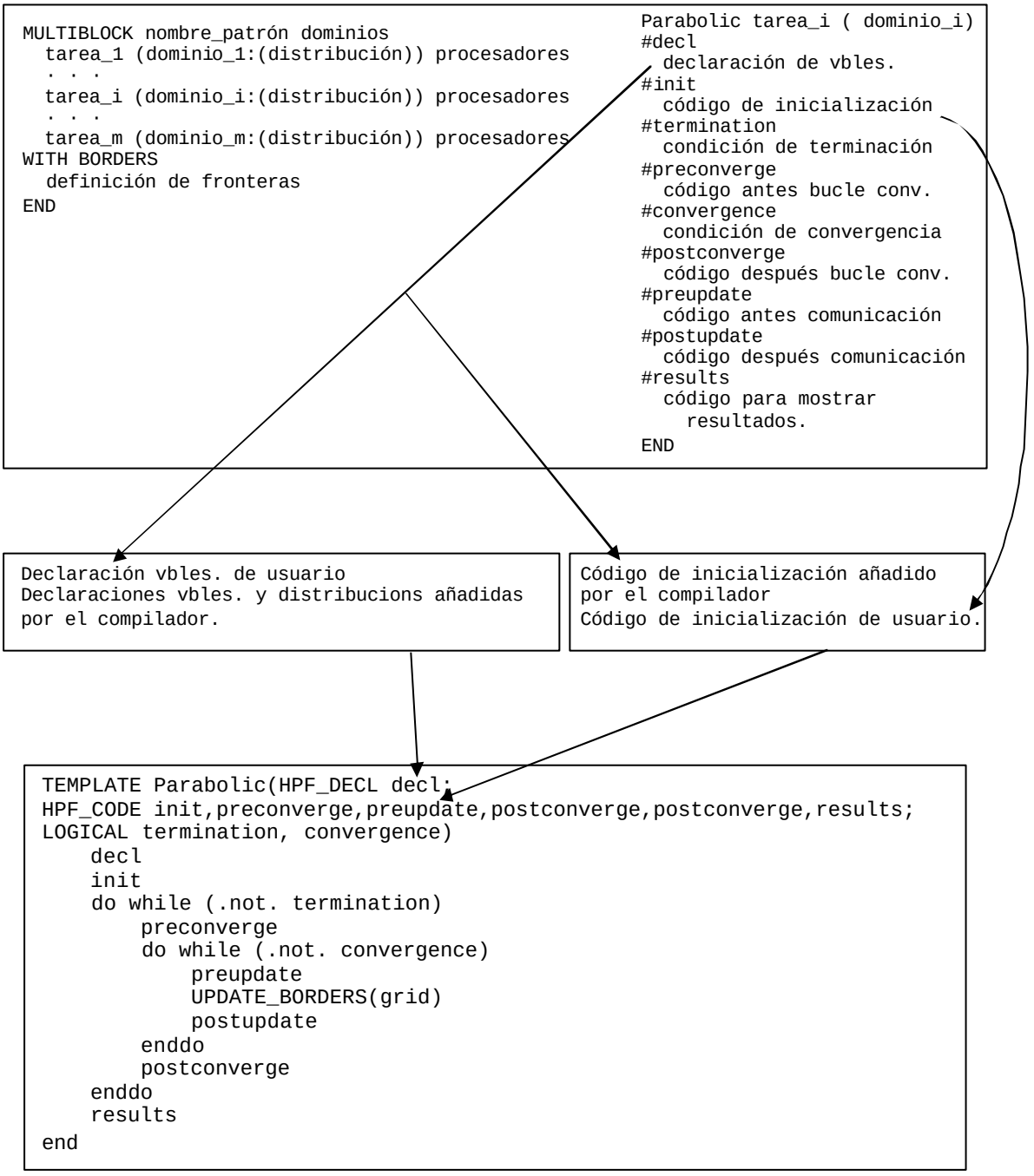


Figura 27. La plantilla Parabolic y su instanciación desde la especificación DIP.

seguido del nombre de la tarea (la que se le ha dado en el patrón). A continuación, se tienen que rellenar las secciones de código necesarias de la tarea utilizando código HPF. Para separar cada sección, se ha utilizado el carácter "#" seguido del nombre de la sección declarada en la plantilla.

La figura también muestra el proceso por el cual se crea la instancia de la plantilla correspondiente. Esta plantilla consta de una sección donde el usuario puede declarar sus variables y constantes, denominada `decl`. A continuación vendrá la parte destinada a la inicialización de estas variables y a algunos cálculos iniciales, `init`. Seguidamente hay dos bucles anidados. El más externo se utiliza para avanzar en el tiempo, mientras que el más interno itera hasta que se alcancen las condiciones de convergencia entre dominios en cada instante de tiempo. La terminación de ambos bucles viene dada por el usuario en la especificación de tarea mediante las expresiones lógicas `termination` y `convergence`. Antes del bucle de convergencia, existe una sección denominada `preconverge` donde el usuario puede escribir el código necesario antes de entrar en el bucle interno. Esta sección se utiliza generalmente para preparar las ecuaciones que se imponen en las fronteras entre dominios. Dentro del bucle de convergencia, la sección `preupdate` se usa habitualmente para realizar los cálculos locales a cada dominio. Esta sección puede o no ser a su vez un bucle dependiendo del método numérico que se esté empleando (iterativo o directo). La actualización de las fronteras entre tareas se realiza mediante la instrucción `UPDATE_BORDERS`, la cual recibe como argumento el `GRID` declarado en la especificación de tarea y que esté asociado al dominio de la tarea. En la sección `postupdate` se realizan los cálculos una vez han llegado los valores de otros dominios, como puede ser el cálculo del error cometido para comprobar si se ha alcanzado la convergencia. La sección `postupdate` se puede utilizar para eliminar ciertos errores numéricos una vez que se ha alcanzado la convergencia entre dominios. Por último, los resultados de la computación se pueden mostrar mediante el código introducido en la sección `results`.

La *instanciación* la realiza el compilador. Éste añade alguna información necesaria tanto a las declaraciones de las variables de usuario como a su inicialización antes de *instanciar* la plantilla. Partiendo de la definición de un dominio, de su distribución especificada por el usuario en el patrón `Multiblock` y de las variables declaradas en la especificación de tarea, el compilador genera lo siguiente:

- La dimensionalidad del dominio (1D, 2D, etc.).
- La dimensionalidad del `GRID` asociado al dominio.
- La distribución del `GRID`.
- La creación dinámica del `GRID`.

El compilador genera automáticamente la dimensionalidad de las variables con atributo `GRID`, a partir de la definición que se ha hecho en el patrón. Por ello, el usuario no necesita especificar la dimensionalidad de estas variables sino que le basta con decir que una variable con atributo `GRID` está asociada a un dominio. Esto se puede realizar como en el ejemplo siguiente:

```
real, GRID(u) :: g, g_old
```

siendo `u` una variable de tipo dominio. De esta forma, se declaran dos variables con atributo `GRID` y tipo base `real` cuya dimensionalidad vendrá marcada por la que se defina en el patrón correspondiente. El compilador generará la declaración adecuada (`GRID1D`, `GRID2D`, etc.), su distribución, su creación dinámica y las instrucciones de comunicación de fronteras para la primera variable declarada con atributo `GRID` en la especificación para un dominio (en el ejemplo sólo para `g` pero no para `g_old`).

Esto último se hace así para permitir la declaración en una misma tarea de varias variables con atributo `GRID` asociadas al mismo dominio. De esta forma, el compilador sólo genera las instrucciones de comunicación de frontera de un `GRID` para cada dominio. Por tanto, en el caso de esta plantilla, se realiza la llamada a la instrucción `UPDATE_BORDERS` únicamente para la variable `g`. Si se desea realizar comunicación de fronteras para otro `GRID`, es necesario declarar otro dominio.

Es de resaltar que si la misma tarea se llama desde el patrón `Multiblock` utilizando diferentes distribuciones de datos, la especificación de tarea puede ser la misma y, en ese caso, el compilador será el encargado de generar instancias diferentes partiendo de la misma plantilla. De esta forma, el compilador de HPF conoce la distribución de las variables en tiempo de compilación y se puede generar código más eficiente. Por otro lado, no es necesario desarrollar subrutinas distintas cuando se quiere

resolver el mismo problema pero con distribuciones distintas de dominios, como ocurría en el ejemplo de la *Figura 14* de la página 75. De acuerdo con nuestros objetivos, se consigue, así, la reutilización de código sin que se vea afectada la eficiencia.

Las plantillas mostradas aquí son independientes de la dimensionalidad del problema. Este aspecto es muy importante cuando se quiere incrementar la precisión de la solución de un problema que ha sido programado, por ejemplo, con dos dimensiones y se quiere pasar a tres. Tanto la plantilla como gran parte de la especificación de la tarea puede ser reutilizada. Obviamente, esto no tiene porqué ser así en algunas plantillas más complejas definidas por el usuario que requieran unos patrones de comunicación en los que la dimensionalidad del problema sí sea determinante en la definición de su plantilla correspondiente.

Tanto las instanciaciones distintas en función de la distribución de los datos como la independencia de la dimensionalidad, no son características únicas de las plantillas `Multiblock`, sino que las plantillas para el patrón `PIPE` también sacan ventaja de estas características como se verá más adelante.

La plantilla `Elliptic` que hemos definido (*Figura 28*) solamente requiere un bucle puesto que, en este caso, se trata de problemas estacionarios. De esta forma, el usuario no tiene que proporcionar las secciones de código `termination`, `preconverge` ni `postconverge`. El proceso de instanciación es similar al mostrado para la plantilla `Parabolic`.

```
TEMPLATE Elliptic (HPF_DECL decl;
                   HPF_CODE init,preupdate, postupdate,results;
                   LOGICAL convergence)
    decl
    init
    do while (.not. convergence)
        preupdate
        UPDATE_BORDERS(grid)
        postupdate
    enddo
    results
end
```

Figura 28. La plantilla Elliptic.

4.1.3.2 Plantillas Pipeline.

El patrón `PIPE` establece una cadena de etapas con flujo de datos, las cuales consumen y producen un flujo de datos de entrada y salida, respectivamente. Así, la plantilla de implementación de una etapa genérica se organiza como un bucle que recibe elementos de un flujo de entrada, ejecuta algún código y envía los datos resultantes al flujo de salida.

Se han considerado dos casos diferentes para manejar un aspecto muy importante en este tipo de aplicaciones, como es el tamaño del flujo de datos, es decir, el número de elementos de que consta. En el primer caso, se asume que todas las etapas

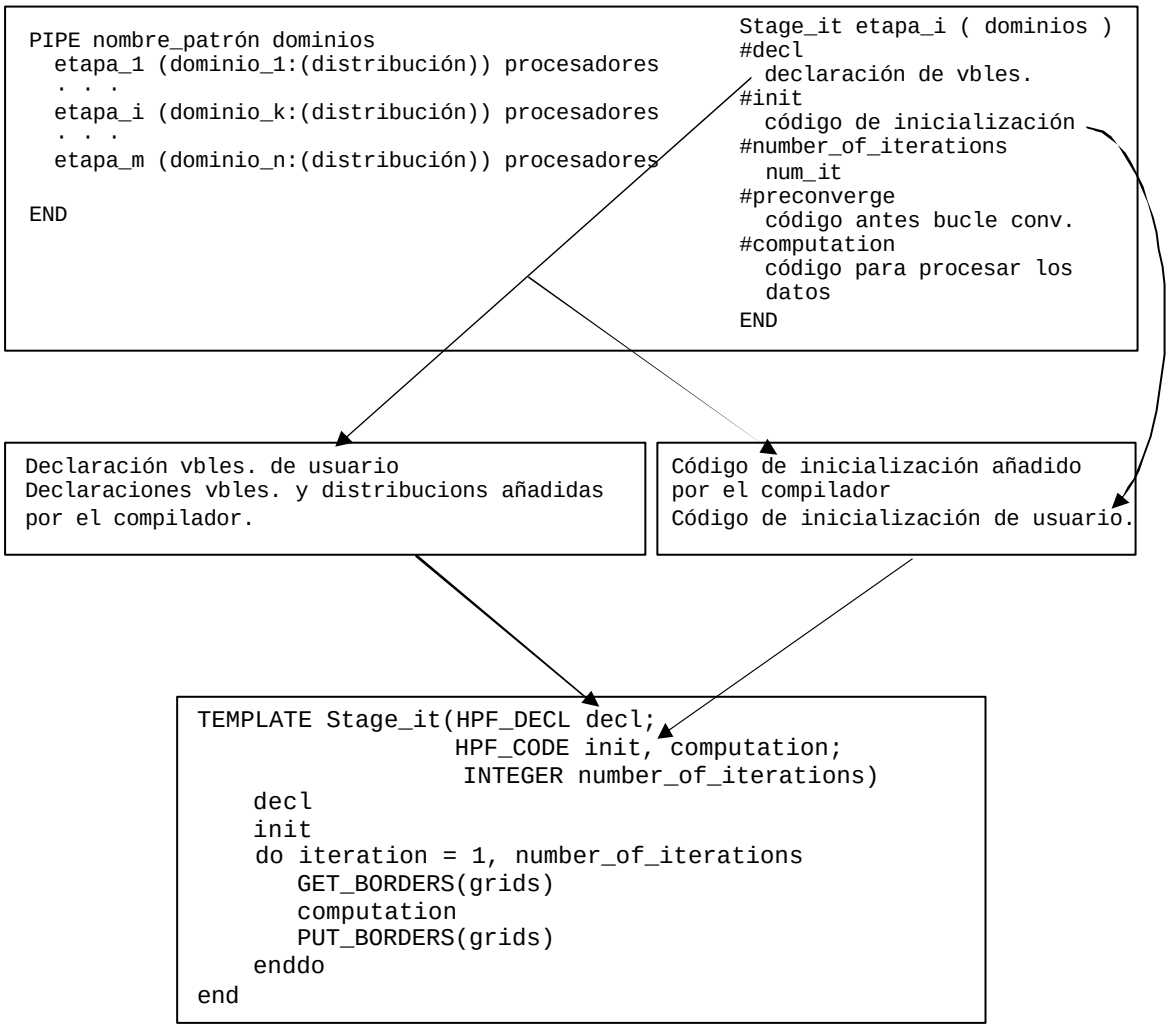


Figura 29. La plantilla de etapa de encauzamiento y su instanciación desde la especificación DIP cuando la longitud del flujo de datos es conocida.

conocen el tamaño del flujo mediante el número de iteraciones que ha de realizar el bucle principal de la etapa y que es establecido por el usuario. En el segundo caso, las etapas no lo conocen, por lo que se ha establecido un protocolo de terminación asociando una marca de terminación a cada elemento del flujo.

La *Figura 29* muestra la plantilla de implementación donde se ha considerado el primer caso. También se puede observar la especificación DIP para el patrón `PIPE` y para una tarea "llamada" desde éste junto con el proceso de instanciación. Nótese que, de esta forma, no se necesitan plantillas especiales para las etapas primera y última del encauzamiento, ya que el compilador conoce, por la información descrita en el patrón y en la especificación de la tarea, cuantas instrucciones `PUT_BORDERS` y `GET_BORDERS` tienen que ser introducidas en el proceso de instanciación.

El usuario ha de escribir el patrón PIPE y la especificación de cada una de las etapas. La especificación comienza con el nombre de la plantilla que se desea usar. En este caso, puesto que el número de iteraciones es conocido, hemos denominado a la plantilla `Stage_it`. A continuación vendrá el nombre de la etapa o tarea y el dominio o dominios que va a utilizar. El usuario rellenará las secciones de código `decl`, `init`, `number_of_iterations` y `computation`. En la primera de ellas se declaran todas las variables que el usuario necesite, entre ellas, los dominios y `GRIDs` asociados. En la sección `init`, se puede introducir código que se tenga que ejecutar una sola vez, como puede observarse en la plantilla de implementación. En la sección `number_of_iterations`, el usuario define, mediante una expresión entera, el número de elementos del flujo de datos, o dicho de otra forma, las iteraciones que debe realizar el bucle. Finalmente, en la sección `computation`, se establece el cómputo a realizar con cada elemento recibido.

Teniendo en cuenta la posición de la etapa dentro del patrón (si es la primera o la última en declarar un dominio) y las variables declaradas en la zona de especificación, el compilador generará las instrucciones `PUT_BORDERS` y `GET_BORDERS` necesarias, pudiendo incluso, no necesitarse alguna de las dos.

La *Figura 30* muestra la plantilla de implementación en el caso de que el tamaño del flujo no sea conocido por las etapas. En este caso, la especificación de tarea no contiene la sección `number_of_iterations`. El bucle se repite hasta que se recibe una marca con el valor especial `end_of_stream`. Esta marca va asociada a cada uno de los elementos del flujo de datos y tendrá el valor `end_of_stream` cuando la etapa anterior termine de enviar datos. Como se observa en la figura, el primer paso será el de recibir la marca y si ésta no indica el final del flujo, se recibirán los datos, se realizará la computación y se enviará la marca junto con los datos procesados. Una vez recibida la marca de fin de flujo, se enviará a la etapa siguiente. Todo este proceso es transparente al programador que solamente ha de especificar la declaración de sus variables, el código de inicialización y el de computación.

```
TEMPLATE Stage_mark(HPF_DECL decl;
                    HPF_CODE init,computation)
    decl
    init
    <receive mark>
    do while (.not. <end_of_stream>)
        GET_BORDERS(grids)
        computation
        <send mark>
        PUT_BORDERS(grids)
        <receive mark>
    enddo
    <send end_of_stream>
end
```

Figura 30. Plantilla de implementación para una etapa de un encauzamiento donde no se conoce el tamaño del flujo de datos.

De esta forma, la primera etapa del encauzamiento (o aquellas en las que sólo aparezcan dominios que no se han usado en etapas anteriores) es la que determina la longitud del flujo de datos, por lo que se necesita una plantilla especial para su implementación, como se muestra en la *Figura 31*. El programador debe proporcionar una expresión lógica en la especificación de tarea con el propósito de controlar el bucle de la primera etapa (el fin de un fichero, una señal recibida, etc.).

```
TEMPLATE First_stage (HPF_DECL decl;
                      HPF_CODE init,computation;
                      LOGICAL termination)
    decl
    init
    do while (.not. termination)
        computation
        <send mark>
        PUT_BORDERS(grids)
    enddo
    <send end_of_stream>
end
```

Figura 31. Plantilla de implementación para la primera etapa de un encauzamiento donde no se conoce el tamaño del flujo de datos.

Finalmente, cuando se usa la directiva `REPLICATE`, se ha considerado una política de envío cíclico o de Round-Robin. De esta forma, cuando se proporciona el número de iteraciones, el compilador decide de forma automática cuantas iteraciones deberán realizar cada una de las instancias de la etapa replicada. Por otro lado, cuando el tamaño del flujo no es conocido, el compilador introduce código especial para la etapa anterior (*emisor*) y posterior (*colector*) a la etapa replicada. Así, cuando el emisor envía la marca `end_of_stream` al final de su ejecución, debe mandar realmente tantas marcas como instancias haya en la etapa replicada. De la misma forma, el colector debe recibir la marca `end_of_stream` de todas las instancias antes de enviar su propia marca.

4.2 Ejemplos.

A continuación se muestran diversos ejemplos de programación que reflejan la adecuación del modelo para la resolución de este tipo de problemas. Algunos de ellos ya han sido resueltos previamente en capítulos anteriores, con lo cual se puede constatar la mejoría obtenida en cuanto a expresividad se refiere. Estos ejemplos pueden servir también para entender mejor las características introducidas sin tener que centrarnos en la descripción de nuevas aplicaciones. A parte de estos, se presentan problemas un poco más complejos cuya resolución sin las características de DIP hubiese sido más engorrosa y que nos sirven para comprender algunos conceptos nuevos de nuestra aproximación.

4.2.1 El Patrón MULTIBLOCK para el Ejemplo de Jacobi.

El Programa 27 muestra el patrón para el ejemplo de Jacobi con paralelismo de datos y tareas tal y como fue descrito en el capítulo 3 (*Figura 14*).

```
1) MULTIBLOCK Jacobi u/1,1,Nxu,Nyu/, v/1,1,Nxv,Nyv/
2)   solve (u:(BLOCK , BLOCK)) ON PROCS (4,4)
3)   solve (v:(BLOCK , *)) ON PROCS (2)
4) WITH BORDERS
5)   u (Nxu,Ny1, Nxu,Ny2 ) <- v (2,1, 2,Nyv)
6)   v (1,1, 1,Nyv ) <- u (Nxu-1,Ny1, Nxu-1, Ny2)
7) END
```

Programa 27. El ejemplo de Jacobi con DIP.

La definición de los dominios se realiza en la línea 1 junto con el tamaño de éstos. La dimensión de los dominios se establece por el número de elementos que aparecen en la definición del dominio (al ser 4 números en este caso, la dimensión es 2). La distribución de los datos se realiza en la llamada a las tareas, líneas 2 y 3, junto con la disposición de los procesadores.

La definición de las fronteras se realiza en las líneas 5 y 6, correspondiéndose con las fronteras de la *Figura 14*.

Las tareas computacionales en este caso se pueden escribir reutilizando el código ya descrito en la solución del problema mediante BCL (Programa 12). Sin embargo, como se describió en el apartado 3.2, es necesario escribir códigos distintos para resolver el mismo problema debido a la distribución distinta de los datos. Para solucionar esto, se puede utilizar la plantilla de implementación descrita anteriormente. De esta forma, el usuario tendría que describir la especificación de la tarea como se muestra en el Programa 28. El compilador DIP se encarga de generar dos instancias de la tarea, una para cada distribución.

```
 1) Elliptic solve (u)
 2) #decl
 3)    domain u
 4)    double precision, GRID(u):: g,g_old
 5)    integer i
 6) #init
 7)    call initGrid (g)
 8)    i=1
 9) #convergence
10)    i > niters
11) #preupdate
12) #postupdate
13)    g_old=g
14)    call computeLocal (g, g_old)
15)    error = computeNorm (g, g_old)
16)    REDUCE (error, maxim)
17)    print *, “Max norm: ”, error
18)    i=i+1
19) #results
20) END
```

Programa 28. Especificación de tarea para el ejemplo de Jacobi.

En la especificación se rellenan las secciones que se corresponden con los argumentos de la plantilla `Elliptic`. Estas secciones serán sustituidas dentro de la plantilla en el proceso de instanciación junto con otro código introducido por el compilador, como es la dimensionalidad de las variables de tipo dominio y `GRID`, la distribución HPF, la creación dinámica de las variables `GRID` y la actualización de la variable `g` entre las tareas. En este ejemplo se dejan algunas secciones vacías (`preupdate` y `results`) puesto que la sencillez del problema hace que no sean necesarias.

En la primera línea se especifica el nombre de la plantilla que se va a utilizar seguida del nombre de la subrutina y de los argumentos que recibe. La variable `u` es de tipo dominio puesto que así se define en la línea 3. La declaración de las variables con atributo `GRID` `g` y `g_old` se realiza asociándoles el dominio `u` en la línea 4, con lo cual, el compilador podrá obtener la dimensión y las fronteras de ambas. Como se explicó anteriormente, sólo se actualizarán las fronteras de la variable `g` al ser la primera en ser declarada, entendiéndose que `g_old` es una variable auxiliar.

En este ejemplo sencillo, se considera que la solución es correcta tras ejecutarse un número predeterminado de iteraciones. Por esta razón, la condición de convergencia

se convierte en una simple comparación entre la variable de control del bucle y la constante `niters`.

Por último, las instrucciones del cuerpo principal del bucle se realizan después de la actualización de la frontera. Obsérvese que la comunicación del error local a cada dominio cometido en cada tarea es realizada mediante la instrucción `REDUCE` de la línea 16, en la cual se asume que la reducción se realiza entre todas las tareas activas.

4.2.2 El Patrón PIPE para la Transformada Rápida de Fourier.

Para la transformada rápida de Fourier en dos dimensiones, el patrón necesario es bastante simple (Programa 29). En la línea 1 se declara el tipo de patrón, el nombre del mismo y el dominio que se va a utilizar como canal de comunicación entre las etapas. A continuación se declaran las dos etapas del encauzamiento. En ellas aparece el dominio y la distribución que tendrán los datos asociados al dominio. Por último, se definen los procesadores en los cuales se ejecutará cada etapa.

```
1) PIPE fft2d d/1,1,N,N/
2)   etapa1 (d:(*,BLOCK)) ON PROCS (4)
3)   etapa2 (d:(BLOCK,*)) ON PROCS (4)
4) END
```

Programa 29. El patrón PIPE para el problema FFT2D.

Si se opta por no utilizar plantillas de implementación, las subrutinas `etapa1` y `etapa2` serían las mismas a las mostradas en el Programa 16 y en el Programa 17, respectivamente.

Una solución alternativa que expone la forma en que se pueden anidar patrones `PIPE` se muestra en el Programa 30. En el patrón principal se han introducido dos nuevas etapas, la primera de ellas realiza la lectura de los datos y envía estos al patrón anidado `FFT2D`. Esta primera etapa es ejecutada por un solo procesador por lo que la distribución del dominio aparece `(*,*)`, es decir, sin distribuir. El patrón anidado `FFT2D` recibe como argumento el dominio `d`. Las tareas computacionales implementadas mediante las subrutinas `etapa1` y `etapa2` tienen que modificarse ligeramente ya que las operaciones de entrada y salida se realizan ahora por otras etapas del encauzamiento. De hecho, sólo

hay que modificar una línea en cada subrutina. Así, la línea 7 del Programa 16 para la `etapa1` debe ser ahora una operación de recepción de datos `GET_BORDERS(a)` en lugar de la lectura que se realizaba, mientras que la línea 12 de la subrutina `etapa2` (Programa 17) debe ser ahora `PUT_BORDERS(b)` en lugar de la escritura de los resultados. Finalmente, la etapa `Output` recoge los elementos del flujo de datos enviados por el `PIPE` anidado y escribe sus resultados.

```
1) PIPE FFT2D d/1,1,N,N/
2)   etapa1 (d: (*,BLOCK)) ON PROCS (4)
3)   etapa2 (d: (BLOCK,*)) ON PROCS (4)
4) END
5) PIPE Solucion_Alternativa d/1,1,N,N/
6)   Input (d:(*,*)) ON PROCS(1)
7)   FFT2D (d)
8)   Output (d:(*,*)) ON PROCS (1)
9) END
```

Programa 30. Solución del FFT2D con anidamiento de patrones.

Aunque la primera versión es más corta, la segunda establece un patrón `PIPE FFT2D` más útil ya que puede ser reutilizado en aplicaciones más complejas tales como la transformada rápida de Fourier en 2 dimensiones con histograma (`FFT2D_Hist`) o la convolución de dos fuentes de imágenes. Ambas son técnicas estándares utilizadas para extraer información de figuras en imágenes.

El Programa 31 muestra el código para el problema `FFT2D_Hist`. Su esquema es parecido al mostrado en la *Figura 26*. En primer lugar, se realiza la transformada de Fourier como se ha explicado previamente mediante el patrón anidado `Flujo` que reutiliza el patrón del ejemplo anterior `FFT2D`. A continuación, se realiza el histograma. Como esta última etapa no ofrece buenos resultados cuando es paralelizada, se ejecuta en un solo procesador, pero se replica 4 veces. De esta forma, cada una de las réplicas realiza el histograma de una cuarta parte de los elementos del flujo de datos de forma secuencial, lo cual puede incrementar notablemente la eficiencia de la solución resultante [Gross y otros 94].

```
 1) PIPE FFT2D d/1,1,N,N/
 2)   etapa1 (d:(*,BLOCK)) ON PROCS (2)
 3)   etapa2 (d:(BLOCK,*)) ON PROCS (2)
 4) END
 5) PIPE Flujo a/1,1,N,N/
 6)   InputImagen(a:(*,*)) ON PROCS(1)
 7)   FFT2D(a)
 8) END
 9) PIPE FFT2D_Hist a/1,1,N,N/
10)   Flujo (a)
11)   REPLICATE (4) hist(a:(*,*)) ON PROCS (1)
12)   Output (a:(*,*)) ON PROCS (1)
13) END
```

Programa 31. FFT2D con histograma replicado.

El algoritmo de convolución de dos fuentes de imágenes se representa gráficamente en la *Figura 32*. Esta aplicación conlleva dos `FFT2D`, una multiplicación de matrices elemento a elemento y una `FFT2D` inversa. Su aplicación se realiza sobre dos flujos de imágenes para generar un solo flujo de salida.

Este algoritmo se puede programar en DIP como se muestra en el Programa 32. En el patrón principal `Convolucion2d`, se declaran los dominios `a`, `b` y `c` con el mismo tamaño. Los dos primeros se utilizan para pasárselos al patrón anidado `Flujo`.

Los datos obtenidos de la transformada de ambas imágenes son enviados a la etapa `FFT2D_1` que también es un patrón anidado, constando de una primera etapa,

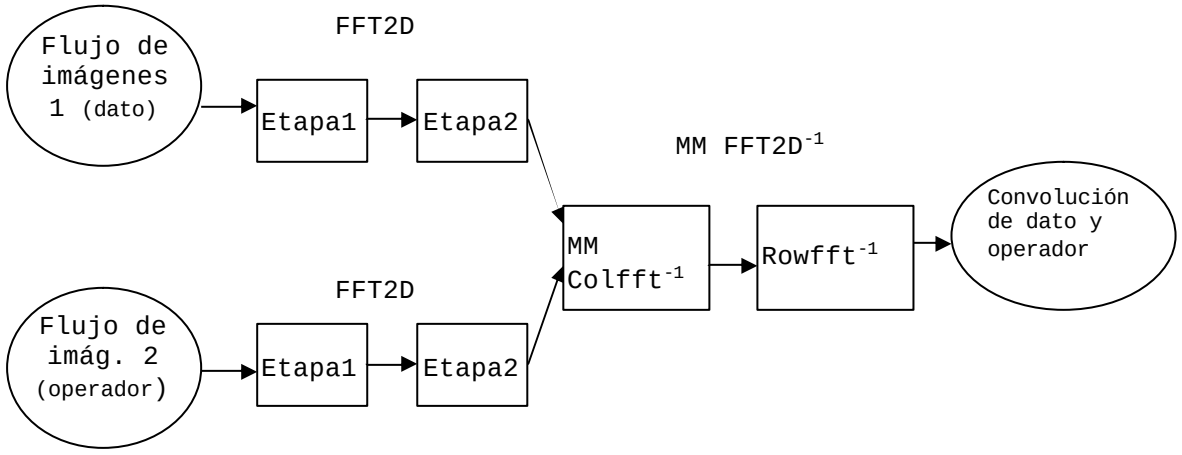


Figura 32. Esquema del algoritmo de Convolución.

`MM_col_fft_1` que realiza la multiplicación de los datos que han llegado elemento a elemento y la transformada inversa por columnas del resultado. A continuación, envía los datos a su segunda etapa que termina la transformada inversa por filas. Por último, la etapa `Output` del patrón principal escribe los resultados.

```
 1) PIPE FFT2D d/1,1,N,N/
 2)   etapa1 (d:(*,BLOCK)) ON PROCS (2)
 3)   etapa2 (d:(BLOCK,*)) ON PROCS (2)
 4) END
 5) PIPE FFT2D_1 a/1,1,N,N/, b/1,1,N,N/ c/1,1,N,N/
 6)   MM_col_fft_1 (a:(*,BLOCK),b:(*,BLOCK),c:(*,BLOCK) ) ON PROCS (2)
 7)   row_fft_1 (c: (BLOCK,*)) ON PROCS (2)
 8) END
 9) PIPE Flujo a/1,1,N,N/
10)   InputImagen(a: (*,*)) ON PROCS(1)
11)   FFT2D(a)
12) END
13) PIPE Convolucion2D a/1,1,N,N/, b/1,1,N,N/, c/1,1,N,N/
14)   Flujo (a)
15)   Flujo (b)
16)   FFT2D_1 (a,b,c)
17)   Output (c:(*,*)) ON PROCS (1)
18) END
```

Programa 32. Convolución de dos flujos de imágenes con DIP y 9 tareas (15 procesadores).

Como puede observarse, las instrucciones 14 y 15 llaman dos veces al mismo patrón y, además, se ejecutan simultáneamente puesto que la segunda etapa no recibe datos de la primera. Esto se puede expresar de forma más clara sustituyendo ambas líneas por la línea 14 del Programa 33. Como las etapas posteriores a un `REPLICATE` obtienen los datos en Round-Robin, la línea 15 obtiene dos veces el mismo dominio (en este caso, `a`) de forma que cuando se hagan los `GET` correspondientes, cada vez se obtienen los datos de una réplica distinta, obteniéndose el mismo resultado que en el programa anterior.

```
 1) PIPE FFT2D d/1,1,N,N/
 2)   etapa1 (d:(*,BLOCK)) ON PROCS (2)
 3)   etapa2 (d:(BLOCK,*)) ON PROCS (2)
 4) END
 5) PIPE FFT2D_1 a/1,1,N,N/, b/1,1,N,N/ c/1,1,N,N/
 6)   MM_col_fft_1 (a:(*,BLOCK),b:(*,BLOCK),c:(*,BLOCK) ) ON PROCS (2)
 7)   row_fft_1 (c: (BLOCK,*)) ON PROCS (2)
 8) END
 9) PIPE Flujo a/1,1,N,N/
10)   InputImagen(a: (*,*)) ON PROCS(1)
11)   FFT2D(a)
12) END
13) PIPE Convolution2D a/1,1,N,N/, c/1,1,N,N/
14)   REPLICATE (2) Flujo (a)
15)   FFT2D_1 (a,a,c)
16)   Output (c: (*,*)) ON PROCS (1)
17) END
```

Programa 33. Convolución de dos flujos de imágenes usando la directiva REPLICATE.

4.2.3 La aplicación NPB-FT.

La transformada rápida de Fourier para un problema de difusión se resolvió utilizando distintos esquemas de computación utilizando BCL en el apartado 3.4. En el Programa 34 se muestra como se puede implementar el último esquema empleado, el mostrado en la *Figura 22*, mediante DIP. Es de destacar la sencillez del patrón en comparación con el proceso coordinador de la solución equivalente que se mostró en el Programa 22 de la página 90.

En el patrón principal (`NPB_FT_3`) se declaran dos etapas, la primera de ellas es la que obtiene los datos iniciales, realiza la transformada y evoluciona en el tiempo. La segunda etapa es la que realiza la transformada inversa (`FFT_Inversa`) mediante una estructura `PIPE` anidada y que está replicada con objeto de sacar partido a otro nivel de paralelismo, como se explicó anteriormente. De esta forma, cada elemento generado por la primera etapa irá a cada una de las instancias del patrón replicado.

El patrón anidado consta a su vez de dos etapas, la primera de ellas realiza la transformada inversa sobre el eje Z y manda sus resultados a la etapa que termina el cálculo con las transformadas sobre los ejes X e Y.

```
1) PIPE FFT_Inversa a/1,1,1,N,N,N/
2)   etapa_par (a:(*,BLOCK,*) ) ON PROCS (P2)
3)   etapa_impar (a:(*,*,BLOCK)) ON PROCS (P3)
4) END
5) PIPE NPB_FT_3 a/1,1,1,N,N,N/
6)   etapa1(a:(*,BLOCK,*)) ON PROCS (P1)
7)   REPLICATE (R) FFT_Inversa(a)
8) END
```

Programa 34. El patrón DIP para el problema NPB-FT con replicación de la transformada inversa.

Las tareas computacionales se pueden implementar de forma sencilla y elegante utilizando las plantillas de implementación para la estructura de encauzamiento. El Programa 35 muestra la especificación de la tarea `etapa1` mediante la plantilla `Stage_it` (puesto que el número de iteraciones es conocido por todas las etapas).

```
 1) Stage_it etapa1(d)
 2) #decl
 3)   domain d
 4)   complex, GRID(d) :: w,v,u
 5)   !hpf$ DISTRIBUTE (*,*,BLOCK) :: u
 6) #init
 7)    call cond_iniciales (u)
 8)    call fft3d (u,v)
 9) #number_of_iterations
10)    T
11) #computation
12)   call evoluciona (v,w,iteration)
13) end
```

Programa 35. La especificación de la primera tarea del problema NPB-FT.

En la parte de declaración se definen las variables con atributo `GRID` `w`, `u` y `v` que tienen asociado el dominio, `d`, que es pasado como parámetro. El compilador sólo generará instrucciones de comunicación para la variable `w`, entendiendo que las variables `v` y `u` son variables auxiliares para la computación. Esta última variable tiene una distribución distinta a la especificada en el patrón (línea 5), con lo cual el compilador tendrá que detectar que ya hay una declaración de distribución para esta variable y no utilizar la declarada en el patrón. Para las otras dos variables, el compilador sí generará su distribución correspondiente.

Los cálculos iniciales, es decir, aquellos que se realizan antes del bucle principal del flujo de datos, son especificados en las líneas 7 y 8. En la línea 12 se llama a la subrutina que genera cada elemento del flujo de datos. Como para este cálculo es necesario conocer la iteración en la que se encuentra, la variable definida en la plantilla `iteration` (ver *Figura 29*) también se pasa como un argumento a esta subrutina.

El compilador utiliza también la información del patrón para generar la instancia que implementa la tarea computacional. Puesto que la tarea `etapa1`, es la primera en la que aparece el dominio `a`, se asume que ésta es la que genera el flujo de datos asociado a ese dominio, con lo cual solamente se introduce la instrucción `PUT_BORDERS(w)` como última instrucción del bucle, no apareciendo instrucción de recepción de datos.

La segunda etapa del patrón principal replica el `PIPE` anidado `FFT_inversa`. Este patrón consta de dos etapas que, por mantener los nombres dados en la solución expuesta en el capítulo 3, se han denominado `etapa_par` y `etapa_impar`. La especificación de la primera de ellas se muestra en el Programa 36. Al igual que en la etapa anterior y en la solución sin las características de DIP, se han omitido algunos detalles de su implementación (como son los cálculos iniciales que son necesarios para realizar la transformada más eficientemente).

```
 1) Stage_it etapa_par(d)
 2) #decl
 3)    domain d
 4)    complex, GRID(d) :: w
 5) #init
 6) #number_of_iterations
 7)    T
 8) #computation
 9)   call fft3d_inversa_ejeZ(w)
10) end
```

Programa 36. La especificación de la etapa par del problema NPB-FT.

En este caso, sólo es necesario declarar una variable con atributo `GRID` que tiene asociado el dominio pasado como argumento (línea 4). El número de iteraciones es declarado en la línea 7, aunque el compilador modificará este valor en función del número de réplicas declaradas en la línea 8 del Programa 34, de forma que cada réplica sólo recibirá una parte del flujo de datos. El cálculo a realizar consiste en las

transformadas inversas unidimensionales a lo largo del eje Z. Puesto que la `etapa_par` es una etapa intermedia en el patrón, el compilador generará una instrucción `GET_BORDERS(w)` al principio del bucle y una instrucción `PUT_BORDERS(w)` al final.

Finalmente, la etapa `etapa_impar` se especifica según se muestra en el Programa 37. En esta tarea se termina el cálculo de la transformada inversa y se comprueban los resultados (líneas 9 y 10). El número de iteraciones a realizar también es modificado por el compilador en función del número de réplicas. Al ser la última etapa del patrón, el compilador sólo insertará una instrucción para recibir datos pero no para enviarlos.

```
 1) Stage_it etapa_impar(d)
 2) #decl
 3)    domain d
 4)    complex, GRID(d) :: u
 5) #init
 6) #number_of_iterations
 7)    T
 8) #computation
 9)   call fft3d_inversa_ejesXY(u)
10)   call check_sum(u)
11) end
```

Programa 37. La especificación de la etapa impar del problema NPB-FT.

4.3 Conclusiones.

En este capítulo se han presentado las nuevas características de alto nivel que permiten la integración del paralelismo de datos y tareas mediante el uso de patrones. De esta forma, se permite la definición de forma concisa y clara de estructuras de computación complejas, como las presentadas en los últimos ejemplos. Los patrones permiten describir la parte de coordinación de una aplicación a alto nivel, de modo que el programador puede cambiar de forma drástica la estructura de la computación haciendo unas pocas modificaciones en el patrón. De esta manera, se pueden probar distintas alternativas de implementación para obtener la más eficiente.

El uso de dominios permite que los tipos de los datos que se deben comunicar no tengan que ser descritos a nivel de coordinación, lo cual incrementa las posibilidades de

reutilización del código. Por otra parte, el uso de los patrones `PIPE` y sus anidamientos, permiten construir aplicaciones complejas de forma estructurada.

Las plantillas de implementación permiten al programador utilizar un nivel más de abstracción, puesto que sólo ha de preocuparse de rellenar las distintas secciones de código. Además, la utilización de estas plantillas en las estructuras encauzadas permite que el usuario pueda cambiar el orden de ejecución de las etapas, de forma que es el compilador el que se encarga de generar las comunicaciones necesarias entre dichas etapas, así como de establecer los protocolos de comunicación cuando se decide replicar una de ellas.

Por último, la construcción de plantillas es muy sencilla, con lo cual el usuario puede crear plantillas nuevas que se ajusten al tipo de aplicación con la que está trabajando.

Para poder comprobar la eficiencia de los ejemplos descritos, se ha construido un compilador, cuya implementación se describe en el capítulo siguiente.

Capítulo 5. Implementación

En los capítulos anteriores se ha mostrado la expresividad del lenguaje BCL, así como su adecuación al tipo de problemas que se tratan en esta tesis. Sin embargo, dada la gran carga computacional asociada a la mayoría de los problemas de este tipo, este lenguaje tiene como objetivo primordial la eficiencia de las aplicaciones resultantes. Por esta razón, se han diseñado unos prototipos del modelo para comprobar la eficiencia de las soluciones obtenidas como resultado de aplicar nuestra aproximación y cuyos resultados se muestran en el capítulo 6.

Los prototipos diseñados e implementados no pretenden ser una versión final del modelo entero, sino unas aproximaciones iniciales y simples de la base fundamental del mismo, con objeto de que nos permitan evaluarlo con respecto a los objetivos de eficiencia planteados al inicio del trabajo y que, además, nos permitan adquirir experiencia tanto a nivel de implementación como desde el punto de vista del diseño de aplicaciones utilizando el modelo, especialmente en cuanto a la integración del paralelismo de datos y tareas se refiere. Obviamente, estos prototipos iniciales deben ser al mismo tiempo fácilmente extensibles con el propósito de incorporar de una manera sencilla todas las características que posee el modelo.

Una estrategia común en la implementación de lenguajes de coordinación es la traducción fuente a fuente llevada a cabo sobre el código resultante de integrar el modelo en un lenguaje base o *"host"* y la construcción de bibliotecas adecuadas con el objeto de enlazarlas con el resultado de esas transformaciones y obtener la aplicación final a ejecutar. Esta es la forma en que se ha implementado el prototipo del compilador de BCL. Para su explicación se detallan primero las características de éste sin el uso de patrones ni plantillas para luego incorporar las características de DIP.

5.1 BCL.

Dada la experiencia previa de nuestro grupo en la implementación de un lenguaje de coordinación [Rubio 98] y con el propósito de familiarizarnos con la problemática de la integración del paralelismo de datos y tareas, se decidió realizar la implementación en dos fases:

1. Determinación del modo de conseguir la integración del paralelismo de datos y tareas.

2. Realización del traductor de BCL y de la biblioteca que implementa las distintas características de BCL (de ahora en adelante BCLIB).

5.1.1 Integración del paralelismo de datos y tareas.

Se hicieron diversas pruebas en la primera fase con objeto de determinar, en primer lugar, si era posible la integración y, en segundo lugar, comprobar si el resultado era eficiente. Puesto que el único compilador de HPF del que se disponía en un principio era el de Digital, se pensó en su utilización como primera aproximación. De esta forma, se tendrían una serie de programas distintos compilados con HPF y ejecutándose de forma simultánea. Para la creación de tareas y la comunicación entre ellas se eligió PVM, dada su adecuación al problema y nuestra experiencia previa con esta biblioteca de paso de mensajes [Álvarez, Díaz, Llopis, Pastrana, Rus, Soler y Troya 96].

Los resultados fueron satisfactorios cuando se realizaron pruebas en las que la comunicación entre tareas era limitada, como es el caso de los problemas de descomposición de dominios en los que la frontera se establecía como unos cuantos datos entre unos pocos procesadores. Sin embargo, cuando se probaron aplicaciones en los que la comunicación es mayor, el resultado pasó a ser decepcionante. Esto fue debido, por un lado, a la arquitectura del sistema con el que estábamos trabajando y, por otro, a la versión de PVM de la que se disponía al comenzar la implementación, en la cual, todas las comunicaciones de una máquina con otra se realizan a través de un proceso denominado PVM `daemon`. La arquitectura hardware con la que se hicieron las pruebas consta de 4 servidores Alpha cada uno de ellos con 4 procesadores que

comparten memoria. La *Figura 33* muestra cómo un servidor Alpha se comunica con otro servidor. Las comunicaciones de una tarea HPF ejecutándose en 4 procesadores del primer servidor han de pasar forzosamente por el `daemon` del primer servidor que comunica los datos al `daemon` del segundo servidor. Esto provoca un cuello de botella que era el que influía negativamente en la eficiencia de la implementación.

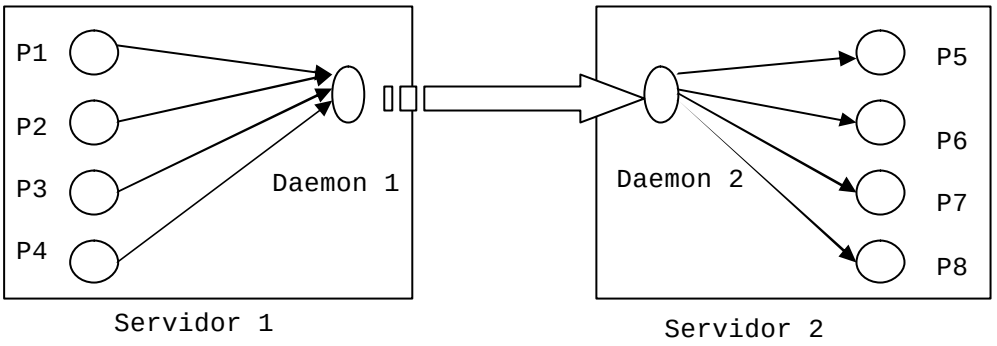


Figura 33. Comunicación entre dos tareas HPF en dos servidores Alpha distintos mediante PVM.

Para resolver este problema se pensó en la sustitución de la biblioteca PVM por MPI, donde la comunicación entre procesos se hace directamente, sin la necesidad de pasar por un proceso intermedio. Lamentablemente, el compilador de HPF y su sistema en tiempo de ejecución no eran compatibles con MPI. Nos pusimos entonces en contacto con Digital para intentar solucionar el problema. Su respuesta fue la de enviarnos, a modo de evaluación, una nueva versión de su compilador (denominado ahora Compaq HPF) que generaba paso de mensajes mediante la biblioteca MPI en lugar de utilizar una biblioteca propia. Sin embargo, ésta no pudo ser la solución puesto que el sistema de tiempo de ejecución no permitía la ejecución de varias tareas HPF a la vez (como ocurría cuando se usaba PVM), ni tampoco la distribución irregular de arrays, ni la ejecución de distintas subrutinas en conjuntos disjuntos de procesadores. Es decir, no cumplía las extensiones aprobadas de HPF 2.0 para la integración del paralelismo de datos y tareas.

Se decidió entonces buscar otro compilador de HPF que permitiese la ejecución simultánea de tareas, o bien, que permitiese la ejecución de distintos trozos de código por distintos procesadores. El compilador elegido fue Adaptor [Brandes 99a], un compilador de libre disposición que permite ambas cosas. Tras comprobar su

adecuación a nuestros requisitos y tras hacer algunas pruebas iniciales (ver apartado 6.1) se decidió utilizarlo como compilador HPF y utilizar MPI y nuestra biblioteca (BCLIB) para la comunicación entre las tareas.

La adopción de este compilador tiene una gran ventaja con respecto a la utilización de un compilador como el de Compaq, que consiste, obviamente, en la portabilidad. Mientras que el de Compaq es totalmente dependiente de la máquina, Adaptor es completamente transportable y ha sido utilizado en una gran variedad de arquitecturas:

```
CM-5 de Thinking Machines
Intel Paragon
Intel iPSC/860
Meiko CS 2
KSR
IBM SP 1 y SP 2
NEC Cenju-3
NEC SX-4
Parsytec
Cray T3D
Cray T3E
Sistemas Multi-procesador SGI
Arquitecturas Fujitsu VPP
Red de estaciones de trabajo SUN (SUN4 y SOLARIS)
Red de estaciones de trabajo HP UX
Red de estaciones de trabajo DEC ALPHA
Red de estaciones de trabajo IBM Risc
PC's ejecutando LINUX
```

Por otro lado, aunque el compilador de Compaq es ligeramente más eficiente que Adaptor cuando el número de procesadores utilizado es pequeño (pues hace uso de las comunicaciones mediante memoria compartida), Adaptor escala mejor que el compilador de Compaq, de modo que, para un número mayor de procesadores, se obtienen mejores resultados.

5.1.2 Traductor.

La *Figura 34* muestra el esquema de traducción de código BCL para conseguir un ejecutable paralelo. En una primera fase, el traductor fuente a fuente de BCL genera código HPF donde se saca partido a las características de las extensiones aprobadas de HPF 2.0 para la ejecución de subrutinas por conjuntos disjuntos de procesadores. En este código también se harán llamadas a la biblioteca BCLIB. Este programa tiene que ser procesado por el compilador de Adaptor que utiliza su biblioteca DALIB

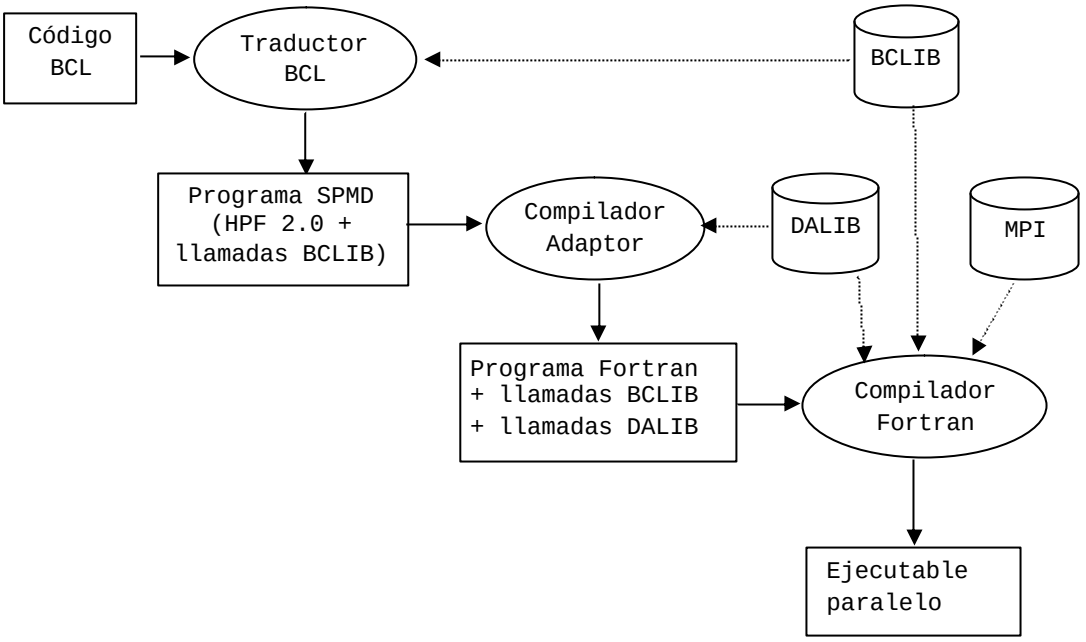


Figura 34. Obtención del ejecutable a partir de código BCL.

(*Distributed Array Library*). Finalmente, a partir del código resultante, el compilador de Fortran es el que se encarga de producir el ejecutable enlazando las bibliotecas BCLIB, DALIB y MPI. Esta última es con la que se implementan las comunicaciones tanto entre tareas como dentro de éstas (Adaptor utiliza también MPI para el paso de mensajes generado al compilar el código HPF).

Puesto que el objetivo principal al realizar el prototipo de traductor es el de evaluar la eficiencia del modelo, no se han implementado todas las características de BCL. Nos hemos centrado en las que nos permitan la codificación de los procesos coordinadores y la comunicación eficiente entre los procesos trabajadores (bien sea por las fronteras definidas o por las variables de convergencia). Para facilitar la tarea al traductor, se han añadido algunas "ayudas" de forma que sea más sencillo distinguir entre las instrucciones Fortran que no deben modificarse y las primitivas que han de traducirse con la ayuda de la biblioteca BCLIB. Estas ayudas consisten en comenzar cada línea donde hay alguna primitiva BCL con el carácter @. De esta forma, al traductor le es más fácil identificarlas y, por tanto, traducirlas.

A continuación se describe la forma en que se han implementado las características principales de BCL.

Los dominios se implementan mediante registros que contienen una valiosa información. Aparte del campo `region`, que contiene la región del dominio, existen otros campos muy importantes como son el número de fronteras de entrada y de salida que tiene un dominio y la información relativa a cada frontera del dominio, la cual se almacena en dos arrays (de entrada y de salida) cuyo tipo base es el tipo de dato frontera.

El tipo frontera es también un registro que contiene, entre otras, la siguiente información: Tipo de frontera (la definida en el coordinador mediante la etiqueta correspondiente), número de procesos a los que hay que enviarle la frontera, identificador (MPI) de proceso de cada uno de ellos y región de la frontera que hay que enviar a cada procesador. Cuando la frontera es de entrada, también hay que almacenar la función a ejecutar cuando lleguen los datos.

Cuando el traductor se encuentra la definición de un dominio, almacena en una tabla interna su nombre y la distribución de cada una de sus dimensiones y crea, además, un array para almacenar los identificadores de procesos en los que se resolverá ese dominio. Cuando se encuentra una frontera, además de rellenar la información correspondiente, almacena en su tabla interna los nombres de los dominios con los que tiene frontera y el nombre del array de los procesadores donde se ejecutarán esos otros dominios. Cuando se encuentra con la instrucción `create` ya puede rellenar el array con los identificadores de los procesos y rellena también los arrays de los demás dominios que tengan frontera en común con él.

Aunque el modelo refleja que el coordinador es un proceso secuencial que se encarga de crear los procesos trabajadores, la implementación se ha realizado de otra forma. Todos los procesadores ejecutarán el proceso coordinador hasta la aparición de la primera instrucción `CREATE`. La creación de procesos se realiza mediante la directiva de la extensión de HPF 2.0 `independent task_region` de modo que cada tarea se ejecute en los procesadores indicados. Así, las instrucciones:

```
CREATE solve ( u , c ) ON p1
CREATE solve ( v , c ) ON p2
```

serán traducidas por un código similar al que se muestra en la *Figura 35*. El traductor inserta código y variables para poder realizar la ejecución en conjuntos disjuntos de

procesadores. La variable `BCL_Num_proc` se utiliza para ir guardando en los arrays de números enteros `p1` y `p2` el identificador del proceso correspondiente (que empieza en 0 al estar implementado sobre MPI). Asumiendo que entre los dominios `u` y `v` se han declarado fronteras previamente y que el traductor ha guardado internamente esta información, así como la distribución de los dominios, el traductor genera las instrucciones de las líneas 10 y 11. Estas instrucciones se utilizan para almacenar los identificadores de los procesos donde se ejecutará `v` (que están almacenados en `p2`) y que necesita el dominio `u` y viceversa.

```
 1) BCL_Num_proc=0
 2) do i_BCL= 1, ubound(p1,1)
 3)    p1(i_BCL)= BCL_Numproc
 4)    BCL_Numproc= BCL_Numproc+1
 5) enddo
 6) do i_BCL= 1, ubound(p2,1)
 7)    p2(i_BCL)= BCL_Numproc
 8)    BCL_Numproc= BCL_Numproc+1
 9) enddo
10) call BCL_set_procs (u,v,p2)
11) call BCL_set_procs (v,u,p1)
12) c%tids(1)=p1(1)
13) c%tids(2)=p2(1)
14) !hpf$ independent task_region
15) !hpf$ on procs (p1(lbound(p1,1):p1(ubound(p1,1)), resident
16)    call solve(u,c)
17) !hpf$ on procs (p2(lbound(p2,1):p2(ubound(p2,1)), resident
18)    call solve(v,c)
19) !hpf$ end task_region
```

Figura 35. Código generado para las dos instrucciones `CREATE`.

Las variables de tipo convergencia contienen los identificadores de los procesos pertenecientes a las tareas a las cuales se les pasa como argumento, de forma similar a como se hace con los dominios. Sin embargo, en este caso, sólo se guarda el identificador del primer proceso de cada tarea (líneas 12 y 13). Hay que tener en cuenta que las variables escalares que se pretenden comunicar tienen el mismo valor en todos los procesos de una tarea HPF. De esta forma, cuando se llama a la instrucción `converge`, el primer proceso de cada tarea es el que se encarga de hacer un envío al primer proceso del resto de las tareas. Esto se realiza mediante una subrutina declarada del tipo `extrinsic (HPF_LOCAL)`. Una vez que el primer proceso de cada tarea tiene un array con los errores de todas las tareas, manda este array al resto de los procesos de su

misma tarea. Cuando termina la comunicación, todos los procesos de todas las tareas llaman a la subrutina pasada como argumento.

Las variables con atributo `GRID`, son traducidas mediante la declaración de dos variables, puesto que un array dinámico no puede formar parte de un registro de Fortran estándar. Así, si en un programa BCL se declara, por ejemplo, una variable de la forma:

```
real, GRID2D ::g
```

el traductor generará código que declara dos variables:

```
type (domain2d) BCL_g_DOMAIN
real, allocatable :: BCL_g_DATA (:,:)
```

De esta manera, cuando se accede a uno de los campos de la variable, el traductor sólo tiene que sustituirla por la variable generada correspondiente. Cuando a un procedimiento se le pasa una variable con atributo `GRID`, se le han de pasar ambas variables. Igualmente, y para que no se produzcan errores de compilación, cuando una de estas variables aparece como parámetro formal de una subrutina, también es necesario declarar las dos variables. En este caso, el traductor no puede escribir la cabecera de un procedimiento o función hasta que no ha analizado el tipo de todos los parámetros formales.

Las instrucciones `PUT` y `GET` se codifican de forma relativamente simple dada la información que contienen las variables de tipo dominio asociadas al `GRID` correspondiente. De esta forma, cada procesador sabe qué región del array ha de mandar a qué procesador de cada tarea con una frontera en común con él. Esto se hará en un bucle para todas las fronteras que contenga y discriminando según el tipo de frontera que se haya indicado en las instrucciones `PUT` y `GET`. No es necesario que las subrutinas sean del tipo `extrinsic` puesto que todos los procesadores realizan la misma operación aunque con datos distintos.

Del resto de los aspectos de BCL se han implementado algunos de ellos mediante la biblioteca BCLIB. Ésta se ha realizado utilizando las características de Fortran 90 para la definición de interfaces y sobrecarga de operadores. De esta forma, cuando se quiere realizar una llamada a la instrucción `GROW` o `SHIFT` no hace falta

indicar la dimensión del dominio o región sino que es el compilador el que elige a qué procedimiento llamar mediante el tipo y número de los argumentos que se le pasan.

5.2 DIP.

La implementación de los programas escritos con las características de DIP se realiza también mediante la traducción fuente a fuente para generar código BCL sin primitivas DIP. Por un lado, los patrones son traducidos para generar el proceso coordinador de un programa BCL, aunque se han tenido que hacer algunas modificaciones a la implementación de este lenguaje para soportar la directiva `REPLICATE` del patrón `PIPE` como se explicará más adelante. Los procesos trabajadores se pueden escribir directamente en BCL, o bien, se pueden usar las plantillas de implementación, para lo cual el compilador necesita la información descrita tanto en el patrón como en la especificación de tarea.

En primer lugar, se va a describir el proceso de traducción de los patrones para obtener un proceso coordinador BCL. Seguidamente se describirá el proceso de instanciación de una plantilla de implementación.

5.2.1 Patrones.

El proceso de traducción es distinto para los dos patrones que se han definido. La traducción de un patrón `MULTIBLOCK` es bastante directa. Partiendo de su cabecera, se obtiene el nombre del programa y los dominios que se van a utilizar. De la declaración del dominio se obtiene también su dimensión. Al analizar las tareas, se obtienen los identificadores de los procesos que se van a ejecutar, lo cual se guarda en un fichero temporal. Se generan las fronteras y, por último, se crean las instrucciones `CREATE` con los datos del fichero temporal.

La generación del proceso coordinador partiendo de un patrón `PIPE` es un poco más complicada, debido a que cambia el modo de definir la comunicación entre las tareas. Ya no se definen fronteras sino que la comunicación se realiza pasando como argumento el mismo dominio a dos etapas. Para solucionar esto, lo que se hace es definir un array de dominios. Cada dominio que aparece en el patrón en cada una de las

llamadas tendrá asociado un índice dentro del array de dominios. Cuando se utiliza el mismo dominio en dos etapas distintas, el traductor genera una frontera entre los dos elementos del array. Así, en el ejemplo que sigue:

```
PIPE ejemplo a/1,1,N,N/
   Etapa1(a:(*,BLOCK)) ON PROCS(2)
   Etapa2(a:(BLOCK,*)) ON PROCS(2)
END
```

el traductor asociará el índice 1 con la primera aparición del dominio a. El índice 2 será para la segunda aparición de este dominio. Así, el traductor generará un código como el que se muestra a continuación:

```
Program ejemplo
DOMAIN2D DIP_domains(2)
DISTRIBUTE (*,BLOCK):: DIP_DOMAINS(1)
DISTRIBUTE (BLOCK,*):: DIP_DOMAINS(2)
PROCESSORS DIP_PROCS1(2), DIP_PROCS2(2)
DIP_DOMAINS(1)=(/1,1,N,N/)
DIP_DOMAINS(2)=(/1,1,N,N/)
DIP_DOMAINS(2) <- DIP_DOMAINS(1)
CREATE Etapa1 (DIP_DOMAINS(1)) ON DIP_PROCS1
CREATE Etapa2 (DIP_DOMAINS(2)) ON DIP_PROCS2
END
```

En el caso de que se utilicen patrones anidados, el traductor reemplaza la llamada dentro del patrón más externo por el código perteneciente al anidado (de una manera similar a como lo hace un ensamblador con una macro). En el proceso de sustitución, se reemplazan también los nombres de los dominios declarados como argumento por los parámetros reales. De esta forma, se obtiene un patrón más grande pero sin llamadas anidadas.

Si se utiliza la instrucción `REPLICATE`, se crean tantos índices dentro del array como réplicas haya. Si una etapa anterior ha de mandar datos a las instancias de esta etapa, se crea una frontera entre el dominio de la etapa emisora y cada una de las instancias de la etapa replicada. Estas fronteras son de un tipo especial que ha tenido que ser introducido en BCL, de modo que cuando la etapa emisora envía su frontera, la comunicación no se hace a todas las instancias sino que solamente se envía a una de ellas. La primera vez que se llame a `PUT_BORDERS` se envía a la primera instancia, la segunda vez a la segunda instancia y así sucesivamente en orden cíclico. Para implementar este tipo de frontera, el dominio debe almacenar internamente el número de la réplica a la cual se envió por última vez.

5.2.2 Plantillas de implementación.

La mayor parte del proceso de instanciación de las plantillas de implementación se ha comentado anteriormente para aclarar su funcionamiento. Solamente queda aclarar algunos aspectos de bajo nivel.

Cuando se compila un patrón, se generan unos ficheros intermedios con información necesaria para el proceso de instanciación. Estos ficheros contendrán el nombre de cada tarea llamada por el coordinador (recuérdese que el compilador puede cambiar el nombre de la subrutina para el caso de distribuciones distintas) e información sobre los dominios y distribuciones de éstos. En el caso del patrón `PIPE` también se almacenará información relativa al uso de cada dominio por parte de una etapa. En concreto, distinguirá si una etapa es la que genera los datos asociados a cada dominio, si es la última o si es una etapa intermedia para ese dominio.

En el proceso de instanciación de una plantilla, se toma de ésta el cuerpo principal, mientras que las apariciones de los argumentos de la plantilla dentro del cuerpo se sustituyen por instrucciones del tipo `INCLUDE` de Fortran. De esta forma, el compilador sólo tiene que generar unos ficheros intermedios que serán incluidos en la instancia. Estos ficheros se generan a partir de la información obtenida al compilar el patrón y la información suministrada en la especificación de tarea.

Cuando en un patrón se llama a una subrutina con dos dominios que están distribuidos de forma distinta, se instancian dos procesos trabajadores distintos. Para ello, el traductor concatena el nombre de la subrutina con la distribución de cada dominio pasado como argumento a ésta. Así, si se crean dos tareas trabajadoras denominadas ambas como `solve` pero pasándole dos dominios distintos, por ejemplo, `u` y `v` con distribuciones distintas, pongamos `(BLOCK,*)` para `u`, la primera llamada se sustituiría por `call solveBLOCK_`, representando el símbolo `_` la distribución `*`. El compilador dará un mensaje informando de este cambio.

Por último, queda por comentar el caso de las plantillas diseñadas para las etapas de un encauzamiento donde no se conoce el tamaño del flujo de datos. Como se mencionó anteriormente, el sistema añade una marca a cada envío, de modo que un valor especial, `end_of_stream`, es el que indica la terminación del flujo. El compilador

detecta la necesidad de esta marca por el tipo de plantilla que se emplea en la especificación de tarea. Estas plantillas contienen instrucciones especiales denominadas `<send mark>` `<receive mark>`, que son sustituidas por un envío/recepción a la etapa siguiente de un valor entero. Este valor es enviado a todos los procesadores indicados en el dominio o dominios declarados en la especificación de tarea.

5.3 Conclusiones.

Para hacer una evaluación del modelo, se ha realizado un prototipo de compilador que hace una traducción fuente a fuente de código BCL a HPF utilizando las características del sistema Adaptor en cuanto a la ejecución de distintos trozos de código por distintos subconjuntos de procesadores. La comunicación entre las tareas se realiza utilizando nuestra propia biblioteca y el sistema de paso de mensajes MPI.

Se han implementado sólo las características más importantes del lenguaje para poder evaluar su comportamiento y poder realizar las primeras pruebas que nos permitan probar las primeras aplicaciones con nuestro lenguaje.

En este capítulo se describe también cómo se realiza la traducción de programas escritos mediante patrones. Cada patrón se traduce de forma distinta generando la parte de coordinación de un programa BCL sin patrones. Si se utilizan plantillas, el código insertado por el compilador es obtenido de la definición del patrón, la plantilla y las especificaciones de tarea.

Gracias a la construcción del prototipo de compilador se han podido realizar las pruebas de eficiencia del modelo y cuyos resultados se presentan en el capítulo siguiente.

Capítulo 6. Resultados

A continuación se muestran algunos ejemplos que se han utilizado para medir la eficiencia de la implementación de BCL. Para obtener los resultados se ha utilizado un cluster de 4 nodos *DEC AlphaServer 4100* conectados mediante *Memory Channel*. Cada nodo tiene 4 procesadores *Alpha 22164* (300 MHz) compartiendo 256 MB de memoria RAM. El sistema operativo es *Digital Unix V4.0D (Rev. 878).*

En primer lugar se presenta un banco de pruebas sencillo en el que dos procesos trabajadores se intercambian un array cuya distribución es distinta en ambas tareas (también denominado *ping-pong*).

Seguidamente se presenta el ejemplo de Jacobi para la ecuación de Laplace y dominios regulares, donde se considera el efecto de tener distinto número de dominios con distinto número de procesadores y se comparan los resultados con los obtenidos con HPF.

Los resultados para la transformada rápida de Fourier en dos dimensiones se muestran en el apartado siguiente para distintos tamaños de matrices y distinto número de procesadores.

En el apartado 6.4 se muestran los resultados obtenidos para el problema de difusión mediante la transformada de Fourier utilizando encauzamiento y replicación de tareas.

Por último, se presenta un problema de descomposición de dominios de una aplicación más compleja que las anteriores. Para ello, se realiza primero un estudio en detalle del mejor método de descomposición de dominios a emplear en función de la exactitud de los resultados y del número de iteraciones necesarias para su convergencia. Posteriormente, se emplea BCL para su paralelización, primero con distinto número de dominios regulares, para después utilizar un dominio global irregular.

6.1 Banco de Pruebas Sencillo.

El Programa 38 muestra el proceso coordinador en BCL del banco de pruebas simple en el que se definen dos dominios `a` y `b`, cada uno de los cuales es pasado como argumento a un proceso trabajador distinto. La distribución de ambos es por columnas y por filas, respectivamente (*Figura 36*). A cada una de las tareas se le asigna la mitad de los procesadores disponibles (por sencillez, se considera una constante predefinida `P` aunque ésta se podría sustituir por una llamada a la función `number_of_processors`). La comunicación entre ambas tareas se define en las líneas 8 y 9 puesto que la comunicación se realiza en ambos sentidos.

Los procesos trabajadores se muestran en el Programa 39. Se declaran los dominios pasados como parámetros, se definen las variables `GRID` de cada trabajador y su distribución. Después se crean los arrays dinámicos con la instrucción de la línea 5 de cada programa y, durante un número de iteraciones, se envían y reciben los datos a la otra tarea (líneas 7 y 8).

```
 1) program benchmark
 2) DOMAIN2D  a,b
 3) PROCESSORS p1(P/2), p2 (P/2)
 4) DISTRIBUTE a(*,BLOCK) ONTO p1
 5) DISTRIBUTE b(BLOCK,*) ONTO p2
 6) a= (/1,1,N,N)
 7) b= (/1,1,N,N)
 8) a <- b
 9) b <- a
10) CREATE bench_1 (a) ON p1
11) CREATE bench_2 (b) ON p2
12) end
```

Programa 38. Proceso coordinador para el banco de pruebas.

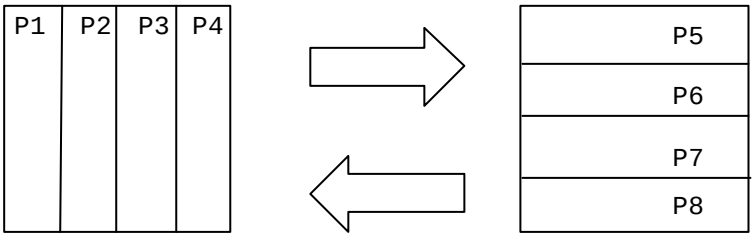


Figura 36. Distribución de datos en el banco de pruebas sencillo.

```
 1) subroutine bench_1(d)
 2) DOMAIN2D  d
 3) double precision, GRID2D :: a
 4) !hpf$ DISTRIBUTE a(*,BLOCK)
 5) a%DOMAIN = d
 6) do i=1,N_ITERS
 7)   PUT_BORDERS (a)
 8)   GET_BORDERS (a)
 9) enddo
10) end
```

```
 1) subroutine bench_2(d)
 2) DOMAIN2D  d
 3) double precision, GRID2D :: b
 4) !hpf$ DISTRIBUTE b(BLOCK,*)
 5) b%DOMAIN = d
 6) do i=1,N_ITERS
 7)   GET_BORDERS (b)
 8)   PUT_BORDERS (b)
 9) enddo
10) end
```

Programa 39. Procesos trabajadores para el banco de pruebas.

Para comparar la eficiencia de la implementación se ha utilizado la versión de Adaptor que permite la comunicación de tareas HPF. El Programa 40 muestra el programa principal de la misma aplicación. Se declaran los procesadores disponibles mediante la instrucción de la línea 2. La línea 3 define el comienzo de la región donde se van a definir las distintas tareas. En la línea 4 se define en qué procesadores se tienen que ejecutar las instrucciones que siguen. La cláusula `resident` indica al compilador que los datos que necesitan esas instrucciones no afectan al resto, es decir, informa al compilador de que la subrutina llamada no realiza efectos laterales. De esta forma, la mitad de los procesadores ejecutan la subrutina `bench_adaptor_1` y la otra mitad, `bench_adaptor_2`.

```
1) program benchmark_adaptor
2) !hpf$ processors procs (P)
3) !hpf$ independent task_region
4) !hpf$ on (procs(1:P/2)),resident
5)    call bench_adaptor_1
6) !hpf$ on (procs(P/2+1:proce)),resident
7)    call bench_adaptor_2
8) !hpf$ end task_region
9) end
```

Programa 40. Programa principal del banco de pruebas para la versión con Adaptor.

```
 1) subroutine bench_adaptor_1
 2) use hpf_task_library
 3) double precision :: a(N,N)
 4) !hpf$ distribute a(*,block)
 5) integer req_send, req_recv
 6) call hpf_task_init()
 7) call hpf_send_init(a,2,req_send)
 8) call hpf_recv_init(a,2,req_recv)
 9) do i= 1, N_ITERS
10)    call hpf_task_comm(req_send)
11)    call hpf_task_comm(req_recv)
12) enddo
13) end
```

```
 1) subroutine bench_adaptor_2
 2) use hpf_task_library
 3) double precision :: b(N,N)
 4) !hpf$ distribute b(block,*)
 5) integer req_send, req_recv
 6) call hpf_task_init()
 7) call hpf_recv_init(b,1,req_recv)
 8) call hpf_send_init(b,1,req_send)
 9) do i= 1, N_ITERS
10)    call hpf_task_comm(req_recv)
11)    call hpf_task_comm(req_send)
12) enddo
13) end
```

Programa 41. Tareas HPF para la versión de Adaptor.

Las dos subrutinas se muestran en el Programa 41. En este caso, el tamaño del array se tiene que conocer en tiempo de compilación (no se han declarado las variables `a` y `b` como dinámicas para no complicar el ejemplo). Después de declarar la distribución de las variables, en la línea 5, se tienen que declarar dos números enteros que harán referencia a las peticiones de comunicación declaradas en las líneas 7 y 8. La línea 6 es un requisito de la biblioteca. Cuando se ejecutan las líneas 7 y 8, las tareas intercambian información sobre la distribución de las variables en la tarea fuente y destino (sino se hiciera así, esta información se tendría que intercambiar por cada iteración). El segundo parámetro de las llamadas de estas dos instrucciones es el número de tarea al que hay que enviar/recibir (la primera tarea será numerada con el 1).

Una vez que las tareas han intercambiado la información necesaria, cuando se ejecuta la línea 10, cada procesador de la primera tarea sabe qué datos tiene que enviar a qué procesador de la segunda y viceversa. Lo mismo ocurre con la línea 11, ya que la información sobre las distribuciones se intercambió al ejecutarse la línea 8.

Nótese que, en BCL, la información sobre las distribuciones se realiza en tiempo de compilación puesto que esta información es suministrada por el coordinador y es pasada a los procesos trabajadores. Además, cuando se realiza la petición de comunicación (líneas 7 y 8 del Programa 41) hay que tener en cuenta el número de tarea a la que hay que enviar o recibir, mientras que en BCL no es necesario (puesto que esto se hace en el coordinador). En este caso muy simple, no parece una gran ventaja, pero en otros problemas donde interactúen un mayor número de tareas, este hecho puede ser

un elemento más que el programador tiene que tener en mente y que puede provocar errores de programación.

En la *Figura 37* se muestra la comparación de los tiempos de ejecución con ambos lenguajes para distintos tamaños de matrices. El número de iteraciones ha sido 500 y los tiempos mostrados (en milisegundos) son por iteración. Como puede apreciarse, a excepción de matrices pequeñas (64 por 64) en las que BCL siempre es mejor, Adaptor es mejor cuando se tienen sólo dos procesadores (no es necesario el

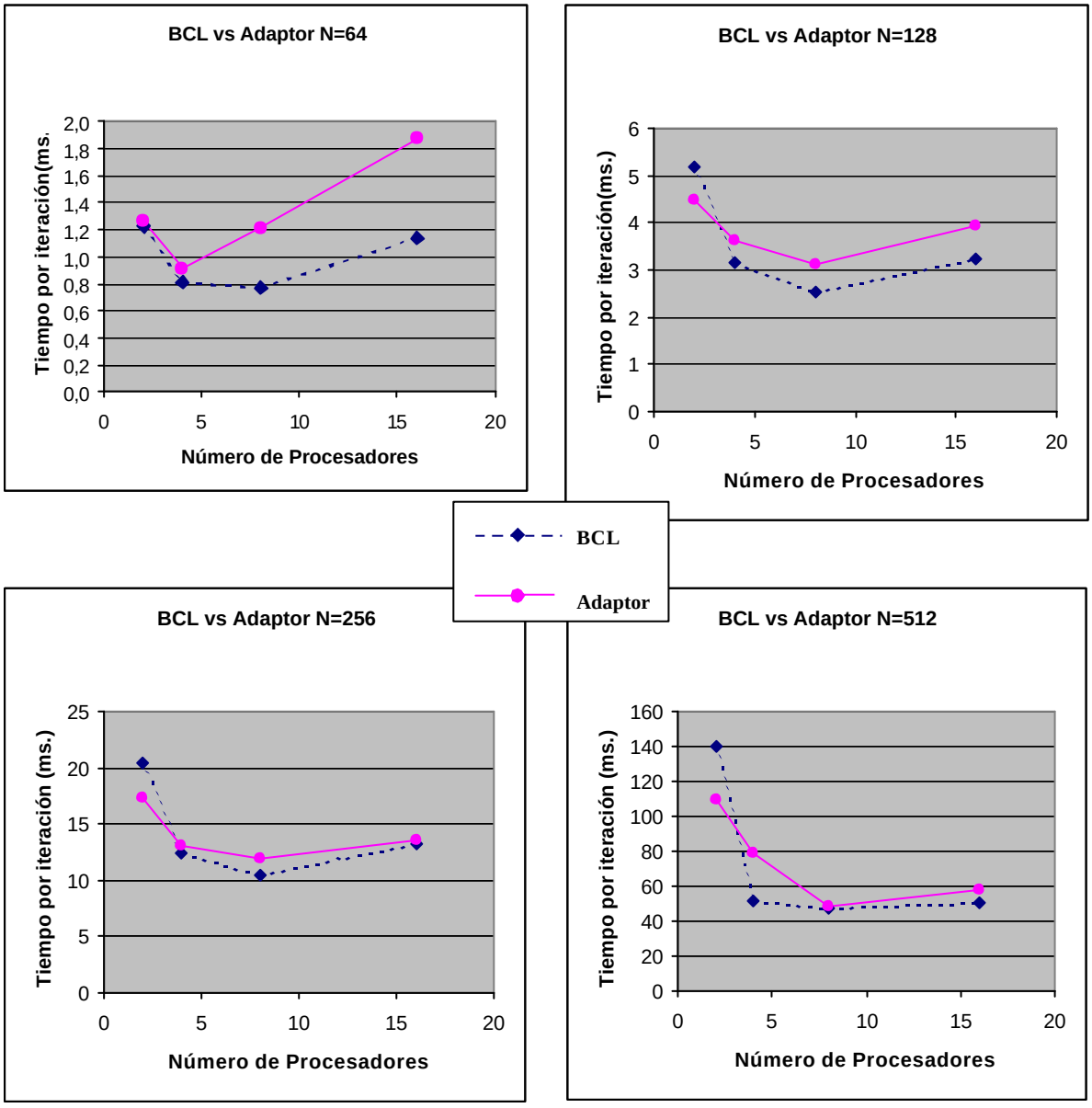


Figura 37. Comparación de los tiempos de ejecución (por iteración) del banco de pruebas simple tanto en BCL como en Adaptor.

intercambio de información puesto que las variables no están distribuidas y, además, no se están integrando el paralelismo de datos y el de tareas, sino sólo el de tareas). Con más procesadores, BCL es ligeramente superior. De esta forma, las ventajas de BCL como lenguaje de coordinación no suponen una perdida de eficiencia, lo cual constituye uno de los objetivos marcados al comienzo del trabajo.

6.2 Método de Jacobi.

El Programa 42 muestra un ejemplo de uso de BCL para la ecuación de Laplace resuelta mediante el método de Jacobi como se describió en el apartado 2.2.3. En este caso se ha utilizado un número de dominios regulares. Este número está definido por una constante, `NDOMAINS`, de modo que los subdominios se declaran como un array de dominios (línea 3) y cuyos tamaños se definen en las líneas 7 a 9 (todos tienen la misma forma y tamaño por lo que la asignación se puede hacer en un bucle). En la línea 4, se define la variable de tipo convergencia que será compartida por los distintos procesos trabajadores. Las fronteras entre los dominios se definen en las líneas 10 a 13. Los procesos trabajadores se crean en las líneas 14 a 16, de nuevo dentro de un bucle. El código de cada proceso trabajador es el mismo que el mostrado en el Programa 2.

```
 1) program jacobi_regular
 2) integer, parameter :: Nx= 128, Ny = 128, NDOMAINS= 4
 3) DOMAIN2D  u(NDOMAINS)
 4) CONVERGENCE c OF NDOMAINS
 5) PROCESSORS p1(number_of_processors()/NDOMAINS,NDOMAINS)
 6) DISTRIBUTE (*,BLOCK) :: u (:)
 7) do i= 1, NDOMAINS
 8)    u(i)=(/1,1,Nx, Ny/)
 9) enddo
10) do i= 1,NDOMAINS-1
11)    u(i)(Nx,1,Nx,Ny) <- u(i+1)(2,1,2,Ny)
12)    u(i+1)(1,1,1,Ny) <- u(i)(Nx-1,1,Nx-1,Ny)
13) enddo
14) do i= 1,NDOMAINS
15)    CREATE solve (u(i),c) ON p1(:,i)
16) enddo
17) end
```

Programa 42. Proceso coordinador para el ejemplo de Jacobi regular.

También se ha implementado el problema utilizando HPF estándar para comparar los resultados. La *Figura 38* compara los tiempos de ejecución con las

implementaciones en HPF y BCL, considerando 2, 4 y 8 dominios con una malla de 128x128 cada uno. El problema se ha probado con 4, 8 y 16 procesadores, con objeto de integrar el paralelismo de datos y tareas. El programa ha sido ejecutado para 20.000 iteraciones.

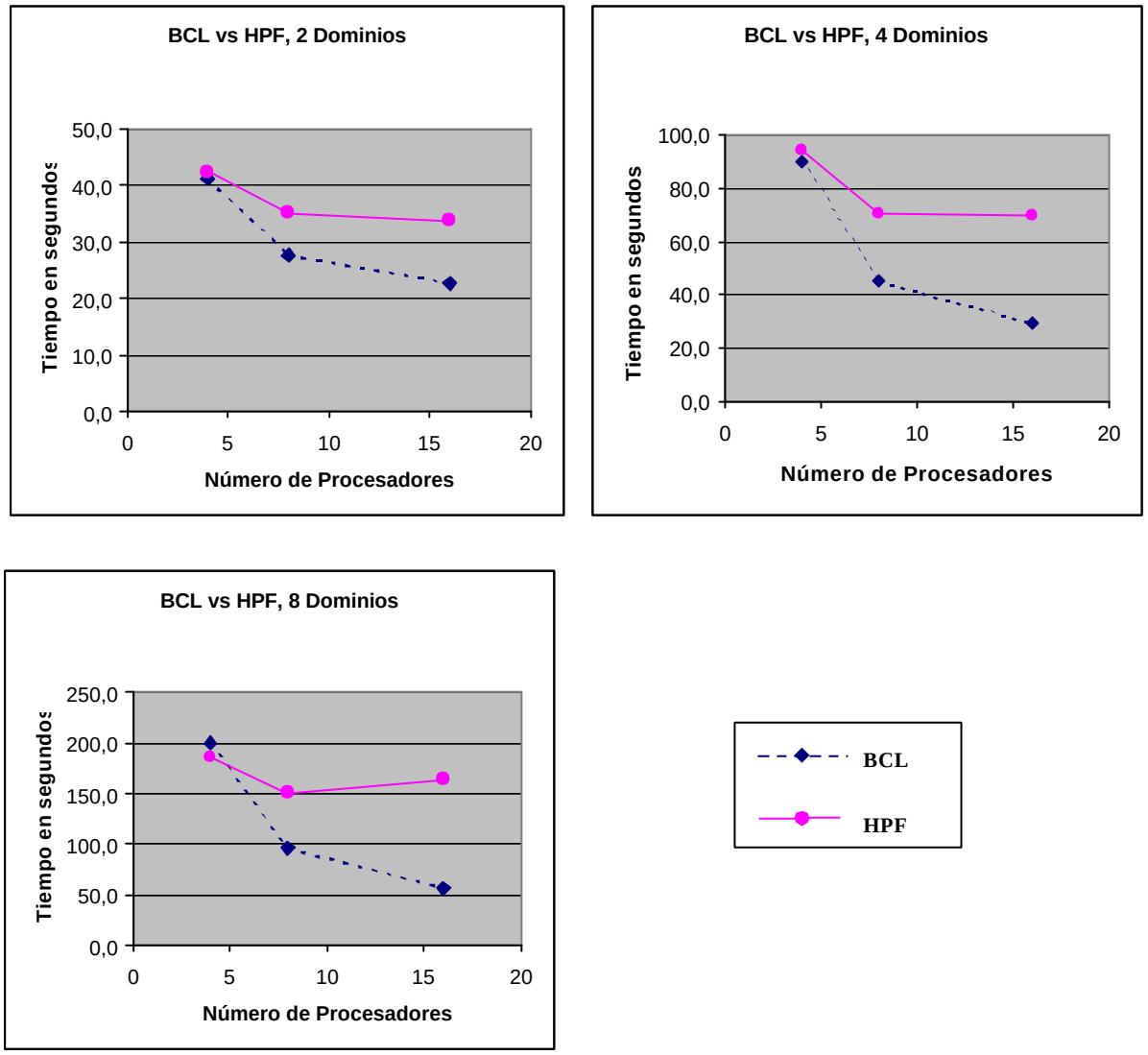


Figura 38. Tiempos de ejecución para las implementaciones de HPF y BCL para el método de Jacobi en función del número de procesadores.

Como se puede observar en la Tabla 1, la integración del paralelismo de datos y tareas ofrece un mayor rendimiento. En esta tabla se presentan tanto los tiempos de ejecución como la relación entre ambos tiempos para los distintos dominios y distinto número de procesadores. Cuando el número de procesadores es igual al de dominios (sólo se realiza paralelismo de tareas), BCL también es más eficiente. Solamente cuando

existen más dominios que procesadores disponibles, BCL ofrece peor rendimiento debido a la sobrecarga del cambio de contexto de procesos “pesados”. Hay que tener en cuenta que cada dominio se ejecuta en un proceso aparte.

		HPF vs. BCL (relación)		
Dominios	**Secuencial**	**4 Procesadores**	**8 Procesadores**	**16 Procesadores**
2	97.05	42.40 / 41.27 (1.03)	35.05 / 27.66 (1.27)	33.73 / 22.67 (1.49)
4	188.88	93.90 / 90.06 (1.04)	70.75 / 45.06 (1.57)	69.61 / 29.28 (2.38)
8	412.48	185.62 / 199.66 (0.93)	150.54 / 95.85 (1.57)	163.67 / 56.43 (2.90)

Tabla 1. Tiempos de ejecución en segundos obtenidos con las implementaciones de HPF y BCL para el método de Jacobi. Entre paréntesis la ganancia obtenida al usar BCL con respecto a HPF.

6.3 FFT en Dos Dimensiones.

El problema FFT2D ha sido implementado tanto en HPF, utilizando paralelismo de datos, como con BCL. Puesto que el código del programa principal de ambos

		HPF vs. BCL (relación)			
Tamaño de Array	**Secuencial**	**2 Procesadores**	**4 Procesadores**	**8 Procesadores**	**16 Procesadores**
32 x 32	1.507	1.178 / 0.953 (1.23)	0.947 / 0.595 (1.59)	0.987 / 0.457 (2.08)	1.601 / 0.921 (1.74)
64 x 64	5.165	3.532 / 3.351 (1.05)	2.189 / 1.995 (1.09)	1.778 / 1.082 (1.64)	2.003 / 1.095 (1.83)
128 x 128	20.536	14.01 / 13.67 (1.02)	7.238 / 7.010 (1.03)	5.056 / 4.081 (1.24)	4.565 / 2.905 (1.57)

Tabla 2. Tiempos de ejecución por iteración en milisegundos obtenidos con las implementaciones de HPF y BCL para FFT2D. Entre paréntesis la ganancia obtenida con BCL.

procesos ha sido descrito previamente (Programa 14 para HPF y Programa 15 a Programa 17 para BCL) aquí sólo se muestran los resultados. Éstos se pueden encontrar en la Tabla 2 para 1000 pasos de tiempo y su representación gráfica puede verse en la *Figura 39*.

BCL es más eficiente que HPF en todos los casos, sin embargo, el rendimiento de HPF se aproxima al de BCL a medida que el problema se hace mayor y el número de procesadores decrece, al igual que ocurre en otras aproximaciones [Foster y otros 97].

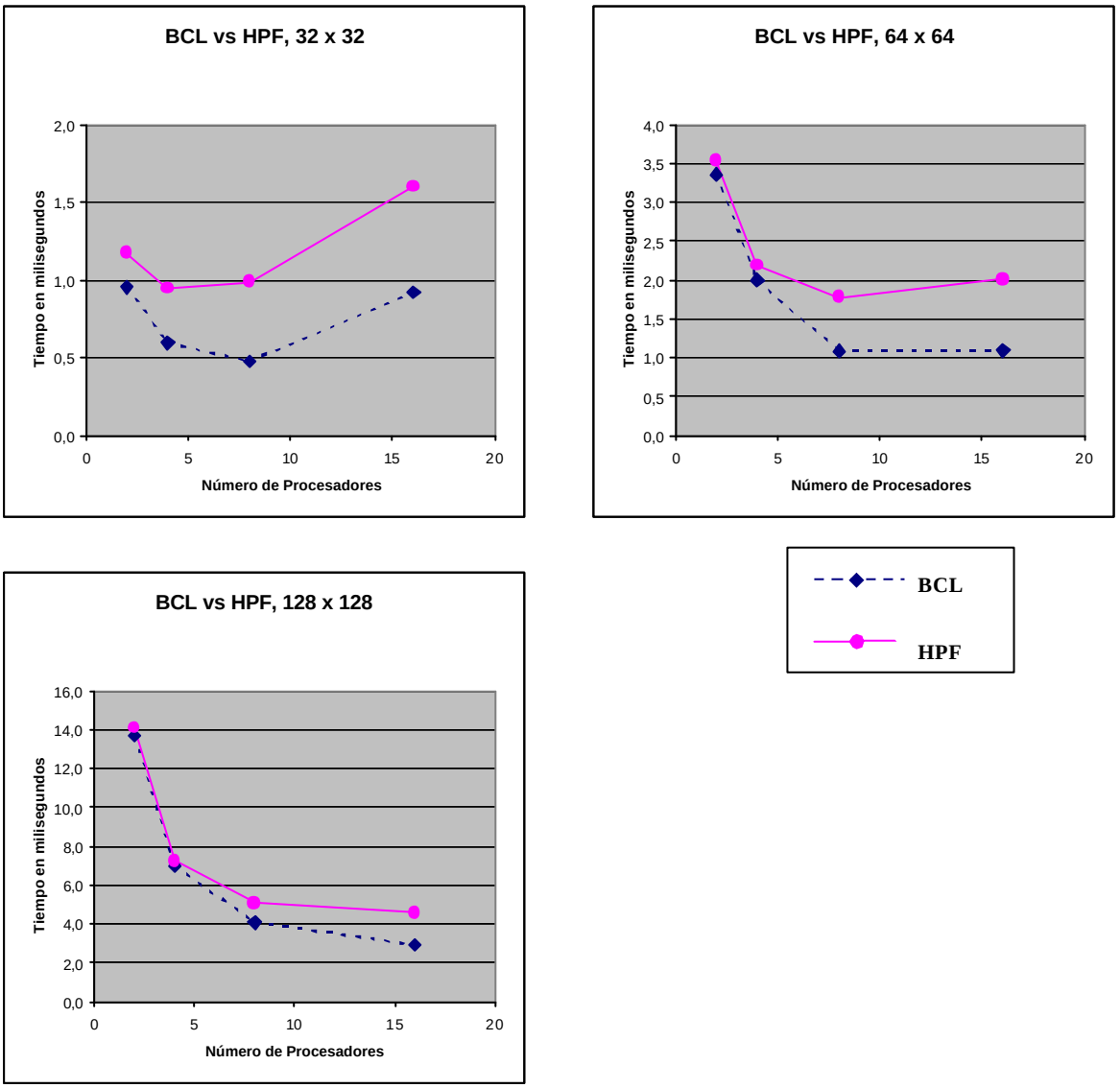


Figura 39. Tiempos de ejecución para las implementaciones de HPF y BCL para la transformada rápida de Fourier en dos dimensiones en función del número de procesadores y para varios tamaños de array.

En esta situación, el rendimiento de HPF es razonablemente bueno, con lo que la integración del paralelismo de datos y tareas no contribuye tanto.

Para demostrar el comportamiento de las tareas según se mostró en la *Figura 17* de la página 79, se han tomado los tiempos a cada una de las fases de una ejecución. Así, se obtuvieron los tiempos de los dos algoritmos para una matriz de tamaño de 128 x 128 y 2 procesadores. La Tabla 3 muestra los tiempos en milisegundos para las dos implementaciones. Hay que hacer notar que en la versión HPF los tiempos de las transformadas por filas y columnas se realizan una después de la otra mientras que en BCL se realizan a la vez. Los resultados se han obtenido dividiendo el tiempo total empleado por el número de iteraciones. Entre paréntesis se encuentra el tiempo empleado por el primer GET de la versión BCL, que como puede observarse es significativamente mayor a la media como se visualizó en la *Figura 17*.

HPF	**FFT por Columnas**	**FFT por Filas**	**Primera Transpuesta**	**Segunda Transpuesta**
	4.733	4.399	4.371	4.401
BCL	**FFT por Columnas**	**FFT por Filas**	**PUT**	**GET (primer GET)**
	9.334	8.381	4.717	4.674 (23.324)

Tabla 3. Tiempos de ejecución en milisegundos por cada iteración de las distintas fases de los algoritmos obtenidos con las implementaciones de HPF y BCL para el problema FFT2D. Entre paréntesis el tiempo empleado en el primer GET.

6.4 Transformada de Fourier para un Problema de Difusión.

El problema NPB-FT descrito anteriormente se ha implementado siguiendo distintas estructuras de cómputo. La primera de ellas es una implementación en HPF estándar sin encauzamiento ni replicación de tareas, tal como se describe en las figuras 18 y 19. Con objeto de poder compararlas hemos denominado a esta implementación `HPF`.

La siguiente implementación probada es la representada en la *Figura 20*. En ésta existen dos etapas, la primera de ellas se encarga de realizar los cálculos iniciales y la

transformada a lo largo del eje Z. La segunda etapa se encarga de la transformada a lo largo de los ejes X e Y. Las pruebas se han realizado para dos tamaños distintos de matriz 32x32x32 y 64x64x64. En ambos casos se ejecutaron 16 pasos de tiempo. Esta implementación la hemos denominado `BCL`.

Además, se ha implementado el esquema de resolución con réplicas de encauzamiento como el mostrado en la *Figura 21*. Se probaron ejecuciones con 2, 4 y 8 réplicas. A estas implementaciones las hemos denominado respectivamente `BCL 2`, `BCL 4` y `BCL 8`.

	NPB-FT 32x32x32. Secuencial 9.47 E-2				
Procesadores	**HPF**	**BCL**	**BCL 2**	**BCL 4**	**BCL 8**
2	5.64E-02	6.02E-02			
4	3.75E-02	3.54E-02	3.53E-02		
8	3.34E-02	2.38E-02	2.07E-02	2.26E-02	
16	4.35E-02	3.73E-2	1.98E-02	1.50E-02	1.52E-02

Tabla 4. Tiempos de ejecución en segundos por cada iteración de las distintas implementaciones para el problema NPB-FT y un tamaño de 32x32x32.

La Tabla 4 muestra los resultados de las ejecuciones para el primer tamaño de matriz considerado. En primer lugar, se muestra el tiempo en secuencial y, a continuación, los tiempos empleados al ejecutar con `HPF` y las implementaciones con BCL antes comentadas. Como cada etapa necesita al menos un procesador, al realizarse `R` réplicas se necesitan al menos `2·R` procesadores. Por esta razón, no se pueden mostrar los resultados de las ejecuciones con réplicas y menor número de procesadores.

La *Figura 40* muestra gráficamente los tiempos de ejecución en función del número de procesadores. Como puede observarse, cuando no se tiene replicación, la ejecución con BCL es peor cuando el número de procesadores es 2 ya que no se saca partido a la integración del paralelismo de datos y tareas. Aunque no se muestra aquí, el primer procesador termina su ejecución rápidamente si el búfer definido para el paso de mensajes es suficientemente grande. Los tiempos mostrados son los del segundo

procesador, es decir, el tiempo empleado por la segunda etapa. Cuando el número de procesadores es mayor, BCL ofrece mejor rendimiento que HPF. Si se utilizan dos réplicas, el rendimiento mejora en todos los casos. Sin embargo, con un número mayor de réplicas, el rendimiento no siempre es mejor. Esto es debido a varias razones. Por un lado, el número de iteraciones, 16, es pequeño, lo que provoca que se note más la redundancia de los cálculos iniciales que se deben realizar en todas las réplicas. Por otro lado, se deja de sacar rendimiento al paralelismo de datos, puesto que, por ejemplo, con 4 réplicas y 8 procesadores, cada etapa es ejecutada por un solo procesador.

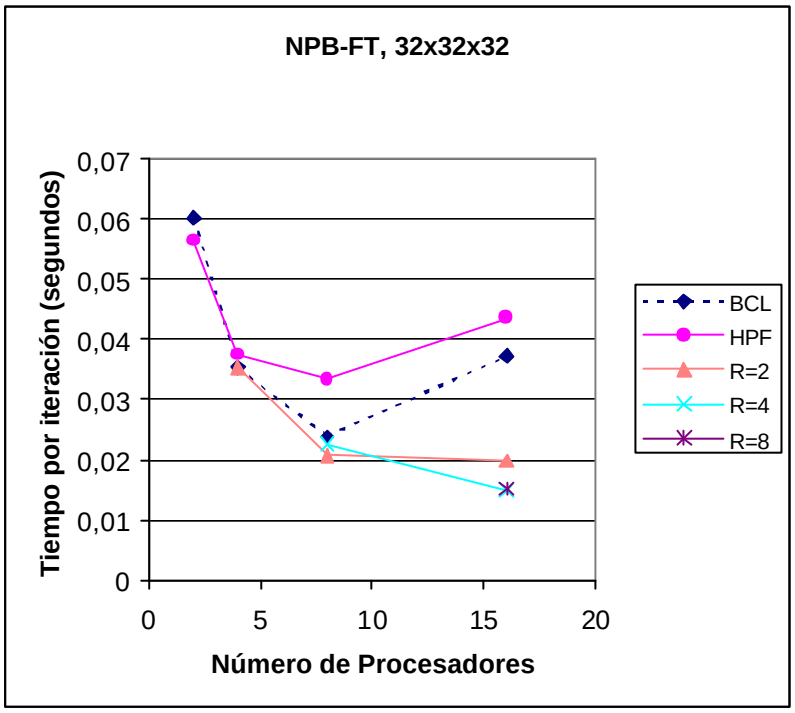


Figura 40. Tiempos de ejecución para las implementaciones de HPF y BCL con réplicas para la aplicación NPB-FT en función del número de procesadores para un array de 32x32x32..

Los resultados para un tamaño mayor, 64x64x64 se muestran en la Tabla 5. Puesto que el cómputo a realizar es mayor, HPF muestra mejores resultados que BCL con 2 y 4 procesadores mientras que a partir de 8, BCL pasa a ser mejor. Como puede observarse en la *Figura 41*, los mejores resultados se obtienen cuando se utilizan 16 procesadores y 2 réplicas. Al igual que ocurría el caso anterior, un número de réplicas demasiado grande puede no ofrecer mejores resultados como ocurre aquí para el caso de 4 y 8 réplicas.

	NPB-FT 64x64x64. Secuencial 9.99 E-01				
Procesadores	**HPF**	**BCL**	**BCL 2**	**BCL 4**	**BCL 8**
2	5.17E-01	6.66E-01			
4	3.39E-01	3.88E-01	4.80E-01		
8	2.39E-01	2.07E-01	2.11E-01	2.78E-01	
16	2.13E-01	1.18E-01	1.13E-01	1.48E-01	1.92E-01

Tabla 5. Tiempos de ejecución en segundos por cada iteración de las distintas implementaciones para el problema NPB-FT y un tamaño de 64x64x64.

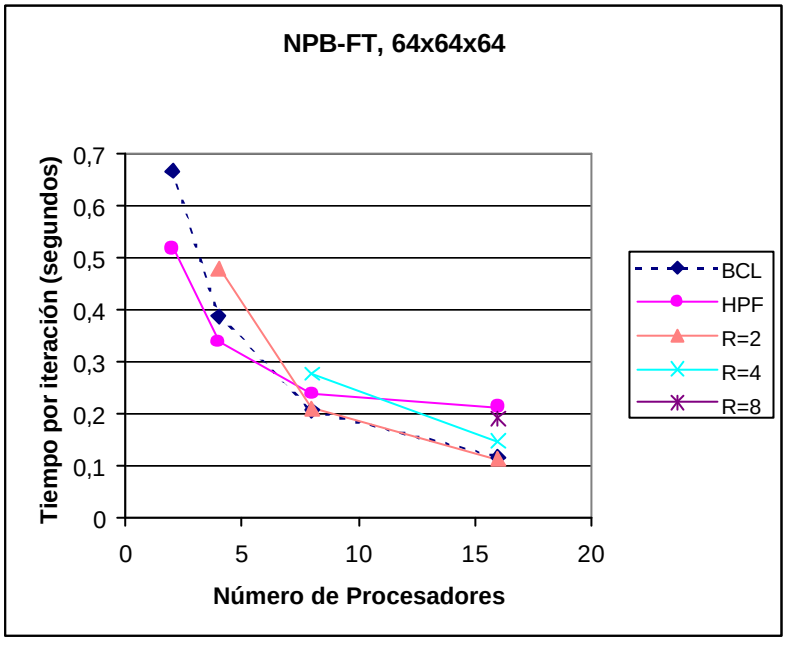


Figura 41. Tiempos de ejecución para las implementaciones de HPF y BCL con réplicas para la aplicación NPB-FT en función del número de procesadores para un array de 64x64x64.

6.5 Una Aplicación Real de Descomposición de Dominios con BCL.

Para comprobar la adecuación de BCL a un problema más complejo que los anteriores, se ha considerado un ejemplo de ecuaciones en derivadas parciales cuyos orígenes se pueden encontrar en la propagación de pulsos en sistemas biológicos así como en el modelo simplificado de la ignición y propagación de una llama en mezclas combustibles, en las que solamente se considera una reacción. Este problema se modela mediante un sistema de ecuaciones no lineales de reacción-difusión y se ha considerado un dominio bidimensional irregular con esquinas reentrantes (*Figura 42*).

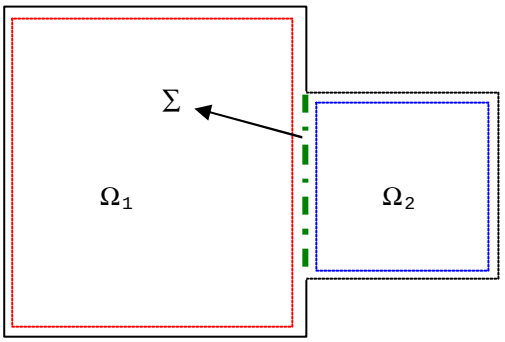


Figura 42. El dominio irregular se descompone en dos dominios regulares Ω_1 y Ω_2. La frontera de la superficie es la que se muestra con línea continua y en ella se cumplen las condiciones de Dirichlet. Σ es la frontera o interfaz entre el dominio 1 y el dominio 2.

Las ecuaciones de este problema son las siguientes:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + F(U) \qquad \textbf{Ecuación 6}$$

donde: $U = \begin{pmatrix} u \\ v \end{pmatrix}$ y $F(U) = \begin{pmatrix} -uv \\ uv - Kv \end{pmatrix}$, siendo **K** una constante.

En este modelo simplificado de ignición y propagación de una llama (o frente), la variable u representaría la concentración adimensional del combustible y v la temperatura adimensional. En la frontera (línea continua en la *Figura 42*) se ha impuesto u= 1 y v= 0**,** permaneciendo constantes en el tiempo (condiciones de Dirichlet); t es el tiempo, y x e y representan las coordenadas cartesianas. Un problema

análogo ha sido estudiado previamente sin descomposición de dominios mediante una variedad de métodos de diferencias finitas en [Ramos 97].

En el artículo [Ramos y Soler 01] se realiza un estudio de los distintos métodos de descomposición de dominios para la resolución de este problema tanto en dominios regulares como irregulares. Se estudian métodos con superposición y sin ella, aplicando, además, distintas condiciones en la frontera entre dominios para cada uno de los métodos (condiciones de Dirichlet, Neumann, Robin y combinaciones de ellas). El resultado de aplicar estos métodos, así como la resolución del mismo problema mediante otros métodos sin descomposición de dominios y la comparación entre ellos, se encuentra en [Ramos, Soler y Troya 98].

Aquí se presenta un resumen del estudio, centrándonos en los dominios irregulares con esquinas reentrantes como los de la *Figura 42*.

Tomando diferencias finitas en la variable t, la Ecuación 6 se puede expresar en los siguientes términos:

$$\frac{U^{n+1}-U^{n}}{\Delta t}=(1-q)\left[\frac{\partial^{2}U^{n}}{\partial x^{2}}+\frac{\partial^{2}U^{n}}{\partial y^{2}}+F(U^{n})\right]+q\left[\frac{\partial^{2}U^{n+1}}{\partial x^{2}}+\frac{\partial^{2}U^{n+1}}{\partial y^{2}}+F(U^{n+1})\right]$$

Ecuación 7

Con lo cual, se tiene:

$$U^{n+1}-\Delta t q\left[\frac{\partial^{2}U^{n+1}}{\partial x^{2}}+\frac{\partial^{2}U^{n+1}}{\partial y^{2}}+F(U^{n+1})\right]=U^{n}+\Delta t(1-q)\left[\frac{\partial^{2}U^{n}}{\partial x^{2}}+\frac{\partial^{2}U^{n}}{\partial y^{2}}+F(U^{n})\right]$$

Ecuación 8

Dado que **F** no depende de t, la Ecuación 6 puede ser linealizada mediante un método implícito donde el término no lineal $F(U^{n+1})$ se aproxima mediante su polinomio de Taylor de primer grado para obtener el siguiente sistema de ecuaciones lineales donde la variable U se ha descompuesto en sus dos componentes:

$$\frac{\Delta u_{i,j}}{\Delta t} = \frac{d_x^{\,2} u_{i,j}^n}{h^2} + q_x \frac{d_x^{\,2} \Delta u_{i,j}}{h^2} + \frac{d_y^{\,2} u_{i,j}^n}{h^2} + q_y \frac{d_y^{\,2} \Delta u_{i,j}}{h^2} - u_{i,j}^n v_{i,j}^n - d\ u_{i,j}^n \Delta v_{i,j} - d\ v_{i,j}^n \Delta u_{i,j}$$

$$\frac{\Delta v_{i,j}}{\Delta t} = \frac{d_x^{\,2} v_{i,j}^n}{h^2} + q_x \frac{d_x^{\,2} \Delta v_{i,j}}{h^2} + \frac{d_y^{\,2} v_{i,j}^n}{h^2} + q_y \frac{d_y^{\,2} \Delta v_{i,j}}{h^2} + u_{i,j}^n v_{i,j}^n - K v_{i,j}^n +$$
$$+ d\,(u_{i,j}^n \Delta v_{i,j} + v_{i,j}^n \Delta u_{i,j} - K \Delta v_{i,j})$$

Ecuación 9

donde $\Delta t = t^{n+1} - t^n$ es el escalón de tiempo, $h = \Delta x = \Delta y$ es el tamaño del escalón en el espacio, $\Delta u_{i,j} = u_{i,j}^{n+1} - u_{i,j}^n$,

$d_x^{\,2} u_{i,j}^n = u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n$ y $d_y^{\,2} u_{i,j}^n = u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n$ son las aproximaciones en diferencias centrales con respecto a ***x*** y a ***y***.

Los valores de $d = q_x = q_y = 0.5$ se corresponden a un método implícito de segundo orden de exactitud en el tiempo.

El sistema de la Ecuación 9 se puede expresar de la siguiente forma:

LHS(u)= RHS(u)

LHS(v)= RHS(v) **Ecuación 10**

donde:

$$\text{LHS(u)} = -q_x \frac{k}{h^2} \Delta u_{i-1,j} + \Delta u_{i,j}\left(1 + 2q_x \frac{k}{h^2} + 2q_y \frac{k}{h^2} + d\ k v_{i,j}^n\right) - q_x \frac{k}{h^2} \Delta u_{i+1,j}$$
$$- q_y \frac{k}{h^2} \Delta u_{i,j-1} - q_y \frac{k}{h^2} \Delta u_{i,j+1} + d\ k u_{i,j}^n \Delta v_{i,j}$$

$$\text{LHS(v)} = -q_x \frac{k}{h^2} \Delta v_{i-1,j} + \Delta v_{i,j}\left(1 + 2q_x \frac{k}{h^2} + 2q_y \frac{k}{h^2} - d\ k u_{i,j}^n + kd\ K\right) - q_x \frac{k}{h^2} \Delta v_{i+1,j}$$
$$- q_y \frac{k}{h^2} \Delta v_{i,j-1} - q_y \frac{k}{h^2} \Delta v_{i,j+1} - d\ k v_{i,j}^n \Delta u_{i,j}$$

$$\text{RHS(u)} = \frac{k}{h^2}(u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n) + \frac{k}{h^2}(u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n) - k u_{i,j}^n v_{i,j}^n$$

$$\text{RHS(v)} = \frac{k}{h^2}(v_{i+,j}^n - 2v_{i,j}^n + v_{i-1,j}^n) + \frac{k}{h^2}(v_{i,j+1}^n - 2v_{i,j}^n + v_{i,j-1}^n) + k u_{i,j}^n v_{i,j}^n - kK v_{i,j}^n$$

Ecuación 11

El inconveniente de la Ecuación 11 es que, al aplicarla, se producen errores de linealización. Una forma de eliminar estos errores consiste en resolver el dominio irregular completo mediante iteraciones de Newton–Raphson, como se muestra en el siguiente apartado. Sus resultados se han utilizado como referencia para medir los errores cometidos con el resto de los métodos estudiados. En los apartados siguientes, se muestran los métodos de descomposición de dominios. Para ello, en cada dominio, se ha empleado la Ecuación 11 ya que permite una implementación rápida y fácil de la solución interior de cada dominio. A continuación, se muestran los métodos empleados para realizar la descomposición de dominios. Aunque el problema considerado aquí presenta gradientes pronunciados en movimiento, las mismas ecuaciones gobiernan el fenómeno físico en el dominio global y las mallas son iguales de finas en cada dominio.

Como se verá en los apartados que siguen, los métodos de descomposición de dominios aprovechan bien la geometría del problema, siendo, además, apropiados para su paralelización. Sin embargo, los métodos sin superposición (apartado 6.5.2) presentan grandes errores debido, entre otras razones, a las singularidades que suponen las esquinas. Para evitar estos errores se emplean los métodos de descomposición de dominios con superposición (apartado 6.5.3).

6.5.1 Resolución mediante Newton-Raphson.

Para disminuir los errores cometidos en la linealización de la Ecuación 6, se aplica el método iterativo de Newton-Raphson.

Llamando *RHS* al término derecho de la igualdad de la Ecuación 8, es decir

$$RHS = \mathrm{U}^{n} + \Delta t(1-q)\left[\frac{\partial^{2}\mathrm{U}^{n}}{\partial \mathrm{x}^{2}} + \frac{\partial^{2}\mathrm{U}^{n}}{\partial \mathrm{y}^{2}} + \mathrm{F}(\mathrm{U}^{n})\right] \qquad \textbf{Ecuación 12}$$

y tomando el superíndice *k* como indicador de iteración, se obtiene la siguiente expresión:

$$\mathrm{U}^{k+1} - \Delta tq\left[\frac{\partial^{2}\mathrm{U}^{k+1}}{\partial \mathrm{x}^{2}} + \frac{\partial^{2}\mathrm{U}^{k+1}}{\partial \mathrm{y}^{2}} + \mathrm{F}(\mathrm{U}^{k+1})\right] = RHS\ ;$$

$$\mathrm{F}(\mathrm{U}^{k+1}) = \mathrm{F}(\mathrm{U}^{k}) + \mathrm{J}(\mathrm{U}^{k})(\mathrm{U}^{k+1} - \mathrm{U}^{k})\ ;\ \text{donde}\ \mathrm{J} = \frac{\partial \mathrm{F}}{\partial \mathrm{U}}\text{, por tanto}$$

$$U^{k+1} - \Delta tq\left[\frac{\partial^2 U^{k+1}}{\partial x^2} + \frac{\partial^2 U^{k+1}}{\partial y^2} + F(U^k) + J(U^k)(U^{k+1} - U^k)\right] = RHS$$

Tomando $\Delta U^{k+1} = U^{k+1} - U^n$, y, por tanto, $U^{k+1} = \Delta U^{k+1} + U^n$ se tiene que:

$$\Delta U^{k+1} + U^n - \Delta tq\left[\frac{\partial^2 U^n}{\partial x^2} + \frac{\partial^2 U^n}{\partial y^2}\right]$$
$$- \Delta tq\left[\frac{\partial^2 \Delta U^{k+1}}{\partial x^2} + \frac{\partial^2 \Delta U^{k+1}}{\partial y^2} + F(U^k) + J(U^k)(\Delta U^{k+1} + U^n - U^k)\right] = RHS$$

$$\Delta U^{k+1} - \Delta tq\left[\frac{\partial^2 \Delta U^{k+1}}{\partial x^2} + \frac{\partial^2 \Delta U^{k+1}}{\partial y^2} + J(U^k)(\Delta U^{k+1})\right]$$
$$= RHS - U^n + \Delta tq\left[\frac{\partial^2 U^n}{\partial x^2} + \frac{\partial^2 U^n}{\partial y^2} + F(U^k) - J(U^k)\Delta U^k\right]$$

Sustituyendo *RHS* por su valor se tiene que:

$$\Delta U^{k+1} - \Delta tq\left[\frac{\partial^2 \Delta U^{k+1}}{\partial x^2} + \frac{\partial^2 \Delta U^{k+1}}{\partial y^2} + J(U^k)\Delta U^{k+1}\right]$$
$$= \Delta t\left[\frac{\partial^2 U^n}{\partial x^2} + \frac{\partial^2 U^n}{\partial y^2} + q\ F(U^k) + (1-q\ )\,F(U^n) - q\ J(U^k)\Delta U^k\right]. \qquad \textbf{Ecuación 13}$$

Este sistema de ecuaciones ha sido resuelto mediante un solo dominio resolviendo el sistema algebraico lineal $A\ \Delta U = RHS$. Para establecer los valores de la matriz A en un dominio irregular, se ha considerado que el dominio está dividido en 4 regiones (*Figura 43*) de las cuales, las regiones 1 y 2 se corresponden con los dominios Ω_1 y Ω_2 de la *Figura 42*, respectivamente. La matriz A se corresponde por bloques con la matriz identidad en los puntos englobados en las regiones 3 y 4, incluyendo los puntos frontera entre las regiones 2-3, 2-4, 1-3 y 1-4. El método utilizado para la resolución de este problema algebraico lineal es el conocido gradiente biconjugado estabilizado (*BiCGSTAB*) cuya descripción detallada puede ser encontrada en [Barret y otros 97]. Aunque la Ecuación 13 se corresponde con un sistema grande de ecuaciones, se trata de sistemas de ecuaciones donde la matriz de coeficientes es dispersa. De hecho, la matriz de $2n^2$ de cada uno de los dominios puede ser almacenada en una matriz de 2n x 6, donde n es el número de puntos interiores.

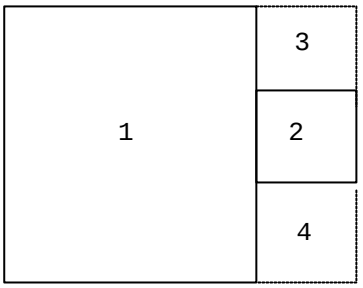


Figura 43. Para resolver el sistema se añaden las regiones 3 y 4 de forma que el dominio pasa a tener una geometría regular.

El método de Newton-Raphson para este dominio tiene el inconveniente de tener que resolver los dominios comprendidos en las regiones 3 y 4, lo cual conlleva un mayor coste tanto en memoria como en tiempo de computo. Además, a las iteraciones del método iterativo del sistema lineal hay que añadir las iteraciones de Newton.

Utilizando este método se ha resuelto la Ecuación 6 con los siguientes valores iniciales para el primer dominio:

$$u(x,\ y,0) = 1 \qquad v(x,\ y,0) = e^{-a\left((x-x_{ign})^2+(y-y_{ign})^2\right)} \qquad a = 1 \qquad -20 \le x \le 20 \qquad -20 \le y \le 20$$

siendo x_{ign}, y_{ign} las coordenadas del punto donde comienza la ignición. Para el segundo

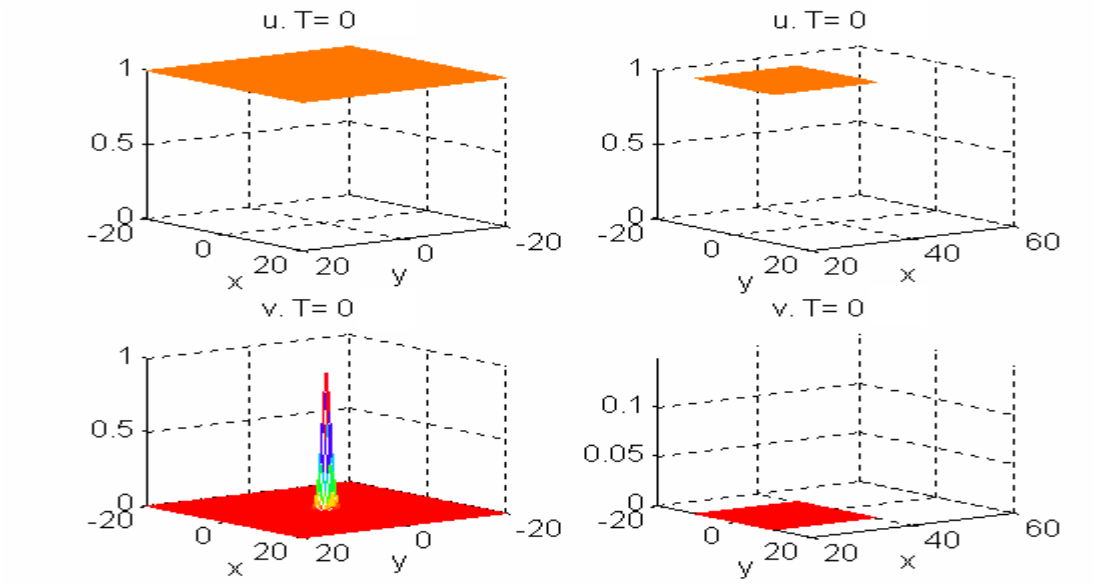


Figura 44. Condiciones iniciales para un dominio con esquinas. Aunque la solución se ha obtenido sin descomposición de dominios, se muestran las variables u *(arriba) y* v *(abajo) separadas en dos gráficas cada una. De esta forma será más fácil explicar los resultados cuando se aplique la descomposición de dominios. Obsérvese que el eje* y *es el mismo para ambos y que el eje* x *del dominio 2 es continuación del eje x del dominio 1.*

dominio se ha considerado: $20 \le x \le 40, -10 \le y \le 10$. Los resultados se muestran desde la *Figura 44* a la *Figura 48*. En estas figuras se ha hecho una rotación del segundo dominio para que pueda observarse mejor la interfaz entre ambos (obsérvese la posición de los ejes X e Y).

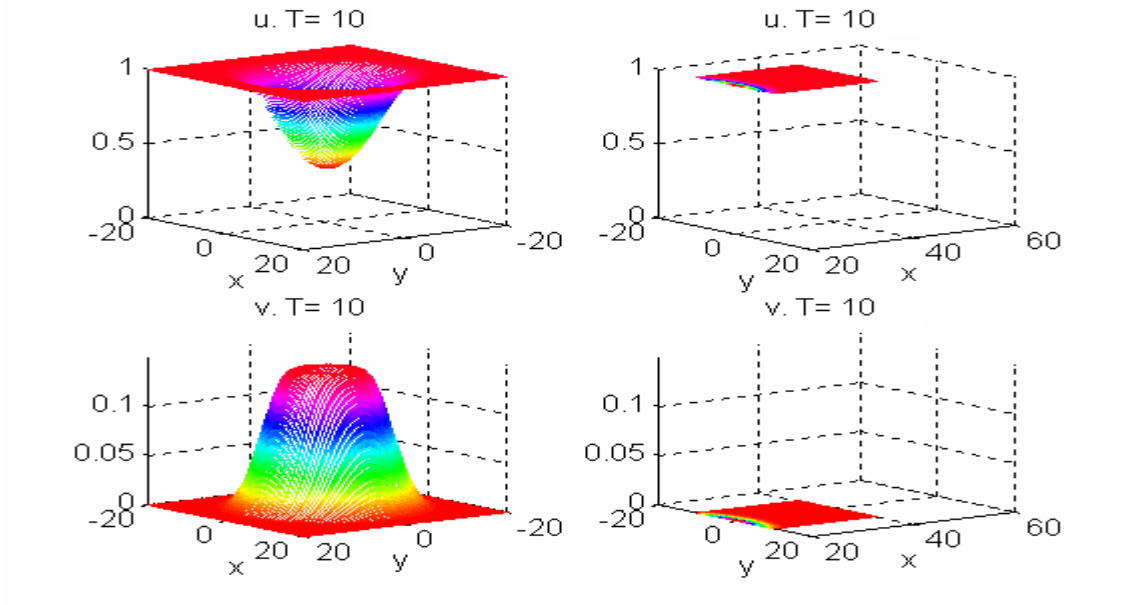


Figura 45. Cuando t = 10, en el dominio 2 todavía no hay reacción, mientras que en el 1 ya es significativa. Obsérvese como la temperatura en el centro del primer dominio (abajo a la izquierda) va disminuyendo debido a que la concentración de combustible en el centro del dominio es menor.

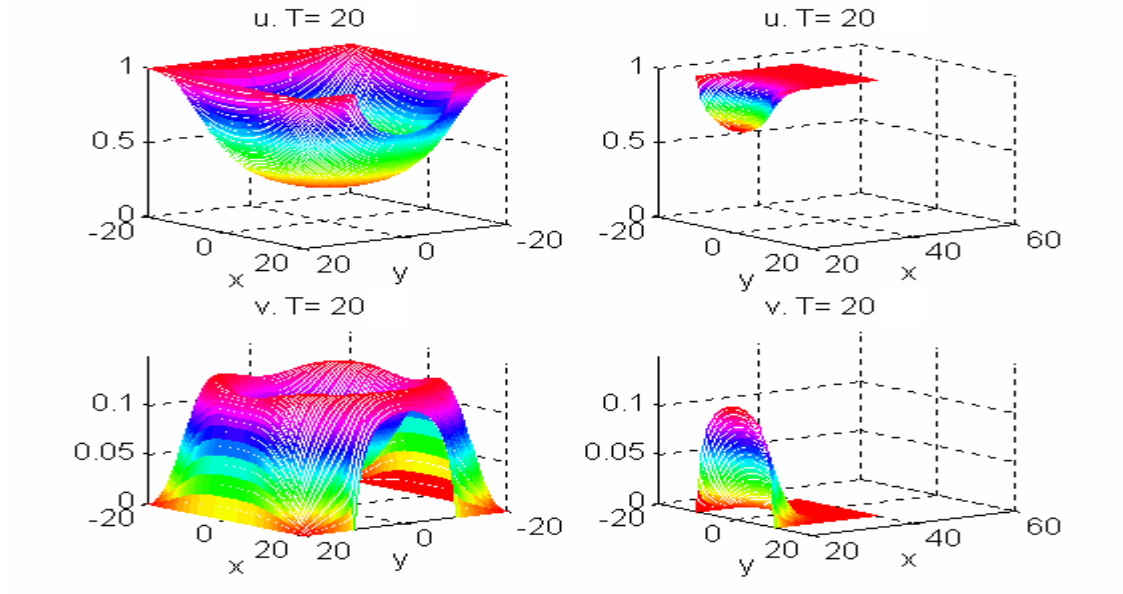


Figura 46. Para t = 20 ya se aprecian cambios en el dominio 2. Este es el momento en el que el frente alcanza la frontera entre los dominios, comenzando la reacción en el segundo de ellos.

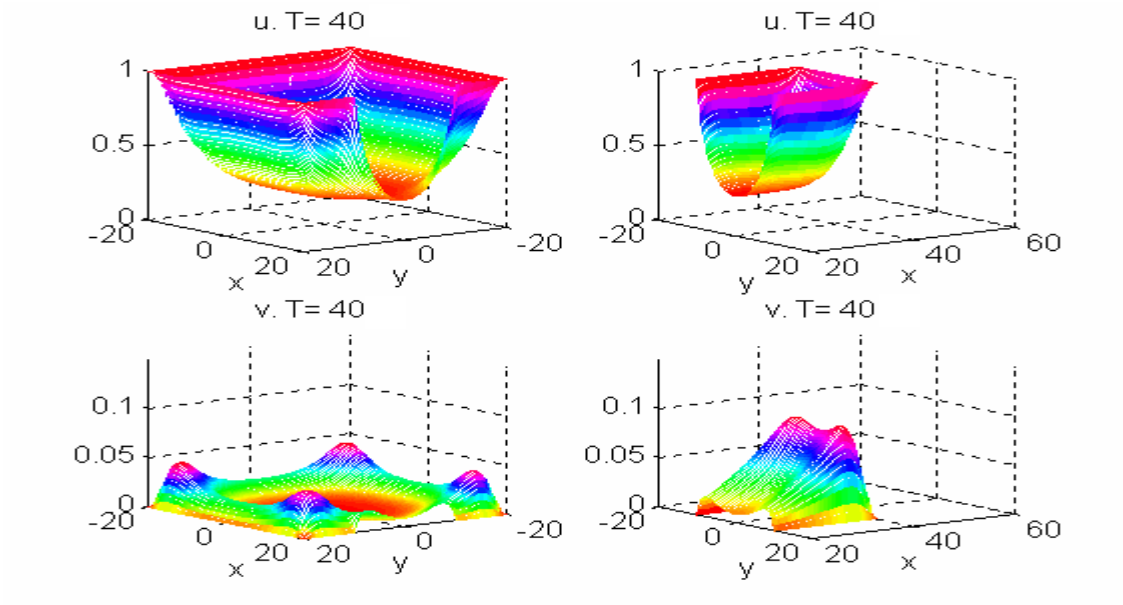


Figura 47. Para t = 40, la reacción está teniendo lugar en el dominio de la derecha, observándose ahora valores más altos de temperatura en las 4 esquinas del dominio de la izquierda debido a que las condiciones de Dirichlet mantienen alto el nivel de combustible en estos puntos.

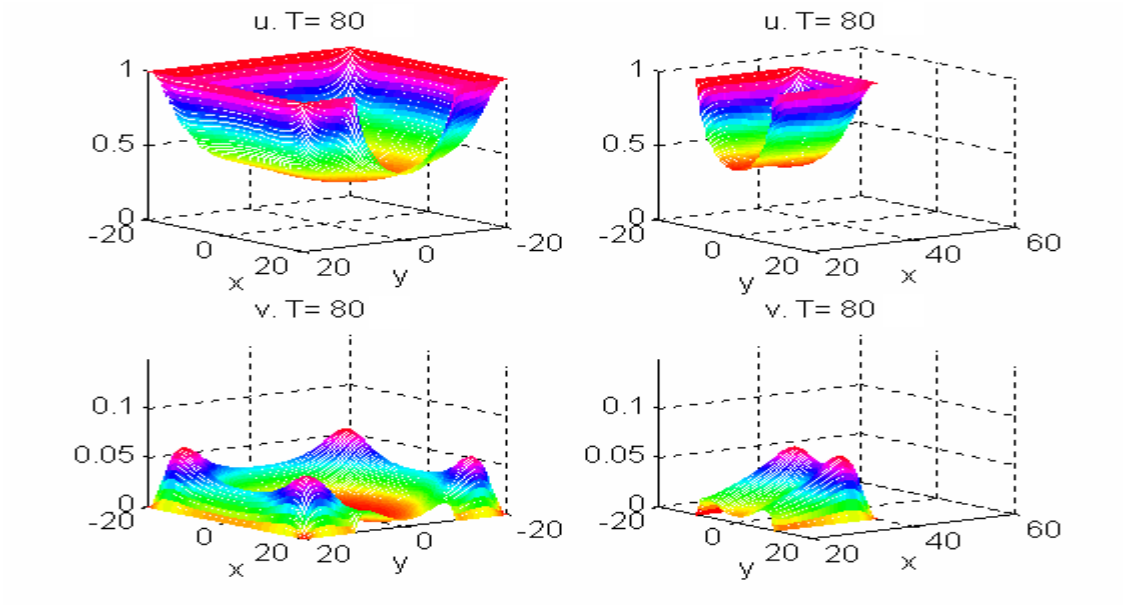


Figura 48. Finalmente, cuando t = 80 se pueden apreciar para la variable v, las 6 esquinas que corresponden a las 6 esquinas de la geometría. Estas son debidas a las condiciones de Dirichlet impuestas en el contorno.

6.5.2 Descomposición de dominios sin superposición.

Los dominios tratados aquí se denominan respectivamente Ω_1 y Ω_2 de manera que $\Omega_1 \cap \Omega_2 = \sum$. La solución en cada dominio se ha obtenido resolviendo las ecuaciones algebraicas lineales sujetas a las condiciones apropiadas a la interfaz común ($\sum$ en la *Figura 42*). Estas condiciones de frontera son tales que tanto la solución como su derivada normal a la frontera deben ser continuas, pero su implementación resulta en métodos diferentes, como se observa a continuación.

En lo sucesivo denominaremos *u* y *v* a los valores de U en Ω_1 y Ω_2 respectivamente para facilitar la descripción de los métodos empleados.

Método de Dirichlet. En este método, las ecuaciones algebraicas se resuelven iterativamente en los puntos interiores de los dos subdominios Ω_1 y Ω_2 usando los valores de U en $\sum$; inicialmente estos valores se corresponden a los del paso de tiempo anterior, pero son actualizados en las iteraciones sucesivas imponiendo que la derivada de la solución normal a la interfaz sea continua. Así, los valores de U en la interfaz deben ser determinados imponiendo que $\frac{\partial u}{\partial x} = \frac{\partial v}{\partial x}$, lo cual puede ser discretizado mediante diferencias finitas de primer orden de exactitud de la siguiente forma:

$$p_L = \frac{u_\Sigma - u_{\Sigma-1}}{\Delta x_1} = p_R = \frac{v_{\Sigma+1} - v_\Sigma}{\Delta x_2}$$

Ecuación 14

donde $U_\Sigma = u_\Sigma = v_\Sigma$ y Σ-1 y Σ +1 denotan las líneas de la malla más cercanas a la interfaz a su izquierda y derecha, respectivamente.

En lugar de usar diferencias finitas de primer orden de exactitud para las derivadas en la interfaz, que degradan la exactitud del método de diferencias finitas de segundo orden en los puntos interiores, se pueden utilizar diferencias finitas de segundo orden para estas derivadas, con lo cual resulta la siguiente expresión:

$$p_L = \frac{3u_\Sigma - 4u_{\Sigma-1} + u_{\Sigma-2}}{2\Delta x_1} = p_R = \frac{4v_{\Sigma+1} - v_{\Sigma+2} - 3v_\Sigma}{2\Delta x_2}$$

Ecuación 15

con esto se tiene que $\Delta U_{\Sigma} = \frac{1}{6}(4\Delta v_{\Sigma+1} - \Delta v_{\Sigma+2} + 4\Delta u_{\Sigma-1} - \Delta u_{\Sigma-2})$ **Ecuación 16**

si $\Delta x_1 = \Delta x_2$.

En ambos casos, se puede determinar U_{Σ} en términos de los valores de U en los puntos interiores de los dos dominios sin superposición, una vez que se obtiene la solución en ambos, y usar el nuevo valor en la siguiente iteración. Este procedimiento iterativo se repite hasta que la norma de la diferencia entre los valores calculados para la frontera sea menor que un cierto ε, es decir, $\left\| u_{\Sigma}^{k+1} - u_{\Sigma}^{k} \right\| \leq \mathrm{e}$. La norma empleada es la raíz cuadrada de la suma de los cuadrados. El método descrito aquí se ha denominado **D1** y **D2** dependiendo de si se usa el método de primer o segundo orden de exactitud en la fórmula para calcular la derivada a la normal de la interfaz entre los dominios.

Método de Neumann. Este método también impone la continuidad de ambas soluciones y de su derivada normal a la interfaz, al igual que el método de Dirichlet, pero, en lugar de usar $u_{\Sigma} = v_{\Sigma}$ cuando se resuelven ambos dominios, se emplea $\frac{\partial u_{\Sigma}}{\partial x} = \frac{\partial v_{\Sigma}}{\partial x}$ y se determina U_{Σ} usando la expresión de primer o segundo orden para las derivadas en la frontera dadas por la Ecuación 14 o la Ecuación 15, respectivamente. Dado que las derivadas normales a la interfaz son continuas, se pueden escribir los valores de las incógnitas en la interfaz como funciones de estas derivadas y reemplazar sus valores en las ecuaciones en diferencias finitas obtenidas en el método anterior. De esta manera, uno puede obtener la solución en todos los puntos interiores de Ω_1 y Ω_2 y determinar u_{Σ} y v_{Σ} de la Ecuación 14 o la Ecuación 15. Ya que durante el proceso iterativo, $u_{\Sigma} \neq v_{\Sigma}$, un valor actualizado de U_{Σ} se determina mediante $U_{\Sigma} = \mu\, u_{\Sigma} + (1-\mu)\, v_{\Sigma}$ con $0 \leq \mu \leq 1$ y este valor se usa para determinar las derivadas normales a la interfaz para la siguiente iteración. Este procedimiento iterativo se repite hasta que se alcanza el criterio de convergencia previamente especificado. En los cálculos presentados aquí, el valor de $\mu = 0.5$ y a estos métodos se les ha denominado **No1** y **No2** dependiendo de si se usa el método de primer o segundo orden de exactitud en la fórmula para calcular la derivada normal a la interfaz entre los dominios.

Hay que hacer notar que la Ecuación 14 y la Ecuación 15 se pueden escribir también de la siguiente manera:

$$p = \frac{v_{\Sigma+1} - u_{\Sigma-1}}{2\Delta x}$$

Ecuación 17

$$p = \frac{4v_{\Sigma+1} - 4u_{\Sigma-1} + u_{\Sigma-2} - v_{\Sigma+2}}{4\Delta x}$$

Ecuación 18

si $\Delta x = \Delta x_1 = \Delta x_2$, lo cual se basa en la continuidad de la solución y de su derivada normal a la interfaz entre los dominios.

En lugar de determinar los valores de la interfaz una vez que los puntos interiores han sido calculados, se pueden incluir éstos en ambos subdominios como una función de la derivada normal a la frontera y, así, obtener un sistema de ecuaciones mayor que el de los métodos **No1** y **No2**. Una vez que la solución de este sistema se obtiene en cada subdominio, *p* es relajado para la siguiente iteración, como se describió en el párrafo anterior, y el procedimiento iterativo se repite hasta que el criterio de convergencia descrito anteriormente se satisface. Este método lo denominamos **N1** y **N2** dependiendo de si se usa el método de primer o segundo orden de exactitud en la fórmula para calcular la derivada normal a la interfaz entre los dominios y requiere una modificación sencilla relativa a la multiplicación matriz-vector en el método *BiCGSTAB* cuando la derivada normal a la interfaz es evaluada mediante las fórmulas de diferencias finitas de segundo orden porque, en este caso, u_{Σ} depende de $u_{\Sigma\text{-}1}$ y $u_{\Sigma\text{-}2}$, mientras que v_{Σ} depende de $v_{\Sigma+1}$ y $v_{\Sigma+2}$, y se debe introducir una nueva diagonal en la matriz de coeficientes.

Método de Robin. En este método, la continuidad de la función y de su derivada normal en la interfaz se imponen mediante el uso de las condiciones de Robin:

$$\frac{\partial u}{\partial x} + l\ u = \frac{\partial v}{\partial x} + l\ v \text{ en } \Sigma$$

Ecuación 19

donde l es una constante, las derivadas se pueden evaluar mediante formulas de primer o segundo orden de exactitud y la solución en la interfaz puede ser determinada una vez que los puntos interiores se conocen o junto con los puntos interiores.

Si se discretizan las derivadas de primer orden mediante diferencias finitas de primer orden, entonces

$$\Delta u_{\Sigma}^{k+1} = \frac{1}{1+ l\Delta x}(\Delta u_{\Sigma-1}^{k+1} + \Delta x b_R^k),$$
$$\Delta v_{\Sigma}^{k+1} = \frac{1}{1- l\Delta x}(\Delta v_{\Sigma+1}^{k+1} - \Delta x b_L^k)$$

Ecuación 20

donde k denota la k-ésima iteración dentro del paso de tiempo y

$$b_R = l\ \Delta v_{\Sigma} + \frac{1}{\Delta x}(\Delta v_{\Sigma+1} - \Delta v_{\Sigma}),$$
$$b_L = l\ \Delta u_{\Sigma} + \frac{1}{\Delta x}(\Delta u_{\Sigma} - \Delta u_{\Sigma-1})$$

Ecuación 21

La Ecuación 20 se puede utilizar para eliminar Δu_{Σ} y Δv_{Σ} de los subdominios Ω_1 y Ω_2, respectivamente, y las ecuaciones resultantes pueden ser resueltas en los puntos interiores. Así, la Ecuación 20 se puede usar para determinar la solución en la interfaz. Sin embargo, ya que durante el proceso iterativo, $u_{\Sigma} \neq v_{\Sigma}$, los valores actualizados de U_{Σ} y b se determinan como

$$U_{\Sigma} = m\,\Delta u_{\Sigma} + (1-m)v_{\Sigma} \quad \textbf{Ecuación 22}$$
$$b = m\,b_L + (1-m)b_R) \quad \textbf{Ecuación 23}$$

con $0 \leq m \leq 1$, y estos valores se utilizan para determinar las derivadas normales a la interfaz para la próxima iteración. Este procedimiento iterativo se repite hasta que el criterio de convergencia se satisface. En los cálculos presentados en esta tesis, $m = 0.5$, y se ha denominado a los métodos **Ro1** y **Ro2** dependiendo de si se usa el método de primer o segundo orden de exactitud, respectivamente, en la fórmula para calcular la derivada normal a la interfaz entre los dominios.

En lugar de determinar los valores de la interfaz una vez que los puntos interiores han sido calculados, se pueden incluir estos en ambos subdominios como una función de b_L y b_R y, así, obtener un sistema de ecuaciones mayor que el de los métodos **Ro1** y **Ro2**. Una vez que la solución de este sistema se obtiene en cada subdominio, b es relajado para la siguiente iteración, como se describió en el párrafo anterior. Este método lo denominamos **R1** y **R2** dependiendo de si se usa el método de primer o segundo orden de exactitud en la fórmula para calcular la derivada normal a la interfaz entre los dominios y requiere una modificación sencilla relativa a la multiplicación matriz-vector en el método *BiCGSTAB* cuando la derivada normal a la interfaz es

evaluada mediante las fórmulas de diferencias de segundo orden, como se explicó anteriormente para el método de Neumann.

Método de Dirichlet-Neumann. Este método es una combinación de los dos primeros métodos descritos y consiste en los siguientes pasos para cada iteración: Utilizando los valores iniciales para la frontera, se obtiene la solución en los puntos interiores de los subdominios y, mediante la continuidad de la derivada de primer orden en la frontera y una fórmula de segundo orden de exactitud, se puede calcular U_Σ como en el método de Dirichlet (Ecuación 16). Este valor, junto con $u_{\Sigma-1}$, $u_{\Sigma-2}$, $v_{\Sigma+1}$, $v_{\Sigma+2}$ se pueden utilizar entonces para calcular una aproximación de segundo orden de exactitud a la derivada normal a la frontera en ambos subdominios (Ecuación 15). La media aritmética de estas derivadas puede ser utilizada para resolver el problema de Neumann resultante, aplicándose entonces la Ecuación 16 de nuevo para calcular los valores en la frontera. Este procedimiento consiste así, en un ciclo de Dirichlet y otro de Neumann por iteración y se repite tantas veces como sea necesario hasta que el criterio de convergencia previamente especificado se alcance. A este método se le ha denominado **DN1** y **DN2** dependiendo de si se usa el método de primer o segundo orden de exactitud en la fórmula para calcular la derivada normal a la interfaz entre los dominios, respectivamente.

Método de Neumann-Dirichlet. Este método es similar al de Dirichlet-Neumann pero primero emplea un ciclo de Neumann donde las derivadas normales a la interfaz se especifican para determinar la solución en la interfaz entre los dominios. Estos valores se usan entonces para determinar una media según la Ecuación 16, y la solución se obtiene entonces por medio del método de Dirichlet. La solución del método de Dirichlet se usa para determinar de nuevo los valores en la frontera con la Ecuación 16. Este procedimiento consiste así en un ciclo de Dirichlet y otro de Neumann por iteración y se repite tantas veces como sea necesario hasta que el criterio de convergencia previamente especificado sea satisfecho. A este método se le ha denominado **ND1** y **ND2** dependiendo de si se usa el método de primer o segundo orden de exactitud en la fórmula para calcular la derivada normal a la interfaz entre los dominios, respectivamente.

Método de Yang. Este método ha sido desarrollado por [Yang 96] y consiste también en dos pasos por iteración. En el primer paso, un subdominio es resuelto con condiciones de frontera de Dirichlet en la interfaz, mientras que el otro subdominio es resuelto con condiciones de Neumann. En el segundo paso, el subdominio que fue resuelto con condiciones de Dirichlet se resuelve ahora con condiciones de Neumann y viceversa. Este procedimiento se repite tantas veces como sea necesario hasta que el criterio de convergencia previamente especificado sea satisfecho. A este método se le ha denominado **Y1** e **Y2** dependiendo de si se usa el método de primer o segundo orden de exactitud en la fórmula para calcular la derivada normal a la interfaz entre los dominios, respectivamente.

6.5.3 Descomposición de dominios con superposición.

El algoritmo de descomposición de dominios con superposición de Schwarz [Schwarz 90] se está aplicando actualmente a una gran variedad de problemas, especialmente a la hora de sacar partido al paralelismo [Bjørstad y Karstad 95][Cai 95]. Este método puede ser aplicado a un dominio irregular como el de la *Figura 42* tomando los dominios que se muestran en la *Figura 49*.

Los algoritmos resuelven los puntos interiores de los dominios Ω_1 y Ω_2 independientemente. Denominamos la interfaz entre Ω_1 y de Ω_2 como $\sum_1$ y la línea entre las dos esquinas reentrantes como $\sum$. Sea N_o el número de puntos en la dirección *x* que tienen el dominio Ω_1 y el Ω_2 en común (al menos dos puntos), de modo que $U_\Sigma = u_N = v_{N_o}$. Tres procedimientos iterativos se han utilizado para obtener el sistema de ecuaciones lineales algebraicas que se han resuelto con el método *BiCGSTAB.*

Método de Dirichlet. En este método, las ecuaciones en Ω_1 en t^{n+1} podrían ser resueltas primero con los valores de las incógnitas en el instante t^n en $\sum$ y los valores obtenidos así en $\sum_1$ ser empleados entonces para obtener la solución en Ω_2. Este procedimiento sería análogo a la técnica iterativa de Gauss-Seidel, que no es paralelizable. Para obtener métodos de descomposición de dominios con superposición que se puedan implementar fácilmente en arquitecturas paralelas, se ha usado un método iterativo de relajación de bloques.

Dado que el valor de u_N no se ha calculado en el dominio 1 (se toman condiciones de Dirichlet), se asume que el valor correcto en la frontera es el calculado en el dominio 2 (donde sí se ha calculado al ser puntos internos). Es decir, se actualizan los valores de la frontera de u_N con los de v_{No}.

Para el segundo dominio se hace lo propio, actualizándose los puntos frontera del dominio 2 que están dentro del dominio 1 (Σ_1) con los valores que se han calculado para el dominio 1 (estos valores no se actualizan en el dominio 1 puesto que aún no se ha avanzado en el tiempo) y se itera de nuevo. Cuando la norma de la diferencia entre los valores calculados en la frontera en dos iteraciones consecutivas es menor que una cierta tolerancia ε, se avanza al siguiente escalón de tiempo, actualizando todos los valores del dominio 2 que están dentro del 1 con los valores calculados en el dominio 1. A este método se le ha denominado **OD**.

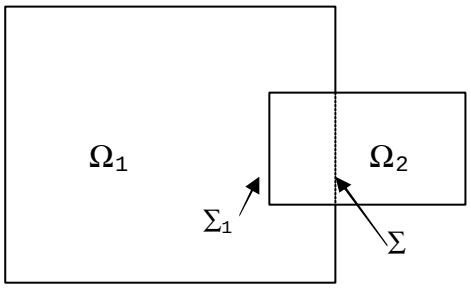


Figura 49. Representación de los dominios para una resolución mediante descomposición de dominios con superposición.

Método de Neumann. Este método es análogo al usado para los dominios sin superposición, excepto que la interfaz es resuelta en ambos subdominios de modo que, cuando la derivada de primer orden en la frontera se aproxima por medio de diferencias finitas de primer orden hacia delante y hacia atrás en Ω_1 y Ω_2, respectivamente, los valores de $v_{\Sigma-1}$ y $u_{\Sigma+1}$ requeridos en la solución de los dominios, se reemplazan por $u_\Sigma + \Delta xp$ y $v_\Sigma - \Delta xp$, respectivamente. Nótese que aquí el número de puntos en común es dos. La solución obtenida del sistema de ecuaciones resultante se emplea para determinar la derivada normal a la interfaz mediante la Ecuación 17, la cual es la media aritmética de sus derivadas a la izquierda y derecha de la interfaz. Este método también es iterativo y se ha denominado **ON1** porque emplea diferencias de primer orden para la evaluación de la derivada normal a la frontera.

Si en lugar de utilizar la Ecuación 14 y la Ecuación 17 se usa la Ecuación 15 y la Ecuación 18 para obtener diferencias finitas de segundo orden de exactitud, el método resultante se denomina **ON2** y emplea diferencias hacia delante y hacia atrás para evaluar la derivada de primer orden en la interfaz entre los dominios Ω_1 y Ω_2, respectivamente.

Método de Robin. Este método es análogo al empleado sin superposición excepto que la interfaz es resuelta en ambos subdominios de modo que la discretización de la Ecuación 19 usando diferencias hacia delante y hacia atrás en el dominio de la izquierda y derecha conlleva a:

$$\Delta u_{\Sigma+1} = \Delta x\, b_R + (1 - l\ \Delta x)\Delta u_{\Sigma}$$

Ecuación 24

$$\Delta v_{\Sigma-1} = -\Delta x\, b_L + (1 + l\ \Delta x)\Delta v_{\Sigma}$$

Ecuación 25

donde:

$$b_R = l\ \Delta v_{\Sigma} + \frac{1}{\Delta x}(\Delta v_{\Sigma} - \Delta u_{\Sigma-1})$$

Ecuación 26

$$b_L = l\ \Delta u_{\Sigma} + \frac{1}{\Delta x}(\Delta v_{\Sigma+1} - \Delta u_{\Sigma})$$

Ecuación 27

La sustitución de la Ecuación 24 y de la Ecuación 25 en las ecuaciones algebraicas correspondientes a la interfaz para ambos dominios permite obtener la solución en todos los puntos, incluidos los de la frontera. Estos valores pueden ser usados entonces para obtener nuevos valores para b_{L} y b_{R}, los cuales se pueden ahora utilizar para obtener un valor relajado para la nueva iteración como sigue:

$$b = \frac{l}{2}\left(\Delta u_{\Sigma} + \Delta v_{\Sigma}\right) + \frac{1}{2\Delta x}\left(\Delta v_{\Sigma+1} - \Delta v_{\Sigma} + \Delta u_{\Sigma} - \Delta u_{\Sigma-1}\right),$$

Ecuación 28

$$b = \frac{l}{2}\left(\Delta u_{\Sigma} + \Delta v_{\Sigma}\right) + \frac{1}{4\Delta x}\left(4\Delta v_{\Sigma+1} - 3\Delta v_{\Sigma+2} - 3\Delta v_{\Sigma}\right) + \frac{1}{4\Delta x}\left(3\Delta u_{\Sigma} - 4\Delta u_{\Sigma-1} + \Delta u_{\Sigma-2}\right)$$

Ecuación 29

donde la Ecuación 28 y la Ecuación 29 se corresponden a diferencias de primer y segundo orden, respectivamente, para las derivadas de primer orden y los métodos

resultantes de la aplicación de estas fórmulas se han denominado **OR1** y **OR2**, respectivamente.

6.5.4 Presentación de resultados.

La Tabla 6 y la Tabla 7 muestran un cuadro resumido con los resultados más relevantes de todas las pruebas que se han realizado y cuya presentación exhaustiva se muestra en [Ramos, Soler y Troya 98]. El tamaño de la malla del primer dominio es de 130 x 130 mientras que para el segundo dominio es de 66 x 66. La tolerancia del método *BiCGSTAB* se fijó en 10^{-12}, el criterio de convergencia ε se fijo en 10^{-10} y los cálculos se realizaron hasta que t = 80, cuando la solución es prácticamente estacionaria. Los tiempos de ejecución mostrados corresponden a una CPU DEC Alpha Server 22164 a 300 MHz. Los errores mostrados son los errores relativos respecto a la solución obtenida empleando un solo dominio mediante el método de Newton-Raphson explicado previamente.

Método	Máx. Error	Tiempo Ejecución	Iteraciones BiCGSTAB
D1	3.2300e-1	5703	159344
D2	1.8550e-1	6121	169196
No1	3.2258e-1	4442	121196
No2	1.8548e-1	4384	121087
N1	3.2299e-1	8242	191479
N2	1.8550e-1	9346	228414
Ro2 l=1	1.8548e-1	7540	145057
Ro2 l=1000	1.8550e-1	7224	135069
R2 l=1	1.8550e-1	11629	309933
DN2	1.8548e-1	8801	198426
ND2	1.8550e-1	8442	192114
Y2	1.8548e-1	7855	195894

Tabla 6. Errores máximos, tiempos de ejecución e iteraciones del método BiCGSTAB para los métodos sin superposición.

La Tabla 6 muestra los resultados para los métodos de descomposición de dominios sin superposición. En estos, la exactitud de los métodos de Dirichlet y Neumann aumentan con la exactitud de la discretización de la derivada de primer orden normal a la interfaz. La exactitud del método de Dirichlet es similar a la del de Neumann pero estos métodos no producen errores idénticos. **No1** conlleva resultados que difieren en el quinto decimal de aquellos de **N1**. Aunque no se muestran aquí, **Ro1** y **R1** conllevan resultados que difieren en el séptimo decimal con respecto a los de **N1**;

Ro1 y **Ro2** obtuvieron resultados que difieren en el error relativo en, al menos, el sexto decimal para los valores de l = 0, 1, 10, 10^2, 10^3 y 10^{30} cuando las medias aritméticas de los resultados para los dominios de la izquierda y derecha se usan para actualizar la interfaz en la siguiente iteración; **R1** y **R2** obtuvieron cada uno resultados idénticos para l = 0 y 1 pero estos métodos no llegaron a converger para $l \geq 10$; para **DN2**, se obtuvieron resultados exactos a los de **Y2**, y estos resultados, en cambio, difieren en el error relativo en, al menos, el sexto dígito de aquellos de **ND2**.

La Tabla 7 muestra los resultados para los métodos de descomposición de dominios con superposición. El método de Dirichlet fue resuelto para 2, 4, 8, 10, 12, 16 y 32 líneas solapadas entre los dominios Ω_1 y Ω_2. Sin embargo, los errores permanecen independientes del solapamiento cuando se utilizan más de 8 líneas, mientras que el tiempo de ejecución aumenta con el número de líneas de solapamiento. Por esta razón, solamente se muestran los valores para 2 y 8 líneas en la Tabla 7. En esta tabla también se muestra que la exactitud de los métodos de Neumann y Robin con superposición es prácticamente independiente del orden de discretización de las condiciones de interfaz. Sin embargo, **OR1** y **OR2** no alcanzan la convergencia para $l \geq 100$; **OR1** y **OR2** ofrecen resultados cada uno que difieren en el error relativo, como máximo en el sexto decimal para l=0, 1 y 10 cuando las medias aritméticas de los resultados para el dominio de la izquierda y derecha se usaron para actualizar la interfaz en la siguiente iteración. La eficiencia de ambos **OR1** y **OR2** empeora a medida que l se incrementa.

Método	Máx. Error	Tiempo Ejecución	Iteraciones BiCGSTAB
OD (2 líneas)	5.5612e-3	8935	177154
OD (8 líneas)	6.7818e-6	4305	106570
ON1	8.8975e-5	3417	92748
ON2	1.2095e-4	4116	96312
OR1 l=1	8.8975e-5	5387	138542
OR2 l=1	1.2079e-4	5068	107722

Tabla 7. Errores máximos, tiempos de ejecución e iteraciones del método BiCGSTAB para los métodos con superposición.

Los métodos sin superposición para los dominios con esquinas reentrantes han sido menos eficientes que aquellos con superposición cuando estos comparten 4, 8, o 10 líneas verticales, excepto para **No1** y **No2**; **ON1** fue el más eficiente; las eficiencias de **No**, **N**, **Ro** y **R** incrementaron mientras que la de **D** disminuyó a medida que se

incrementó la exactitud de la discretización de las condiciones de la frontera. La eficiencia de tanto **Ro1** como **Ro2** empeoró primero a medida que l pasó de 0 a 10 y aumentó después con los valores de 10 a 10^{30}.

Como puede comprobarse, los errores cometidos en los métodos de descomposición de dominios sin superposición en dominios con esquinas reentrantes son mucho mayores que los cometidos cuando existe superposición. Además, estos errores son también mucho mayores que los que se comenten en dominios regulares [Ramos y Soler 01]. Como puede observarse en la *Figura 50*, los mayores errores se presentan precisamente cerca de las esquinas. Esto es debido a varias razones:

- Primero, las esquinas son puntos singulares.
- Segundo, los isocontornos de la función, es decir, las líneas a lo largo de las cuales la función tiene el mismo valor, no son perpendiculares a la frontera, especialmente cuando el frente la alcanza, según se puede apreciar en la *Figura 51*. Sería más apropiado, por tanto, en lugar de forzar la continuidad de las derivadas respecto a la normal a la frontera, esto es $\frac{\partial u}{\partial \mathrm{n}} = \frac{\partial v}{\partial \mathrm{n}}$, forzar la derivada

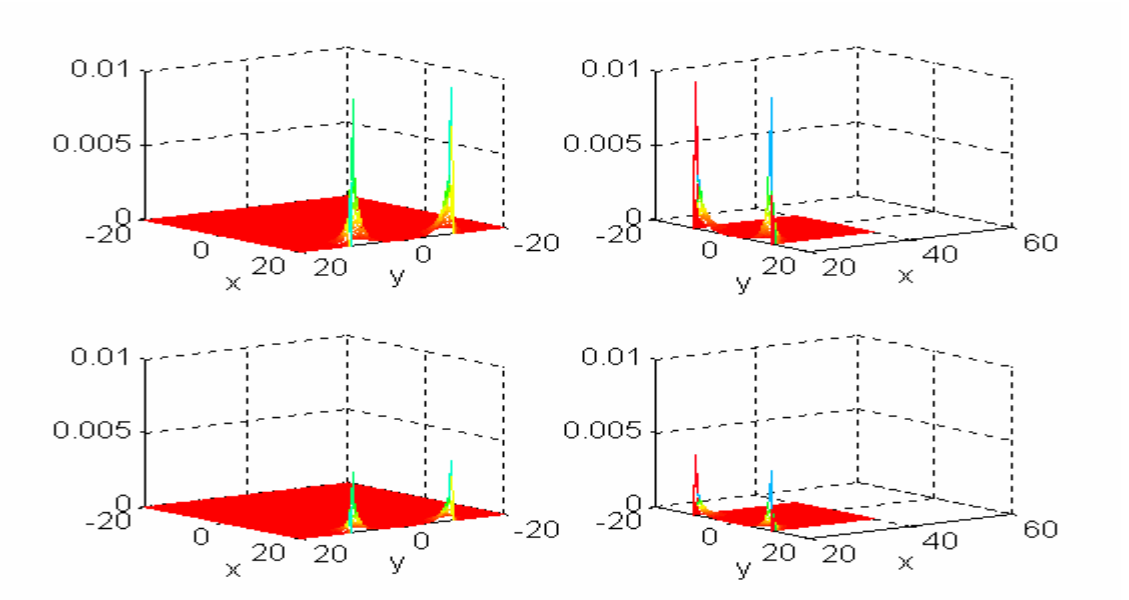


Figura 50. Diferencia entre los valores obtenidos mediante Newton-Raphson y Neumann $O(h^2)$ sin superposición. Arriba, las diferencias para la variable u. *Abajo, para la variable* v.

a lo largo del isocontorno, es decir $\frac{\partial u}{\partial s} = \frac{\partial v}{\partial s}$, según se puede apreciar en la *Figura 52*.

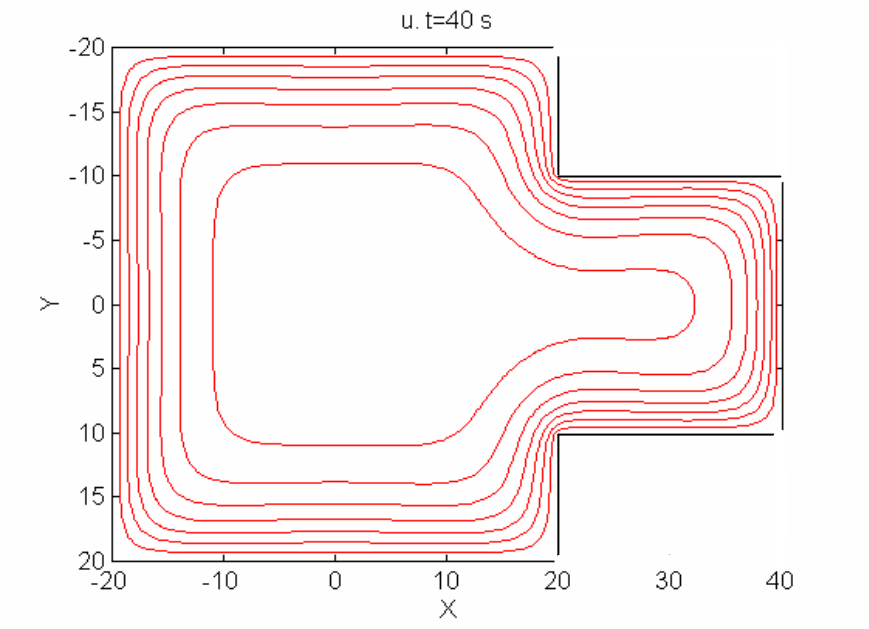


Figura 51. Isocontornos de la variable u. *Cuanto más cercanos a las esquinas, menos perpendiculares son a la frontera.*

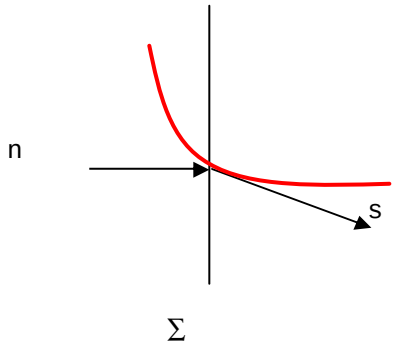


Figura 52. El vector n representa la normal a la frontera mientras que s es la dirección del isocontorno en la frontera.

- Tercero, y más importante, las condiciones de continuidad y suavización impuestas en la interfaz entre los subdominios sólo requieren continuidad de la función y de la derivada mientras que las ecuaciones en diferencias parciales deberían ser satisfechas allí. Para entender estos resultados, considérese una

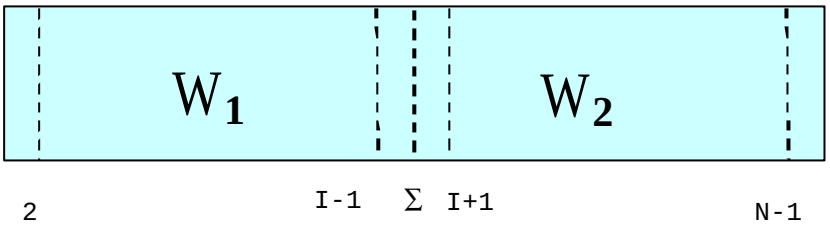


Figura 53. Simplificación del problema a 1 dimensión

simplificación del problema a 1 dimensión (*Figura 53*). En ese caso, la Ecuación 6 quedaría como:

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2} + F(U)$$

Ecuación 30

que al discretizarla mediante diferencias finitas daría:

$$\frac{U_i^{n+1} - U_i^n}{\Delta t} = \frac{1}{h^2}(U_{i+1}^{n+1} - 2U_i^{n+1} + U_{i-1}^{n+1}) + F(U_i^{n+1})$$

Ecuación 31

en los puntos interiores, es decir, para i=2,3 ..., I-1; i=I+1,I+2, ..., N-1.

$$U_I^{n+1} - U_I^n = \frac{\Delta t}{h^2}(U_{I+1}^{n+1} - 2U_I^{n+1} + U_{I-1}^{n+1}) + \Delta t\, F(U_I^{n+1})$$

Ecuación 32

Sin embargo, al aplicar la descomposición de dominios sin superposición, se está forzando la continuidad de la función y la derivada, es decir:

$$U\big|_{\Sigma^-} = U\big|_{\Sigma^+} \quad \text{y} \quad \frac{\partial U}{\partial x}\Big|_{\Sigma^-} = \frac{\partial U}{\partial x}\Big|_{\Sigma^+}$$

Ecuación 33

las cuales implican que $U_{I+1}^{n+1} - 2U_I^{n+1} + U_{I-1}^{n+1} = 0$

Ecuación 34

Obviamente, la Ecuación 32 y la Ecuación 34 no coinciden, lo que provoca que cambios locales y efectos de la reacción no se estén incluyendo debido a que la malla es finita. A la misma conclusión se llega matemáticamente:

Integrando la Ecuación 30 se tiene que

$\int_{x_I-e}^{x_I+e}\left(\frac{\partial U}{\partial t} - \frac{\partial^2 U}{\partial x^2} - F(U)\right)dx = 0$ como puede apreciarse en la *Figura 54*.

Aplicando el teorema del valor medio, se tiene que

$$2e\,(\frac{\P\ \tilde{U}}{\P\ t} - F(\tilde{U})) - \frac{\partial U}{\partial x|_{\Sigma^+}} + \frac{\partial U}{\partial x|_{\Sigma^-}} = 0\,,$$

de lo cual se obtiene que sólo cuando $e \to 0$ (lo que es imposible en un ordenador puesto que el número de puntos de la malla sería infinito) se consigue la continuidad de la derivada, es decir, $\frac{\partial U}{\partial x|_{\Sigma^+}} = \frac{\partial U}{\partial x|_{\Sigma^-}}$.

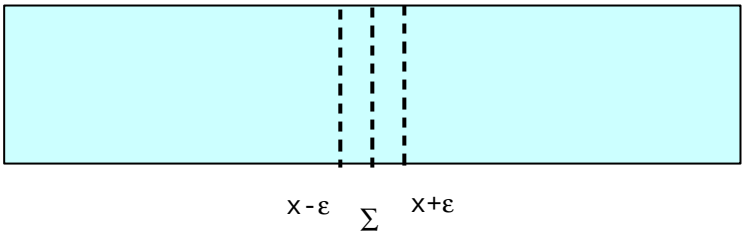


Figura 54 La interfaz está comprendida entre x-e y x+e

6.5.5 Estudio de eficiencia con BCL.

Una vez obtenidos los resultados sobre la exactitud y haber optado por los métodos de descomposición con superposición por ser los que ofrecen mejores resultados de precisión tanto para dominios regulares como irregulares, se ha realizado la implementación del problema con distintas geometrías y número de procesadores para estudiar la eficiencia de la solución utilizando BCL. Para el caso de dominios

		HPF vs. BCL (relación)		
Dominios	**Secuencial**	**4 Procesadores**	**8 Procesadores**	**16 Procesadores**
2	249.34	115.27 / 102.75 (1.12)	87.68 / 66.35 (1.32)	98.23 / 59.33 (1.66)
4	604.95	246.88 / 238.09 (1.04)	191.72 / 106.15 (1.81)	219.71 / 72.18 (3.04)
8	1215.43	496.72 / 564.62 (0.88)	403.43 / 155.93 (2.59)	460.52 / 124.32 (3.70)

Tabla 8. Tiempos de ejecución en segundos obtenidos con las implementaciones de HPF y BCL para el problema de reacción-difusión con dominios regulares. Entre paréntesis la ganancia obtenida al usar BCL respecto a HPF.

regulares, se ha utilizado el mismo proceso coordinador que el utilizado para el problema de Jacobi en el Programa 42 de la sección 6.2. Únicamente se ha modificado el número de puntos de superposición a emplear. La Tabla 8 muestra los resultados obtenidos para distinto número de dominios (2, 4 y 8) y para distintos números de procesadores. El tamaño del dominio ha sido 64 x 64 y el sistema avanza hasta que la variable t obtiene el valor 80. La *Figura 55* muestra gráficamente estos resultados.

Al igual que ocurre con el método de Jacobi para la ecuación de Laplace, BCL ofrece mejores resultados que HPF a excepción del caso en que el número de dominios es superior al de procesadores, por la misma razón a la explicada anteriormente. La eficiencia de BCL respecto a HPF es aún mejor que en el método de Jacobi debido a la mayor carga computacional que ha de realizar cada uno de los procesadores.

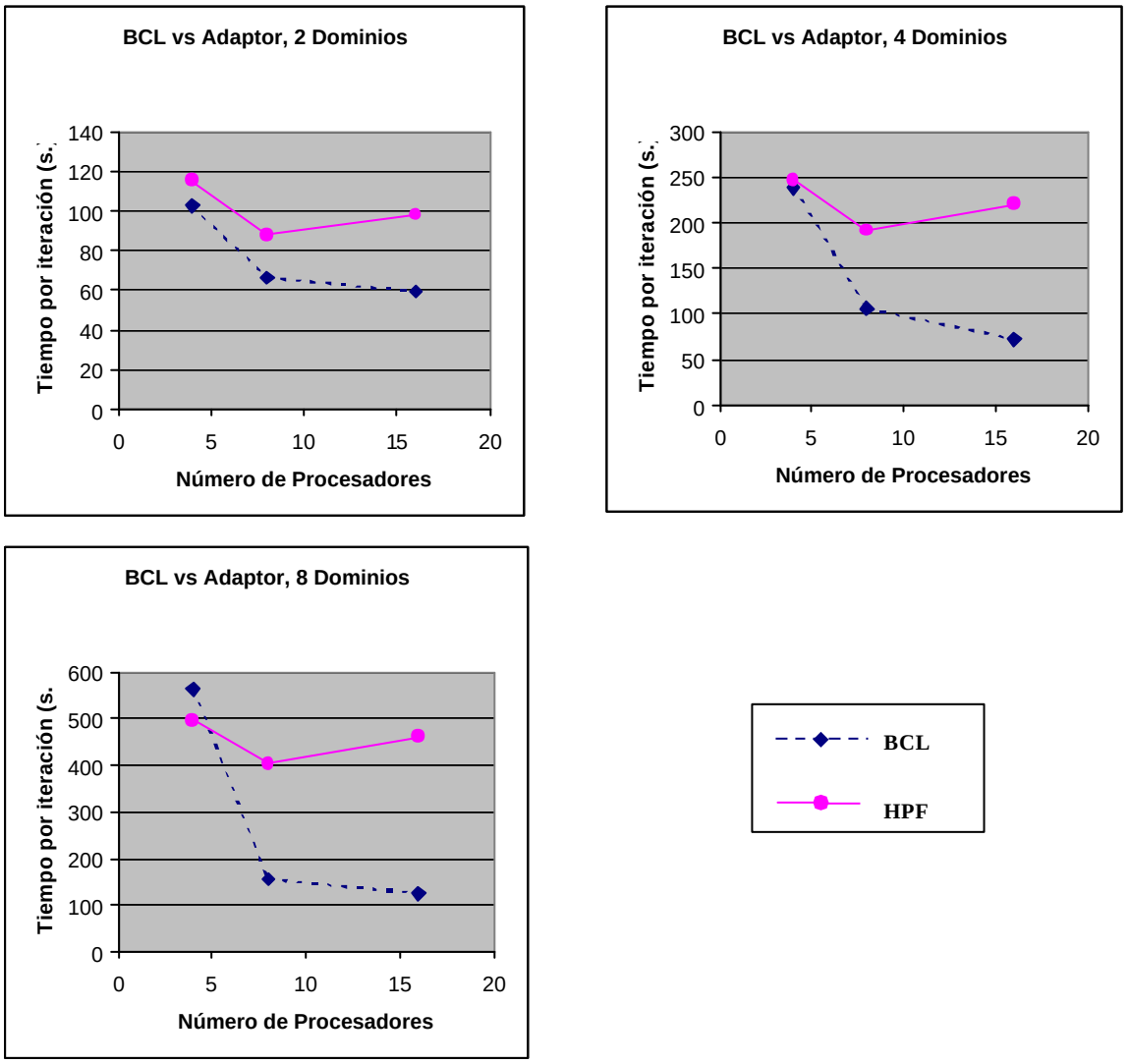


Figura 55. Tiempos de ejecución para las implementaciones de HPF y BCL para el problema de reacción-difusión con dominios regulares en función del número de procesadores.

Para el caso irregular, se ha realizado la implementación del problema con un dominio como el de la *Figura 5* de la página 52. El código de esta aplicación utilizando superposición y condiciones de Dirichlet se encuentra en el Anexo. La Tabla 9 compara los resultados obtenidos para los 3 dominios cuadrados que forman la superficie irregular mostrada en dicha figura. La comparación de forma gráfica de los resultados se ofrece en la *Figura 56*.

En este caso (al contrario que en el de Jacobi) el número de dominios es fijo y se ha variado el número de puntos de la malla. Así, por ejemplo, en la primera fila de la Tabla 9, el tamaño para el dominio `l` es de 64 x 64, para `m` es de 32 x 32 y para `r` es de 64 x 64. En el caso de HPF todos los procesadores ejecutan cada dominio. En el caso de BCL, cuando se utilizan 5 procesadores, 2 de ellos ejecutan el dominio `l`, otros 2 el dominio `r` y 1 procesador, el dominio `m`; para 9 procesadores la distribución de procesadores es 4/1/4 y para 16 procesadores es 7/2/7. Nótese que cuando se usan 5 procesadores, el que ejecuta el dominio central está ocioso una gran cantidad de tiempo puesto que su número total de puntos es la cuarta parte de la de los otros dos. Sin embargo, BCL también ofrece mejor rendimiento que HPF con 5 procesadores excepto para el caso de tamaños grandes, donde, como es bien sabido, HPF mejora su rendimiento. En todos los demás casos probados, BCL ofrece mejores resultados.

		HPF vs. BCL (relación)		
Tamaños de Malla	**Secuencial**	**5 Procesadores**	**9 Procesadores**	**16 Procesadores**
64 / 32 /64	0.21	0.28 / 0.16 (1.75)	0.31 / 0.14 (2.21)	0.29 / 0.13 (2.23)
128 / 64 /128	2.07	1.34 / 1.05 (1.28)	1.16 / 0.67 (1.73)	1.05 / 0.54 (1.94)
256 / 128 / 256	21.12	11.14 / 11.88 (0.94)	8.88 / 7.14 (1.24)	6.87 / 4.31 (1.59)

Tabla 9. Tiempos de ejecución en horas obtenidos con las implementaciones de HPF y BCL para el problema de reacción-difusión con la geometría de la *Figura 5*

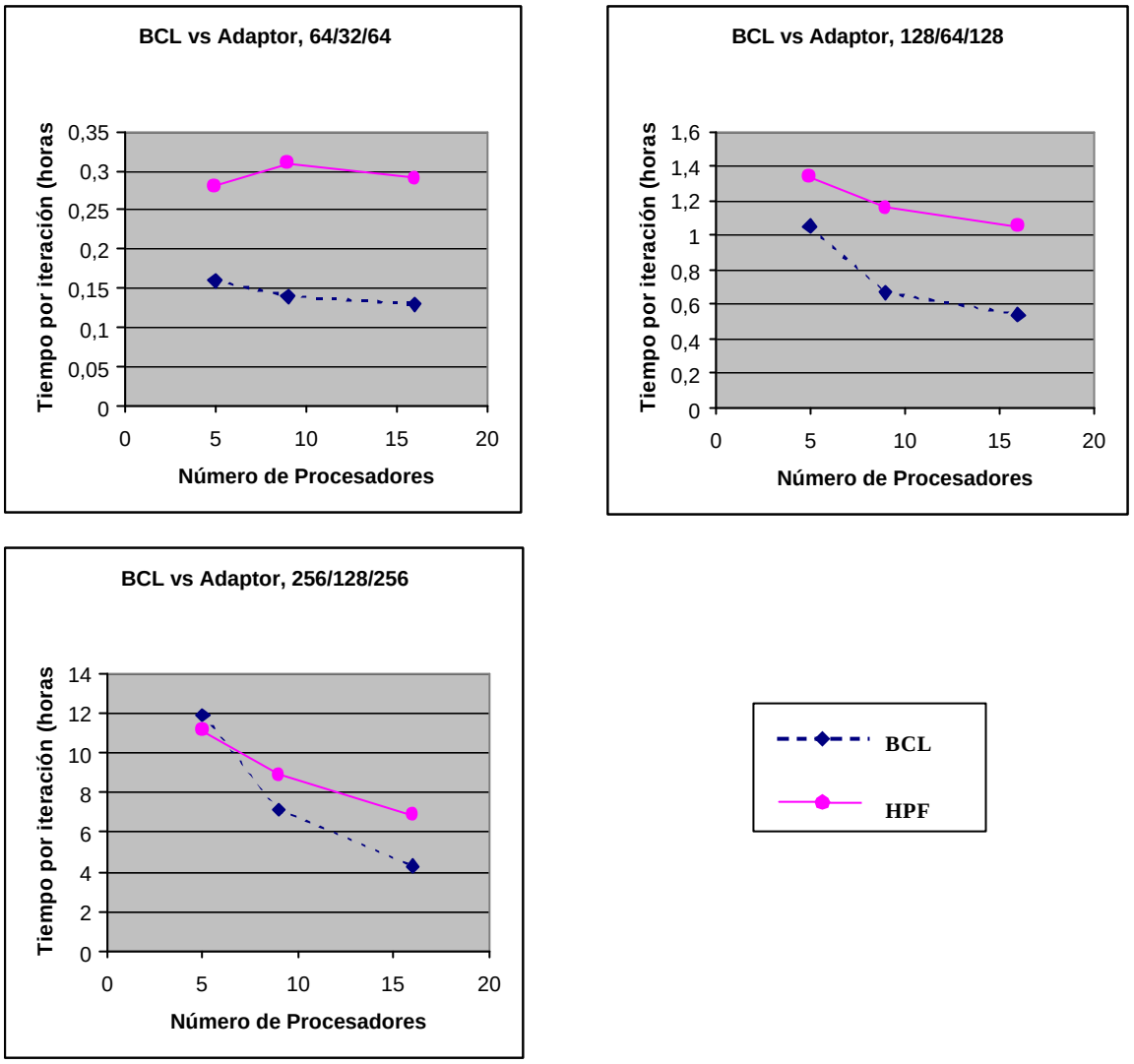


Figura 56. Tiempos de ejecución para las implementaciones de HPF y BCL para el problema de reacción-difusión con dominios irregulares en función del número de procesadores.

Las *Figura 57* a *Figura 61* muestran los resultados de la ejecución de la solución del problema. En esta ocasión no se ha rotado ningún dominio para que se pueda observar más claramente la evolución del frente desde el dominio denominado `l` hasta el dominio `r` a través del dominio central, `m`.

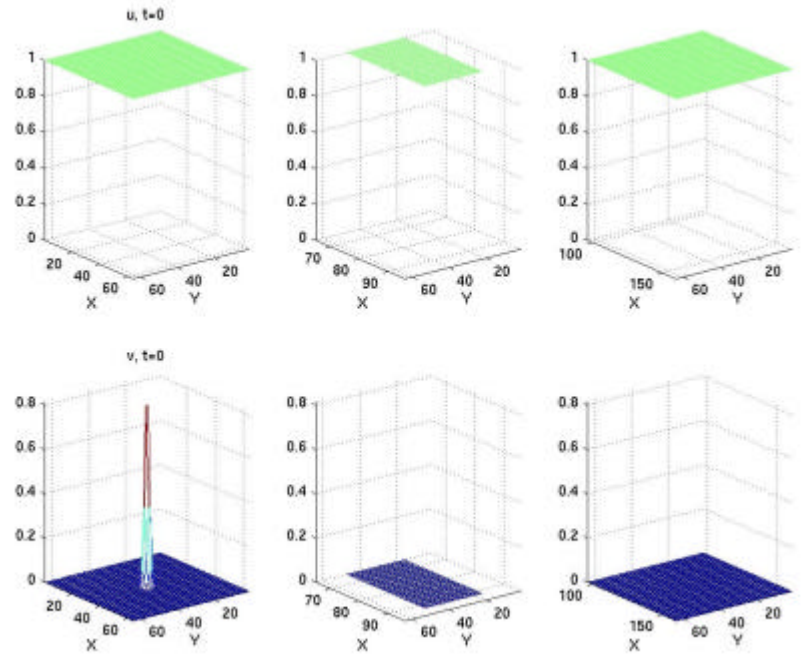


Figura 57. Condiciones iniciales para los tres dominios en el problema de reacción-difusión. De izquierda a derecha los dominios `l`, `m` *y* `r`. *Arriba la variable* `u` *(concentración de combustible) y abajo la variable* `v` *(temperatura).*

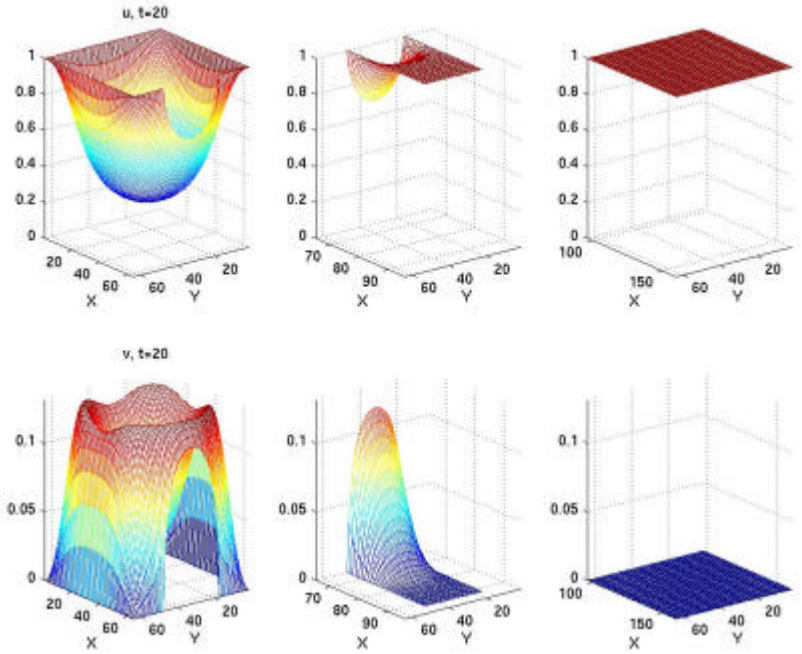


Figura 58. Para t = 20 ya se aprecian cambios en el dominio m. Este es el momento en el que el frente alcanza la frontera entre los dominios.

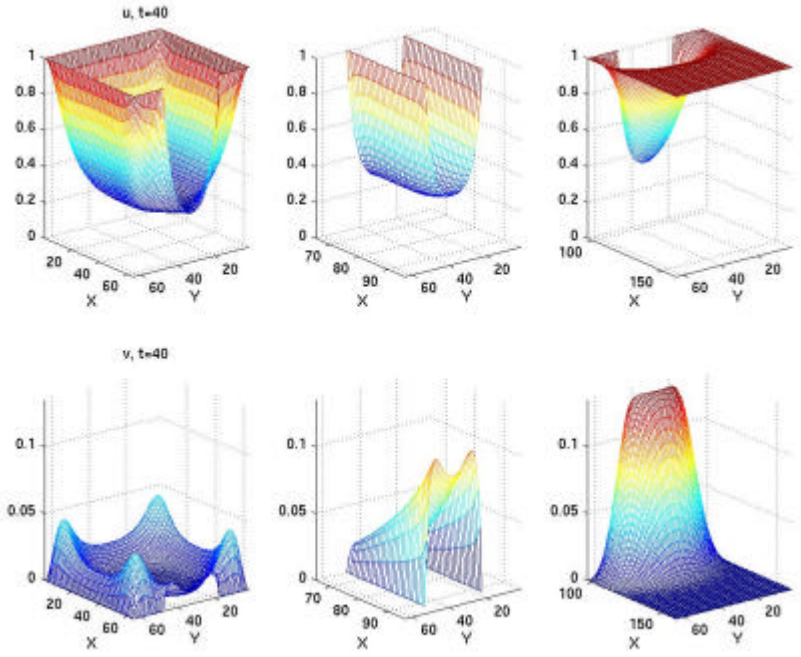


Figura 59. Para t = 40, el frente se ha propagado a lo largo del dominio `m` *y ha llegado al dominio* `r`.

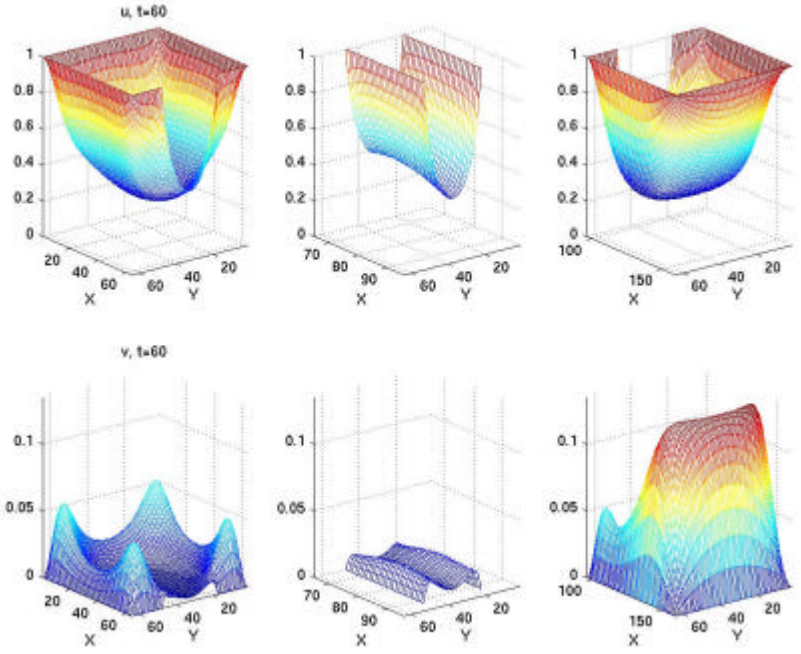


Figura 60. Para t = 60, la reacción está teniendo lugar en el dominio de la derecha, observándose ahora valores más altos de temperatura en las 4 esquinas del dominio de la izquierda debido a que las condiciones de Dirichlet mantienen alto el nivel de combustible en estos puntos.

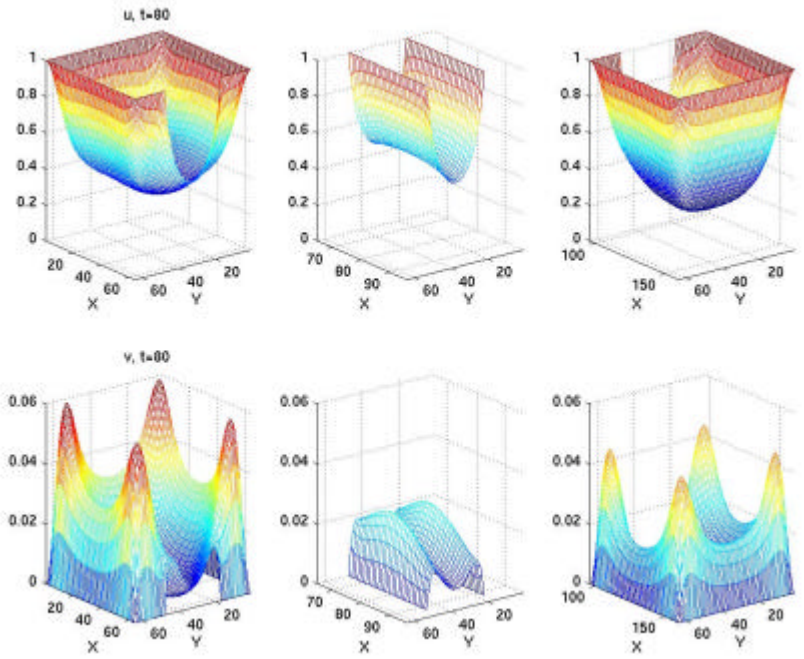


Figura 61. Cuando t = 80 se pueden apreciar para la variable v, las 8 esquinas que corresponden a las 8 esquinas de la geometría. Estas son debidas a las condiciones de Dirichlet impuestas en el contorno.

6.6 Conclusiones.

En este capítulo se muestran los resultados experimentales obtenidos tras la implementación de cinco ejemplos. El primero de ellos, el banco de pruebas sencillo, muestra que las ventajas de BCL como lenguaje de coordinación no suponen una pérdida de eficiencia, lo cual constituye uno de los objetivos marcados al comienzo del trabajo.

En el resto de los ejemplos se muestra cómo la integración del paralelismo de datos y tareas es particularmente útil cuando el tamaño de datos no es muy grande y se emplea un número mayor de procesadores. Esto es particularmente interesante en ciertas aplicaciones en las que el tamaño de datos es pequeño y se requiere una gran eficiencia, como puede ser el caso de la transformada de Fourier para un sistema de visión por computador en tiempo real.

El cuarto ejemplo, la aplicación NPB-FT, muestra cómo con nuestra aproximación se puede sacar partido a distintos niveles de paralelismo de una forma

sencilla a una aplicación relativamente compleja. De nuevo, los mejores resultados se obtienen cuando el tamaño es pequeño y se emplea un número mayor de procesadores. De los resultados se comprueba cómo la replicación puede ser interesante siempre y cuando el número de iteraciones a realizar por cada réplica sea suficientemente grande.

En los ejemplos del método de Jacobi, FFT2D y NPB-FT se comprobó que la salida de la versión secuencial del programa y la de la versión utilizando BCL fuese exactamente la misma.

También se ha comprobado la adecuación de BCL a una aplicación más compleja en la que se han realizado numerosas implementaciones para obtener el mejor método de descomposición de dominios para el problema propuesto. Se comprueba que, en este sistema, los métodos sin superposición no son adecuados al incurrir en errores de precisión muy grandes. Los métodos con superposición son más adecuados y son los empleados para medir la eficiencia en dominios regulares e irregulares.

Al medir la eficiencia, se comprueba que con dominios regulares los mejores resultados se obtienen al sacar mayor provecho a la integración del paralelismo de datos y tareas, es decir, con mayor número de procesadores. Cuando se dispone del mismo número de procesadores que de dominios, también se obtienen mejores resultados con BCL que con HPF. Únicamente cuando existen más dominios que procesadores disponibles, HPF presenta mejores resultados puesto que en BCL cada dominio se ejecuta en un proceso distinto y los procesadores tienen que cambiar de contexto en cada iteración.

Con dominios irregulares, también se consigue una mejor eficiencia, siempre y cuando se realice una correcta asignación de tareas a procesadores. En los ejemplos presentados, se observa que BCL ofrece muy buenos resultados en comparación con HPF excepto en el caso en el que la asignación es poco adecuada y el tamaño de los datos es muy grande.

Capítulo 7. Conclusiones y trabajo futuro

En este trabajo se ha presentado un nuevo modelo de coordinación para la resolución de problemas científicos y de ingeniería basado en los métodos de descomposición de dominios. Estos métodos consisten en dividir el dominio global de una aplicación en distintos subdominios, los cuales serán resueltos cada uno de forma relativamente independiente. Esta independencia parcial es la que permite la resolución en paralelo de los distintos subdominios, necesitándose solamente algunas comunicaciones entre los procesos que los resuelven.

Dada la complejidad que generalmente ofrece este tipo de aplicaciones, se intenta facilitar la tarea del programador mediante la separación, por un lado, de los aspectos de comunicación y sincronización entre los distintos procesos y, por el otro, la parte de cómputo. En la parte de coordinación se establecen los dominios y las fronteras que existen entre ellos. Estas últimas serán las causantes de la comunicación entre los procesos que resuelven cada subdominio. Por esta razón se ha denominado al lenguaje BCL (*Border-based Coordination Language*). En la parte de cómputo, sólo se tienen que introducir unas pocas instrucciones para indicar el punto donde se deben actualizar las fronteras.

BCL proporciona un modelo sencillo de paralelismo pensado para ser utilizado por una clase de usuarios que, aún sabiendo programar, suelen ser reacios a aprender lenguajes paralelos de alto nivel que, por un lado, pueden distraerles de las características complejas de su aplicación y, por otro, pueden no ofrecer una gran eficiencia.

Puesto que el lenguaje más utilizado en este tipo de aplicaciones es Fortran, se ha utilizado éste como lenguaje base de nuestra aproximación. De esta forma, el usuario no necesita un gran esfuerzo para aprender a manejar el modelo y se asegura, además,

poder reutilizar sin ningún problema la gran cantidad de programas y bibliotecas escritos en este lenguaje.

En nuestra aproximación, también se proporcionan una serie de características adicionales que sacan partido del concepto de dominio para aumentar la expresividad del lenguaje y facilitar la codificación de las dos partes de las que consta una aplicación: la de coordinación y la de cómputo.

Para asegurar la eficiencia del modelo, se permite sacar partido a la integración del paralelismo de datos y tareas mediante la adopción del nuevo estándar de paralelismo de datos, HPF. Esto no supone complicar mucho el lenguaje ya que HPF proporciona una forma bastante sencilla y cómoda de expresar el paralelismo de datos. Mediante su integración con BCL, se permite utilizar el paralelismo de tareas de una forma muy intuitiva, de modo que el sistema constará de una serie de tareas (cada una resolviendo un dominio con paralelismo de datos) que se ejecutan concurrentemente y que se comunican mediante el envío de los datos pertenecientes a las fronteras establecidas en la parte de coordinación de la aplicación. Esta declaración de las fronteras entre las tareas, así como la especificación de la distribución de los dominios dentro de cada tarea HPF a nivel de coordinación, es la clave de una implementación eficiente, puesto que cada procesador conocerá en tiempo de compilación qué trozo de su dominio tendrá que enviar a cada procesador de otra tarea.

Como se ha visto con varios ejemplos, BCL también permite la definición de otros problemas científicos, distintos de los de descomposición de dominios, que se pueden beneficiar de la integración del paralelismo de datos y tareas y cuyo patrón de comunicación está basado en el intercambio de matrices. Así, se pueden definir de forma relativamente sencilla esquemas complejos de cómputo mediante la definición de los dominios de cada tarea y las fronteras entre estos.

El uso de la programación paralela estructurada nos permite añadir nuevos mecanismos a BCL para dar una solución de más alto nivel a la definición de esquemas complejos de computación. De esta forma, se incluyen patrones para la definición de la parte de coordinación de la aplicación y plantillas para la de cómputo.

Los patrones permiten que el programador pueda cambiar de forma drástica la estructura de computación de una aplicación haciendo unas pocas modificaciones en el patrón. De esta forma, se pueden probar distintas alternativas de implementación para obtener la más eficiente. El tipo de los datos no se declara en esta parte, lo cual incrementa las posibilidades de reutilización del código al separarse los aspectos de coordinación de los de cómputo.

Se han presentado algunas plantillas de implementación que son útiles para que el programador pueda utilizar un nivel más de abstracción a la hora de definir la parte de cómputo de la aplicación. Además de éstas, se pueden definir nuevas plantillas de forma sencilla para distintos esquemas de ejecución de forma que, una vez establecidas, el usuario sólo tenga que rellenar las distintas secciones para obtener programas distintos. Es posible también, definir plantillas que sean independientes de la dimensionalidad del problema, de modo que cambiando el patrón y las especificaciones de cada tarea se puede pasar un problema, por ejemplo, de 2 a 3 dimensiones.

Además, la utilización de estas plantillas en las estructuras encauzadas permite que el usuario pueda cambiar el orden de ejecución de las etapas, de forma que el compilador se encargue de generar las comunicaciones necesarias entre dichas etapas, así como de establecer los protocolos de comunicación cuando se decide replicar una de ellas.

También se han comentado las características más importantes de la implementación del primer prototipo que se ha realizado del compilador y que nos ha permitido evaluar la eficiencia del modelo en el capítulo de resultados. De estos resultados se observa cómo la sobrecarga del sistema es muy pequeña y que la ventaja de la integración del paralelismo de datos y tareas es especialmente significativa con tamaños de datos pequeños y muchos procesadores.

Utilizando BCL se ha podido realizar un estudio de un sistema de ecuaciones de reacción-difusión para obtener el método de descomposición de dominios más adecuado en dominios irregulares. Se ha comprobado que, en este sistema, los métodos con superposición son más adecuados tanto en precisión como en eficiencia.

BCL proporciona un modelo de paralelismo explícito parcialmente abstracto en el que se intenta dar una solución de alto nivel a la definición de las distintas tareas y a la comunicación y sincronización entre ellas. De esta forma, el programador se puede abstraer en gran medida de estos aspectos, dado el modelo de paralelismo que proporcionan, por un lado BCL y, por otro, HPF.

Las contribuciones que aporta BCL como lenguaje de coordinación se enumeran a continuación:

a) Es un lenguaje de coordinación diseñado para los problemas basados en descomposición de dominios, pero también es útil para definir e implementar otras aplicaciones.

b) Se separa de forma clara de la parte de coordinación de la de cómputo de una aplicación. De este modo, el programador no tiene que pensar en una cuando escribe la otra, incrementándose, además, la reutilización de ambas partes.

c) La parte de coordinación y la de cómputo se escriben en el mismo lenguaje (con algunas variaciones).

d) La parte de coordinación proporciona información sobre la futura distribución de los datos entre los procesadores de cada tarea HPF, lo que permite obtener el esquema de comunicación entre los distintos procesadores en tiempo de compilación, lo que lo hace más eficiente.

e) No se necesitan cambios en el compilador de HPF utilizado.

f) En nuestro sistema se trabaja con dominios en lugar de canales con tipo, lo que permite no tener que especificar los tipos de los datos a nivel de coordinación.

g) La utilización de patrones permite describir esquemas de computación complejas a alto nivel. Nuestra aproximación es la única que dispone de un patrón `MULTIBLOCK`, lo que la hace adecuada para definir problemas de descomposición de dominios.

h) La eliminación de las palabras clave `IN`, `OUT` o `INOUT` que utilizan otros modelos para el anidamiento de patrones hace que en nuestro sistema se pueda cambiar

de manera sencilla el orden de ejecución de las etapas de un encauzamiento. Será el compilador el que decida si se han de enviar o recibir los datos asociados a un dominio dependiendo de su situación en el patrón.

Esta tesis abre las puertas a la realización de distintos trabajos futuros. En primer lugar, obviamente, está la tarea de la implementación total del modelo, con el compilador de BCL primero sin su extensión DIP para luego ampliarlo con las construcciones relativas a los patrones y plantillas de implementación. El compilador se realizará para varias arquitecturas hardware distintas, puesto que el modelo así lo permite. Una vez terminado el compilador para el ordenador con arquitectura SMP, se realizará otro compilador para un sistema distribuido consistente en estaciones Linux unidas con una red de alta velocidad Mirinet.

Uno de los trabajos que está en marcha actualmente es la realización de un entorno de programación gráfico que permita la definición de patrones y plantillas de modo que se facilite la tarea del programador. Este entorno permitirá también la asignación de procesos a los procesadores disponibles y la representación gráfica de las fronteras entre las distintas tareas HPF.

Los patrones definidos en este trabajo se corresponden al tipo de problemas con los que hemos trabajado. Queda una puerta abierta para la definición de otros patrones que permitan la definición de manera estructurada de otros tipos de aplicaciones. Como ejemplo, se podrían mencionar aquellos que permitan afrontar problemas científicos con patrones de comunicación distintos, como pueden ser los problemas multimalla (*multigrid*) o aquellos en los que la precisión requerida en cada dominio es distinta, lo que conlleva a que las fronteras tengan tamaños distintos para cada dominio (*unstructured meshes*).

Otra línea de investigación en la que estamos interesados es la de utilizar el sistema para ejecutar distintas tareas en ordenadores geográficamente distribuidos. Esta es una línea de investigación en la que se está realizando un gran esfuerzo actualmente en el sentido de utilizar recursos geográficamente distantes como una única unidad de potencia computacional [Foster y Kesselman 99] [Buyya y Baker 01]. De esta forma, se podría utilizar el concepto de dominio y, especialmente, el de frontera para definir los costes de las comunicaciones entre las tareas, hacer migraciones de procesos de unos

ordenadores a otros e, incluso, utilizar los dominios para realizar sistemas tolerantes a fallos mediante la introducción de instrucciones que permitan salvar el estado de todo el sistema en un instante dado.

Referencias

[Adams y otros 92]

Adams, J.C., Brainerd, W.S., Martin, J.T., Smith, B.T., Wagener, J.L. *Fortran 90 Handbook*. Mc Graw Hill, 1992.

[Agarwal y otros 94]

Agarwal, R.C., Gustavson, F.G., Zubair, M. An Efficient Parallel Algorithm for the 3-d FFT NAS Parallel Benchmark. *Proceedings of SHPCC'94*, pp. 129-133, 1994.

[Agrawal y otros 95]

Agrawal, G., Sussman, A., Saltz, J. An Integrated Runtime and Compile-Time Approach for Parallelizing Structured and Block Structured Applications. *IEEE Transactions on Parallel and Distributed Systems*, vol. 6 (7), pp. 747-754, 1995.

[Álvarez, Díaz, Llopis, Pastrana, Rus, Soler y Troya 96]

Álvarez, J.M., Díaz, M., Llopis, L., Pastrana, J., Rus, F., Soler, E., Troya, J.M. Practical Parallelization Strategies of a Thermohydraulic Code. *International Conference on Supercomputation in Nonlinear and Disordered Systems*, El Escorial, Madrid, pp. 254-257, 1996.

[ANSI 78]

American National Standards Institute, Inc. *American National Standard Programming Language FORTRAN*. ANSI x3.9-1978, 1978.

[Arbab y otros 93]

Arbab, F., Herman, I., Spilling, P. An Overview of Manifold and its Implementation. *Concurrency: Practice and Experience*, vol. 5 (1), pp. 23-70, 1993.

[Arbab 96]

Arbab, F. The IWIM Model for Coordination of Concurrent Activities. En Ciancarini, P. y Hankin, C. (eds), *First International Conference on Coordination Models, Languages and Applications (Coordination'96)*, Cecena, Italia. Lecture Notes in Computer Science, vol. 1061, pp. 34-56. Springer-Verlag, 1996.

[Bacci y otros 95]

Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneshi, M. P^3L: A Structured High-Level Parallel Language and its Structured Support. *Concurrency: Practice and Experience*, vol. 7 (3), pp. 225-255, 1995.

[Bailey y otros 94]

Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Finiberg, S., Frederickson, P. Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., Weeratunga, S. The NAS Parallel Benchmarks. *Technical Report RNR-94-007*, NASA Ames Research Center, 1994. http://www.nas.nasa.gov/Research/Reports/Techreports/1994/rnr-94-007-abstract.html.

[Bal y Haines 98]

Bal, H.E., Haines, M. Approaches for Integrating Task and Data Parallelism. *IEEE Concurrency*, vol. 6 (3), pp. 74-84, 1998.

[Barret y otros 97]

Barret, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., Vorst, H. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1997.

[Bjørstad y Karstad 95]

Bjørstad, P.E., Karstad, T. Domain Decomposition, Parallel Computing and Petroleum Engineering. *Domain–Based Parallelism and Problem Decomposition Methods in Computacional Science and Engineering*, pp. 39-56. David E. Keyes, Youcef Saad and Donald G. Truhlar (eds.). SIAM, 1995

[Brandes 97]

Brandes, T. ADAPTOR Programmer's Guide, Version 5.0. *Internal Report, GMD-SCAI*, Sankt Augustin, Alemania, Abril 1997.

[Brandes 99a]

Brandes, T. ADAPTOR Programmer's Guide, Version 7.0, *Internal Report, GMD-SCAI*, Sankt Augustin, Alemania, Diciembre 1999. ftp://ftp.gmd.de/GMD/adaptor/docs/pguide.ps

[Brandes 99b]

Brandes, T. Exploiting Advanced Task Parallelism in High Performance Fortran via a Task Library. *Proceedings of Euro-Par'99, Parallel Processing*, Toulouse, Francia, pp. 833-844, 1999.

[Briham 88]

Briham, E.O. *The Fast Fourier Transform and its Applications*. Prentice-Hall International, 1988.

[Buyya y Baker 01]

Buyya, R., Baker, M. *Grid Computig- GRID 2000. Proceedings of the First IEEE/ ACM International Workshop*. Bangalore, India. Lecture Notes in Computer Science, vol. 1971. Springer Verlag, 2001.

[Cai 95]

Cai, X.C. A Family of Overlapping Schwarz Algorithms for Nonsymmetric and Indefinite Elliptic Problems. *Domain–Based Parallelism and Problem Decomposition Methods in Computacional Science and Engineering*, pp. 1-19. David E. Keyes, Youcef Saad and Donald G. Truhlar editors. SIAM, 1995.

[Carriero y Gelernter 90]

Carriero, N., Gelernter, D. *How to Write Parallel Programs: A First Course*. MIT Press, Cambridge, Massachusetts, USA, 1990.

[Chandy y Kesselman 93]

Chandy, K, Kesselman, C. CC++: A Declarative Concurrent Object-Oriented Programming Notation. *Research Directions in Concurrent Object Oriented Programming*, pp. 281-313. The MIT Press, 1993.

[Chapman y otros 92]

Chapman, B., Mehrotra, P., Zima, H. Programming in Vienna Fortran. *Scientific Programming*, vol. 1 (1), pp. 31-50, 1992.

[Chapman y otros 97]

Chapman, B., Haines, M., Mehrotra, P., Zima, H., Rosendale, J. Opus: A Coordination Language for Multidisciplinary Applications. *Scientific Programming*, vol. 6 (2), pp. 345-362, 1997.

[Chassin y otros 00]

Chassin de Kergommeaux, J., Hatcher, P.J., Rauchwerger, L. (Eds.). Parallel Computing for Irregular Applications. *Parallel Computing*, vol. 26 (13-14), 2000.

[Ciancarini y Hankin 96]

Ciancarini, P., Hankin, C. *First International Conference on Coordination Models, Languages and Applications* (Coordination'96), Cecena, Italia. Lecture Notes in Computer Science, vol. 1061. Springer-Verlag, 1996.

[Ciancarini 97]

Ciancarini, P. *Coordination Models, Languages, Architectures and Applications: A Personal Perspective.* URL: http://www.cs.unibo.it/~cianca/coord_ToC.html.

[Ciarpaglini y otros 00]

Ciarpaglini, S., Folchi, L., Orlando, S., Pelagatti, S., Perego, R. Integrating task and data parallelism with taskHPF. *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00),* Las Vegas, Nevada, USA. CSREA Press, pp. 2485-2492, Junio 2000.

[Cole 89]

Cole, M. Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Cambridge, Massachusetts, USA, 1989.

[Cole y Zavanella 00]

Cole, M., Zavanella, A. Activity Graphs: A Model-Independent Intermediate Layer for Skeletal Coordination. *15th Annual ACM Symposium on Applied Computing (SAC'00).* Special Track on Coordination Models, Languages and Applications, Villa Olmo, Como, Italia. ACM Press, pp. 255-261, Marzo, 2000.

[Cooley y Tukey 65]

Cooley, J., Tukey, J. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, vol. 19, pp. 297-301, 1965.

[Darlington y otros 93]

Darlington, J., Ghanem, M., To, H.W. Structured Parallel Programming. *Proc. Of Massively Parallel Programming Models Conference*, Berlín, Alemania. IEEE Computer Society Press, pp. 160-169, 1993.

[Darlington y otros 95]

Darlington, J., Guo, Y., To, H.W., Yang, J. Functional Skeletons for Parallel Coordination. *Proceedings of Europar'95*, Estocolmo, Suecia. Lecture Notes in Computer Science, vol. 966, pp. 55-69. Springer-Verlag, 1995.

[Díaz, Llopis, Pastrana, Rus y Soler 96]

Díaz, M., Llopis, L., Pastrana, J., Rus, F., Soler, E. Thermohydraulic Code Paralelization. *International Conference on Systems Engineering*, Las Vegas (USA) pp. 860-865, 1996.

[Díaz, Rubio y Troya 94]

Díaz, M., Rubio, B., Troya, J.M. Implementation Issues of a Distributed Real-Time Logic Language. Brogi, A. y Hill, P. (Eds.) *ICLP'94 Post-Conference Workshop on Integration of Declarative Paradigms*, pp. 106-119, 1994.

[Díaz, Rubio y Troya 96]

Díaz, M., Rubio, B., Troya, J.M. Distributed Programming with a Logic Channel based Coordination Model. *The Computer Journal,* vol. 39 (10), pp. 876-889, 1996.

[Díaz, Rubio y Troya 97]

Díaz, M., Rubio, B., Troya, J.M. DRL: A Distributed Real-Time Logic Language. *Computer Languages*, vol. 23 (2-4), pp. 87-120, 1997.

[Díaz, Rubio y Troya 98]

Díaz, M., Rubio, B., Troya, J.M. Multilingual and Multiparadigm Integration of a Tuple Channel Based Coordination Model. *13th ACM Symposium on Applied Computing (SAC'98)*. Special Track on Coordination Models, Languages and Applications, Atlanta, Georgia, USA. ACM Press, pp. 194-196, 1998.

[Díaz, Rubio, Soler y Troya 99]

Díaz, M., Rubio, B., Soler, E., Troya, J.M. Using Coordination for Solving Domain Decomposition-based Problems. *Informe Técnico de Investigación ITI 99/14*. Departamento de Lenguajes y Ciencias de la Computacion. Universidad de Málaga, 1999

[Díaz, Rubio, Soler y Troya 00a]

Díaz, M., Rubio, B., Soler, E., Troya, J.M. BCL: A Border-based Coordination Language. *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, Las Vegas, Nevada, USA. CSREA Press, pp. 753-760, Junio 2000.

[Díaz, Rubio, Soler y Troya 00b]

Díaz, M., Rubio, B., Soler, E., Troya, J.M. Integration of Task and Data Parallelism: A Coordination-based Approach. *International Conference on High Performance Computing (HiPC 2000)*, Bangalore, India. Lecture Notes in Computer Science, vol. 1970, pp. 173-182. Springer-Verlag, 2000.

[Díaz, Rubio, Soler y Troya 01a]

Díaz, M., Rubio, B., Soler, E., Troya, J.M. DIP: a Pattern-based Approach for Task and Data Parallelism Integration. *Symposium on Applied Computing (SAC 2001).* Las Vegas, Nevada, USA, 2001.

[Díaz, Rubio, Soler y Troya 01b]

Díaz, M., Rubio, B., Soler, E., Troya, J.M. Integrating Task and Data Parallelism by means of Coordination Patterns. *6th International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS'2001).* San Francisco, California, USA. Lecture Notes in Computer Science, vol. 2026, pp. 16-27. Springer-Verlag, 2001.

[Dinda y otros 94]

Dinda, P., Gross, T., O'Hallaron, D., Segall, E., Stichnoth, J., Subhlok, J., Webb, J., Yang, B. The CMU task parallel program suite. *Technical Report CMU-CS-94-131*, School of Computer Science, Carnegie Mellon University, Marzo 1994.

[Drashansky, Joshi y Rice, 95]

Drashansky, T.T., Joshi, A., Rice, J.R. SciAgents - An Agent Based Environment for Distributed Cooperative Scientific Computing. *Proceedings of Seventh International Conference on Tools with Artificial Intelligence*, Herndon, Virginia, USA, pp. 452-459, 1995.

[Dubois 99]

Dubois, P.F. Scientific Components are Coming. *IEEE Computer*, vol. 32 (3), pp. 115-116, 1999.

[Everaars y Arbab 96]

Everaars, C.T.H., Arbab, F. Coordination of Distributed/Parallel Multiple-Grid Domain Decomposition. *IRREGULAR'96*, Lecture Notes in Computer Science, vol. 1117, pp. 131-144. Springer-Verlag, 1996.

[Everaars y Koren 98]

Everaars, C.T.H., Koren, B. Using Coordination to Parallelize Sparse-Grid Methods for 3D CFD Problems. *Parallel Computing*, vol. 24, pp. 1081-1106, 1998.

[Everaars y otros 96]

Everaars, C.T.H., Arbab, F., Burger F.J. Restructuring Sequential Fortran Code into a Parallel/Distributed Application. *International Conference on Software Maintenance, 1996*. También en: *Technical Report CS-R9628*, Centrum voor Wiskunde en Informatica, 1996.

[Fink 98]

Fink, S.J. *A Programming Model for Block-Structured Scientific Calculations on SMP Clusters*, Ph. D. Dissertation, Universidad de California en San Diego, 1998. ftp://ftp.cs.ucsd.edu/pub/scg/papers/1998/thesis.ps.gz.

[Fink y Baden 98]

Fink, S.J., Baden, S. Efficient Run-Time Support for Irregular Block-Structured Applications. *Journal of Parallel and Distributed Computing*, vol. 50, pp. 61-82, 1998.

[Foster 95]

Foster, I. *Designing and Building Parallel Programs.* Addison-Wesley, 1994.

[Foster y Chandy 92]

Foster, I. Chandy, K.M. Fortran M. A Language for Modular parallel Programming. *Technical Report MCS-P327-0992*, Argonne National Laboratory, 1992.

[Foster y Kesselman 99]

Foster, I., Kesselman, C. *The Grid. Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[Foster y otros 94]

Foster, I., Avalani, B., Choudhary, A. Xu, M. A Compilation System that Integrates High Performance Fortran and Fortran M. *Proceedings of 1994 Scalable High Performance Computing Conference*, pp. 293-300. IEEE Computer Society, 1994.

[Foster y otros 96]

Foster, I., Kohr, D., Krishnaiyer, R., Choudhary, A. Double Standards. Bringing Task Parallelism to HPF via the Message Passing Interface. *Proceedings of Supercomputing'96*. ACM Press, 1996.

[Foster y otros 97]

Foster, I., Kohr, D., Krishnaiyer, R., Choudhary, A. A Library-Based Approach to Task Parallelism in a Data-Parallel Language. *Journal of Parallel and Distributed Computing*, vol. 45 (2), pp. 148-158, 1997.

[Freeman y Phillips 92]

Freeman, T., Phillips, C. *Parallel Numerical Algorithms*, pp. 267-276. Prentice-Hall International, 1992.

[Frumkin y otros 98]

Frumkin M., Jin, H., Yan, J. Implementation of NAS Parallel Benchmarks in High Performance Fortran. *Technical Report NAS-98-009*, NASA Ames Research Center, 1998.
http://www.nas.nasa.gov/Research/Reports/Techreports/1998/nas-98-009-abstract.html.

[Geist y otros 94]

Geist, A. Beguelin, A., Dongarra, J. Jiang, W., Manchek, R. Sunderam, V. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*. The MIT Press, 1994.

[Gelernter 85]

Gelernter, D. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems,* vol. 7 (1), pp. 80-112, 1985.

[Gropp y otros 94]

Grop, W., Lusk, E. Skjellum, A. *Using MPI: Portable Parallel Processing with the Message-Passing Interface*. The MIT Press, Cambridge, Massachusetts, 1994.

[Gross y otros 94]

Gross, T. O'Hallaron, D., Subhlok, J. Task Paralellism in a High Performance Fortran Framework. *IEEE Parallel & Distributed Technology*, vol. 3, pp. 16-26, 1994.

[Hassen y Bal 96]

Hassen, S.B., Bal, H.E. Integrating Task and Data Parallelism Using Shared Objects. *10th ACM International Conference on Supercomputing*, pp. 317-324, 1996.

[Hassen y otros 98]

Hassen, S.B., Bal, H.E., Jacobs, C.J. A Task and Data Parallel Programming Language Based on Shared Objects. *ACM Transactions on Programming Languages and Systems*, vol. 20 (6), pp. 1131-1170, 1998.

[Hiranandani y otros 91]

Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C. An Overview of the Fortran D Programming System. *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, California, USA. Springer-Verlag, Agosto, 1991.

[Houstis y otros 89]

Houstis, E.N., Rice, J.R., Papatheodorou, T.S. Parallel ELLPACK: An Expert System for Parallel Processing of Partial Differential Equations. *Math. Comp. Simulation*, vol. 31, pp. 497-508, 1989.

[HPFF 93]

High Performance Fortran Forum. *High Performance Fortran Language Specification, version 1.0*. Mayo, 1993.

[HPFF 97]

High Performance Fortran Forum. *High Performance Fortran language specification, version 2.0*. Enero, 1997. URL: http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/index.cfm

[INMOS 94]

INMOS Ltd. *Occam Programming Manual*. Prentice-Hall, Englewood Cliffs, New Jersey, 1994.

[Koelbel y otros 94]

Koelbel, C., Loveman, D., Schreiber, R., Steele, G., Zosel, M. *The High Performance Fortran Handbook*. MIT Press, 1994. http://www.nas.nasa.gov/Software/NPB.

[Malone y Crowston 94]

Malone, T.W., Crowston, K. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, vol. 26, pp. 87-119, 1994.

[May y otros 86]

May, D., Shepherd, R., Keane, C. Communicating Process Architecture: Transputers and Occam, *Future Parallel Computers*, Lecture Notes in Computer Science, vol. 272, pp. 35-81. Springer-Verlag, 1986

[Merlin y otros 96]

Merlin, J.H., Carpenter, D.B., Hey, A.J.G. SHPF: A Subset High Performance Fortran Compilation System. *Fortran Journal*, pp. 2-6, Marzo/Abril, 1996.

[Merlin y otros 98]

Merlin, J.H., Baden, S. B., Fink, S. J. and Chapman, B. M. Multiple Data Parallelism with HPF and KeLP. *HPCN'98*, Amsterdam, Holanda. Lecture Notes in Computer Science, vol. 1401, pp. 828-839. Springer-Verlag 1998.

[Metcalf y Reid 90]

Metcalf, M., Reid, J. *Fortran 90 Explained*. Oxford Science Publications, 1990.

[MPIF 95]

Message Passing Interface Forum. *MPI. A Message-Passing Interface Standard*. University of Tennessee, Knoxville, Tennessee, USA, Junio, 1995.

[NAS 00]

Numerical Aerodynamic Simulation. *NAS Parallel Benchmark, NPB, version 2.3.* NASA Ames Research Center. Moffett Field, California, USA, 2000.

[Orlando y otros 00a]

Orlando, S., Palmerini, P., Perego, R. Mixed Data and Task Parallelism with HPF and PVM. *Cluster Computing,* Baltzer Science Publishers, vol. 3 (3), pp.201-213, 2000.

[Orlando y otros 00b]

Orlando S., Palmerini, P., Perego, R. Coordinating HPF Programs to Mix Task and Data Parallelism. *15th Annual ACM Symposium on Applied Computing (SAC'00),* Villa Olmo, Como, Italia. Special Track on Coordination Models, Languages and Applications. ACM Press, pp. 240-247, Marzo, 2000.

[Orlando y Perego 99]

Orlando, S., Perego, R. $COLT_{HPF}$, a Run-Time Support for the High-Level Coordination of HPF Tasks. *Concurrency: Practice and Experience,* vol. 11 (8), pp. 407-434, 1999.

[Papadopoulos y Arbab 98]

Papadopoulos, G.A., Arbab, F. Coordination Models and Languages. *Advances in Computers*, vol. 46, pp. 329-400. Academic Press, 1998.

[Pelagatti 98]

Pelagatti, S. *Structured Development of Parallel Programs.* Taylor and Francis, 1998.

[Press y otros 96]

Press, W., Teukolsky, S., Vetterling, W., Flannery, B. *Numerical Recipes in Fortran 90*. Cambridge University Press, 1996.

[Ramaswamy y Banerjee 95]

Ramaswamy S., Banerjee, P. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. *The Fifth Symposium on the Frontiers of Massively Parallel Computations,* pp. 342-349, 1995.

[Ramos 97]

Ramos, J.I. Linearization Methods for Reaction-Diffusion Equations: Multidimensional Problems. *Applied Mathematics and Computation,* vol. 88, pp. 225-254, 1997.

[Ramos y Soler 01]

Ramos, J.I., Soler, E. Domain Decomposition Techniques for Reaction Diffusion Equations in Two-Dimensional Regions with Re-entrant Corners. *Applied Mathematics and Computation*, vol. 118 (2-3), pp. 189-221, 2001.

[Ramos, Soler y Troya 98]

Ramos, J.I., Soler, E., Troya, J.M. Comparación de Métodos para la Resolución de una Ecuación de Reacción-Difusión en un Dominio de Geometría Compleja. *Informe técnico de investigación, ITI 98/20*. Departamento de Lenguajes y Ciencias de la Computación. Universidad de Málaga, 1998.

[Rauber y Rünger 99]

Rauber, T., Rünger, G. A Coordination Language for Mixed Task and Data Parallel Programs. *14th Annual ACM Symposium on Applied Computing (SAC'99)*. Special Track on Coordination Models, Languages and Applications, San Antonio, Texas, USA. ACM Press, pp. 146-155, Febrero, 1999.

[Rubio 98]

Rubio, B. *TCM: Un Modelo de Coordinación basado en Canales de Tuplas*. Tesis Doctoral. Departamento de Lenguajes y Ciencias de la Computación. Universidad de Málaga. Septiembre, 1998.

[Schwarz 90]

Schwarz, H. A. *Gesammelte Mathematische Abhandlungen*, vol. 2, pp.133-143, Springer, Berlin, 1890.

[Skillicorn y Talia 98]

Skillikorn, D., Talia, D. Models and Languages for Parallel Computation. *ACM Computing Surveys*, vol. 30 (2), pp.123-169, Junio, 1998.

[Smith y otros 96]

Smith, B., Bjørstard, P., Gropp, W. *Domain Decomposition. Parallel Multilevel Methods for Elliptic P.D.E.'s.* Cambridge University Press, 1996.

[Subhlok y Yang 97]

Subhlok, J., Yang, B. A New Model for Integrated Task and Data Parallel Programming. *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Las Vegas, Nevada, USA, pp. 1-12, 1997.

[Wegner 96]

Wegner, P. Coordination as Constrained Interaction. En Ciancarini, P. y Hankin, C. (eds) *First International Conference on Coordination Models, Languages and Applications (Coordination'96)*, Cecena, Italia. Lecture Notes in Computer Science, vol. 1061, pp. 28-33. Springer-Verlag, 1996.

[Yang 96]

Yang, D. A Parallel Iterative Nonoverlapping Domain Decomposition Procedure for Elliptic Problems. *IMA Journal of Numerical Analysis*, vol. 16, pp. 75-91, 1996

[Zima y otros 92]

Zima, H., Brezany, P., Chapman, B., Mehrotra, P., Schwald, A. Vienna Fortran. A Language Specification. *Technical report ICASE Interim Report 21*, ICASE NASA Langley Research Center, Virginia, USA, 1992.

Anexo. Codificación de un Problema de Reacción-Difusión.

A continuación se muestra el ejemplo de reacción difusión que ha servido como uno de los bancos de prueba para evaluar el modelo. La descripción del problema se puede encontrar en el apartado 5 del capítulo 6, mientras que su geometría se corresponde con la de la *Figura 5*. En primer lugar se muestra el patrón `MULTIBLOCK` para la definición de este problema.

```
MULTIBLOCK reaction_diffusion  left /0,0, Ncl,Nrl /
                               center/Ncl-1,Nrm1,Ncm+1,Nrm2/,
                               right/Ncm,0, Ncr, Nrl/

  solve(left:(*,BLOCK), ignition) ON PROCS(2)
  solve(center:(*,BLOCK), initial) ON PROCS(1)
  solve(right:(*,BLOCK), initial) ON PROCS(2)

WITH BORDERS
  left(Ncl,Nrm1 ,Ncl,Nrm2) <- center(_)
  center(Ncl-1,Nrm1 ,Ncl-1,Nrm2) <- left(_)
  center(Ncm+1,Nrm1, Ncm+1,Nrm2) <- right(_)
  right(Ncm,Nrm1, Ncm,Nrm2) <- center(_)

END
```

En el patrón se declaran los tres dominios en los que se ha decompuesto la geometría global del problema. Los hemos denominado aquí `left`, `center` y `right` por claridad. Junto con su declaración, se define la región del plano que comprenden, por ejemplo, para el dominio `left`, la región se extiende desde el punto `0,0` al `Ncl,Nrl` (número de columnas de `left`, número de filas de `left`).

Seguidamente aparece el nombre de las tareas que se van a ejecutar. En este caso, habrá una tarea por dominio. A cada tarea se le pasan dos argumentos, el dominio que va a resolver junto con su distribución y la subrutina de inicialización de los puntos del dominio. Como puede verse en la *Figura 57*, el dominio más a la izquierda se

inicializa de forma distinta a los otros dos. Por último, se especifican los procesadores sobre los que se va a ejecutar.

Las fronteras entre dominios se definen en la sección `WITH BORDERS`. Puesto que los dominios se han definido según su posición en el plano, la región de frontera en ambos dominios es la misma, por lo que en la variable de tipo dominio que está a la derecha del operador `<-`, se puede poner la expresión "_", que indica que la región es la misma que la definida en la variable de tipo dominio a la izquierda del operador de frontera.

Seguidamente se presenta la especificación de usuario para las tareas trabajadoras. La plantilla de implementación con la que se instancia es la que hemos denominado `Parabolic`. No se muestran todos los procedimientos sino sólo los más significativos. Después de la especificación se explica cada una de las secciones.

```
Parabolic solve(d, initialize)
#decl
  use BiCG
  domain d
  external initialize
  real, parameter :: tottime = 80., tstep = 0.2, toler = 1.0e-10
  type vble
     double precision :: u,v
  end
  type (vble), GRID(d) :: dg,g,rhs,dg_old
  type (vble), dim(6), GRID(d) :: ab
  real time
  logical conver

#init
  call initialize(g)
  time = 0.
  dg = 0.

#termination
  time > tottime

#preconverge
  time = time + tstep
  call set_values(ab,rhs,g)
  conver = .false.

#convergence
  conver

#preupdate
  call modify_rhs(rhs,dg)
  call save_borders(dg,dg_old)
  call BiCG(ab,rhs,dg)
```

```
#postupdate
  error = compute_error(dg,dg_old)
  REDUCE(error,maxim)
  if (error < toler) conver = .true.

#postconverge
  g = g + dg

#results
  call show_results(g)

END
```

En la zona de declaraciones se incluye el uso del módulo `BiCG` que es donde está implementado la subrutina que resuelve el sistema de ecuaciones lineales mediante el método denominado gradiente biconjugado estabilizado. A esta rutina se le han añadido directivas HPF para permitir la distribución de los datos. A continuación se declaran los argumentos pasados en la cabecera, es decir, el dominio `d` y la subrutina con la cual se inicializarán los valores de la variable con atributo `GRID` asociada al dominio. Después, se definen las constantes necesarias que indican el valor final que debe alcanzar la variable de tiempo, el escalón de tiempo empleado y el error máximo tolerado en la convergencia entre dominios.

Puesto que en el problema de reacción difusión están involucradas dos variables, es necesario que cada punto del dominio incluya esos dos valores. Para ello, se ha definido el tipo `vble` que sirve de tipo base para las variables con atributo `GRID`. A continuación se declaran estas variables, la primera de ellas `dg`, es la que contiene la diferencia del valor de las variables en dos escalones de tiempo, ΔU. Al ser la primera en ser declarada, será la que se comunique con otros dominios. La variable `g` es la que contiene el valor de la variable U de la Ecuación 6. La variable `rhs` contiene el valor del lado derecho de la expresión $AB\ \Delta U = RHS$, donde AB es la matriz de coeficientes del sistema de ecuaciones lineales correspondientes a la Ecuación 11. Los valores de `dg` en la frontera se guardan de una iteración a otra en la variable `dg_old` para poder medir el error local cometido. Puesto que la matriz AB es dispersa, sólo necesita 6 diagonales, por lo que se declara como un `GRID` cuyo tipo base es un array de 6 columnas de tipo base `vble`. La variable lógica `conver` es la que indica la convergencia del método por cada paso de tiempo.

En la sección `#init` se inicializan los valores de `g`, `dg` y `time`. La condición de terminación del programa se alcanza cuando la variable `time` llegue a tener un valor superior al de la constante `tottime` como se indica en la sección `#termination`.

Las instrucciones necesarias antes de comenzar el bucle de convergencia se escriben en la sección `#preconverge`. Estas instrucciones se ejecutarán una vez por paso de tiempo. En la subrutina `set_values` se actualizan los valores de las variables `ab` y `rhs` con los de `g` según la Ecuación 11. La variable `conver` se pone a `false` para entrar en el bucle de convergencia.

Dentro de la sección `#preupdate` se escriben las instrucciones que se repetirán por cada iteración del bucle de convergencia y antes de la actualización de las fronteras. La subrutina `modify_rhs` es la que fija los valores de la variable `rhs` según los valores de `dg` (correspondientes a la actualización de la frontera en la iteración anterior) según el método de descomposición de dominios empleado. Los valores en la frontera para la iteración actual se guardan mediante la subrutina `save_borders`. Finalmente, para concluir esta sección, se llama a la subrutina que resuelve el sistema algebraico lineal.

Después de la actualización de las fronteras se llama a la subrutina `compute_error` que mide la diferencia en las fronteras entre la iteración anterior y la actual. Este error es reducido con respecto a las otras tareas mediante la instrucción `REDUCE`, de forma que el error global es el máximo de los errores locales, en valor absoluto, cometidos en cada tarea.

Una vez terminado el bucle de convergencia, se ejecuta la instrucción `g = g + dg` que actualiza los valores de la variable para el paso de tiempo siguiente. Esta instrucción se ejecuta una sola vez por escalón de tiempo. Por último, una vez alcanzado el final del programa, se muestran los resultados.

Tras el proceso de instanciación, se eliminan las características de DIP, generándose un proceso coordinador y otro trabajador, cuyos códigos se muestran a continuación.

```
program reaction_diffusion
  DOMAIN2D left, center,  right

  left =(/0,0, Ncl,Nrl /)
  center = (/Ncl-1,Nrm1,Ncm+1,Nrm2/)
  right = (/Ncm,0, Ncr, Nrl/)

  left(Ncl,Nrm1 ,Ncl,Nrm2) <- center(_)
  center(Ncl-1,Nrm1 ,Ncl-1,Nrm2) <- left(_)
  center(Ncm+1,Nrm1, Ncm+1,Nrm2) <- right(_)
  right(Ncm,Nrm1, Ncm,Nrm2) <- center(_)
  CREATE solve(left:(*,BLOCK), ignition) ON PROCS(2)
  CREATE solve(center:(*,BLOCK), initial) ON PROCS(1)
  CREATE solve(right:(*,BLOCK), initial) ON PROCS(2)

END


subroutine solve (d, initialize)
  use BiCG
  domain2d d
  external initialize
  real, parameter :: tottime = 80., tstep = 0.2, toler = 1.0e-10
  type vble
     double precision :: u,v
  end
  type (vble), GRID2d :: dg,g,rhs,dg_old
  type (vble), dim(6), GRID2d :: ab
!hpf$ distribute (*,BLOCK) :: dg, g, rhs, dg_old
!hpf$ distribute (*,BLOCK) :: ab

  real time
  logical conver

  dg%DOMAIN = d
  g%DOMAIN = d
  rhs%DOMAIN = d
  dg_old%DOMAIN = d
  ab%DOMAIN = d

  call initialize(g)
  time = 0.
  dg%DATA = 0.

  do while (.not. (time > tottime))
       time = time + tstep
       call set_values(ab,rhs,g)
       conver = .false.
       do while (.not. conver)
            call modify_rhs(rhs,dg)
            call save_borders(dg,dg_old)
            call BiCG(ab,rhs,dg)
            UPDATE_BORDERS(dg)
            error = compute_error(dg,dg_old)
            REDUCE(error,maxim)
            if (error < toler) conver = .true.
        enddo
        g%DATA = g%DATA + dg%DATA
    enddo
    call show_results(g)
end subroutine solve
```