Universidad Politécnica de Cataluña
Departamento de Arquitectura de Computadores
DAC

Estrategias de Descomposición en Dominios para entornos Grid

Tesis Doctoral

Presentada por:
Beatriz Otero Calviño

Dirigida por:
Dr. José María Cela Espín

Dedicatoria

A mi padre, mi mayor apoyo, quien
con su amor incondicional me ha ayudado
en todos los momentos difíciles de mi vida.

A mi madre, quien con su dedicación,
me ha enseñado que la constancia debe
formar parte de todo lo que hacemos.

A mis hermanos, quienes me han hecho ver
que todos los sueños pueden hacerse realidad
en la medida en que trabajemos para lograrlos.

A mis amigos, Roger y Antonio, quienes me han
acompañado y apoyado siempre.

Agradecimientos

Es difícil recordar, después de tanto tiempo, todas las personas que de alguna manera contribuyeron para que finalizara este trabajo. Muchas de ellas, me ayudaron técnicamente y otras me brindaron momentos de verdadera alegría. Sin embargo, quisiera agradecer la confianza de mis amigos, *todos fantásticos*. Particularmente quiero agradecerle:

A **Roger** y **Anto** por apoyarme siempre, por estar allí, por escucharme y orientarme en los peores momentos.

A **Angels** por sus intensos momentos de diálogo, por su colaboración y su disposición para ayudarme.

A **Esther** por sus palabras de ánimo y sus constantes consejos.

Además, quisiera agradecer el apoyo que me han brindado siempre mis compañeros de la UPC, especialmente **Xavi**, **Frank**, **Oliver**, **Carmelo**, **Edu**, **Alex Pajuelo**, **Joseph Ma.**, así como también, los profesores del Departamento, particularmente **Juanjo**, **Pau**, **Fermín, David, Marta, Lorenzo** y **Alex**. Gracias a todos por sus palabras.

Por último, quisiera agradecer:

A **Rosa Ma. Badía** y a **Jesús Labarta** por su disposición para analizar las trazas.

A **José Ma. Cela**, mi director de tesis, por su tiempo y su colaboración.

Al **Departamento de Arquitectura de Computadores** por permitirme utilizar los recursos para trabajar en mi tesis.

A **muchos otros** que en este momento no recuerdo y que también contribuyeron con este trabajo.

Índice

Índice de figuras

Índice de tablas

Resumen

Diversas simulaciones computacionales en ciencias básicas e ingeniería resuelven numéricamente ecuaciones en derivadas parciales (EDP's). Esta resolución podría ameritar una discretización del problema con respecto al espacio y al tiempo, utilizando métodos muy conocidos, como por ejemplo: el método de los elementos finitos y el método de diferencias centrales.

Particularmente, estamos interesados en realizar simulaciones numéricas basadas en elementos finitos con integración explícita en el tiempo. De esta manera, nuestro algoritmo consta de un bucle principal, que itera (avanza a la solución) en pasos de tiempo discretos, y en cuyo interior, se realiza una operación producto matriz por vector y un producto escalar. Estas operaciones suelen realizarse elemento a elemento, por lo que no se requiere ensamblar una matriz y un vector en cada iteración. Sin embargo, la operación producto matriz por vector representa entre el 85% y el 90% del tiempo total de una iteración. Por lo tanto, optimizar el cálculo de esta operación numérica es importante para reducir los tiempos de ejecución de las simulaciones explícitas.

Tradicionalmente, la paralelización de estas simulaciones implica el uso de, modelos de programación con paso de mensajes y la utilización de la técnica de descomposición en dominios para realizar el particionamiento de los datos. Así, las simulaciones numéricas explícitas son ejecutadas en potentes supercomputadores.

Sin embargo, en muchos casos, no se dispone de la tecnología adecuada (acceso a supercomputadores) para realizar estas simulaciones numéricas explícitas. De esta forma, en este trabajo planteamos la posibilidad de utilizar el Grid como un cluster dedicado a realizar cálculo paralelo.

En general, las simulaciones numéricas que abordamos determinan las

deformaciones que sufren los materiales sobre su estructura cuando son sometidas a fuerzas de impacto. Sin embargo, nuestro trabajo no se limita únicamente a estas simulaciones, también pueden considerarse otras simulaciones que demanden cálculo intensivo, como por ejemplo, las simulaciones de ondas sísmicas, que estudian el efecto que producen los terremotos sobre la superficie terrestre. El tamaño de estos problemas es inmenso y puede llegar a desbordar, en varios órdenes de magnitud, los supercomputadores existentes; tales problemas podrían usar grandes Grids para ejecutarse.

Ante tales situaciones, surgió interés en utilizar la tecnología Grid para realizar las simulaciones numéricas explícitas. De esta forma, es posible contar con tecnología a bajo coste, sin necesidad de comprar o tener acceso a costosos supercomputadores. El problema surge cuando planteamos la idea de ejecutar, con una eficiencia razonable, aplicaciones de granularidad media sobre entornos Grid. El objetivo de la tesis es dar solución a este problema.

Actualmente, las simulaciones explícitas de elementos finitos usan la técnica de descomposición en dominios con particiones balanceadas para realizar la distribución de los datos. Sin embargo, esta distribución de los datos presenta una degradación importante del rendimiento de las simulaciones explícitas cuando son ejecutadas en entornos Grid. Esto se debe principalmente, a que en un ambiente Grid tenemos comunicaciones heterogéneas, muy rápidas dentro de una máquina y muy lentas fuera de ella. De esta forma, una distribución balanceada de los datos se ejecuta a la velocidad de las comunicaciones más lentas. Para superar este problema proponemos solapar el tiempo de la comunicación remota con el tiempo de cálculo. Para ello, dedicaremos algunos procesadores a gestionar las comunicaciones más lentas, y el resto, a realizar cálculo intensivo. Este esquema de distribución de los datos, requiere que la descomposición en dominios sea no balanceada, para que, los procesadores dedicados a realizar la gestión de las comunicaciones lentas tengan apenas carga computacional.

Nuestras propuestas son:

1. **Distribución irregular de los datos:** Hasta ahora la distribución de los datos se realizaba de forma balanceada entre los procesadores y se ejecutaba sobre entornos homogéneos. Proponemos:
    - realizar distribuciones no balanceadas de los datos dentro de una misma máquina y entre máquinas.

    Cada máquina esta formada por un conjunto de procesadores.

2. **Planificación de las comunicaciones remotas conscientes del Grid:** La asignación de las cargas de trabajo para cada máquina se realiza considerando las comunicaciones remotas. La idea es solapar el cálculo con las comunicaciones más lentas. La distribución de los datos se organiza de manera que determinados procesadores se encarguen de realizar las comunicaciones remotas, mientras que otros procesadores realizan cálculo.

En este trabajo se han analizado diferentes estrategias para distribuir los datos y mejorar el rendimiento de las aplicaciones en entornos Grid. Las estrategias de distribución estáticas analizadas son:

1. ***U-1domains***: Inicialmente, el dominio de los datos es dividido proporcionalmente entre las máquinas dependiendo de su velocidad relativa. De esta forma, cada partición de los datos obtenida es llamada subdominio. Posteriormente, en cada máquina, los datos son divididos en *nprocs-1* partes, donde *nprocs* es el número de procesadores total de la máquina. En este caso, la cantidad de carga computacional asignada a cada procesador es proporcional a su velocidad relativa con respecto al procesador que realiza el mayor número de operaciones por segundo. Cada subdominio es asignado a un procesador y cada máquina dispone de un único procesador para gestionar las comunicaciones remotas con otras máquinas.

2. ***U-Bdomains***: Igual que en el caso anterior, el particionamiento de los datos se realiza en dos fases. La primera fase es equivalente a la realizada para la distribución *U-1domains*. La segunda fase, divide, proporcionalmente, cada subdominio de datos en *nprocs-B* partes, donde *B* es el número de comunicaciones remotas con otras máquinas (dominios especiales). En este caso, cada máquina tiene más de un procesador para gestionar las comunicaciones remotas. El número de comunicaciones remotas es variable y depende de las relaciones que existan entre los dominios de datos.

3. ***U-CBdomains***: En esta distribución, se crean tantos dominios especiales como comunicaciones remotas, igual que en el caso de la distribución *U-Bdomains*. Sin embargo, ahora los dominios especiales son asignados a un único procesador dentro de la máquina. De esta forma, cada subdomino de datos es dividido en *nprocs-1* partes. La gestión de las comunicaciones remotas se realiza concurrentemente mediante *threads*.

Para evaluar el rendimiento de las aplicaciones sobre entornos Grid utilizamos Dimemas. Dimemas es una herramienta desarrollada por el CEPBA[1] que permite predecir el tiempo de ejecución de las aplicaciones sobre entornos Grid. El modelo de comunicación de Dimemas descompone el tiempo de comunicación en cinco componentes. Los valores para cada uno de estos componentes, son determinados considerando: valores experimentales utilizados en trabajos de investigación similares y las redes informáticas actuales.

En cada caso, evaluamos el rendimiento de las aplicaciones para diferentes configuraciones del entorno Grid y distintos tipos de mallas de elementos (tiras y cajas). Inicialmente, especificamos la configuración del entorno Grid y el valor de los parámetros que definen la arquitectura. Luego, ejecutamos Dimemas, utilizando una traza instrumentada de la aplicación. Dimemas determina el tiempo total de ejecución de la aplicación para la configuración del entorno Grid que ha sido

[1] European Center for Parallelism of Barcelona, www.cepba.upc.edu

especificada.

Cada distribución de los datos propuesta fue evaluada considerando diferentes configuraciones y parámetros del entorno Grid. Los resultados al aplicar nuestras propuestas fueron comparados con los obtenidos al utilizar la clásica distribución de datos: la distribución balanceada. Los resultados obtenidos muestran que:

- La distribución *U-1domains* reduce los tiempos de ejecución hasta un 45% (configuración con dos máquinas) respecto a la distribución balanceada. Sin embargo, esta distribución no resulta efectiva para entornos Grid compuestos por más de dos máquinas, ya que las comunicaciones remotas de una máquina con otras máquinas se realizan en secuencia en cada iteración del método.

- La distribución *U-Bdomains* muestra ser más eficiente, ya que reduce el tiempo de ejecución hasta un 84%, respecto a la distribución balanceada. Además, esta distribución resulta más efectiva para particiones en forma de caja que para particiones en forma de tiras. Sin embargo, la escalabilidad de ésta distribución es moderada, debido a que puede llegar a tener un gran número de procesadores que no realizan cálculo. Estos procesadores únicamente gestionan las comunicaciones remotas. Esta distribución resulta apropiada si más del 50% de los procesadores en una máquina realizan cálculo.

- La distribución *U-CBdomains* reduce los tiempos de ejecución hasta un 30%, respecto a la distribución balanceada, pero no resulta ser tan efectiva como la distribución *U-Bdomains*. Sin embargo, esta distribución incrementa la utilización de los procesadores que no realizan cálculo entre un 25% y un 75%, es decir, disminuye la cantidad de procesadores ociosos.

Este trabajo esta organizado en cuatro capítulos. En el capítulo 1 describimos los modelos de programación paralela y comentamos las situaciones en las que resulta más apropiado utilizar cada modelo.

En el capítulo 2 definimos el entorno Grid y planteamos la posibilidad de ejecutar simulaciones numéricas explícitas en estos entornos.

En el capítulo 3 definimos el problema, junto con las herramientas y métodos utilizados.

En el capítulo 4 describimos las estrategias de distribución de los datos que han sido propuestas para mejorar el rendimiento de las simulaciones explícitas en los entornos de programación Grid.

Por último, en el anexo A mostramos detalles de los resultados obtenidos para cada distribución de los datos y para diferentes configuraciones del entorno Grid. Además, comentamos las conclusiones y los trabajos futuros que se plantean a raíz de este trabajo.

Capítulo 1

Modelos de programación paralela

Un modelo de programación es una colección de abstracciones de programación que le brindan al programador una visión simplificada y transparente de la arquitectura. De esta forma, los modelos de programación son mecanismos disponibles para el programador que permiten expresar la estructura lógica de un programa.

Para mejorar el rendimiento de las aplicaciones sobre arquitecturas paralelas es necesario determinar el modelo de programación más adecuado. Esta elección influye en:

1. La complejidad, el costo de desarrollo y el mantenimiento del programa.
2. El rendimiento, la implementación y la estructura de la paralelización.

En este capítulo describimos los modelos de programación paralela más utilizados. Entre estos modelos se encuentran: los modelos de programación para

memoria compartida, los modelos de programación para memoria distribuida, los modelos de programación basados en el paralelismo de datos (*Data-Parallel)* y los modelos de programación híbridos. Para cada caso, comentamos las ventajas, las desventajas y las características que definen las situaciones en donde resulta más apropiado utilizar cada uno de éstos modelos de programación.

1.1 Introducción

A comienzos de los años 80 se creía que el rendimiento de una aplicación podía mejorarse únicamente creando procesadores más rápidos y eficientes. Sin embargo, el procesamiento paralelo desafía esta idea, uniendo dos o más procesadores para resolver un mismo problema. Es a principios de los años 90, cuando surge la tendencia a utilizar supercomputadores paralelos y redes de computadoras. El uso de las redes de computadores toma fuerza, debido a:

1. la presencia de estaciones de trabajo (PC's) con grandes capacidades de cómputo y ,
2. los avances tecnológicos de las redes.

Estos progresos hacen que las redes de computadoras resulten atractivas para realizar procesamiento paralelo de alto rendimiento a bajo coste. El principal atractivo de estos sistemas es que son baratos. Además, disponen de componentes software estándares (MPI, PVM) para realizar implementaciones en ambientes de programación paralelos. Además, este sistema resulta escalable porque puede adaptarse al presupuesto disponible y a las necesidades de cómputo requeridas por la industria [3].

Por otra parte, los tipos de aplicaciones que demandan procesamiento paralelo surgen de problemas en áreas tales como la ciencia y la ingeniería. Estos problemas tienen alto impacto económico y científico [13]. Sin embargo, el desarrollo de aplicaciones paralelas es una tarea compleja. Este desarrollo depende de la disponibilidad de herramientas software y del entorno.

De esta manera, la programación paralela resulta exitosa, si el software paralelo

es capaz de ocultar los detalles de la arquitectura, la red de comunicaciones y la tolerancia a fallos. Además, debe existir una estandarización para lenguajes de alto nivel que simplifique la forma de hacer paralelismo.

Básicamente existen dos tipos de programación paralela. Estos tipos son:

1. **Paralelismo implícito**: Este tipo de paralelismo es característico de algunos lenguajes de programación y de compiladores paralelizadores. En este caso, el usuario no interviene en la planificación de los cálculos ni en la localidad de los datos.

   Entre los lenguajes de programación implícitos se mencionan: *Higher Order Functional Hasbell*, *Maude*, *OBJ*, *Unity*, *PPP*, *AND/OR*, *REDUCE/OR*, *Opera*, *Palm*, *P3L*, *Cole*, *Darlington*, *Celland*, *Carpet*, *CDL*, *Ceprol*, *Cristal*, *Sisal*, *Id*, *Concurrent Prolog*, *Delta-Prolog*, *Strand*, *Multilisp*, *pSETL*, *Gamma*, *PEI*, *APL*, *MOA*, *Nial*, *AT*, *CamlFlight*, *NESL*.

2. **Paralelismo explícito**: En este tipo de paralelismo el programador es el responsable de la descomposición y la comunicación de las tareas, así como también del mapeo de las mismas en los procesadores.

   Este tipo de paralelismo se basa en la suposición de que el programador explota el paralelismo de una aplicación particular. Según esto, el paralelismo explícito tiene una mayor eficiencia sobre los lenguajes de programación paralelos y sobre los compiladores que utilizan paralelismo implícito. Este tipo de paralelismo es característico de:

   a. **Los lenguajes con primitivas para el paralelismo**. Entre estos lenguajes se encuentran: *Linda*, *Compositional*, *C++*, *Opus*, *HPF*, *Ada*, *SR*, *OCCAM*, *Java*, *pC++*, *Fortran 90*.
   b. **Los compiladores con directivas para el paralelismo**, y
   c. **Las librerías para el manejo del paralelismo**. Entre estas librerías se mencionan: Express p4, MPL, PVM, MPI, OpenMP, threads (POSIX threads, SOLARIS threads, Win32 threads, entre otros).

1.1.1 Estrategias para diseñar aplicaciones paralelas

Básicamente existen tres estrategias para diseñar aplicaciones paralelas. La primera esta basada en la paralelización automática, la segunda en el uso de librerías paralelas y la tercera está relacionada con el desarrollo de la aplicación. A continuación detallamos cada una de éstas estrategias.

1. **Paralelización automática**: Esta estrategia libera al programador de paralelizar tareas. El compilador es el responsable de generar el código objeto paralelo e integrarlo en el código del programador. Sin embargo, la eficiencia de esta paralelización puede ser baja, ya que depende del compilador.

2. **Utilización de librerías de cálculo paralelizadas**: La idea básica de esta estrategia es la de encapsular código paralelo común a muchas aplicaciones y colocarlo en librerías paralelas, que pueden ser implementadas eficientemente. Estas librerías pueden ser de dos tipos: las que usan tipos de datos abstractos y las que permiten una implementación paralela de rutinas matemáticas. Estas últimas librerías son muy utilizadas en la ciencia y la ingeniería para realizar simulaciones numéricas.

3. **Implementación de la aplicación paralela:** Esta estrategia ofrece libertad al programador para elegir el lenguaje y el modelo de programación. Sin embargo, en muchos casos es imposible reutilizar el código paralelo.

La figura 1.1 muestra los componentes de la programación paralela y esquematiza la secuencia de pasos necesarios para obtener una implementación paralela utilizando las estrategias de diseño comentadas anteriormente.

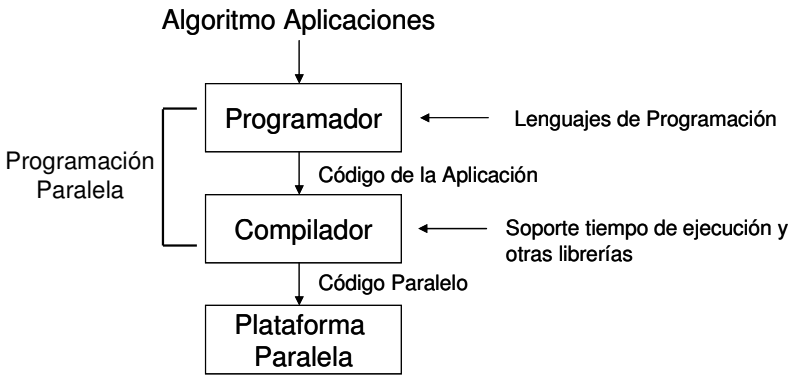


Fig. 1.1 Programación paralela

1.1.2 Granularidad

Para garantizar un buen rendimiento del programa paralelo es necesario considerar los costes asociados a las ejecuciones paralelas de las tareas, como por ejemplo, la creación y la gestión de tareas, la migración de las tareas a procesadores remotos y los costes de comunicación. De esta forma, el paralelismo de una aplicación esta relacionado con la granularidad.

La granularidad se define como el cociente entre la cantidad de operaciones que puede realizar un proceso y el tiempo de comunicación requerido [32]. Existen diversas clasificaciones de los niveles de paralelismo [16] [33], entre ellas destacamos las siguientes:

a) **Granularidad muy fina:** Paralelismo a nivel de instrucciones de lenguaje máquina.
b) **Granularidad media:** Paralelismo a nivel de subrutinas.
c) **Granularidad gruesa:** Paralelismo a nivel de aplicación.

El primer nivel de paralelismo está relacionado con el compilador, mientras que los dos niveles siguientes corresponden al programador.

Si la granularidad de las aplicaciones es "pequeña", puede ocurrir que los costes asociados a las ejecuciones paralelas de las tareas superen el beneficio de la ejecución paralela. De esta forma, la granularidad que explotemos al paralelizar

depende de los costes asociados a la ejecución paralela de las tareas. Por lo tanto, la granularidad de las tareas determina el tipo de ejecución (paralela o secuencial). En nuestro caso, el coste de comunicación de las ejecuciones paralelas de la aplicación no supera el beneficio obtenido de la ejecución paralela.

Por otra parte, los costes de sincronización/comunicación en máquinas de memoria compartida son comparativamente pequeños, del orden de los microsegundos (μs). Sin embargo, en arquitecturas de memoria distribuida estos costes son mayores, ya que en algunos casos se alcanza el orden de los milisegundos (ms). En redes de computadores estos costes pueden ser enormes, del orden de los minutos. Por lo tanto, es necesario en cada arquitectura, explotar el grano de paralelismo.

Hasta ahora hemos relacionado la arquitectura y el tipo de granularidad que presentan las aplicaciones. Esta relación determina cuando resulta provechoso cambiar de ejecuciones secuenciales a ejecuciones paralelas. A continuación presentaremos los modelos de programación y determinaremos en qué casos resulta mas adecuado la utilización de cada uno de ellos.

Existen diversos modelos de programación utilizados para el desarrollo de aplicaciones paralelas. Básicamente, estos modelos de programación se clasifican en:

- modelos de programación *Data-Parallel*,
- modelos de memoria compartida,
- modelos de paso de mensajes y
- modelos híbridos.

1.2 Modelos de programación paralela

1.2.1 Modelo Data Parallel

El modelo de programación *Data Parallel* es un modelo de programación utilizado en aplicaciones numéricas. Este modelo de programación presenta las siguientes características:

- El trabajo paralelo esta orientado a la paralelización de un conjunto de

datos que se organizan en estructuras comunes.

- Todas las tareas realizan la misma operación y trabajan colectivamente sobre la misma estructura de datos.
- Sin embargo, cada tarea modifica su partición de datos en la estructura.

En arquitecturas de memoria compartida, todas las tareas pueden acceder a la estructura de datos a través del espacio de direcciones compartido. Por otra parte, en arquitecturas de memoria distribuida, la estructura de datos se divide y reside en la memoria local de cada tarea.

La programación de éste modelo se realiza utilizando constructores que permiten la paralelización de los datos. Estos constructores los proveen librerías y/o directivas de compilación. Entre las implementaciones más conocidas para este modelo de programación se mencionan: Fortran 90 (librerías) y HPF [23] (directivas de compilación). Estas implementaciones están disponibles en la mayoría de las plataformas paralelas.

Particularmente, este tipo de modelo de programación se utiliza en arquitecturas SIMD (*Single Instruction Multiple Data*). Sin embargo, puede ser un excelente modelo para soportar y desarrollar aplicaciones Grid de alto rendimiento, donde los programadores especifican el paralelismo de los datos en término de un grupo de procesos y distribución de las operaciones.

En este modelo de programación el paralelismo de los datos puede ser de dos tipos:

- **Concurrencia explícita:** En este tipo de paralelismo el programador aporta las directivas para distribuir los datos y guiar la paralelización. Este modelo de programación es aplicable sobre arquitecturas SIMD/SPMD y otras arquitecturas paralelas, tales como: máquinas vectoriales y arquitecturas UMA/NUMA.
- **Concurrencia implícita:** En este tipo de paralelismo el compilador es capaz de detectar iteraciones independientes que pueden ser ejecutadas concurrentemente.

1.2.2 Modelo de memoria compartida

En los modelos de programación paralela para memoria compartida las tareas tienen un espacio de memoria común, donde leen y escriben asincrónicamente. El acceso a memoria es controlado utilizando mecanismos tales como: semáforos, monitores, etc.

Para estos modelos de programación no es necesario especificar explícitamente la comunicación de los datos entre las tareas productoras y las tareas consumidoras, por lo que el desarrollo de los programas se simplifica.

Las comunicaciones en los sistemas de memoria compartida tienen baja latencia y grandes anchos de banda. La portabilidad de estos modelos es pobre, pero resultan fáciles de programar comparados con los modelos de programación para memoria distribuida.

El modelo de programación para memoria compartida es utilizado en arquitecturas SISD (*Single Instruction Single Data*), pero también puede utilizarse en arquitecturas MIMD (*Multiple Instruction Multiple Data*) de memoria compartida.

Las arquitecturas MIMD con memoria compartida son sistemas de multiprocesamiento simétrico, donde múltiples procesadores comparten memoria y hacen uso de un mismo sistema operativo. Entre las arquitecturas de este tipo se mencionan: SGI/Cray Power Challenge, SGI/Cray 90, SGI/Onyx, ENCORE, MULTIMAX, SEQUENCE y BALANCE, entre otros.

Entre los modelos de programación para memoria compartida destacan, principalmente:

1. Modelos de programación *multi-threaded* con *POSIX Threads* (*Pthreads*).
2. Modelos de programación usando *OpenMP*.

Un *thread* es un semi-proceso independiente que ejecuta un conjunto de instrucciones [20] [21]. En el entorno UNIX, un *thread* existe dentro de un proceso y utiliza los recursos del proceso. La figura 1.2 muestra un ejemplo de un proceso con dos *threads*.

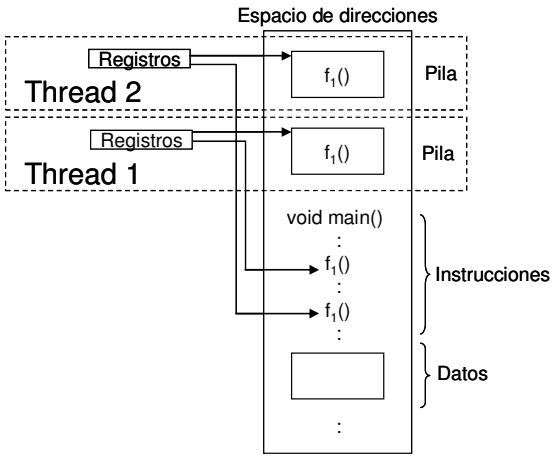


Fig. 1.2 Ejecución de un programa con un proceso y dos *threads*

Un proceso puede tener múltiples *threads*, pero todos los *threads* comparten los mismos recursos de este proceso. Como consecuencia, si algún *thread* cambia el estado de alguno de éstos recursos, el cambio es percibido por el resto de los *threads* asociados al proceso.

Para garantizar la exclusividad en las operaciones y controlar el acceso simultáneo a los datos, el programador tiene la responsabilidad de sincronizar estas operaciones. A diferencia de los procesos que tienen su propio espacio de direcciones, un *thread* comparte su memoria con otros *threads*. Un conjunto de *threads*, ejecutados en paralelo, tienen acceso al mismo espacio de variables globales y al mismo conjunto de descriptores de ficheros. Por lo tanto, los modelos de programación para memoria compartida requieren constructores (semáforos) que controlen el acceso al espacio de direcciones compartido. De esta forma, se garantiza la cohesión de los datos y la exclusión mutua en el acceso a los mismos.

Por otra parte, cuando ejecutamos múltiples *threads* en una máquina con un solo procesador, hablamos de ejecuciones concurrentes. Sin embargo, cuando ejecutamos un proceso con múltiples *threads* sobre un sistema de multiprocesamiento hablamos de paralelismo.

Debido a que los *threads* deben ejecutarse en la misma máquina, este modelo de programación es muy utilizado en arquitecturas de memoria compartida. Sin

embargo, los *threads* pueden ejecutarse en arquitecturas de memoria distribuida con soporte software de memoria compartida.

La ventaja de utilizar múltiples *threads* en un modelo de programación de memoria compartida permite que las operaciones puedan realizarse en paralelo. Además, la comunicación entre dos *threads* es generalmente más rápida que la comunicación entre dos procesos. Sin embargo, debido a que todo el conjunto de *threads* usa el mismo espacio de direcciones, esto supone que si alguno de ellos corrompe el contenido de esa memoria, el resto de los *threads* se perjudican de esta corrupción de datos.

Las implementaciones de *threads* no son novedosas actualmente en computación. Históricamente los vendedores de hardware implementaron sus propias versiones de *threads*. Estas versiones diferían de las de otros vendedores, lo que impedía desarrollar aplicaciones portables a otras máquinas. De esta forma, se desarrollaron dos estándares importantes de *threads*: *POSIX threads* y *OpenMP*.

1.2.2.1 API POSIX *threads*

Los POSIX *threads* son creados para estandarizar la programación con *threads*. En sistemas *UNIX*, esta interfaz se estandarizó en el IEEE POSIX 1003.1c (1995). Las implementaciones desarrolladas con este estándar son llamadas POSIX *threads* o *Pthreads*. Los *Pthreads* definen una API (Application Programming Interface) que incluye tipos de datos y librerías de funciones en Lenguaje C [2] [17] [26].

Los *Pthreads* requieren mucha atención por parte del programador para manejar detalles de implementación, ya que el paralelismo es explícito.

Los *Pthreads* son utilizados para mejorar el rendimiento del programa. De esta forma, si los comparamos con los procesos:

- Generan menos *overhead* por parte del sistema operativo, ya que, hacen un menor uso de los recursos del sistema,
- La comunicación entre *Pthreads* es más eficiente y más fácil de utilizar

que la comunicación entre procesos.

- Ofrecen paralelismo en un sistema de uniprocesamiento.

Las aplicaciones con *Pthreads* ofrecen otras ventajas ajenas al paralelismo, tales como, el solapamiento de cálculo con operaciones de entrada/salida y la planificación de las tareas en tiempo real.

1.2.2.2 API OpenMP

A mediados de los 90, los vendedores de máquinas de memoria compartida ofrecen la posibilidad de que los usuarios puedan indicar qué bucles pueden paralelizarse. De esta forma, surge la necesidad de crear estándares para escribir programas paralelos. ANSI X3H5 es la primera aproximación de éste estándar (1994). Sin embargo, este estándar no tuvo mucha aceptación, ya que estaba orientado a máquinas de memoria distribuida.

A pesar del fracaso de ANSI X3H5, en 1997 surge el estándar *OpenMP*. Este estándar fue apoyado por la mayoría de vendedores, incluyendo *Cray*, *Compaq*, *IBM*, *Sun* y *SGI*.

OpenMP proporciona un estándar para una gran variedad de arquitecturas de memoria compartida. Además, establece un conjunto simple y limitado de directivas para la programación, incrementando el paralelismo de la aplicación [28] [29]. *OpenMP* soporta paralelismo de granularidad fina y gruesa. Este estándar utiliza el modelo *fork-join* de ejecución de paralelismo. En este modelo, todos los programas comienzan como un proceso, llamado *master thread*. El *master thread* se ejecuta secuencialmente hasta que encuentra la región paralela especificada por el constructor *fork*. En este punto, el *master thread* crea un conjunto de *threads* paralelos. De esta forma, cuando la ejecución en la zona paralela es completada y todos los *threads* se sincronizan y terminan, el *master thread* continúa. La figura 1.3 describe gráficamente este proceso.

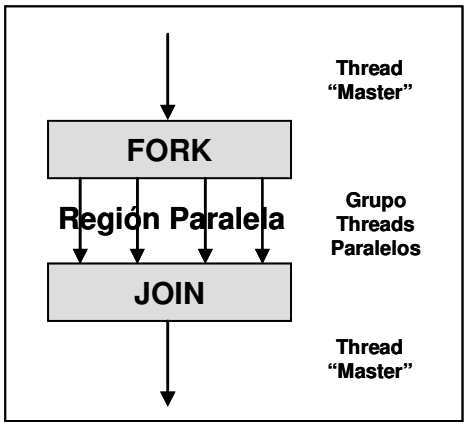


Fig. 1.3 Modelo de programación *OpenMP*

El paralelismo en *OpenMP* es especificado utilizando un conjunto de directivas de compilación disponibles en el Lenguaje C/C++ y Fortran. El programador introduce directivas de compilación, de manera que, el compilador interpreta las directivas y genera llamadas a librerías que paralelizan la región especificada. De esta forma, es posible variar el número de *threads* dinámicamente al ejecutar las regiones paralelas. Por otra parte, el programador es responsable de gestionar la sincronización de los *threads* y la dependencia de los datos.

Recientemente, se ha implementado una versión de *OpenMP* para Java llamada *JOMP* [1].

Otros modelos de programación para memoria compartida tales como: *Java Threads*, *Linda*, *Shemen* (para *Cray*), *High Performance Fortran* (*HPF*) y *Remote Threads* pueden consultarse en [12].

1.2.3 Modelo de paso de mensajes

En los inicios de la programación paralela, algunos investigadores observaron que la clave para obtener alto rendimiento en máquinas de memoria compartida, consistía en asignar los datos a diferentes localidades en las memorias del procesador. De esta forma, se maximizaba la localidad de los datos, minimizando la comunicación entre los procesadores [9]. Entonces, si cada cálculo del programa se realizaba donde residían sus datos, el programa se ejecutaba con

alta eficacia. De aquí, surge la idea de dividir el dominio de los datos en subdominios para asignarlos a diferentes procesadores. De esta forma, a cada procesador se le asigna un proceso.

Debido a que la mayoría de máquinas paralelas tienen sistemas de memoria distribuida y requieren comunicación explícita para obtener datos de memorias remotas, la utilización de librerías con paso de mensajes ha sido considerada una estrategia de programación importante [6]. Para utilizar librerías que permiten el paso de mensajes, el programador necesita una versión del programa a ejecutar en todos los procesadores.

El modelo de paso de mensajes se puede usar en arquitecturas de memoria distribuida y en arquitecturas de memoria compartida. Normalmente, este modelo se utiliza en arquitecturas MIMD (*Multiple Instruction Multiple Data*) de memoria distribuida, pero también es posible utilizarlo en arquitecturas SIMD (*Single Instruction Multiple Data*). Entre las arquitecturas SIMD se mencionan: Cray 1, Cray 2 y CIBER 205. Ejemplos de arquitecturas MIMD de memoria distribuida son: IBM SP2 y SGI/Cray T3D/T3E.

En sistemas de memoria distribuida, el paso de mensajes es un método de programación utilizado para realizar el intercambio de datos entre los procesadores. En estos sistemas, los datos son enviados de un procesador a otro utilizando mensajes. El procedimiento básico para realizar el paso de mensajes consiste en: crear los procesos para cada máquina y *enviar/recibir* los mensajes. La figura 1.4 describe el mecanismo de paso de mensajes entre dos procesadores.

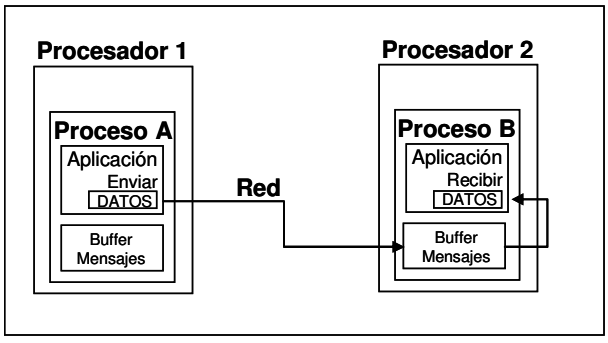


Fig. 1.4 Paso de mensajes entre dos procesadores

El paso de mensajes involucra la transferencia de datos desde un proceso que envía (*proceso A*) a un proceso que recibe (*proceso B*). El proceso que envía, necesita conocer la localización, el tamaño y el tipo de los datos, así como el proceso destino. Las comunicaciones remotas son mucho más lentas que las comunicaciones locales, por lo que el objetivo del programador consiste en minimizar las comunicaciones remotas.

En el modelo con paso de mensajes la transferencia de datos entre procesos requiere operaciones cooperativas entre los procesos que comunican, es decir, cada envío deberá tener su correspondiente recepción. La comunicación en el modelo de paso de mensajes comprende dos aspectos: la comunicación de datos entre el emisor y el receptor, y la sincronización del proceso emisor con el proceso receptor.

El estilo de programación *enviar/recibir* fue apoyado por sistemas hipercubo [8]. Sin embargo, este estilo resultó inadecuado para la programación paralela con arquitectura independiente, debido a que cada máquina ofrecía una interfaz de comunicación distinta a la de otras máquinas.

Con la llegada de la máquina paralela virtual (PVM) [11] y MPI [14], estándares para el paso de mensajes sobre máquinas homogéneas, es posible escribir programas utilizando el modelo de pase de mensajes sobre arquitecturas independientes.

Entre las ventajas que ofrece el modelo de paso de mensajes se mencionan [14]:

- **Universalidad**: El modelo de paso de mensajes se adapta a una gran diversidad de plataformas.
- **Versatilidad**: El modelo de paso de mensajes es un modelo útil y completo que permite escribir algoritmos paralelos. Este modelo proporciona el control que le falta al modelo *Data Parallel* y a los modelos basados en directivas de compilación.
- **Rendimiento:** El modelo de paso de mensajes le permite al programador

asociar datos específicos a procesos permitiendo que tanto el compilador como la gestión de memoria sean eficientes.

Existen diversas herramientas software para realizar el paso de mensajes, entre ellas se mencionan:

- Lenguaje especial de programación paralela (Occam).
- Extensión de un lenguaje secuencial de alto nivel (CC++, Fortran M (conjunto de extensiones para Fortran apropiados para el desarrollo de programas paralelos [7])).
- Lenguaje secuencial de alto nivel (C, Fortran) con librerías de paso de mensajes (MPI, PVM). Estas librerías ofrecen procedimientos externos para realizar el envío y la recepción de los mensajes entre los procesos.

En [12], se comentan herramientas y lenguajes de programación para memoria distribuida tales como: *Ada*, *MPI*, *PVM*, *Distributed Computing Environment* (DCE), *Distributed Java* (*Sockets, Remote Procedure Call (RPC), URLs, Java Space, JMPI, JPVM*).

A continuación describimos las características generales de los principales estándares para el paso de mensajes: MPI y PVM.

1.2.3.1 API Parallel Virtual Machine (PVM)

PVM es un entorno de programación que permite trabajar con un conjunto de máquinas heterogéneas, conectadas en red, como si se tratase de una máquina paralela virtual.

PVM realiza la gestión de tareas en la máquina virtual paralela, el encaminamiento de los mensajes y la conversión de los datos entre computadores. Esta gestión es transparente al usuario. El usuario escribe su aplicación como un conjunto de tareas que cooperan entre sí. Las tareas acceden a los recursos PVM mediante una librería de funciones. Esta librería permite: inicializar y finalizar tareas, realizar la comunicación y la sincronización entre tareas, y gestionar la máquina virtual [11] [31]. Principalmente, PVM esta

compuesta por:

- un proceso *daemon,* que reside en todos los computadores de la máquina virtual. El proceso *daemon* es el encargado de crear la máquina virtual para poder ejecutar aplicaciones PVM.
- una librería de funciones, y
- una consola.

Las tareas PVM tienen un identificador único. De esta forma, el destino de un mensaje se referencia mediante el identificador de la tarea destino.

En PVM, el programador escribe uno o más programas secuenciales que contienen funciones de PVM. Cada programa secuencial se compila para cada arquitectura. Los programas ejecutables generados en cada máquina, son almacenados en espacios que puedan ser accedidos por el *daemon* local de PVM.

Para iniciar la ejecución de la aplicación, el programador arranca desde un terminal la tarea encargada de arrancar otras tareas. El modelo de comunicación de PVM permite que una tarea pueda enviar un mensaje a cualquier otra. Los mensajes se etiquetan y la comunicación puede ser: punto-punto, múltiple o por difusión a un grupo.

1.2.3.2 API Message Passing Interface (MPI)

MPI es una especificación de paso de mensajes aceptada como estándar por todos los fabricantes de computadores.

El objetivo principal de MPI es proporcionar un estándar para escribir programas (Lenguaje C, Fortran) con paso de mensajes. De esta forma, se pretende mejorar la portabilidad, el rendimiento, la funcionalidad (115 rutinas) y la disponibilidad de las aplicaciones.

La tabla 1.1 contiene un resumen histórico del proceso de desarrollo de éste estándar.

Tabla 1.1. Historia cronológica del estándar MPI

Período	Logros
1980-1990	Desarrollo de librerías para el paso de mensajes entre diferentes máquinas. Las aplicaciones no son portables a otras arquitecturas. Las herramientas software son incompatibles. Surge la necesidad de desarrollar un estándar para escribir programas paralelos.
1992	Especificación de las características para realizar el estándar. Se organiza MPI Forum [4].
1993	Presentación del estándar MPI (versión inicial)
1994	Versión final del estándar MPI. Definido para los lenguajes de programación C y Fortran 77.
1996	Desarrollo de MPI-2. Definido para los lenguajes de programación C++ y Fortran 90.

La versión MPI-2 [25], es una extensión del estándar MPI. Esta versión, permite: la comunicación en una sola dirección, la gestión dinámica de los procesos y el acceso remoto a la memoria (RMA). El modelo RMA esta basado en el concepto de procesos que tienen su propia memoria local. Sin embargo, este modelo separa la comunicación de la sincronización. Un proceso puede leer o escribir los datos de otros procesos accediendo a su propia área de datos sin intercambiar mensajes. La transferencia de los datos se realiza en una sola dirección y no requiere la aceptación del otro proceso. Sin embargo, la coherencia y el acceso a la memoria deben sincronizarse explícitamente.

Este modelo de programación puede ser utilizado tanto en arquitecturas de memoria compartida como en arquitecturas de memoria distribuida [19]. El modelo de paso de mensajes en una máquina de memoria compartida puede implementarse utilizando las librerías que provee el estándar MPI 1.1 [24]. Sin embargo, el uso de estas librerías introduce grandes latencias que perjudican el rendimiento de la aplicación. Por esta razón, la plataforma de utilización de este estándar, esta dirigida a sistemas de memoria distribuida incluyendo máquinas paralelas, *SMP clusters*, estaciones de trabajo y redes heterogéneas.

En el modelo de programación de paso de mensajes, el paralelismo es explícito debido a que el programador es el responsable de identificar correctamente el paralelismo e implementarlo utilizando los constructores adecuados que provee MPI. Sin embargo, existen factores que afectan el rendimiento de las aplicaciones distribuidas con paso de mensajes. Estos factores están relacionados con:

1. ***La arquitectura***:
    - la velocidad de reloj y el número de procesadores;
    - la configuración de la memoria *cache*;
    - los adaptadores de red (latencia y ancho de banda) y
    - las características del sistema.
2. ***La red de interconexión:***
    - el hardware (routers, switch, Ethernet);
    - los protocolos de comunicación (TCP/IP, UDP/IP);
    - el enrutamiento y la contención de la red.
3. ***La aplicación:***
    - la eficiencia y la escalabilidad del algoritmo;
    - la comunicación y el balanceo de la carga;
    - los patrones de memoria;
    - la entrada/salida;
    - el tamaño de los mensajes;
    - los tipos de rutinas utilizadas (*blocking*, *non-blocking*, comunicación punto-punto y otras colectivas de comunicación).
4. ***La implementación de MPI:***
    - los protocolos utilizados para el paso de mensajes;
    - la sincronización del envío y recepción de los mensajes, y
    - la eficiencia de la implementación utilizada en el desarrollo de la rutina [5] [10] [15] [18].

MPI ofrece un buen rendimiento y una eficiencia razonable, mientras que PVM sacrifica el rendimiento para obtener flexibilidad. Sin embargo, MPI no permite interoperatividad entre diferentes implementaciones.

En arquitecturas de memoria compartida, las implementaciones de MPI, no utilizan una red de comunicaciones entre tareas, en su lugar, y por razones de rendimiento utilizan copias de memoria.

1.2.4 Modelos híbridos

Estos modelos combinan dos o más modelos de programación paralela. Un ejemplo bastante común de los modelos de programación paralela híbridos resulta de la combinación del modelo de paso de mensajes con el modelo *Pthreads* u *OpenMP*. Otro ejemplo de estos modelos es la combinación del modelo *Data Parallel* con el modelo de paso de mensajes, ya que implementaciones de F90 y HPF sobre arquitecturas de memoria distribuida utilizan el paso de mensajes para transmitir los datos entre las tareas, de forma transparente al programador.

Otros modelos de programación paralela, tales como *Single Program Multiple Data* (SPMD) y *Multiple Program Multiple Data* (MPMD) pueden construirse a partir de los modelos de programación paralela mencionados anteriormente.

1.2.4.1 Distributed Shared Memory (DSM)

En un esfuerzo por simplificar el modelo de programación para máquinas paralelas, muchos investigadores han utilizado modelos de programación de memoria compartida distribuida (DSM) como una vía para esconder algunas de las complejidades de manejar memoria y comunicación [22] [27]. En este modelo, el programador considera un único espacio de direcciones y tiene acceso a los datos dentro de ese espacio, como si se tratase de una máquina con un solo procesador. Sin embargo, el hardware o el software es el responsable de gestionar cualquier comunicación remota. En estos modelos, la granularidad y la latencia son grandes. Por lo tanto, es necesario utilizar

modelos de consistencia de memoria.

El programador es libre de utilizar métodos de programación paralela estándares basados en multiprocesos, tales como paquetes de *threads* y bucles paralelos. El principal problema de estos modelos es que para que trabajen bien, requieren ayuda del programador, ya que es necesario conocer la disposición de memoria. Otra desventaja importante de este modelo, es que la información de control entre procesos, requerida para mantener la coherencia de la memoria, tiene un porcentaje significativo de la cantidad total de comunicación.

En resumen, los modelos de programación paralela en los que nos centraremos, son modelos de programación paralela para memoria compartida y para memoria distribuida, ya que el resto de los modelos de programación presentados son una combinación de estos.

Los modelos de programación paralela no son específicos para una arquitectura. De hecho, cualquiera de los modelos de programación puede ser ejecutado sobre cualquier arquitectura. Por ejemplo:

- un modelo de programación para memoria compartida, en una máquina con memoria distribuida es *Kendall Square Research* [30].
- un modelo de paso de mensajes en una máquina de memoria compartida, es MPI sobre SGI Power Challenge.

La tabla 1.2 esquematiza la relación entre los modelos de programación y la organización de la arquitectura.

Tabla 1.2. Relación entre los modelos de programación paralela y la arquitectura

	Modelo memoria compartida	**Modelo paso de mensajes**
Arquitectura memoria compartida	Combinación natural Poco escalable Fácil de programar	Poco escalable Difícil de programar
Arquitectura memoria distribuida	Combinación poco natural Difícil de escalar Fácil de programar	Combinación natural Escalable Difícil de programar

En general, la eficiencia y el rendimiento de las aplicaciones son factores que determinan el tipo de modelo de programación a utilizar. A continuación estudiaremos las condiciones bajo las cuales resulta más eficiente la elección de un determinado modelo de programación.

1.3 Situaciones y modelos de programación

Existe una relación estrecha entre el modelo de programación paralela utilizado y las características de la aplicación (algoritmo, estructuras de datos, granularidad).

Un factor importante para determinar la elección correcta del modelo de programación es la granularidad de la aplicación, es decir, la relación entre el tiempo de cálculo y el tiempo de comunicación con otros procesos. Por ejemplo, cuando la granularidad de la aplicación es fina, los modelos de programación paralela para memoria distribuida son ineficientes debido a los costes de las comunicaciones remotas. Sin embargo, estos modelos son eficientes para aplicaciones de grano medio con algoritmos regulares. Los algoritmos regulares presentan una buena localización de los datos, lo que permite realizar un particionamiento, en tiempo de compilación, de las estructuras de datos y conocer su patrón de comunicación.

Las aplicaciones irregulares que presentan baja localidad de los datos y cálculo desequilibrado, resultan mas sencillas de programar con el modelo de *multithreading (en modelos de memoria compartida)*; ya que, transfieren datos utilizando espacios de direcciones únicas y la carga, se puede balancear mediante mecanismos dinámicos. Sin embargo, los costes de sincronización, para garantizar la coherencia de los datos en memoria, pueden ser altos. Además, pueden aparecer problemas adicionales, como por ejemplo el *false sharing*.

Sin embargo, independientemente del tipo de aplicación, necesitamos conocer la relación entre el cálculo y la comunicación, para determinar, si resultará efectiva su ejecución para la arquitectura y el modelo de programación elegido.

La tabla 1.3 contiene el tiempo medio de cálculo y de comunicación para el que

resulta apropiado utilizar cada modelo de programación. Estos tiempos fueron estimados a partir del rendimiento de aplicaciones prácticas.

Por otra parte, en importante recordar que el tiempo de cálculo depende de la cantidad de operaciones aritméticas de la aplicación, mientras que, el tiempo de comunicación representa el coste de realizar la transferencia de los datos entre los nodos conectados.

Tabla 1.3. Relación de los tiempos de cálculo y de comunicación para los modelos de programación paralela

Modelo de programación paralela	Tiempo de cálculo	Tiempo de comunicación
Memoria compartida	10 µs	1 µs
Memoria distribuida	1 ms	100 µs
Grid	1 min	10 s

En este trabajo, nos centraremos en aplicaciones de grano medio/grueso que tradicionalmente han sido programadas usando el modelo de paso de mensajes. Concretamente, nos centraremos en simulaciones explícitas de elementos finitos para aplicaciones industriales. Hasta ahora, estas aplicaciones son ejecutadas utilizando costosas máquinas paralelas.

Capítulo 2

Entornos de programación Grid

El Grid es un tipo de sistema paralelo y distribuido que permite la ejecución de aplicaciones que requieren grandes capacidades computacionales. Para este ambiente se han propuesto diversos modelos de programación, pero no existe un consenso claro que indique el modelo de programación más idóneo. Esta elección, como veremos, dependerá de las características de las aplicaciones.

En este capítulo comentamos los avances que ha tenido el Grid desde sus inicios hasta la actualidad. Particularmente comentaremos las diversas definiciones que se le han atribuido, su historia, el desarrollo de herramientas para éstos ambientes y las causas que motivan la necesidad de utilizar Grid. Concluimos el capítulo determinando el modelo de programación más adecuado para implementar nuestras aplicaciones.

2.1 Definición del entorno Grid

Grid es una infraestructura *hardware-software* que proporciona consistencia y acceso a grandes capacidades computacionales. El Grid es independiente de la distribución geográfica de los recursos y de los usuarios.

El Grid permite compartir, seleccionar y agregar recursos distribuidos a través de múltiples dominios administrativos. Los recursos son gestionados dependiendo de su disponibilidad, capacidad, rendimiento y costo, considerando además, los requerimientos de calidad de servicio que exigen los usuarios [9]. En [11] se presenta un estudio que resume las definiciones del Grid que han sido propuestas por diversos investigadores en el área. La figura 2.1 esquematiza gráficamente la estructura computacional del entorno Grid.

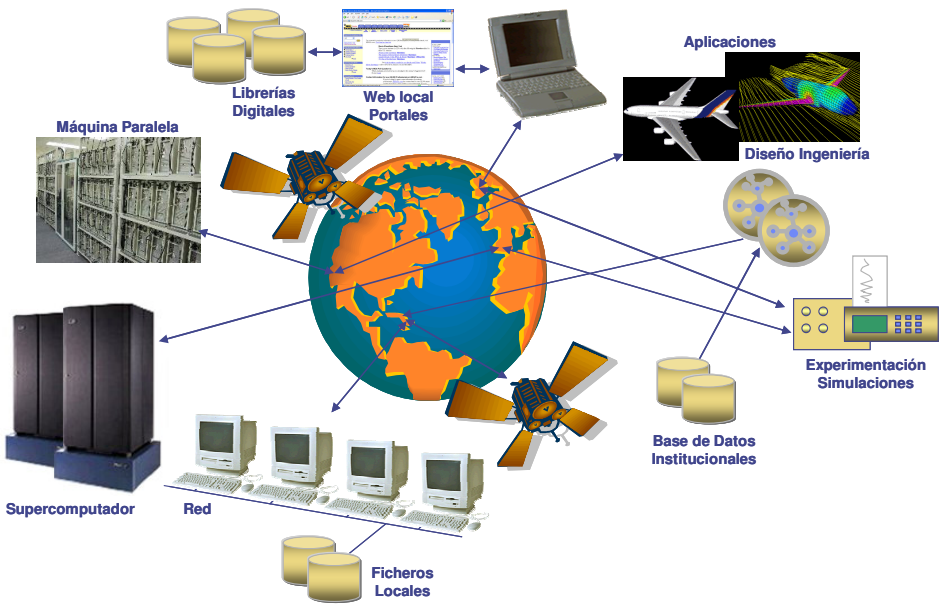


Fig. 2.1 Grid computacional

Debido al rápido crecimiento de Internet y del Web, muchos proyectos han comenzado a explotar el Web como infraestructura para ejecutar aplicaciones distribuidas y paralelas de grano grueso. En este contexto, el Web tiene la capacidad de actuar como plataforma para realizar trabajo paralelo, y de colaboración.

Las aplicaciones Grid utilizan recursos que no se encuentran situados físicamente en un mismo lugar. De esta forma, el Grid no es solamente una infraestructura que ejecuta cálculo. Es, además, una tecnología que puede enlazar y unificar diversos recursos distribuidos geográficamente, que van desde los supercomputadores paralelos a computadores personales.

Por otra parte, si comparamos el entorno Grid con el ambiente paralelo estándar observaríamos que difiere en muchos aspectos, ya que:

1. Un computador paralelo, en general, es homogéneo, mientras que el Grid puede incorporar procesadores de diferentes tipos y con diversos tamaños de memoria.
2. Un computador paralelo generalmente, tiene una red de comunicaciones dedicada y optimizada, caracterizada por grandes anchos de banda. Sin embargo, el Grid puede tener una red de comunicaciones heterogénea y desequilibrada, abarcando diversas redes y una variedad de conexiones Internet. Estas conexiones caracterizan anchos de banda y latencias variables en el tiempo y en el espacio.
3. Un computador paralelo tiene una configuración estable, mientras que la disponibilidad de un recurso en el Grid puede ser variable.
4. Un computador paralelo ejecuta un único sistema operativo y proporciona utilidades básicas tales como: sistema de ficheros y controlador de recursos. En contraste, el Grid contiene una colección de máquinas que integran diferentes utilidades y sistemas operativos.

La figura 2.2 muestra los componentes del Grid que hacen posible el acceso a los recursos. Entre estos componentes destacan:

1. La colección de recursos distribuidos que pueden ser utilizados vía Internet. En esta categoría se encuentran computadores, sistemas operativos (UNIX, Windows), dispositivos de almacenamiento, bases de datos e instrumentos científicos especiales.
2. La gestión de servicios para procesamiento remoto, ubicación de los recursos, información, seguridad y aspectos relacionados con la calidad del servicio.
3. Los ambientes que permiten el desarrollo de aplicaciones y herramientas de programación, así como también, la gestión de recursos y la planificación de tareas.
4. Los portales Web desarrollados para que los usuarios puedan visualizar y obtener los resultados al ejecutar sus aplicaciones utilizando recursos específicos.

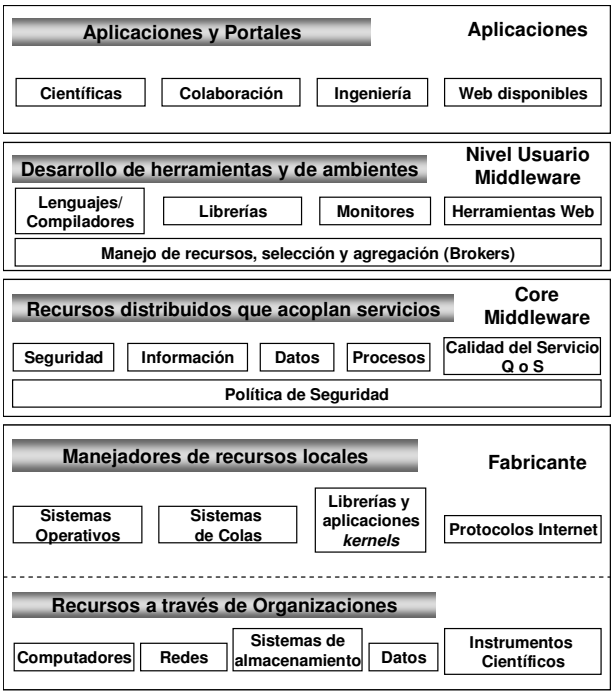


Fig. 2.2 Principales componentes del Grid

2.2 Necesidad de utilizar el Grid en las simulaciones numéricas de ingeniería

La simulación numérica de fenómenos físicos derivados de la ciencia y la ingeniería requiere la utilización de modelos matemáticos complejos. Ejemplos de este tipo de aplicaciones son las: simulaciones para el diseño de turbinas, simulaciones de vuelo, simulaciones del estampado de chapa [79], simulaciones de impactos [35], simulaciones astrofísicas [64], simulaciones climáticas [59], etc. (Más detalles en [45]). Estas simulaciones numéricas requieren alta precisión en los resultados para resolver detalles a escala fina [28] [77], lo que justifica el uso de hardware especializado. Entre las opciones de hardware especializado se encuentran:

1. **Los supercomputadores.** Para este caso, las comunicaciones entre procesos son relativamente pequeñas. Además, no se requiere la adaptación de los algoritmos al ambiente de programación. Sin embargo, la adquisición del hardware es costosa, por lo las empresas no pueden disponer de ellos.
2. **El Grid.** Esta opción resulta más barata que la anterior, ya que depende del servicio de conexión que se contrate y de la compra o reutilización de ordenadores de gran potencia [28]. Todo esto contribuye a que este hardware sea fácil de utilizar. Este hardware puede estar geográficamente distribuido formando un Grid computacional. Sin embargo, cuando utilizamos entornos Grid para realizar las simulaciones, podemos tener algunos inconvenientes. Entre estos inconvenientes destacan las altas latencias y la tolerancia a fallos. Ambos problemas representan temas importantes de investigación [7] [8] [15]. Adicionalmente, para trabajar en este entorno de programación es necesario reprogramar los códigos.

Cada una de las opciones anteriores tiene ventajas y desventajas de utilización. En nuestro caso, nos interesa disponer de hardware especializado para realizar las

simulaciones considerando un rendimiento razonable. De esta manera, la opción mas apropiada para realizar las simulaciones numéricas es la utilización del entorno Grid, ya que la mayoría de las empresas pueden adquirir hardware a bajo coste y realizar sus simulaciones.

Por otra parte, las aplicaciones ejecutadas sobre los entornos de programación Grid han sido clasificadas considerando sus características [28]. La tabla 2.1 resume las categorías identificadas.

Tabla 2.1. Características y tipos de aplicaciones Grid

Categoría	Ejemplos	Características
Supercomputación distribuida	Simulaciones interactivas para entrenamiento y logística militar. Simulaciones de procesos complejos como: dinámica estelar y aplicaciones químicas.	Problemas de grandes dimensiones que realizan cálculo intensivo.
Aplicaciones desacopladas	Animaciones computacionales, criptografía, estudios paramétricos. Simulaciones moleculares de cristales líquidos.	Obtener rendimiento de recursos ociosos para aumentar el rendimiento de las aplicaciones.
Sobre demanda	Instrumentación médica. Procesamiento de datos desde satélites meteorológicos y procesamiento de imágenes en tiempo real.	Recursos remotos integrados con cálculo local, para mejorar los tiempos de ejecución.
Datos intensivos	Datos físicos y astronómicos. Pronóstico del tiempo.	Síntesis de información obtenida de grandes bases de datos
Colaborativas	Exploración de datos geofísicos, diseño de sistemas de control de incineradores industriales, creación de aplicaciones de realidad virtual para entretenimiento y educación.	Soporte de comunicación o trabajo conjunto entre múltiples aplicaciones.

De esta forma, nuestras simulaciones numéricas pertenecen a la categoría de supercomputación distribuida. Sin embargo, estas aplicaciones no son completamente desacopladas, por lo que necesitamos utilizar técnicas de reordenación de los datos para distribuirlos convenientemente en el entorno Grid. Como veremos, nuestras propuestas de distribuciones de los datos, mejorarán el rendimiento de la aplicación, ya que fueron pensadas para solapar cálculo con comunicaciones remotas.

2.3 Historia y evolución del Grid

Durante la década de los 80's, se realizaron investigaciones y desarrollos *hardware-software* para computadores paralelos [9]. Los desarrollos *software* permitían gestionar la comunicación entre procesos y la ejecución de códigos en entornos paralelos. Las aplicaciones eran diseñadas aprovechando las características de las arquitecturas: memoria compartida y memoria distribuida. La infraestructura computacional a gran escala proporcionaba una herramienta fundamental para conseguir nuevos avances tecnológicos [65]. Estos desafíos y el trabajo multidisciplinario proporcionaron un modelo de colaboración que ha tenido un gran impacto en la ciencia: el entorno Grid.

Inicialmente se pensó que el Grid tendría mas éxito en la computación paralela aplicada a *clusters* acoplados distribuidos geográficamente. En la práctica, el Grid ha sido utilizado como una plataforma para integrar aplicaciones acopladas con diversos componentes que pueden ejecutarse en máquinas paralelas. La evolución del Grid se describe en tres generaciones. A continuación detallamos los avances tecnológicos de cada generación [21].

2.3.1 Primera generación

Los esfuerzos iniciales del Grid comenzaron como proyectos que pretendían vincular diferentes sitios de supercomputación. De esta forma, el origen del término Grid es atribuido al proyecto CASA, desarrollado por US en 1989. Particularmente, éste término se le atribuye al director de NCSA, Larry Smarr [17].

A mediados de los años 90, surge la necesidad de realizar supercómputo en entornos Grid. El objetivo era proporcionar recursos computacionales que permitieran mejorar el rendimiento de las aplicaciones.

Los proyectos más representativos de esta generación fueron: FAFNER [23] e I-WAY [31]. A continuación describimos las características más importantes de cada uno de estos proyectos.

Proyectos desarrollados

FAFNER (Factoring via Network-Enabled Recursion): Este proyecto se inició en 1995 por los *Laboratorios Bellcore*, la *Universidad de Syracuse* y otras organizaciones. FAFNER permitía la factorización de claves públicas encriptadas (RSA) utilizando una técnica numérica llamada *Number Field Sieve* (NFS) y servidores Web.

De esta forma, cuando se envía un mensaje encriptado por RSA, el emisor busca la clave pública de cifrado del receptor y cuando el mensaje llega al receptor, éste se encarga de descifrarlo utilizando la clave oculta. Los mensajes enviados utilizando el algoritmo RSA se representan mediante números. Su funcionamiento se basa en el producto de dos números primos superiores a 10^{100} elegidos al azar para formar la clave de descifrado. La seguridad de este algoritmo radica en que no existe una forma rápida de factorizar un número tan grande en sus factores primos utilizando los tradicionales computadores. FAFNER proporcionaba solución a este problema de factorización y abría camino a otros proyectos para realizar supercómputo vía Web.

I-WAY (Information Wide Area Year): Este proyecto se inició en 1995, y fue presentado en la Conferencia *Supercomputing'95*. I-WAY fue una red experimental de alto rendimiento que integraba los recursos (máquinas y herramientas) de los mayores centros computacionales de los Estados Unidos. Este proyecto, permitió el desarrollo de un planificador de recursos conocido con el nombre de *Computational Resource Broker* (CRB). I-WAY requirió la adaptación de herramientas, tales como *Nexus* [32], para poder trabajar. *Nexus* soportaba mecanismos de configuración automática que permitían elegir las configuraciones de red más apropiadas, dependiendo de la tecnología utilizada. Ejemplo de esto fue la elección del protocolo de comunicación que resultaba más apropiado para Internet (TCP-IP o AAL5)). Básicamente, este proyecto consideraba diferentes tipos de aplicaciones, entre ellas: supercomputación, acceso a recursos remotos, realidad virtual, video y Web.

Tanto FAFNER como I-WAY integraban recursos para realizar cálculo intensivo. Sin embargo, estos proyectos se diferenciaban en:

1. El tipo de aplicaciones. FAFNER se adaptaba a una aplicación que permitía realizar la factorización de un número en factores primos, por lo que era una aplicación paralela que no dependía de una interconexión rápida. Por otra parte, I-WAY fue diseñada para hacer frente a una amplia gama de aplicaciones de alto rendimiento que necesitaban una conexión rápida y utilizar potentes recursos. No obstante, ambos proyectos carecían de escalabilidad [21].
2. Los recursos. En FAFNER sus clientes utilizaban máquinas remotas de bajo rendimiento. I-WAY integraba recursos de diferentes centros de supercomputación.

En general, los sistemas de primera generación ofrecieron la posibilidad de compartir recursos para realizar cálculo intensivo y mejorar el rendimiento de las aplicaciones.

2.3.2 Segunda generación

Los primeros inicios del Grid, fundamentaron sus esfuerzos en la necesidad de integrar diferentes centros de supercomputación. El proyecto I-WAY consigue este objetivo. Sin embargo, el Grid incluyendo nuevos estándares y tecnologías. Esto hizo que el Grid fuera una infraestructura distribuida utilizada por aplicaciones que requerían grandes cantidades de datos y cálculo intensivo.

La segunda generación trabaja la integración de aplicaciones software ejecutadas sobre ambientes heterogéneos distribuidos [27] [33] [34]. Estas aplicaciones consideran grandes volúmenes de datos por lo que requieren gran potencia. Básicamente, esta generación confronta tres puntos importantes:

1. **Heterogeneidad:** El Grid está formado por múltiples recursos de naturaleza variada.
2. **Escalabilidad:** El Grid puede contener una cantidad variable de recursos. Esto plantea el problema de la degradación potencial del rendimiento al aumentar los recursos. En consecuencia, las aplicaciones que requieran gran cantidad de recursos distribuidos geográficamente, deberán ser implementadas explotando la localidad de acceso a los recursos y la tolerancia a grandes latencias.
3. **Adaptabilidad:** Debido a la gran cantidad de recursos que involucra el Grid, la probabilidad de que falle alguno de estos recursos es alta. Por lo tanto, ante un(os) fallo(s), los gestores de recursos o las mismas aplicaciones deberán disponer de mecanismos que permitan adaptar su comportamiento/ejecución dinámicamente.

De esta forma, la segunda generación toma en cuenta como características de diseño: los servicios de comunicación, los servicios de información, los servicios de nombres, los sistemas distribuidos de ficheros, la seguridad y la autorización, la tolerancia a fallos y el estado del sistema, el manejo y la planificación de los recursos. A continuación describimos los principales proyectos desarrollados en esta generación.

Proyectos desarrollados

Los proyectos desarrollados en esta generación fueron clasificados en categorías. Estas categorías son: tecnologías *core*, sistemas objetos distribuidos, planificadores y gestores de recursos, portales Grid, sistemas integrados y sistemas de comunicación punto-punto.

a) Tecnologías *core*

En esta categoría se encuentran los proyectos que desarrollaron la infraestructura básica del Grid. Entre los principales proyectos de esta categoría se mencionan:

Globus y Legion.

Globus: Este proyecto es el esfuerzo de investigación desarrollado por diferentes instituciones. Permitía la construcción de Grid computacionales [26] [83]. Surge a partir del proyecto I-WAY y proporcionaba una infraestructura software que permitía gestionar recursos distribuidos geográficamente como si se tratase de una única máquina virtual. Un elemento importante de Globus es *Globus Toolkit*. *Globus Toolkit* define los servicios y las capacidades básicas requeridas para construir un Grid computacional. Está constituido por un conjunto de componentes que implementan servicios básicos, tales como, la seguridad, las comunicaciones, la localización y la gestión de los recursos. La evolución de Globus continúa con la introducción de *Open Grid Services Architecture* (OGSA) [34], una arquitectura Grid basada en servicios Web y Globus.

Legion: Este proyecto fue un proyecto desarrollado por la Universidad de Virginia. Legion proporcionó la infraestructura software para que un sistema de máquinas heterogéneas, geográficamente distribuidas, pudiera interactuar [42] [43] [55]. Se diferenció de Globus, en que, gestionaba sus recursos como objetos. Esta metodología permitió que Legion heredase todas las ventajas del diseño orientado a objeto, tales como, la abstracción y el encapsulamiento de los datos. En el 2001, Legion cambia su nombre a Avaki.

b) Sistemas objetos distribuidos

En esta categoría, se comentan los principales proyectos que contribuyeron a proporcionar la infraestructura software para la integración y gestión de recursos distribuidos. Entre los principales proyectos que definen esta categoría se encuentran: Jini-RMI y *Common Component Architecture Forum*.

Jini-RMI: Este proyecto fue diseñado para proporcionar una infraestructura software para entornos distribuidos [49]. Jini permite la comunicación entre los dispositivos utilizando un mecanismo de Java RMI. En Jini un dispositivo o servicio

software se puede integrar a la red y formar parte de ella. Entre sus funciones se destacan, la búsqueda y el acceso a un servicio, además de la monitorización del estado de los recursos remotos y del sistema.

Common Component Architecture Forum: Este proyecto definió el conjunto mínimo de características estándares necesarias en la construcción de componentes *frameworks*. Estos componentes podían ser desarrollados por diferentes grupos institucionales. La idea era que estas componentes pudiesen integrarse y adaptarse para reutilizar códigos, acelerando de ésta forma, el desarrollo de una aplicación [6]. Las tecnologías de Globus y Legion explicadas en la categorización anterior, permitían implementar servicios sin necesitad de utilizar estas componentes.

c) Planificadores y gestores de recursos

Planificadores: En esta categoría se describen los principales proyectos que permitieron realizar la planificación de los recursos.

Los principales proyectos de esta categoría fueron: Condor, PBS, SGE y LSF.

Condor: Este proyecto es un software que permite ejecutar lotes de tareas sobre en una gran variedad de plataformas UNIX. Entre las características principales de Condor se destacan la localización automática de los recursos, la asignación automática de trabajo, y la migración de procesos. Condor permite monitorizar la actividad de todos los recursos que participan en el cálculo, así como también, desarrollar, implementar y evaluar mecanismos (políticas) que proporcionan altos rendimientos de procesamiento, considerando diferentes recursos distribuidos [81]. En la actualidad este proyecto se encuentra integrado por 30 facultades y por personal que trabaja en la Universidad de Wisconsin-Madison.

Portable Batch System (PBS): Este proyecto es un sistema que permite gestionar peticiones y cargas de trabajo [68].

Originalmente fue desarrollado por la NASA y opera sobre diferentes plataformas

UNIX. PBS se adapta a una amplia variedad de políticas administrativas y proporciona un modelo de seguridad.

Sun Grid Engine (SGE): Este proyecto es un planificador de recursos cuyo software esta basado en Codine/GRM [80]. Cuando se realiza la petición de un trabajo, el usuario debe realizar la especificación de recursos que dicho trabajo necesita. SGE identifica la cola del servidor en la que debe ser encolado el trabajo. Los trabajos esperan ser atendidos en las colas de los servidores que proporcionan el servicio que han solicitado. Cuando la cola se encuentra lista para desencolar un nuevo trabajo, SGE determina cuál será el siguiente trabajo que será atendido. Esta elección, depende de la prioridad del trabajo o del tiempo que lleve en cola.

Load Sharing Facility (LSF): Este proyecto es un sistema utilizado para el manejo de sistemas comerciales [67]. LSF permite monitorizar y analizar recursos y cargas de trabajo sobre redes heterogéneas considerando la tolerancia a fallos.

Gestores: En esta categoría se describen los proyectos más destacados que permitieron gestionar los recursos. Entre los principales proyectos se mencionan: SRB y Nimrod-G.

Storage Resource Broker (SRB): Este proyecto fue desarrollado por el centro de supercomputación de San Diego (SDSC) con el objetivo de proporcionar acceso uniforme al almacenamiento distribuido. SRB es gestionado por un administrador con autorización para crear grupos de usuarios. Además, soporta *metadata* (objeto de información) asociados a sistemas de ficheros distribuidos, tales como localización, tamaño y fecha de creación de la información. Esto hace que SRB resulte atractivo para ser utilizado sobre entornos Grid [69] [74].

Nimrod-G: Este proyecto monitoriza la ejecución de los trabajos asignados a los recursos que gestiona [1] [13].

d) Portales Grid

Los proyectos de esta categoría permiten acceder a recursos específicos desde un dominio particular utilizando una interfaz Web. Entre los principales proyectos que definen esta categoría se mencionan: NPACI HotPage, SDSC Grid Port Toolkit y Grid Portal Development Kit.

NPACI HotPage: Este proyecto es un portal de usuario diseñado para permitir el acceso a los recursos. Esto permite que los recursos puedan ser vistos como un Grid o como recursos individuales. Específicamente, este portal proporciona información y acceso a los recursos, así como servicios que gestionan el manejo de los mismos. NPACI HotPage consulta el estado de los recursos y tiene un mecanismo que permite asociar las aplicaciones a los recursos [46].

SDSC Grid Port Toolkit: Este proyecto esta formado por un conjunto de herramientas reutilizables para desarrollo de portales Gris que utiliza infraestructura *HotPage* [75]. Los principales componentes de Grid Port Toolkit son: los portales de servicios Web y las aplicaciones API. Los portales de servicios Web proporcionan seguridad de conexión al Grid, mientras que las aplicaciones API proveen una interfaz Web que ayuda a los usuarios finales a desarrollar portales modificados para requisitos particulares.

Grid Portal Development Kit: Este proyecto promueve el desarrollo común de componentes y de utilidades necesarias para facilitar la construcción de portales. Además, permite la interacción de diferentes portales utilizando la misma infraestructura [63].

e) Sistemas Integrados

Un conjunto de grupos internacionales han integrado de forma coherente los componentes desarrollados para Grid y han formado sistemas integrados. Entre los proyectos desarrollados para esta categoría se mencionan: Cactus, DataGrid, UNICORE, WebFlow.

Cactus: Este proyecto fue diseñado para ser utilizado por ingenieros y científicos. Cactus presentaba una estructura modular que permitía la ejecución de aplicaciones paralelas considerando arquitecturas distribuidas y desarrollando códigos entre diferentes grupos [4].

DataGrid: Es un proyecto financiado por la Comunidad Europea que proporcionaba un entorno Grid para realizar cálculo intensivo y consulta de bases de datos distribuidas [22].

UNICORE: Es un proyecto que permitía la fácil utilización de GUI, considerando una arquitectura abierta basada en el trabajo abstracto. UNICORE proporcionaba seguridad y un mínimo de interferencia con los procesos administrativos locales. UNICORE utilizaba tecnologías Web y Java. Además, permitía que el usuario modificara o especificara la estructura del Grid haciendo uso de una interfaz gráfica sobre estaciones de trabajo UNIX o PC *Windows* [5] [85].

WebFlow: Es una extensión del modelo Web para entornos distribuidos de área amplia (WAN) [3] [44]. Su principal objetivo estuvo orientado a publicar módulos computacionales en el Web. De esta forma, los módulos WebFlow podían ser reutilizados por otros usuarios. Esto permitía la construcción/composición de las aplicaciones.

f) Comunicación punto-punto (P2P)

El modelo de comunicación punto-punto trata el problema de cuellos de botella que caracteriza al modelo cliente-servidor, permitiendo una mejor escalabilidad [19]. Las máquinas se pueden comunicar directamente y gestionar tareas sin utilizar servidores centrales. El proyecto desarrollado mas destacado para esta categoría es JXTA.

JXTA: Este proyecto fue creado por Sun Microsystems (2001). JXTA desarrolla infraestructura y aplicaciones para realizar comunicación punto-punto. Más que un

software, JXTA es una especificación que permite describir un sistema de red [50].

En la segunda generación las tecnologías *core,* fueron utilizadas para desarrollar servicios orientados a dar solución a aplicaciones que requerían cálculo intensivo, precisión y rendimiento eficiente. Esta generación, también consideró el desarrollo de un conjunto de herramientas y de utilidades, las cuales permitieron desarrollar servicios de alto nivel tanto para las aplicaciones como para los usuarios (portales Web, interfaces).

2.3.3 Tercera generación

La segunda generación proporcionó la interoperatividad necesaria para realizar cálculo a gran escala. Sin embargo, para realizar esto era necesario considerar el desarrollo de las aplicaciones Grid para que reutilizaran componentes y recursos. Esto debía hacerse de forma flexible. Por esta razón, en esta generación, se incrementa la utilización de modelos orientados a objeto. Básicamente, los sistemas de tercera generación se orientan al servicio, permiten *metadata* y exhiben características propias de *autonomic computing*.

El desarrollo de nuevos modelos de servicio estaba orientado a adaptar el cálculo de la aplicación ante posibles cambios/fallos, considerando servicios Web y cálculo basado en agentes. Entre los desarrollos aportados por esta generación al Grid, se mencionan: Servicios Web, Open Grid Services Architectures y Agentes.

Proyectos desarrollados

Servicios Web: Este proyecto crea servicios estándares considerando especificaciones concretas, tales como el tipo de protocolo (XML), el lenguaje de descripción de los servicios (WSDL) y la especificación para el registro distribuido del servicio Web (UDDI). Entre las herramientas consideradas para desarrollar éstos servicios se mencionan:

Web Services Flow Lenguage (WSFL): Fue propuesto por IBM, y define *workflows* como una combinación de servicios Web [87].

XLANG: Fue desarrollado por Microsoft y permite realizar transacciones complejas que pueden requerir múltiples servicios Web [88].

Open Grid Services Architecture (OGSA) framework: OGSA es un proyecto que apoya la creación, el mantenimiento, y el uso de servicios de organizaciones virtuales. Un servicio se define como una red que proporciona recursos, programas, bases de datos e inclusive otras redes. Se espera que futuras implementaciones de *Globus Toolkit* estén basadas en la arquitectura OGSA [34].

Agentes: Como hemos visto hasta ahora, los servicios Web proporcionan la interoperatividad entre los recursos que integran el Grid, y OGSA adapta estos servicios al Grid. Sin embargo, la integración de estos servicios Web no proporciona soluciones a problemas que se presentan en grandes aplicaciones distribuidas, como por ejemplo la tolerancia a fallos. Por esta razón, se piensa en otros modelos orientados al servicio que solucionen estos problemas. Se propone un modelo de servicio basado en agentes. En este modelo, el Grid es visto como un conjunto de componentes (recursos) que interactúan entre sí intercambiando mensajes. El modelo basado en agentes se diferencia de los modelos orientados a servicios, en que, permite una adaptación dinámica a los cambios en el entorno Grid cuando se ante posibles fallos de los recursos. En este sentido, se desarrollaron estándares para la interacción de agentes y de sistemas basados en agentes [82].

Hasta aquí, hemos comentado todas las herramientas desarrolladas hasta la actualidad para trabajar sobre entornos Grid. Todos los proyectos desarrollados contribuyen a fortalecer la utilización de los recursos y el rendimiento de las aplicaciones distribuidas de gran escala. Sin embargo, es necesario adaptar los códigos de las aplicaciones a estos entornos. A continuación describimos las herramientas y los modelos de programación para Grid más utilizados hasta el momento.

2.4 Programación en entornos Grid

El objetivo principal de la programación Grid es el estudio de modelos de programación, herramientas y métodos que permitan el desarrollo de algoritmos portables a arquitecturas distribuidas.

La programación Grid exige contar con propiedades y capacidades que no tiene la programación secuencial o incluso la programación paralela. Además, la programación Grid debe permitir realizar operaciones sobre estructuras de datos distribuidas en diferentes máquinas.

Un programador de aplicaciones Grid debe poder gestionar el cálculo sobre ambientes heterogéneos y dinámicos. Además, debe poder diseñar la interacción de servicios remotos, bases de datos y recursos *hardware*.

A pesar de que existen herramientas que permiten el desarrollo de aplicaciones Grid, existe un consenso generalizado que admite que tales herramientas resultan insuficientes para realizar desarrollos eficientes de códigos en éstos ambientes.

Un entorno Grid se utiliza cuando se necesita realizar cálculo a gran escala de manera eficiente (alto rendimiento). Para hacer esto, resulta necesario obtener un balance entre la carga de trabajo (cálculo) y la comunicación entre máquinas. Actualmente, esto se realiza gestionando el cálculo, la comunicación y la localidad de los datos. Para realizar esto se utiliza el modelo de programación con paso de mensajes o, llamadas a procedimientos remotos (RMI). Esto exige que el programador sea consciente de la transferencia de los datos entre las máquinas o procesadores que realizan la comunicación. Para conseguir altos rendimientos en estos ambientes se requieren aplicaciones tengan granularidad gruesa.

Considerando lo anterior, a continuación describimos las características de los modelos de programación en entornos Grid junto con las herramientas y los modelos de programación mas conocidos hasta el momento [52].

2.4.1 Características de los modelos de programación

El término modelos de programación no se refiere a lenguajes de programación. Un modelo de programación puede referirse a diferentes tópicos, como por ejemplo, un lenguaje de programación, una librería MPI o una herramienta.

A continuación se comentan las propiedades más comunes en los modelos de programación actuales [76]. Los modelos de programación Grid también heredan éstas propiedades.

1. **Portabilidad, interoperatividad y adaptabilidad**

   Los lenguajes de programación de alto nivel permiten desarrollar códigos independientes en cada procesador. Los modelos de programación Grid permiten desarrollar códigos con portabilidad similar permitiendo además la utilización de servicios ubicados en diferentes localizaciones con una funcionalidad equivalente. La idea de utilizar una gran variedad de servicios y de códigos introduce la idea de interoperatividad. El Grid permite considerar una arquitectura abierta y extensible, lo que implica un ambiente distribuido que soporta diferentes protocolos, servicios y software.

   La portabilidad de los códigos y la interoperatividad entre los recursos derivan en adaptabilidad. Un programa Grid puede adaptarse a diferentes configuraciones dependiendo de la disponibilidad que se tenga de los recursos.

2. **Rendimiento**

   El rendimiento de las aplicaciones en entornos Grid está determinado por la influencia del ancho de banda y la latencia. Estos factores pueden influir en el rendimiento de las aplicaciones y en la utilización de los recursos. La comunicación y el cálculo pueden ser soportadas en entornos Grid, pero estos podrían dificultar mejorar el rendimiento de las aplicaciones acopladas.

3. **Tolerancia a fallos**

   Las aplicaciones Grid deben adaptarse a posibles fallos de los recursos o de las comunicaciones entre máquinas, y por lo tanto, tal situación debe considerarse al programar éstas aplicaciones.

4. **Seguridad**

   Las aplicaciones Grid son ejecutadas en una amplia variedad de dominios. Cada uno de estos dominios, requiere una administración adecuada para gestionar los recursos que proporciona. Para lograr esto, es necesario integrar en las aplicaciones mecanismos que permitan garantizar tanto la autentificación del usuario como la seguridad del dominio.

2.4.2 Herramientas y modelos de programación para Grid

Las últimas décadas de investigación y desarrollo en programación distribuida y paralela y en el diseño de sistemas distribuidos, han permitido adaptar algunos de los modelos de programación para desarrollar aplicaciones que se ejecutan en entornos Grid.

A continuación comentamos el conjunto de herramientas, lenguajes, modelos de programación y ambientes desarrollados para el Grid [53] [76].

1) Modelos de estado compartido

Estos modelos generalmente utilizan lenguajes síncronos y modelos de ejecución pensados para máquinas de memoria compartida o distribuida, considerando una red de interconexión con grandes anchos de banda y bajas latencias.

Para anchos de banda pequeños y grandes latencias, las herramientas actuales no resultarían efectivas. Sin embargo, existen modelos de programación basados en estados compartidos en los cuales los productores y los consumidores de datos están desacoplados.

Entre éstos modelos se mencionan:

a) **JavaSpaces:** JavaSpaces es una implementación de Java basada en el concepto de espacio de Linda. Un espacio se define como un repositorio de objetos que se encuentran accesibles vía red. Los procesos utilizan el repositorio como un mecanismo de intercambio de objetos con otros procesos. En éste modelo, una aplicación es vista como una colección de procesos que comunican entre ellos, colocando y obteniendo objetos. Un programador que quiera utilizar esta herramienta para diseñar aplicaciones Grid deberá enfocar las estructuras de datos distribuidas como objetos almacenados. Actualmente existen implementaciones de JavaSpaces para Grid utilizando Java y basadas en Globus [70] [72].

2) Modelos de paso de mensajes

Para modelos con paso de mensajes los procesos se ejecutan en espacios de direcciones separados, y la información entre los procesos se intercambia utilizando mensajes. Cuando se utiliza este modelo, la paralelización de aplicaciones puede resultar engorrosa pero el programador dispone del control total de la aplicación. Estos modelos requieren que el programador sepa donde tendrá lugar la comunicación. Sin embargo, esto puede llegar a ser complicado de determinar en entornos Grid.

a) **MPI y sus variantes.** La interfaz de paso de mensajes MPI [60] [61] es el estándar más utilizado que contiene librerías que permiten enviar y recibir mensajes entre procesos. Se han desarrollado nuevas implementaciones y variantes de MPI [25] [37] [47] [51]. Una de las más importantes para entornos Grid es MPICH-G2.

MPICH-G2 [25] es una implementación de MPI disponible para Grid que utiliza servicios de Globus y permite que los programadores puedan acoplar múltiples máquinas de diferentes arquitecturas utilizando MPI. Esta implementación libera al usuario de la engorrosa tarea de especificar los detalles del sitio de conexión.

La popularidad de MPI ha hecho que tenga una gran cantidad de variantes, que pueden ser utilizadas en entornos Grid. Esto permite gestionar dinámicamente los procesos y hace que las operaciones colectivas se realicen de forma más eficiente. La librería *MagPIe* [51], por ejemplo, implementa operaciones colectivas de MPI (*broadcast*, *barrier* y *reduce)*. Estas implementaciones consideran optimizaciones para entornos Grid. Además, existen aplicaciones paralelas de MPI que pueden ejecutarse sobre plataformas Grid utilizando *MagPIe*.

PACX-MPI [37] es otra implementación de MPI que optimiza las operaciones colectivas y soporta la comunicación entre máquinas.

b) Paso de mensajes unidireccional. Se pueden realizar comunicaciones MPI unidireccionales utilizando MPI-2 [61], aunque también se pueden realizar comunicaciones en las dos direcciones [10]. Para las comunicaciones unidireccionales, no es necesario realizar una operación *de envío* antes de realizar una operación *de recepción* (comunicaciones asíncronas).

Entre las herramientas actuales que permiten la comunicación unidireccional mencionamos a Nexus [30]. Nexus soporta múltiples protocolos disponibles para ambientes Grid.

3) Modelos RPC y RMI

Cuando se utilizan modelos de programación con paso de mensajes, el programador necesita conocer explícitamente los argumentos de las primitivas de comunicación para iniciar el envío de los datos. La semántica asociada a cada tipo de mensaje se define estáticamente en la aplicación. Los modelos con paso de mensajes en una sola dirección, no cumplen con esta especificación, ya que no necesitan hacer una recepción para poder enviar datos. Esto permite que el proceso emisor realice procesamiento remoto.

Los modelos de Remote Procedure Call (RPC) y Remote Method Invocation (RMI) proporcionan éstas capacidades, pero estructuran la interacción entre

emisor y receptor más como un lenguaje constructor que como la llamada a una librería que transfiere datos entre dos puntos de transmisión. Los modelos RPC y RMI son mecanismos simples que permiten gestionar cálculo remoto. Estos modelos pueden utilizarse para construir otras herramientas útiles en la programación Grid, tales como componentes *frameworks* y servicios de redes.

Ejemplos actuales de algunas implementaciones de estos modelos en ambientes Grid son GridRPC [62] y Java RMI [38].

4) Modelos híbridos

Algunas aplicaciones pueden ejecutar *multithreaded* sobre un espacio de direcciones compartido, y transferir datos y flujos de control entre diferentes máquinas. Tal situación se presenta en *clusters* de multiprocesadores simétricos sobre entornos Grid. A continuación se mencionan algunos de éstos modelos:

a) OpenMP y MPI

OpenMP [66] es una librería que soporta programación paralela sobre máquinas paralelas con memoria compartida. Esta librería ha sido desarrollada por un consorcio de vendedores con el objetivo de producir una interfaz estándar de programación para máquinas paralelas con memoria compartida que utilizan lenguajes como Fortran, C y C++. OpenMP tiene primitivas que permiten definir datos compartidos y la sincronización de procesos.

La combinación de OpenMP y MPI en aplicaciones orientadas a *clusters* de multiprocesadores simétricos sobre ambientes Grid, ha sido considerada por varios grupos [78]. Una consideración importante, es el orden de secuenciamiento de las primitivas de OpenMP y MPI. Por ejemplo, si consideramos primitivas de OpenMP y luego de MPI, se necesita que MPI sea un *thread-safe* o que la aplicación maneje explícitamente el acceso a la librería de MPI. Por el contrario, si se colocan primitivas de MPI por encima de OpenMP, necesitamos sincronización adicional. Esto limita la cantidad

de paralelismo que OpenMP pueda realizar.

b) OmniRPC

OmniRPC [73] fue diseñado específicamente como un *thread-safe* RPC facilitado para *clusters* y Grid. OmniRPC utiliza OpenMP para gestionar la ejecución paralela del *thread*, mientras que usa Globus para gestionar la interacción Grid, en vez, de utilizar paso de mensajes para comunicar entre diferentes máquinas.

c) MPJ

Message Passing Java (MPJ) utiliza modelos de paso de mensajes simétricos en vez de utilizar modelos asimétricos como RPC/RMI [16]. MPJ presenta algunas limitaciones. Por ejemplo:

- Implementaciones de MPJ sobre librerías nativas de MPI proporcionan buenos rendimientos pero rompen con los modelos de seguridad de Java.
- Implementaciones de MPJ en Java proporcionan bajos rendimientos.

5) Modelos punto-punto

El cálculo punto-punto permite compartir recursos y servicios por intercambio directo entre sistemas [40]. Tanto los modelos de comunicación punto-punto como las tecnologías Grid, están dirigidas a compartir de forma flexible máquinas y recursos de red.

JXTA es un conjunto de protocolos diseñado para cálculo punto-punto [39]. Estos protocolos permiten realizar comunicación punto-punto sin necesidad de entender o gestionar redes. Estas características hacen que JXTA sea un modelo para implementar servicios Grid punto-punto y aplicaciones [70].

6) Estructuras, componentes de modelos y portales

Junto con las librerías propias del lenguaje de programación se encuentran

entornos que facilitan el desarrollo de aplicaciones distribuidas. Entre ellas se mencionan:

- Cactus: proporciona estructuras modulares para cálculos físicos [14].
- CORBA: es una herramienta estándar la cual permite gestionar operativamente objetos [20] [48].
- Legion: es una herramienta que proporciona objetos con identificadores únicos [56].

7) Modelos de servicio Web

Las tecnologías actuales para Grid se han desarrollado orientadas a proporcionar servicios. OGSA [34] define un conjunto estándar de mecanismos para crear, nombrar y descubrir servicios, además de soportar la integración de los servicios sobre diferentes plataformas.

8) Modelos coordinados

Estos modelos permiten integrar componentes heterogéneas, de manera que, una aplicación pueda ejecutarse en sistemas paralelos y distribuidos [58]. Los componentes de coordinación ofrecen comunicación entre procesos independientes de la arquitectura. Un lenguaje de coordinación ofrece composición de mecanismos e incluye restricciones sobre las formas de paralelismo.

Los lenguajes de composición para Grid [24] [36] [84] proporcionan un modelo para la composición de programas y podrían implementar optimizaciones entre módulos, considerando características de interconexión y máquinas que proveen ejecuciones eficientes sobre Grid.

2.4.3 Grid y modelos de programación paralela

El entorno Grid difiere de los sistemas paralelos y secuenciales en su rendimiento, costo, confiabilidad y características de seguridad. Estas características motivan el desarrollo de nuevas clases de métodos y algoritmos para resolver problemas. Sin

embargo, no existe un consenso claro que determine cual es el modelo de programación adecuado para entornos Grid, aunque si esta claro que se utilizarán diversos modelos.

La tabla 2.2 resume las ventajas y las desventajas que tendrían los modelos de programación para implementar aplicaciones en entornos Grid [28].

Tabla 2.2. Ventajas y desventajas de la aplicabilidad de los modelos de programación en entornos Grid

Modelo	Ejemplos	Ventajas	Desventajas
Paralelismo de datos	HPF, HPC++	Paralelización automática	Restricciones de aplicabilidad
Paso de mensajes	MPICH-G2, PACX-MPI	Alto rendimiento	Bajo nivel
Orientado Objeto	CORBA, DCOM, Java RMI	Soportado para diseñar grandes sistemas	Rendimiento
Llamada a Procedimientos Remotos	DCE, ONC	Simplicidad	Aplicabilidad restringida
Sistemas de colas distribuidos	Condor, LSF, Nimrod	Facilidad de uso	Aplicabilidad restringida
Agentes	Aglets, Telescript	Flexibilidad	Rendimiento, robustez

Como lo muestra la tabla 2.2, un acercamiento a los modelos de programación en Grid consiste en adaptar los modelos de programación que han sido probados en ambientes secuenciales o paralelos. Por ejemplo, un sistema distribuido Grid con MPI, extendería el modelo de paso de mensajes [29] e implicaría la utilización de un sistema de ficheros con acceso remoto utilizando la interfaz estándar de UNIX [86].

La mayoría de estos modelos propuestos, tienen la ventaja de permitir reutilizar código de aplicaciones sin realizar muchas modificaciones para adaptarlas al nuevo entorno. Sin embargo, esto puede generar problemas de rendimiento significativos, debido a que los modelos no se adaptan bien a los entornos distribuidos. Básicamente estos problemas están asociados a la heterogeneidad, las grandes latencias y los constantes cambios de estos entornos.

Por otra parte, otros modelos han sido construidos sobre tecnologías efectivas en la computación distribuida, tales como la llamada a procedimientos remotos (RPC) o técnicas basadas en CORBA. Estas tecnologías tienen ventajas importantes en la

ingeniería de *software*, debido a que sus propiedades de encapsulación facilitan la construcción modular de los programas y la reutilización de componentes existentes. Sin embargo, no se conoce con exactitud si estos modelos pueden aplicarse sobre aplicaciones complejas que involucren el uso de cientos o de miles de procesadores.

Los entornos Grid podrían motivar el origen de nuevos modelos de programación y de servicios. Por ejemplo, sistemas de colas distribuidos, como Condor [57] y Nimrod [2], que soportan métodos que resuelven problemas de estudios paramétricos, donde los problemas son divididos en tareas independientes. Los sistemas de comunicación ordenada de grupo (*group-ordered communication systems*) corresponden a otro modelo importante para entornos Grid dinámicos e impredecibles; éstos proporcionan servicios para gestionar grupos de procesos y para entregar mensajes a los miembros del grupo. Modelos de programación basados en agentes representan otra posibilidad aparentemente buena para entornos Grid; donde, los programas son construidos como entidades independientes que exploran la red buscando datos o realizando otras tareas.

El paralelismo de tareas resulta importante para los Grid computacionales, porque diversas tareas pueden asignarse a diferentes nodos del Grid. Existen muchas variantes de Fortran que soportan paralelismo de tareas, especialmente aquellas diseñadas para operar con paquetes de *threads* y computadores de memoria compartida. El estándar OpenMP provee extensiones del lenguaje que soportan paralelismo de tareas sobre sistemas de memoria compartida.

En ambientes Grid, formados por procesadores heterogéneos, cada programa es visto como una tarea y es asignado a un nodo en el Grid. Las interconexiones entre tareas pueden especificarse utilizando notación convencional o alguna interfaz gráfica como AVS [18], CODE [12], o Khoros [71].

Los ambientes Grid requerirán una revisión de los modelos de programación actuales y nuevos modelos que se adapten a las características específicas de las aplicaciones y de los entornos. De esta manera, se requieren nuevas técnicas para expresar algoritmos complejos, además de herramientas que permitan asociar éstos

algoritmos a la arquitectura. Esto permitirá traducir peticiones de usuario a requisitos de recursos y realizar los cambios en la estructura y el estado del sistema.

La complejidad creciente del uso de aplicaciones y del sistema incrementa la reutilización de códigos. Por lo tanto, el estudio de las técnicas para la construcción de códigos aplicables a entornos Grid resulta muy importante.

En este trabajo, planteamos la posibilidad de realizar simulaciones explícitas de elementos finitos, utilizando el modelo de programación de paso de mensajes. Debido a la cantidad de operaciones numéricas que realizan estas aplicaciones, es factible pensar en ejecutarlas sobre entornos Grid. En este caso, el Grid es visto como un cluster disponible para realizar cálculo paralelo. De esta manera, prescindimos de los costosos supercomputadores para realizar las simulaciones.

Sin embargo, nuestras simulaciones realizan cálculo acoplado, lo que exige realizar una distribución adecuada de los datos, para ocultar las grandes latencias de comunicación entre máquinas. En este trabajo planteamos diferentes estrategias de particionamiento de los datos para ejecutar, con una eficiencia razonable, simulaciones explícitas sobre entornos Grid.

Capítulo 3

Problema y herramientas

Muchos de los problemas estudiados por diversas disciplinas de la ingeniería tales como electromagnetismo, dinámica de fluidos, dinámica estructural y muchas otras, involucran el estudio de magnitudes que evolucionan no solamente en el tiempo sino también en el espacio.

Particularmente, los problemas que abordamos en este trabajo son problemas de contacto, tales como el estampado de chapa y las colisiones de coches. Estos problemas son modelados utilizando ecuaciones diferenciales en derivadas parciales y condiciones fronteras que se imponen a las funciones incógnitas. Para resolver numéricamente estos problemas, es necesario realizar una discretización del problema en el espacio y en el tiempo. La discretización en el espacio es obtenida utilizando el método de elementos finitos, mientras que la discretización en el tiempo (discretización temporal) se determina usando el método de diferencias finitas.

En el método de los elementos finitos (MEF), el dominio espacial del problema es dividido en poliedros llamados elementos finitos. Para cada elemento, la función incógnita es modelada utilizando un polinomio de grado n, donde n suele variar entre 1 y 6. La interrelación y las características de estos elementos definen la malla de elementos.

La discretización espacial de la malla de elementos transforma la ecuación diferencial en un sistema de ecuaciones diferenciales ordinarias (EDO), que contienen derivadas respecto al tiempo. De esta manera, para completar el proceso de discretización, se sustituyen las derivadas temporales por fórmulas que involucran, diferencias de valores de la función incógnita sucesivos en el tiempo. Dependiendo de la fórmula de diferencias finitas utilizada para discretizar en el tiempo, obtendremos un método de solución implícito o explícito. Usualmente, el empleo de fórmulas adelantadas origina métodos explícitos, mientras que las fórmulas regresivas dan lugar a métodos implícitos.

En los métodos implícitos, la discretización del sistema de EDOs genera una sucesión de sistemas de ecuaciones lineales (SEL), uno por cada paso de tiempo. Por lo tanto, para obtener la solución discreta de la ecuación en derivadas parciales (EDP) se necesita resolver un SEL en cada instante de tiempo.

Por otro lado, en los métodos explícitos, la solución de la EDP en el instante de tiempo $t + \Delta t$, evaluada en el punto X, es calculada mediante la aplicación directa de una ecuación en diferencias. Dicha ecuación, define la solución discreta en puntos vecinos a X que han sido calculados previamente.

Hoy en día, todas las simulaciones basadas en resolver EDPs, que usan computadores paralelos con miles de CPUs, emplean como método de solución el método explícito. Como veremos, nuestras aplicaciones utilizan éstos métodos, de manera que, nos interesa realizar una paralelización eficiente de ellos. Para esto, utilizaremos la técnica de descomposición en dominios.

Las secciones siguientes de este capítulo describen cómo realizar todo el proceso

anterior. Comenzaremos por determinar cómo realizar la discretización en el espacio del problema utilizando el método de elementos finitos. Luego, describiremos cómo realizar la discretización en el tiempo del problema utilizando el método de diferencias centrales. Posteriormente, describiremos el método explícito y las aplicaciones que resolveremos, junto con la técnica de descomposición utilizada para realizar la paralelización del método. Por último, comentaremos los detalles del simulador, junto con las configuraciones y los parámetros que definen los entornos Grid sobre los que trabajamos.

3.1. Solución numérica de las ecuaciones diferenciales parciales

Una EDP es una ecuación en la que intervienen las derivadas parciales de una función con al menos dos variables independientes. El orden más alto de las derivadas espaciales indica el *orden* de la ecuación. La solución de la ecuación en derivadas parciales es una función que satisface la ecuación diferencial. Por ejemplo, la expresión matemática de la EDP de segundo orden con variables independientes, $(x_1, x_2, ..., x_n, t) = (x,t) \in \Re^n x \Re$ es una relación entre la función incógnita $u(x,t)$ y sus derivadas parciales, de manera que:

$$F\left(x, t, u, \frac{\partial u}{\partial x_1}, ..., \frac{\partial u}{\partial x_n}, \frac{\partial u}{\partial t}, \frac{\partial^2 u}{\partial x_1 \partial x_1}, \frac{\partial^2 u}{\partial x_1 \partial x_2}, ..., \frac{\partial^2 u}{\partial x_1 \partial x_t}, \frac{\partial^2 u}{\partial t \partial t}\right) = 0$$

donde se asume continuidad de las derivadas mixtas $\frac{\partial^2 u}{\partial x_i \partial x_j}$, $\forall i \neq j$.

De esta forma, para que la función $\phi : \Omega \subset \Re^n x \Re \rightarrow \Re$ sea la solución de la ecuación F en un abierto de $\Re^{n+1}$, denotado por Ω, se debe cumplir:

$$\forall (x,t) \in \Omega : \; F\left(x, t, \phi(x,t), \phi_{x_1}(x,t), ..., \phi_{tt}(x,t)\right) = 0$$

Por otra parte, cada EDP tiene condiciones iniciales asociadas a la variable temporal y, condiciones frontera, relativas a las variables espaciales. La existencia y la unicidad de la solución analítica de la EDP dependen de la definición apropiada de estas condiciones. Sin embargo, a pesar de que muchos de estos problemas carecen

de una solución analítica, es posible determinar la solución discreta utilizando herramientas numéricas, particularmente, utilizando el método de elementos finitos.

La figura 3.1 ilustra el proceso de análisis del método de elementos finitos. Este proceso esta formado por tres etapas, denominadas: pre-procesamiento, simulación y post-procesamiento.

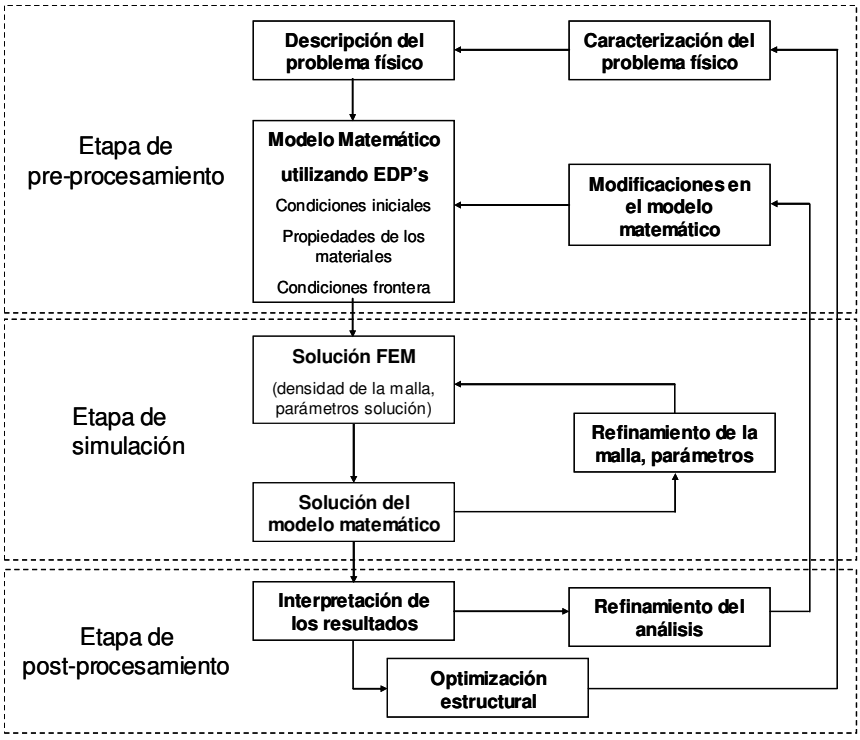


Fig. 3.1 Proceso de análisis del problema utilizando elementos finitos

La etapa de pre-procesamiento define matemáticamente el problema físico, junto con la discretización del dominio. En esta etapa se define: la geometría, las propiedades de los materiales, las condiciones iniciales, las condiciones fronteras, entre otras. Además, se elige la EDP que resulta más apropiada para resolver el problema. Luego, el dominio del problema es particionado en elementos generando un conjunto de ficheros que contienen la malla.

La etapa de simulación determina los valores numéricos de la función incógnita para todos los puntos de la malla. Finalmente, la etapa de post-procesamiento muestra los resultados obtenidos y realiza una estimación del error cometido para determinar, si es necesario, refinar la malla inicial y realizar un nuevo análisis.

3.2. Discretización espacial mediante elementos finitos

El método de los elementos finitos es un procedimiento general de discretización de problemas continuos expresados matemáticamente. Generalmente, este método es utilizado en problemas físicos cuyas fronteras son irregulares o cuyos parámetros presentan discontinuidades [67]. Los problemas con éstas características, resultan complejos de solucionar utilizando los métodos tradicionales de diferencias finitas. Sin embargo, el método de los elementos finitos considera las condiciones fronteras y su interrelación, de manera que facilita el tratamiento de estos problemas. Comúnmente, el MEF es utilizado para obtener la discretización espacial de los problemas de contacto. A continuación, describiremos cómo el MEF realiza la discretización espacial para un problema sencillo.

Inicialmente, el dominio Ω del problema es dividido en ${}^{t}\Omega^{e}$ poliedros. Estos poliedros reciben el nombre de elementos finitos. De esta forma, el dominio original del problema es aproximado por la unión de elementos geométricamente sencillos:

- triángulos y cuadriláteros (en dos dimensiones), o
- tetraedros, paralelepípedos, prismas y pirámides (en tres dimensiones).

La figura 3.2 muestra una aproximación del dominio del problema, utilizando elementos triangulares. En lo sucesivo, nos referiremos a esta aproximación del dominio como malla.

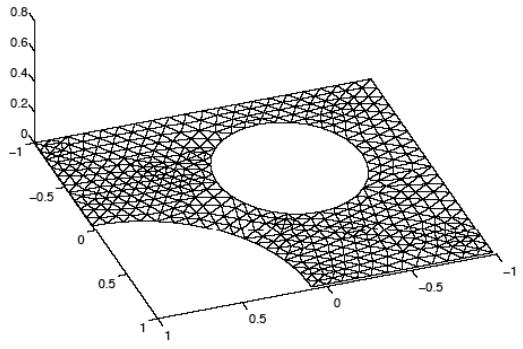


Fig. 3.2 Discretización del dominio del problema utilizando triángulos

Sobre cada elemento finito, la función incógnita de la EDP se aproxima mediante un polinomio de grado n. Este polinomio se construye utilizando la siguiente expresión matemática:

$$\tilde{U}_E(x) = \sum_{i=1}^{m} u_i \phi_i^n(x)$$

donde

$\tilde{U}_E(x)$: es la función incógnita de la EDP en el elemento E,

$\phi_i^n(x)$: es un polinomio interpolador de *Lagrange* de orden n en el elemento E; este polinomio vale 1 en el punto *i-ésimo* y 0 en todos los demás puntos, y

u_i : es el valor de la función en el punto *i-ésimo* del elemento.

La figura 3.3 muestra como la malla de la figura 3.2 es aproximada utilizando funciones lineales continuas a trozos.

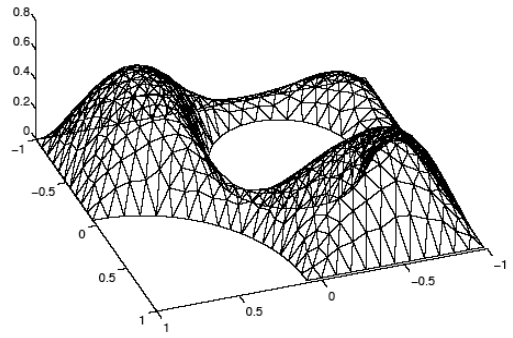


Fig. 3.3 Función lineal aplicada a la malla de la fig. 3.2

Es necesario hacer notar que para definir un polinomio interpolador de *Lagrange* de orden n necesitamos m puntos, donde $m = n + 1$ en una dimensión o $m = \frac{(n+1)(n+2)}{2}$ en dos dimensiones.

Por otra parte, los puntos elegidos para definir las funciones $\phi_i^n(x)$, generalmente, son los vértices y los puntos medios de las caras y los lados de los poliedros. Sin embargo, los puntos elegidos podrían ser cualquiera de los puntos

internos del elemento.

Finalmente, obtenida la solución numérica de la EDP, es necesario determinar si el error numérico de aproximación obtenido, cumple con la tolerancia de aproximación exigida. De esta forma, si se requiere disminuir el error que se comete al resolver la EDP podemos aumentar el grado de los polinomios $\phi_i^n(x)$ (*p-refinamiento*) o bien, disminuir el tamaño de los elementos (*h-refinamiento*).

Normalmente, el procedimiento más usado para disminuir el error de aproximación es el *h-refinamiento*, ya que permite aumentar la precisión de la solución sin tener que cambiar el programa. En este caso, únicamente se necesita cambiar los datos de entrada del programa.

Por otra parte, para discretizar la siguiente EDP:

$$L(U(x)) = f(x)$$

es necesario utilizar la condición de *Galerkin*. De esta forma, cada elemento de la malla esta determinado por la siguiente expresión:

$$\int_E L(U_E(x))\phi_i(x)dx = \int_E f(x)\phi_i(x)dx \qquad \forall i = 1\ldots m$$

Por simplicidad, asumamos que el operador L es lineal, de manera que:

$$\sum_{j=1}^{m} u_i \left(\int_E L(\phi_j(x))\phi_i(x)dx\right) = \int_E f(x)\phi_i(x)dx \quad \forall i = 1\ldots m$$

La expresión matemática anterior, define un sistema de ecuaciones lineales, para cada elemento, donde la matriz de coeficientes, $A \in \Re^{mxm}$ es densa. Estos sistemas, no son independientes, ya que, la solución de la EDP es continua, es decir, el valor de la función aproximada es el mismo en la frontera de elementos. De esta forma, para resolver el problema, es necesario que todas las matrices y los vectores elementales estén ensamblados.

Las integrales sobre los elementos, son calculadas utilizando las fórmulas de integración numérica de *Gauss,* que son exactas para polinomios de orden n con $\frac{(n+1)}{2}$ puntos.

3.3. Discretización en el tiempo

Para resolver numéricamente EDPs con derivadas temporales, es necesario realizar una discretización en el tiempo. La metodología a seguir para realizar esta discretización, es la misma que la especificada en el caso anterior, sólo que ahora los coeficientes u_i son funciones del tiempo. De esta forma, si tenemos la siguiente EDP:

$$\frac{\partial U(t,x)}{\partial t} + L(U(t,x)) = f(t,x)$$

y si asumimos que $\tilde{U}_E(t,x)$ es una aproximación de $U(t,x)$ usando la base ϕ_i:

$$\tilde{U}_E(t,x) = \sum_{i=1}^{m} u_i(t)\phi_i^n(x)$$

obtenemos, utilizando la condición de Galerkin que:

$$\sum_{j=1}^{m} \frac{du_j(t)}{dt} + \sum_{j=1}^{m} u_j(t)\left(\int_E L(\phi_j(x))\phi_i(x)dx\right) = \int_E f(t,x)\phi_i(x)dx \quad \forall i = 1\ldots m$$

Es decir, ahora tenemos un sistema de m ecuaciones diferenciales ordinarias en lugar de un sistema lineal. De nuevo, estos sistemas elementales no son independientes. Para obtener la solución del sistema de ecuaciones diferenciales, se requiere un método de integración numérica. Para realizar esto, es necesario utilizar el método de diferencias finitas. Los métodos de diferencias finitas más utilizados son: el método de diferencias centrales y el método de *Newmark*. A continuación describimos brevemente cada uno de ellos.

3.3.1 Método de diferencias centrales

Considere el siguiente sistema de ecuaciones con un grado de libertad:

$$^t a + w_o^{2\,t} u = {}^t f \tag{3.1}$$

y las siguientes condiciones iniciales,

$$^o u = \overline{u}$$
$$^o v = \overline{v}$$

donde

${}^{t}a$: es el vector de aceleración,

w_o : es el coeficiente de tensión,

${}^{t}f$: es el vector de fuerza,

${}^{t}u$: es el vector incógnita de desplazamientos,

${}^{0}v$: es el vector de velocidad relativa inicial, y

${}^{0}u$: es el vector de desplazamientos inicial.

Para resolver la ecuación 3.1 usando el método de diferencias centrales, dividimos el dominio de tiempo $[0,T]$ en subdominios, incrementando el paso de tiempo. Los subdominios t_{i-1}, t_i, t_{i+1} son denotados respectivamente por ℓ, τ, t, de esta forma, $\Delta\tau = \tau - \ell$ define el paso de tiempo en el subdominio $[t_{i-1}, t_i]$ mientras que $\Delta t = t - \tau$ define el paso de tiempo en el subdomino $[t_i, t_{i+1}]$.

Por otra parte, la aceleración en el instante de tiempo τ es calculada como:

$$ {}^{t}a = \frac{2(v_b - v_a)}{(\Delta\tau + \Delta t)} \tag{3.2}$$

donde ${}^{t}v_a$ y ${}^{t}v_b$ representan los vectores de velocidad en a y b, y se definen, utilizando el método de diferencias centrales como:

$$ {}^{t}v_a = \frac{({}^{\tau}u - {}^{\ell}u)}{\Delta\tau} \tag{3.3}$$

$$ {}^{t}v_b = \frac{({}^{t}u - {}^{\tau}u)}{\Delta t} \tag{3.4}$$

Estas ecuaciones (3.2, 3.3 y 3.4) permiten obtener la siguiente ecuación:

$$ {}^{t}u = {}^{\tau}u + {}^{\tau}v_a \Delta t + \frac{\Delta t(\Delta\tau + \Delta t)^{\tau}a}{2} \tag{3.5}$$

De esta manera, para determinar el desplazamiento en el tiempo $t = \Delta t$, es necesario calcular el desplazamiento en el tiempo τ. Este desplazamiento se define como: ${}^{\Delta t}u = {}^{0}u + \Delta t\,{}^{0}v + \frac{\Delta t^2}{2}{}^{0}a$, con ${}^{0}a = {}^{0}f - w_0^2\,{}^{0}u$ (ecuación 3.1). Este

método de solución obtenido para determinar el desplazamiento en el instante de tiempo t, utilizando diferencias centrales, se denomina método explícito [67].

Por otra parte, para satisfacer la condición de estabilidad [9] [41], el tamaño del paso de tiempo (Δt) no puede exceder un valor crítico Δt_{cr}. Por ejemplo, para sistemas de ecuaciones con un grado de libertad, como el definido en la ecuación 3.1, Δt_{cr} es:

$$\Delta t_{cr} = \frac{2}{w_0} \tag{3.6}$$

Para sistemas de ecuaciones discretizados con varios grados de libertad, el tamaño del paso crítico Δt_{cr} depende del mayor valor del coeficiente. De esta forma, si denotamos como w_m el componente de mayor frecuencia del sistema discretizado, la condición de estabilidad esta determinada por la siguiente expresión:

$$\Delta t \leq \Delta t_{cr} = \frac{2}{w_m} \tag{3.7}$$

3.3.2 Método de *Newmark*

Consideremos ahora la solución de la ecuación 3.1 utilizando el método de *Newmark* [9] [41].

En el método de *Newmark*, la aceleración, la velocidad, y el desplazamiento en el tiempo t están relacionados por las siguientes ecuaciones:

$$^{t}u = {}^{\tau}u + \Delta t\,{}^{\tau}v + \frac{1}{2}\Delta t^{2}\left[(1-2\beta)^{\tau}a + 2\beta\,{}^{t}a\right] \tag{3.8}$$

$$^{t}v = {}^{\tau}v + \Delta t\left[(1-\gamma)^{\tau}a + \gamma\,{}^{t}a\right] \tag{3.9}$$

donde β y γ son parámetros determinados para cada aplicación. De esta forma, si $\beta = 0$ y $\gamma = \frac{1}{2}$, el método de *Newmark* es equivalente al método de diferencias

centrales. Sustituyendo la ecuación 3.8 en la ecuación 3.1, obtenemos el vector de aceleración ${}^{t}a$ en el tiempo t, y de esta manera, es posible determinar los vectores ${}^{t}u$ y ${}^{t}v$. Si $\beta \neq 0$, el sistema de ecuaciones en el tiempo t define un método de solución implícito.

Por otra parte, las condiciones de estabilidad del método de *Newmark* son:

$$\Delta t < \infty \text{ (Estable incondicionalmente) si } 2\beta \geq \gamma \geq 0.5 \tag{3.10}$$

$$\Delta t \leq \frac{T_{\min}}{\left[2\pi\left(\frac{\gamma}{2}-\beta\right)^{\frac{1}{2}}\right]} \qquad \text{si } \beta < \frac{\gamma}{2} \text{ y } \gamma \geq 0.5 \tag{3.11}$$

donde $T_{\min}$ denota el período de tiempo más pequeño del sistema discretizado. Detalles de la estabilidad numérica del método de *Newmark* pueden consultarse en [9] y [41].

3.3.3 Condición de *Courant*: tamaño del paso y estabilidad

En la sección 3.3.1, comentamos que el método de diferencias centrales es un método condicionalmente estable. Por esta razón, los métodos explícitos tiene el inconveniente de que su estabilidad es condicional, exigiendo que el paso de tiempo Δt_i no exceda el valor del paso crítico Δt_{cr}, determinado por la condición de *Courant*. De esta forma, Δt_{cr} representa la longitud máxima del paso que garantiza la convergencia del método a la solución de la EDP y puede estimarse considerando el nivel del elemento (máximo autovalor). El máximo autovalor del sistema de elementos finitos, denotado por $\lambda_{\max}$, se aproxima por [67]:

$$\lambda_{\max} = \underset{n=1}{\overset{e_N}{Max}}\left(\lambda^n_{\max}\right) \tag{3.12}$$

donde $\lambda^n_{\max}$ representa el máximo autovalor asociado al elemento n. Entonces, el paso crítico está determinado por la siguiente expresión:

$$\Delta t_{cr} = \frac{2}{\sqrt{\lambda_{\max}}} \tag{3.13}$$

Sin embargo, es importante considerar que la presencia de fuerzas de contacto modifica el cálculo de los autovalores. Para problemas de contactos, resulta conveniente utilizar el teorema de los círculos de *Gerschgorin* para estimar $\lambda^n_{\max}$ [67]. Para el resto de los casos y dependiendo de la malla que discretiza el dominio de solución de la EDP, es posible obtener otras aproximaciones para calcular $\lambda^n_{\max}$ [14]. De esta forma para cada grado de libertad en U , tenemos:

$$M_{II}{}^{t}a + {}^{\tau}\lambda_I u = {}^{\tau}R_I \tag{3.14}$$

de manera que, Δt_{cr} tiene la siguiente expresión:

$$\Delta t^I_{cr} = 2\sqrt{\frac{M_{II}}{{}^{\tau}\lambda_I}} \leq 2\sqrt{\frac{M_{II}}{\left|{}^{\tau}k_{II}\right| + \sum_{J=1, J\neq I}^{N_d}\left|{}^{\tau}k_{IJ}\right|}} \tag{3.15}$$

donde k_{II}, k_{IJ} denotan, respectivamente, el elemento (I,I) e (I,J) de la matriz ${}^{\tau}K$. De esta manera, la longitud del paso Δt_{cr} se define como [14]:

$$\Delta t_{cr} = \frac{2}{(1-2\theta)\lambda_{\max}} \quad 0 \leq \theta \leq \frac{1}{2} \tag{3.16}$$

La ecuación 3.16 es conocida como condición de *Courant*. Sin embargo, esta estimación de Δt_{cr} es computacionalmente ineficiente, por lo que, se determina una nueva expresión de Δt_{cr} a partir, de la combinación de las ecuaciones 3.13 y 3.15 [56]. De esta forma, la expresión para Δt_{cr} queda determinada por la siguiente expresión:

$$\Delta t_{cr} = Min\left(\Delta t^e_{cr}, \Delta t^c_{cr}\right) \tag{3.17}$$

donde $\Delta t^e_{cr} = \dfrac{2}{\sqrt{\underset{n=1}{\overset{{}^{e}N}{Max}}\left(\lambda^n_{\max}\right)}}$ y $\Delta t^c_{cr} = \underset{I=1}{\overset{N_c}{Min}}\left(\Delta t^I_{cr}\right)$ (3.18)

con N_c el total de grados de libertad. Para más detalles sobre las expresiones

matemáticas que definen el tamaño del paso puede consultarse [1].

Los métodos de cálculo explícitos resultan en la práctica sencillos y robustos, ya que la condición de *Courant* se puede establecer de forma automática con un esquema de paso variable. Comúnmente, en aplicaciones de elementos finitos, se utilizan métodos de uno o dos pasos. Particularmente, si $\theta = 0$ en la ecuación 3.16, el método es de un paso.

A continuación, detallaremos los diferentes esquemas de simulación utilizados al resolver EDP's usando los métodos explícitos.

3.3.4 Métodos explícitos

Como se especificó anteriormente, los métodos explícitos son métodos que constituyen la forma más directa de avanzar en el tiempo. Se basan en establecer las ecuaciones dinámicas en el instante de tiempo t_{n+1} para determinar los valores de las variables u_{n+1} [25].

Los métodos explícitos pueden utilizar dos tipos de esquemas de simulación:

1. Esquema sin ensamblado.

   Este tipo de esquema utiliza un algoritmo de cálculo, elemento por elemento, de manera que no precisa ensamblar un sistema global de ecuaciones. Esto evita el almacenamiento de matrices con coeficientes globales.

   Sin embargo, el número de operaciones que realiza este esquema es mayor que el requerido por el esquema ensamblado, ya que, en este caso, cada coeficiente de la matriz global se descompone en tantos coeficientes como elementos vecinos tiene un nodo. Para problemas en tres dimensiones esto puede implicar un incremento de hasta treinta veces más el número de operaciones.

   Para este tipo de esquemas, el algoritmo de simulación es el siguiente:

Para todos los pasos de tiempo

Para todos los elementos en la malla

Extraer del vector global x al vector elemental $x^{(e)}$

$y^{(e)} = A^{(e)} x^{(e)}$

Expandir del vector elemental $y^{(e)}$ al vector global y

Fin para

Calcular residual

Actualizar condiciones de contorno

Calcular próximo paso de tiempo Δt

Fin para

2. Esquema con ensamblado.

A diferencia del caso anterior, este tipo de esquema requiere ensamblar un sistema global de ecuaciones en cada iteración del método.

El algoritmo de simulación para este esquema es el siguiente:

Para todos los pasos de tiempo

Para todos los elementos en la malla

Ensamblar $A^{(e)}$: $A = A + A^{(e)}$

Fin para

$y = Ax$

Calcular residual

Actualizar condiciones de contorno

Calcular próximo paso de tiempo Δt

Fin para

Independientemente del esquema de simulación utilizado, en cada iteración del método se efectúa una operación producto matriz por vector. Esta operación, en comparación con el resto de las operaciones, representa entre el 80% y el 99% del tiempo de ejecución de las simulaciones. De esta forma, paralelizar eficientemente esta operación es un tema importante a la hora de ejecutar las simulaciones explícitas. Por esta razón, y como veremos mas adelante, se utilizan

técnicas de descomposición en dominios, donde la matriz y el vector se dividen convenientemente para realizar, de forma eficiente, la operación producto matriz por vector.

3.4. Aplicaciones

Con el objetivo de identificar el comportamiento estructural de la colisión entre vehículos, las deformaciones de materiales, y el estampado de piezas, se han desarrollado herramientas de análisis computacional; específicamente, programas de análisis por elementos finitos que resuelven problemas no lineales. La no linealidad de estos fenómenos se debe a: la plasticidad de los materiales, los grandes desplazamientos y los efectos derivados del contacto entre chapas. Debido a la no linealidad que presentan estos problemas, el proceso de solución numérica requiere la utilización de sofisticados algoritmos numéricos y de recursos computacionales de gran potencia. La cantidad de cálculo asociada a estas simulaciones es enorme, por ejemplo: las simulaciones de colisión de coches requieren de 300.000 a 15.000.000 elementos.

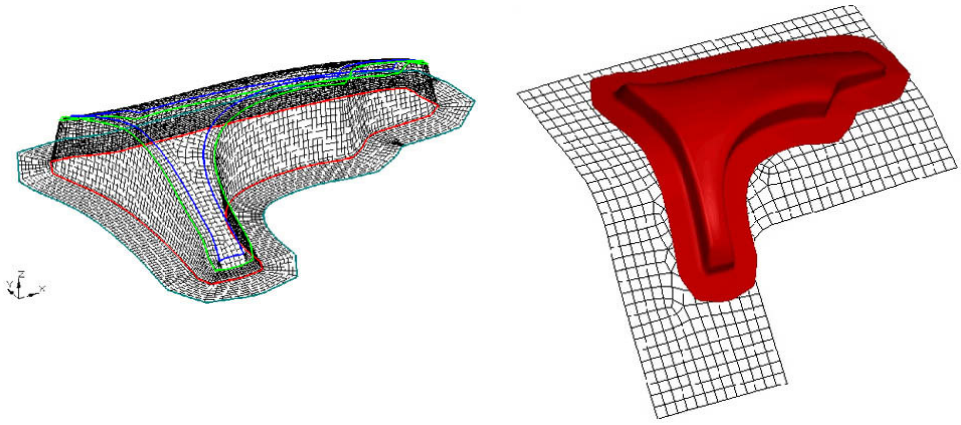
Todo esto, conlleva a que muchos fabricantes acepten la simulación como una parte importante del proceso de diseño y construcción de prototipos. De esta forma, muchas compañías automotoras utilizan la simulación numérica para realizar el proceso de estampado de chapa, o bien, para determinar las deformaciones en la estructura de un coche producidas por un impacto.

Por otra parte, en este trabajo se consideran aplicaciones distribuidas que efectúan operaciones producto matriz-vector. Estas aplicaciones se derivan de la discretización de EDP's para mallas estructuradas y no estructuradas, utilizando el método de elementos finitos y métodos explícitos para obtener la solución numérica del problema.

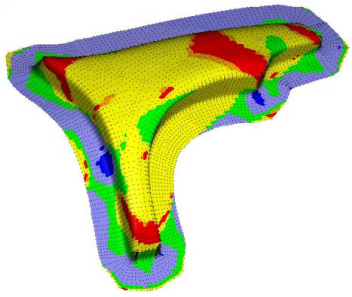
A continuación realizamos una descripción de las aplicaciones industriales simuladas en este trabajo.

3.4.1 Estampado de chapa

El estampado de chapa es un proceso complejo, muy utilizado en la industria automovilística, que requiere, cada vez más, mejorar sus tiempos de producción. De esta manera, es necesario aumentar el número de piezas fabricadas conservando la calidad del producto. La figura 3.4 muestra un ejemplo del proceso de estampación para fabricar una pieza de la defensa del coche.



(a) (b)



(c)

Fig. 3.4 Simulación del estampado de una pieza del coche: (a) Malla discretizada (b) Definición, ajuste y moldeado de la pieza (c) Estampado de la pieza

La forma de abordar esta problemática plantea mejorar el diseño de la pieza a fabricar, el desarrollo del sistema de producción (prensas, troqueles) y la optimización del propio proceso de estampación.

Durante el proceso de diseño, la predicción del rendimiento que genera un troquel para el estampado de chapa reduce el número de ensayos experimentales optimizando el proceso de estampado. Además, a medida que el mercado presiona para realizar una entrega de troqueles más rápida, reduciendo costes de producción, los grandes fabricantes de coches delegan la responsabilidad del diseño a sus proveedores.

Los fabricantes de troqueles son pequeñas empresas sometidas a fuertes presiones que deben asumir entre sus propios costes de producción, el coste de utilizar la simulación. Típicamente, los troqueles son diseñados por personal altamente cualificado que sabe, el tipo de repujo que se necesita para producir una determinada pieza. De esta manera, basado en un diseño inicial, el ajuste fino de la chapa, se efectúa utilizando el troquel y fresando manualmente el troquel hasta que la lámina coincida con la solicitada por el contratista. Sin embargo, en casos complejos, es muy difícil producir, por intuición, un buen diseño del troquel. Por esta razón, se utiliza la simulación numérica como una herramienta que proporciona la información cuantitativa para disminuir las modificaciones sobre la estructura durante el proceso de fabricación [64].

Aproximadamente, una simulación ideal tardaría alrededor de 12 horas, por lo que sería factible realizar esta simulación durante la noche, cuando los recursos están disponibles. Esto permitiría a los fabricantes de troqueles minimizar sus costes de producción.

Para simulaciones más complejas, el tiempo de simulación puede aumentar. Sin embargo, en estos casos, es posible utilizar la simulación paralela. De esta forma, en vez de realizar un largo proceso iterativo, los fabricantes pueden minimizar el tiempo de simulación, por lo que pueden invertir más tiempo en analizar el modelo, antes de hacer los primeros prototipos del troquel. Esto reduce tanto el

número de iteraciones como el tiempo total de diseño y fabricación del troquel.

Para realizar las simulaciones paralelas, es necesario que la empresa cuente, o bien, con el acceso a una máquina paralela, o bien con una red de computadores accesibles. En estos casos, la posibilidad más viable es la segunda, ya que actualmente resulta factible que la pequeña empresa pueda contar con varios ordenadores conectados a través de una red Ethernet. Esto permite romper las barreras económicas de los pequeños fabricantes de troqueles, ofreciéndoles una solución rápida y de bajo coste a sus problemas de simulación.

3.4.2 Colisiones de coches

Las compañías automotoras necesitan realizar cientos de pruebas y rigurosas evaluaciones sobre estructuras sometidas a impactos. Estas pruebas consumen tiempo, son complejas y muy costosas. De esta forma, hasta que el primer prototipo no se construye, las simulaciones, de las colisiones de un coche, son la única herramienta disponible para analizar el comportamiento de la estructura del coche. Además, las colisiones simuladas minimizan la participación de vehículos reales. Esto permite ahorrar mucho dinero.

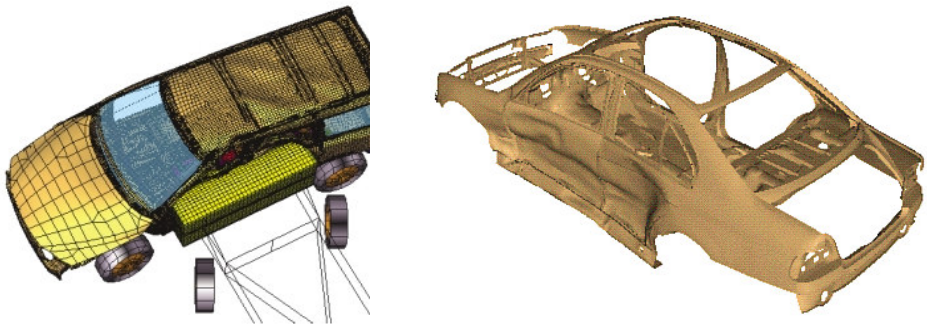
Estos estudios se realizan debido a que los accidentes ocasionales no se pueden evitar. Por lo tanto, resulta deseable que la estructura del coche se comporte de manera tal, que el daño a las personas y la pérdida económica se reduzcan al mínimo. El término "*crashworthiness*" se utiliza para denotar la capacidad que tiene una estructura de aguantar una colisión sin sufrir mucha distorsión. La mayoría de los análisis estructurales del "*crashworthiness*" están asociados a la industria automotora.

En sus inicios, y debido a que no existían otras opciones, los análisis estructurales del "*crashworthiness*" utilizaban acercamientos experimentales y/o analíticos. Posteriormente, con el desarrollo del método de elementos finitos y los avances tecnológicos de la computadora es posible hacer análisis numérico de "*crashworthiness*". Actualmente, se pueden realizar estudios conceptuales y

paramétricos, sin necesidad de tener que fabricar la estructura.

Para realizar estos estudios, es necesario plantear las expresiones matemáticas que definen el problema. Estas expresiones contienen relaciones que definen las características y las propiedades de los materiales de la estructura del coche.

Los modelos que permiten representar el problema de colisiones entre coches son muy diversos y complejos, por ejemplo, en [49] y [50] se desarrollan modelos que simulan el efecto que sufre la estructura del coche al chocar frontalmente con un objeto (figura 3.5). Adicionalmente, en [13] y [21] se deducen modelos que simulan la colisión entre dos coches utilizando el método de elementos finitos (figura 3.6). En ambos casos, estos modelos representan fenómenos de impacto sobre estructuras dinámicas.



(a) (b)

Fig. 3.5 Estudio del impacto lateral de un coche contra un bloque: (a) Malla discretizada (b) Deformación de la estructura del coche

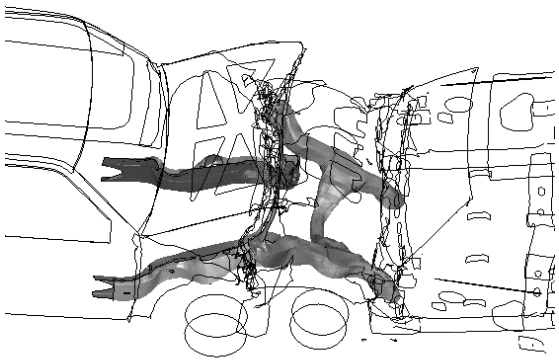


Fig. 3.6 Colisión entre dos coches

El comportamiento dinámico de los materiales sólidos se rige por una serie de ecuaciones diferenciales básicas que expresan el balance de diferentes magnitudes. El balance de la cantidad de movimiento está determinado por la ecuación de *Cauchy*. Esta ecuación relaciona entre sí los siguientes parámetros: la fuerza, la densidad, la velocidad y la tensión. El balance del momento cinético obliga a que el tensor sea simétrico. El balance de la energía, utiliza el primer principio termodinámico, que relaciona la energía interna (por unidad de masa), el tensor velocidad de deformación, la densidad de fuentes de calor y el flujo calorífico. Estas relaciones son válidas para cualquier material, sin embargo, no son suficientes para definir completamente el comportamiento del material. Por esta razón, es necesario añadir relaciones particulares dependiendo del tipo y del régimen al que es sometido el material. Estas relaciones son ecuaciones mecánicas, que permiten expresar la tensión en función de la deformación (Ley de *Hooke*) y obtener ecuaciones térmicas que describen el flujo de calor [59]. En impactos a velocidades medias, se producen deformaciones plásticas del material, así como rotaciones y desplazamientos del mismo. Estos fenómenos hacen que el modelo sea no lineal [10] [11] [24] [62]. De esta forma, y debido a la complejidad de los modelos, no se pueden obtener soluciones analíticas del problema. Por esta razón, utilizamos métodos numéricos basados en la discretización del espacio, tales como el método de los elementos finitos [11]. El problema básico consisten en obtener la solución en el instante t_n, a partir de los valores obtenidos en instantes anteriores [18] [40].

Para realizar el análisis estructural de las colisiones de coches y el proceso de estampación de chapa, se utiliza la ecuación de desplazamientos [59]. Esta ecuación determina la solución numérica tanto de problemas lineales como no lineales [8] [68] [69]. La discretización de la ecuación de desplazamiento, usando el método de elementos finitos, tiene la siguiente expresión matemática [67]:

$$[M]\{\ddot{u}\}+[K]\{u\}=\{F^a\} \tag{3.19}$$

donde

$[M]$: es la matriz de masa,

$[K]$: es la matriz de tensión,

$\{\ddot{u}\}$: es el vector de aceleración,

$\{u\}$: es el vector de desplazamiento, y

$\{F^a\}$: es el vector de fuerzas.

Para obtener la solución numérica de esta ecuación, se utiliza el método de diferencias centrales e integración numérica en el tiempo, dando lugar a la siguiente ecuación:

$$\{u_{n+1}\} = [A]\{u_n\} \tag{3.20}$$

Esta ecuación define el método explícito que determina la solución numérica de nuestras aplicaciones. Detalle sobre las características de la matriz $[A]$ y el vector $\{u_n\}$ pueden consultarse en [1] y [67].

3.4.3 Aplicación científica: Movimientos sísmicos

Las aplicaciones anteriores, son aplicaciones industriales en donde la simulación numérica determina el efecto que producen las fuerzas de contacto sobre sus estructuras. A continuación, comentamos otro tipo de aplicación de carácter científico, en donde las simulaciones explícitas también son necesarias. Nos referimos a las simulaciones sísmicas, que estudian el efecto de la propagación de ondas sísmicas sobre la superficie terrestre. Analizar la propagación de ondas sísmicas sobre la tierra, es una de las pocas vías que existen para estudiar los efectos que producen los terremotos en la superficie terrestre.

Igual que en los casos anteriores, para realizar las simulaciones de esta aplicación científica, es necesario aproximar el fenómeno físico a un modelo matemático utilizando EDPs. Particularmente, en este caso se utiliza la ecuación de onda. La solución numérica del problema se determina usando métodos explícitos

de integración en tiempo.

Hasta ahora, estas simulaciones sólo se podían realizar haciendo uso exclusivo de los supercomputadores, lo que dificultaba poder realizar muchas simulaciones. Por ejemplo, en [55] se realizaron simulaciones sísmicas que utilizaron 1944 procesadores del supercomputador *Earth Simulator*, lo que representaba el 38% de ocupación de la máquina (243 nodos de 640). Recientemente, esta misma aplicación ha utilizado 4200 procesadores del supercomputador *MareNostrum*.

En este trabajo proponemos diferentes estrategias de balanceo de carga para resolver problemas de EDP utilizando métodos de solución explícitos sobre entornos Grid. Esto permitirá que muchas de las simulaciones puedan ejecutarse sobre entornos distribuidos sin necesidad de utilizar costosos supercomputadores.

Por otra parte, las simulaciones sísmicas utilizan métodos de elementos finitos de alto orden. Estos métodos son apropiados para encontrar la solución de problemas hiperbólicos lineales, tales como, la propagación de la onda. En estos casos, el campo de onda es interpolado por polinomios de *Lagrange* de alto orden sobre puntos de integración de *Gauss-Lobatto-Legendre* [52] [53] [54]. En estas simulaciones se usan elementos finitos cúbicos formados por 125 puntos.

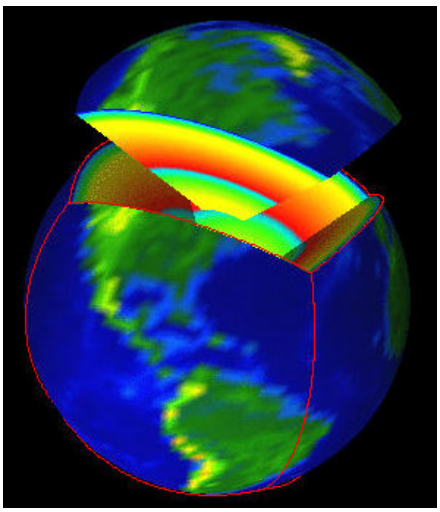


Fig. 3.7 Representación de la superficie terrestre

La figura 3.7 muestra una visión global de la malla de elementos obtenida al aproximar el dominio del problema. De esta manera, la superficie de la tierra esta representada por una esfera cuyas caras son hexaedros. Este tipo de representación se denomina *esfera cúbica*.

De esta manera, cada cara de la *esfera cúbica* es subdividida en *patches*. La figura 3.8 señala cada lado de la esfera utilizando una línea de color rosado, y muestra como cada lado es dividido en 18x18 *patches*, ilustrados en la gráfica por cuadrados de menor tamaño y con diferentes colores. En total, se obtienen 1944 *patches*, cada uno de los cuales es asignado a un procesador. La figura 3.9 muestra la ampliación del *patch* formado por 48x48 elementos. De esta manera, el número total de elementos en la malla es 82.000 millones (5.200 millones de puntos).

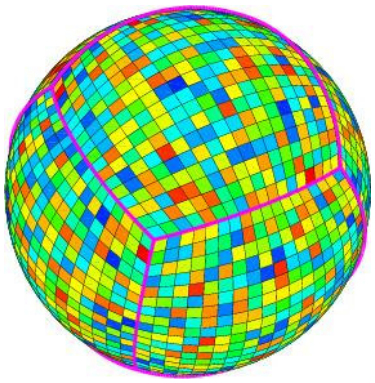


Fig. 3.8 Subdivisión de la *esfera cúbica* en 1944 *patches*

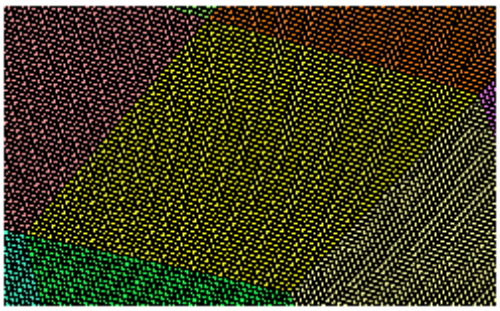


Fig. 3.9 Elementos de un *patch* (48x48)

Hasta ahora todos estos problemas han sido ejecutados en entornos homogéneos utilizando máquinas paralelas o bien, redes de computadores. Sin embargo, debido a las necesidades actuales de las compañías, la complejidad de los modelos y el tamaño de los problemas, la distribución de los datos requiere entornos distribuidos de gran potencia. Por esta razón, los entornos Grid resultan apropiados para realizar tales simulaciones.

Por otra parte, muchas de las empresas cuentan con equipos de computación, ubicados en las diferentes sucursales, de los que sólo se utiliza entre el 3% y el 5% de su capacidad. Para aumentar éste porcentaje de utilización, podríamos ejecutar simulaciones durante la noche o en horas en las que no se utilizan tales de equipos. De esta forma, la empresa podría contar con un grupo de ordenadores distribuidos, que funcionen como una gran máquina paralela. Esto permitiría aprovechar los equipos y no tener que invertir dinero en la compra de costosas máquinas paralelas. De esta manera, si consideramos:

- la complejidad de los diseños,
- la demanda de tecnología para resolver los problemas, y
- el aprovechamiento inapropiado de los recursos

entonces, una ejecución eficiente de las simulaciones podría realizarse utilizando máquinas distribuidas geográficamente.

3.5. Paralelización de los métodos explícitos

3.5.1 Descomposición en dominios

La descomposición en dominios es una técnica muy utilizada en la resolución de EDPs. Esta técnica permite paralelizar eficientemente la operación producto matriz por vector que se realiza en cada iteración del método explícito para nuestras aplicaciones. En muchas otras ocasiones esta técnica puede llegar a reducir la complejidad numérica del SEL cuando se utilizan métodos de solución implícitos [48].

La técnica de descomposición en dominios divide la malla de elementos en subgrafos, llamados dominios, utilizando algoritmos de particionamiento de grafos [12] [16] [19] [44] [46] [60] [61]. De esta forma, cada dominio de datos esta asociado a un proceso paralelo y, la comunicación entre los dominios se efectúa realizando el intercambio de los datos entre los procesadores de dominios vecinos.

Por otra parte, el problema de generar una descomposición en dominios óptima, es en general, un problema *NP-completo* [22]. El calificativo óptimo, hace referencia al hecho de que la partición debe garantizar que la carga computacional por dominio está perfectamente balanceada. De ésta manera, un particionador óptimo de los datos que utilice la técnica de descomposición en dominios debe minimizar:

- el costo de las comunicaciones locales distribuyendo los elementos del dominio entre los procesadores,
- el costo de las comunicaciones remotas entre dominios vecinos,
- el tamaño de las fronteras y,
- la cantidad de mensajes enviados entre los dominios vecinos.

En este sentido, se han realizado muchas investigaciones en el área de descomposición en dominios para paralelizar el análisis de elementos finitos. Los principales métodos heurísticos implementados pueden consultarse en [7] [12] [20] [47] [51] [58] [60] [63]. Estos métodos balancean la carga computacional, de manera que, la cantidad de comunicaciones entre los procesadores de diferentes dominios sea mínima.

La figura 3.10 ilustra un ejemplo de una malla de elementos finitos en dos dimensiones con 36 grados de libertad y una partición en 4 dominios. Estos dominios son determinados utilizando algoritmos de particionamiento de grafos. Las líneas discontinuas de la figura 3.10 definen la cantidad de elementos de cada dominio de datos.

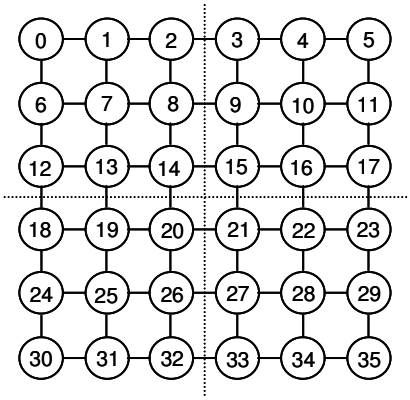


Fig. 3.10 Grafo asociado a la malla de elementos finitos

De esta forma, obtenida la partición de datos se realiza la numeración de los nodos en cada dominio. Inicialmente, se numeran todos los nodos que no son frontera. Luego, se numeran los nodos frontera. La figura 3.11 muestra la numeración de los nodos e identifica los nodos frontera utilizando círculos, rellenos de color gris.

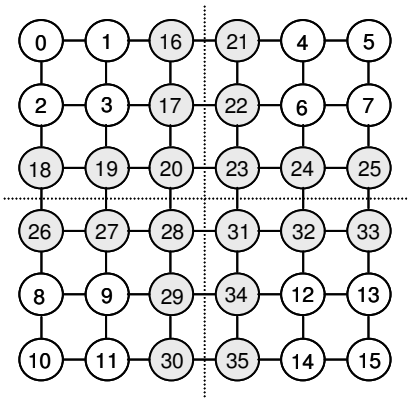


Fig. 3.11 Numeración de los nodos de la malla

Esta nueva numeración de los nodos, origina una reordenación de los elementos en la malla, generando una matriz en bloques cuya estructura es similar a la mostrada en la figura 3.12.

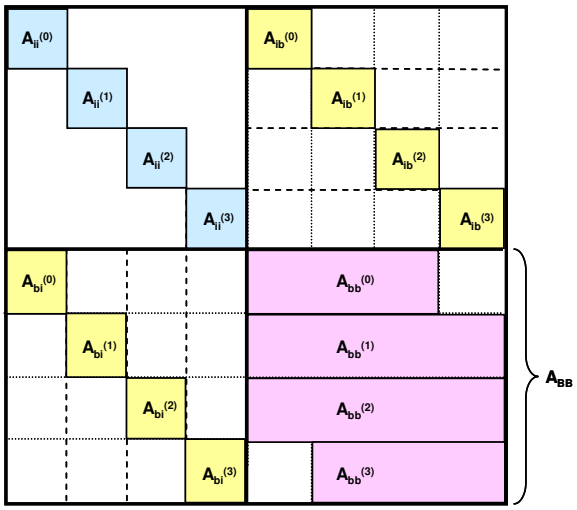


Fig. 3.12 Estructura en bloques de la matriz

Esta estructura en bloques de la matriz, hace que la operación producto matriz por vector ($y = Ax$), que se realiza en cada iteración del método explícito pueda paralelizarse, ya que presenta la siguiente estructura:

$$\begin{pmatrix} y_i^{(1)} \\ y_i^{(2)} \\ y_i^{(3)} \\ y_i^{(4)} \\ y_B \end{pmatrix} = \begin{pmatrix} A_{ii}^{(1)} & & & & A_{ib}^{(1)} \\ & A_{ii}^{(2)} & & & A_{ib}^{(2)} \\ & & A_{ii}^{(3)} & & A_{ib}^{(3)} \\ & & & A_{ii}^{(4)} & A_{ib}^{(4)} \\ A_{bi}^{(1)} & A_{bi}^{(2)} & A_{bi}^{(3)} & A_{bi}^{(4)} & A_{BB} \end{pmatrix} \begin{pmatrix} x_i^{(1)} \\ x_i^{(2)} \\ x_i^{(3)} \\ x_i^{(4)} \\ x_B \end{pmatrix}$$

Por último, cada partición de datos esta asociada a un dominio de descomposición (figura 3.13).

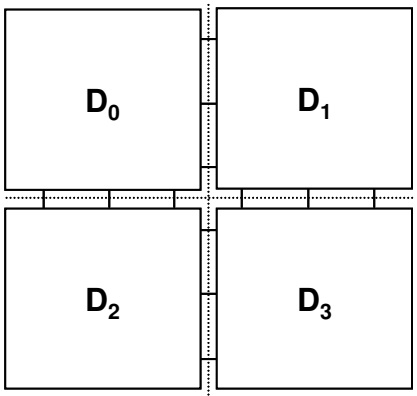


Fig. 3.13 Dominios de datos

3.5.2 METIS

Para realizar la descomposición en dominios existen múltiples herramientas software, entre las que se encuentran: Chaco [27] [33], JOSTLE [32], librerías de particionamiento de grafos [36] [65], PART [38] y METIS [42] [57]. De todas estas herramientas, la que más destaca por sus tiempos de ejecución es METIS [39]. Además, METIS permite obtener particiones de datos no balanceadas, lo que resulta de interés para nosotros, ya que dependiendo de la velocidad relativa del procesador, necesitaremos obtener particiones de datos heterogéneas.

Para determinar las velocidades relativas de los procesadores, ejecutamos cualquier SPEC (*benchmarks*), obteniendo la velocidad pico del procesador. De esta forma, al procesador con velocidad relativa más alta, le haremos corresponder la velocidad 1. El resto de las velocidades relativas se obtienen a partir de la relación de proporcionalidad de sus velocidades con respecto a la velocidad más alta.

Por otra parte, METIS es una herramienta software que permite el particionamiento de grafos y el particionamiento de mallas de elementos finitos utilizando algoritmos multinivel [26] [44] [45] [46].

Los algoritmos de particionamiento de grafo tradicionales calculan una partición del grafo operando directamente sobre este. Esto hace que los algoritmos resulten lentos y generen particiones de datos que no son óptimas. Sin embargo, los algoritmos de particionamiento multinivel, reducen, en cada nivel el tamaño del grafo y realizan particiones sobre grafos más pequeños. Estos subgrafos son utilizados, posteriormente, para construir una partición del grafo original. En la fase de reducción del grafo, METIS emplea algoritmos que generan particiones de alta calidad. Finalizada esta etapa, se realiza el refinamiento del subgrafo hasta alcanzar el nivel inicial. Durante el refinamiento, METIS trabaja definiendo la porción del grafo y delimitando la frontera del subgrafo [45]. De esta forma, la principal ventaja de METIS es la rapidez en la obtención de las particiones y la calidad de las

mismas.

Sin embargo, a pesar del éxito que han tenido los algoritmos de particionamiento multinivel, existen aplicaciones científicas que no pueden usar directamente el algoritmo de particionamiento de METIS. Ejemplo de estas aplicaciones, son aquellas donde se producen grandes deformaciones de la malla, o donde existe contacto entre las mismas. En estos casos, es necesario añadir a METIS una capa de software que tome en cuenta la dirección espacial de las deformaciones, es decir, que considere también la geometría del problema [43].

3.6. Dimemas

Dimemas es la herramienta que utilizaremos para simular el comportamiento de nuestras aplicaciones en entornos Grid. Dimemas es una herramienta desarrollada por el CEPBA[1] que se ha extendido para simular entornos Grid [2] [5] [17].

Para trabajar con Dimemas es necesario definir dos ficheros de entrada. El primer fichero, contiene una traza de la aplicación, mientras que el segundo define la configuración del entorno Grid.

Para obtener el fichero de traza, se ejecuta la aplicación paralela utilizando alguna versión instrumentada de MPI. Esta ejecución puede realizarse en cualquier máquina, por lo que Dimemas no requiere máquinas sofisticadas para generar este fichero. Por otra parte, el fichero de configuración contiene detalles sobre las características de la arquitectura, tales como: el número de nodos, la latencia y ancho de banda entre los nodos.

Dimemas genera un fichero de salida que contiene los tiempos de ejecución de la aplicación simulada para los valores de los parámetros especificados en el fichero de configuración. Además, es posible obtener una representación gráfica de la ejecución paralela de la aplicación [17].

La figura 3.14 muestra la secuencia de pasos que son necesarios para obtener

[1] European Center for Parallelism of Barcelona, www.cepba.upc.edu

el fichero de salida. Este fichero contiene los tiempos de ejecución de la aplicación en el entorno Grid definido.

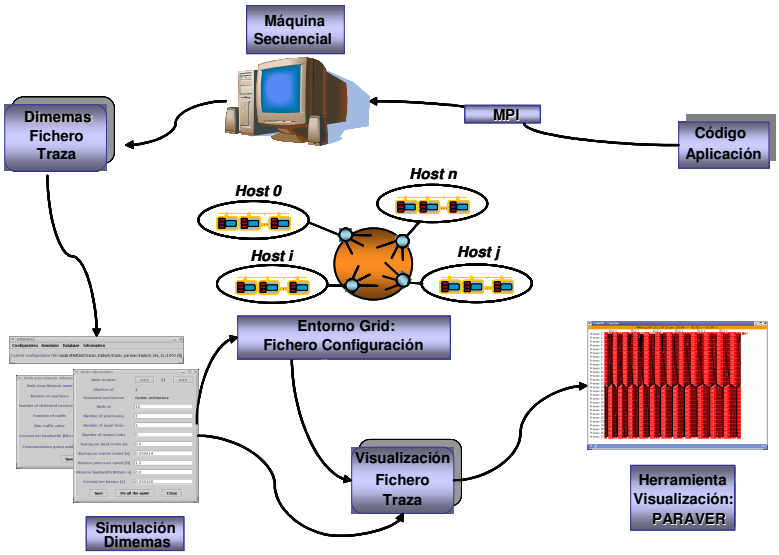


Fig. 3.14 DImemas: Generación del fichero de salida

3.6.1 Modelo de comunicación de Dimemas

El simulador Dimemas implementa un modelo sencillo de comunicación punto-punto [2] [5]. Los mensajes de datos son enviados a través de conexiones dedicadas dependiendo del tamaño del mensaje y del tipo de operaciones [23].

Para conexiones dedicadas entre dos máquinas, se necesita definir los valores de la latencia y del ancho de banda. El retardo del mensaje entre dos máquinas se evalúa utilizando fórmulas determinadas especificadas en [4].

Básicamente, el modelo de comunicación de Dimemas descompone el tiempo de comunicación en cinco componentes. Estas componentes son:

1. El tiempo de latencia

   Es el primer paso requerido para iniciar la comunicación. Este tiempo representa el retardo que sufre el mensaje cuando es enviado a otro procesador, por lo que es un tiempo fijo y particular para cada máquina.

2. El tiempo de contención de los recursos

   Este tiempo depende de la disponibilidad de los enlaces y de los buses en la máquina. Para simular este tiempo, Dimemas bloquea los enlaces y los buses utilizados en la comunicación al transferir el mensaje. Finalizada la comunicación, Dimemas libera los recursos bloqueados [3]. Después de este período, el proceso que envía puede continuar realizando otros cálculos.

3. El tiempo de transferencia

   Este tiempo depende del tamaño del mensaje y del ancho de banda de la conexión, de manera que esta definido por la siguiente relación:

$$T_{Enviar} = \frac{Tamaño\ mensaje}{Ancho\ de\ banda}$$

4. El tiempo de contención de la WAN

   Este tiempo depende del tráfico en la red. Para conexiones dedicadas, el tiempo de contención, para cada máquina en la red externa, está compuesto por: la latencia inicial y el ancho de banda. Ambos valores son modificados por una función que estima el tráfico en la red para cada instante de tiempo. De esta manera, la función de tráfico, denotada por $f(tráfico)$, descompone el tráfico en: tráfico externo ($Tráfico_{ext}$) y tráfico interno ($Tráfico$).

El tráfico externo esta definido como el tráfico enviado a la red externa por otras aplicaciones. Este tráfico se genera usando una función de tiempo, denotada por $f(horas)$. Esta función devuelve un valor entre 0 y 1. De esta forma, para obtener el tráfico externo, el valor obtenido por la función es multiplicado por el ancho de banda de la red externa. Detalles de la función $f(horas)$ pueden consultarse en [4].

Por otra parte, el tráfico interno, es definido como el tráfico generado por la misma aplicación en la red. Este valor es calculado como la media aritmética del tráfico interno enviado a la red en los $(n-1)$ instantes de tiempo anteriores.

Los valores del tráfico externo e interno están ponderados por factores γ y β [4], de manera que:

$$f(tráfico) = \gamma\, Tráfico_{ext} + \beta\, Tráfico_{\text{int } er}$$

y la contención de la WAN esta determinada por la siguiente expresión:

$$T_{WAN} = \left(\frac{1 - f(tráfico)}{f(tráfico)} \right) * T_{Enviar}$$

5. El tiempo de vuelo

Este tiempo es utilizado para las comunicaciones remotas y representa el tiempo de transmisión del mensaje al destino, sin consumir tiempo de procesador. El tiempo de vuelo depende de la distancia entre las máquinas que comunican y es diferente en cada par de ellas. De esta manera, el tiempo de vuelo está representado por una matriz de $N_{hosts} x N_{hosts}$, donde N_{hosts} es el número total de procesadores en el entorno Grid [3].

La figura 3.15 el modelo de comunicación punto-punto entre dos procesadores de máquinas diferentes considerando los cinco componentes anteriores.

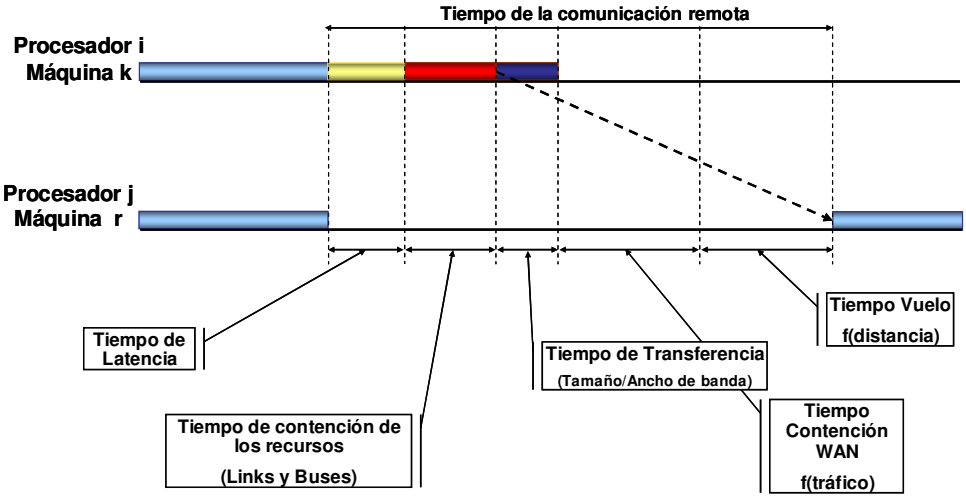


Fig. 3.15 Modelo de comunicación punto-punto

3.6.2 Parámetros y configuraciones del simulador

Cuando realizamos estudios de rendimiento de aplicaciones distribuidas en ambientes Grid, resulta necesario definir los valores de los parámetros que intervienen en el modelo de comunicación del simulador [4]. En nuestro caso, necesitamos determinar los valores de: latencia y ancho de banda entre máquinas, entre nodos y entre procesadores. Los criterios para definir éstos valores son bastante amplios y dependen de múltiples factores, tales como: el servicio de conexión a la red, el medio de transmisión (cable, fibra óptica, etc.) y la distancia a la que se encuentren las máquinas que interconectan.

A continuación presentamos los valores más utilizados, tanto para las comunicaciones externas, comunicaciones entre máquinas, como para las comunicaciones internas, entre nodos y entre procesadores. Estos valores fueron determinados a partir de casos reales, redes actuales y trabajos relacionados.

Iniciamos el estudio, comentando los niveles de comunicación de Dimemas

[17]. Posteriormente, mostramos los valores del ancho de banda en conexiones ADSL, redes LAN, MAN y WAN. Realizamos estudios similares para determinar los valores de la latencia. Por último, especificamos el rango de valores para los parámetros que utilizaremos en nuestros experimentos: la latencia y el ancho de banda.

3.6.2.1 Niveles de comunicación de Dimemas

Dimemas define dos niveles de comunicación [6]. Básicamente, estos niveles son:

1. **Comunicaciones internas**: Representa las comunicaciones entre procesos MPI asignados a procesadores dentro de la misma máquina. De esta manera, las comunicaciones internas pueden ser de dos tipos:
    a. **Comunicaciones entre nodos**: Comunicaciones entre procesadores de diferentes nodos de la misma máquina.
    b. **Comunicaciones entre procesadores**: Comunicaciones entre procesadores dentro del mismo nodo.
2. **Comunicaciones externas**: Representan las comunicaciones entre procesos MPI asignados a procesadores de diferentes máquinas.

Por ejemplo, la figura 3.16 muestra los niveles de comunicación para un entorno Grid compuesto de dos máquinas, donde, la primera máquina esta formada por dos nodos compuestos de dos máquinas, y la segunda máquina, tiene un solo nodo con cuatro procesadores. Las letras a y b en la figura 3.16, denotan el tipo de comunicación interna.

En nuestros experimentos, el nivel de comunicación interno ésta determinado por las comunicaciones entre procesadores, ya que sólo definimos un nodo para cada máquina.

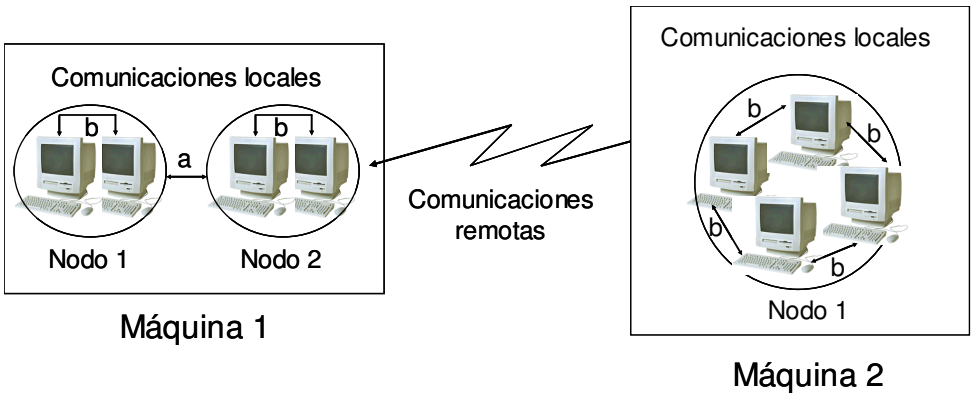


Fig. 3.16 Niveles de comunicación de Dimemas

De esta manera, para definir los entornos Gris de nuestras aplicaciones, necesitamos conocer los valores del ancho de banda y de la latencia para cada nivel de comunicación. La comunicación externa es un componente clave en el rendimiento de muchas aplicaciones distribuidas, debido a la presencia de altas latencias y a la influencia del tráfico en la red. De esta forma, los parámetros que caracterizan el rendimiento de las comunicaciones remotas son: el ancho de banda y la latencia. A continuación realizamos un estudio para determinar los valores de éstos parámetros.

3.6.2.2 Ancho de banda

El ancho de banda se define como la cantidad de datos que pueden enviarse en un período de tiempo determinado, a través de un circuito de comunicación. De esta manera, cuanto mayor sea el ancho de banda, mayor será la cantidad de datos que pueden enviarse por unidad de tiempo.

El ancho de banda depende del tipo de enlace de comunicación y de la distancia a la que se encuentren las máquinas que comunican. La unidad métrica del ancho de banda es el Megabits por segundo (Mbps).

Las redes de transmisión de datos pueden clasificarse dependiendo de su tamaño en redes de tipo: LAN, MAN o WAN. A continuación comentaremos los

valores del ancho de banda para una de estas redes. La tabla 3.1 muestra los valores del ancho de banda para redes tipo LAN y MAN tomando en cuenta la ubicación física de las máquinas que comunican y el medio de transmisión utilizado.

Tabla 3.1. Ancho de banda para redes LAN y MAN

Red	Distancia física máxima entre CPUs	Medios de transmisión	Ancho de banda
LAN	100 m	Par trenzado no blindado. Categoría 5 (UTP) (Ethernet 10BASE-T y 1000BASE-TX)	10 Mbps
	100 m	Par trenzado no blindado. Mejorado de categoría 5 (UTP). (Ethernet 10BASE-T, Fast Ethernet, 100BASE-TX,1000BASE-SX))	100 Mbps
	185 m	Cable coaxial 50 ohmios (Ethernet 10BASE2, ThinNet)	10 Mbps a 100 Mbps
	500 m	Cable coaxial 50 ohmios (Ethernet 10BASE5, ThickNet)	10 Mbps a 100 Mbps
	100 m	Inalámbrico	11 Mbps
MAN	2000 m	Fibra óptica multimodo (62,5/125 mm) 100BASE-FX, 1000 BASE-SX	100 Mbps
	3000 m	Fibra óptica monomodo (núcleo de 9/125 mm) 1000 BASE-LX	1000 Mbps (1 Gbps)

Considerando la tabla anterior, observamos que las redes LAN pueden llegar a tener valores del ancho de banda que oscilan entre 10 Mbps y 100 Mbps. Actualmente, las redes LAN que utilizan como medio de transmisión el par trenzado (UTP categoría 5) y la fibra óptica, utilizan el estándar Gigabit Ethernet, por lo que pueden transmitir datos a 1 Gbps. Otro ejemplo de redes de área local, es la red Myrinet [35] que transmite a 1.2 Gbps. Sin embargo, actualmente más del 85% de las redes todavía utilizan el estándar Ethernet [31].

Por otra parte, en la tabla 3.1 observamos que las redes MAN, definidas como redes entre campus o redes entre municipios, tienen anchos de banda entre 100 Mbps y 1000 Mbps.

Por último, para redes tipo WAN los valores del ancho de banda están determinados por el tipo de comunicación utilizada. La tabla 3.2 muestra éstos valores para cada tipo de servicio, de manera que especifica la velocidad de

transmisión de la red de comunicaciones utilizando determinados estándares. Por ejemplo, OC-x determina la velocidad de las redes de fibra óptica, al igual que E-x (estándar europeo) y T-x (estándar americano) [28].

Tabla 3.2. Ancho de banda para diferentes servicios en redes WAN

Tipo de servicio	Ancho de banda
Módem	0.056 Mbps
RDSI	0.128 Mbps
Frame Relay	0.056 Mbps – 1.544 Mbps
T1	1.544 Mbps
T3	44.736 Mbps
E1	2.048 Mbps
E3	34.368 Mbps
STS-1(OC-1)	51.840 Mbps
STS-3(OC-3)	155.251 Mbps
STS-48(OC-48)	2.488320 Gbps
OC-192	9.95328 Gbps

Como observamos en la tabla 3.2, los valores del ancho de banda para redes WAN oscilan entre 0.056 Mbps y 10 Gbps. Estos valores dependen del tipo de servicio y de los protocolos de comunicación. Ejemplos de este tipo de redes son:

- la red GÉANT, red de investigación europea, que actualmente permite transferir datos a 10 Gbps [30].
- la red troncal nacional RedIRIS, donde las velocidades de comunicación entre las diferentes comunidades autónomas pueden ser de: 155 Mbps, 622 Mbps y 2.5 Gbps [37].

Hasta ahora hemos comentado los valores teóricos más utilizados del ancho de banda en redes LAN, MAN y WAN. Nuestro interés se centra en determinar los valores más representativos del ancho de banda para utilizarlos en los entornos Grid de nuestras simulaciones. Por esta razón, a continuación comentamos los valores experimentales de éste parámetro, en trabajos de investigación que realizan simulaciones sobre entornos Grid. Por ejemplo, en [20] [47] [51] [58] y [63] realizan simulaciones para estudiar el comportamiento de diferentes aplicaciones

de elementos finitos sobre supercomputadores. En todos estos trabajos los valores del ancho de banda oscilan entre 10 Mbps y 100 Mbps.

Adicionalmente, en [2] se realizan simulaciones para predecir el rendimiento de aplicaciones bioinformáticas sobre entornos Grid utilizando Dimemas. Sin embargo, a pesar de que estas aplicaciones no coinciden con las nuestras, citamos este trabajo para comentar los valores del ancho de banda utilizados en las simulaciones. Estos valores fueron: 70 Kbps, 100 Kbps y 200 Kbps.

Por otra parte, los entornos Grid que definiremos caracterizan situaciones reales de la mediana empresa donde, en la mayoría de los casos, no se dispone ni de redes dedicadas ni de redes WAN que transmitan a 10 Gbps. En este sentido, parece que lo más adecuado es pensar en redes MAN cuyas velocidades de transmisión se adaptan a las conexiones ADSL [29]. Las conexiones ADSL más comunes, tienen velocidades máximas de transferencia que oscilan entre 0.256 Mbps y 0.512 Mbps, aunque actualmente se puede llegar a 2 Mbps. Sin embargo, todavía existen servidores que operan a 0.064 Mbps, por lo que, los valores del ancho de banda externo pueden oscilar entre 0.064 Mbps y 2 Mbps. Por esta razón, en nuestras simulaciones elegimos tres valores representativos de este rango de valores. De esta manera, los valores del ancho de banda entre máquinas son: 0.064 Mbps, 0.3 Mbps y 2 Mbps.

Similarmente, para definir los valores internos del ancho de banda, utilizamos los valores presentes en las redes LAN y los utilizados en las simulaciones ejecutadas sobre arquitecturas paralelas. De esta manera, el rango de valores de éste parámetro oscila entre 10 Mbps y 100 Mbps. Sin embargo, en nuestras simulaciones utilizamos 100 Mbps, ya que actualmente las redes LAN pueden transmitir a esta velocidad o inclusive superar este valor.

3.6.2.3 Latencia

La latencia se define como el tiempo inicial que se necesita para realizar la transferencia entre dos procesadores. Este tiempo es cuantificado en

microsegundos (μs). Al igual que en el caso anterior, iniciamos el estudio comentando los valores de la latencia en redes LAN, MAN y WAN. Esto nos permitirá determinar los valores que utilizaremos en nuestras simulaciones. Necesitamos determinar dos rangos de valores, aquellos que definirán la latencia entre máquinas y los que definirán la latencia entre los procesadores de una misma máquina.

En las redes LAN, la latencia entre máquinas es muy pequeña y sus valores se encuentran en el orden de los microsegundos. Por ejemplo, el valor de la latencia en la red Myrinet es de 3.2 μs [35]. Sin embargo, se registran valores experimentales de redes Ethernet (a 10 Gbps) que utilizan valores de la latencia entre 41 μs y 57 μs [34]. Otros estudios [47], realizan evaluaciones del rendimiento de las aplicaciones utilizando diferentes protocolos de comunicación (TCP/IP, M-VIA y cLan), y los valores de la latencia oscilaron entre 100 μs y 10000 μs. Similarmente, en [15], se realizan simulaciones en entornos distribuidos (redes LAN) que evalúan el rendimiento de las aplicaciones utilizando diferentes librerías de comunicación para realizar el paso de mensajes. En estas simulaciones, los valores de la latencia fueron: 150 μs y 300 μs. Considerando los resultados anteriores, observamos que el rango de valores experimentales para la latencia se encuentra acotado entre decenas y cientos de microsegundos.

En las redes MAN, con estándares de fibra óptica (FDDI y DBBQ), los valores de la latencia se encuentran en el orden de unos pocos milisegundos (ms).

En las redes WAN, los valores de la latencia dependen de la distancia a la que se encuentren las máquinas que interconectan, así como también del tráfico en la red de comunicaciones. De esta forma, si se consideran conexiones para máquinas ubicadas entre Europa y América o inclusive dentro de España, los valores de la latencia pueden oscilar entre 10 ms y 500 ms. Las conexiones por satélites utilizando anchos de banda de 50 Mbps tienen latencias entre 250 ms y 300 ms.

Hasta aquí, hemos identificado para cada tipo red el rango de valores de la

latencia. Entonces, para comunicaciones internas, parece razonable elegir valores de la latencia que se encuentren en el orden de las decenas de microsegundos. Particularmente, en nuestros experimentos utilizaremos 25 µs.

Por otra parte, observamos que la latencia en las comunicaciones externas varía desde decenas a cientos de milisegundos. Por esta razón, en nuestras simulaciones los valores de latencia externa elegidos fueron 10 ms y 100 ms. Estos valores son típicos de las redes MAN y de las redes WAN de distribución geográfica en España.

3.6.2.4 Otros parámetros

En esta sección, comentamos los valores de otros parámetros que son necesarios para definir el entorno Grid de nuestras simulaciones [6]. Estos parámetros están relacionados con: el modelo de tráfico y el número de links para cada máquina.

- **Modelo de tráfico:** Como lo indicamos cuando describimos el modelo de comunicación de Dimemas, el modelo de tráfico esta compuesto por dos tipos de tráfico. El primero de ellos, está relacionado con el tráfico externo de la red y el segundo depende del tráfico generado por la propia aplicación. Los valores del tráfico externo e interno están ponderados por factores γ y β.

  En nuestras simulaciones, las comunicaciones dependen principalmente de la red externa y no del tráfico generado por la propia aplicación. Por esta razón, en nuestras simulaciones, ponderamos la influencia del tráfico externo en un 99% ($\gamma = 0.99$) y la influencia del tráfico interno en 1% ($\beta = 0.01$).

- **Links:** En nuestras simulaciones, suponemos que todas las máquina que definen el entorno Grid disponen de tantos links como comunicaciones

externas tenga esa máquina con otras máquinas. En nuestras simulaciones, necesitamos como máximo disponer de seis links por máquinas (para más detalle ver la tabla 4.6 del capítulo 4). Esta cantidad denota la máxima cantidad de comunicaciones externas con diferentes máquinas que puede realizar una determinada máquina. Cada link gestiona todas las comunicaciones remotas que tiene la máquina con una máquina particular. Como veremos, disponer de un link para realizar estas comunicaciones remotas es una restricción que impone la estrategia de distribución de datos *U-Bdomains*.

La tabla 3.3 contiene los valores de los parámetros de comunicación utilizados en nuestras simulaciones. La columna interna define los valores de los parámetros para las comunicaciones internas, mientras que, la columna externa, define los valores de éstos parámetros para las comunicaciones remotas.

Tabla 3.3. Parámetros de comunicación

Parámetros	Internos (entre procesadores)	Externos (entre máquinas)
Latencia (L)	25 µs	1ms, 10 ms y 100 ms
Ancho de Banda	100 Mbps	64 Kbps, 300 Kbps y 2 Mbps
Tráfico	$\beta = 0.01$	$\gamma = 0.99$

Capítulo 4

Descomposición en dominios para entornos Grid

Debido a su papel crítico en el rendimiento de aplicaciones paralelas, el problema de balanceo de carga ha sido estudiado recientemente para entornos Grid. Hasta ahora, se han desarrollado herramientas para realizar el balanceo de carga en ambientes paralelos [8] [38]. Muchos de estos trabajos están pensados para los tradicionales sistemas distribuidos y no son apropiados para entornos Grid [19].

El cálculo sobre ambientes Grid presenta heterogeneidad de plataformas y entornos dinámicos, los cuales representan obstáculos para la aplicación directa de las técnicas de balanceo de carga convencionales. De esta forma, este capítulo plantea el problema de cómo adaptar estos esquemas de balanceo de carga y propone nuevas alternativas para realizar la distribución de los datos en entornos Grid.

4.1 Trabajos anteriores

Para que las simulaciones científicas se ejecuten de manera eficiente en entornos de computación distribuidos, necesitamos particionar la malla de elementos finitos entre los diferentes procesadores de manera que, la cantidad de cálculo esté balanceada, y la comunicación entre procesadores sea mínima.

Esta sección incluye un resumen de las investigaciones realizadas hasta el momento en el área de balanceo de carga de los datos en entornos distribuidos y en entornos Grid. Como veremos, muchas de las herramientas implementadas para balancear los datos en entornos distribuidos han sido extendidas, mejoradas o adaptadas para ser utilizadas en entornos Grid.

4.1.1 Balanceo de carga en entornos distribuidos

El balanceo de carga en entornos distribuidos se realiza considerando diferentes algoritmos de particionamiento de grafos que permiten dividir y repartir los datos de forma eficiente en éstos entornos [6] [14] [15] [18] [25] [34] [37] [46].

Actualmente existe una gran variedad de algoritmos de particionamiento, sin embargo, para obtener particiones óptimas de los datos es necesario utilizar algoritmos de particionamiento multinivel [2] [16] [21] [24] [28] [44]. Versiones paralelas de estos algoritmos han sido desarrolladas para entornos distribuidos [20] [22] [23] [38]. Estas versiones son capaces de escalar miles de procesadores y generar particiones para mallas con billones de elementos.

Sin embargo, a pesar del éxito que estos algoritmos de particionamiento han tenido, existe un número importante de aplicaciones científicas que no pueden ser paralelizadas de forma eficiente. Ejemplos de tales aplicaciones son las simulaciones científicas de: colisiones de coches, deformación de materiales y estampado de chapa. La razón de que estas aplicaciones no puedan ser paralelizadas de forma eficiente se debe a que al particionar los datos no se toman en cuenta la geometría de las mallas y se obtienen particiones de datos que generan *overheads* en la comunicación. Para solventar este problema se han planteado algoritmos de

particionamiento que toman en cuenta la geometría de la malla al particionar los datos. Detalles de estos algoritmos pueden consultarse en [1] [33]. De esta forma, la gran mayoría de los particionadores actuales realizan particionamiento multinivel considerando la geometría de las mallas.

Sin embargo, a pesar de la gran variedad de algoritmos de particionamiento, no existe un algoritmo que sea efectivo para todas las aplicaciones. Esto se debe a que el problema del balanceo de carga es complejo y depende de las características de cada aplicación [3]. Por esta razón, no existen muchas referencias bibliográficas que determinen el particionador de datos mas adecuado [5] [35].

Como consecuencia de la gran variedad de algoritmos de particionamiento, existen una gran cantidad de herramientas que utilizan diferentes algoritmos de particionamiento para realizar el balanceo de la carga de los datos en entornos distribuidos [4] [8] [17] [26] [39] [40] [42] [45]. Sin embargo, de todos estos particionadores, el particionador que más destaca por su rendimiento es METIS, ya que genera particiones de datos óptimas. METIS es una herramienta software que permite efectuar el particionamiento de grafos y de mallas de elementos finitos utilizando algoritmos de multinivel [16] [21] [24]. Los algoritmos de particionamiento multinivel, reducen en cada nivel el tamaño del grafo y realizan particiones sobre grafos más pequeños. Estos subgrafos son utilizados, posteriormente, para construir una partición del grafo original. Por el contrario, los algoritmos de particionamiento tradicionales calculan la partición operando directamente sobre el grafo. Esto hace que estos algoritmos resulten lentos y que las particiones obtenidas no sean óptimas.

Por otra parte, casi todos los particionadores generan particiones de datos balanceadas. Este tipo de partición resulta adecuada para entornos homogéneos donde todas las comunicaciones tienen el mismo coste. Sin embargo, METIS permite realizar particiones de datos no balanceadas de manera que es posible considerar diferentes costes de comunicación y máquinas heterogéneas. De esta manera, lo único que hay que determinar es la relación de proporcionalidad que define la carga computacional que le corresponde a cada procesador y a cada máquina.

Existen trabajos, como por ejemplo [43], que sugieren desbalancear la carga de los datos en entornos distribuidos. Los resultados experimentales, utilizando redes de computadores homogéneos, muestran que permitir cargas de los datos no balanceadas, contribuye a reducir la comunicación y el tiempo de ejecución de las aplicaciones.

Como observamos, la gran mayoría de los particionadores fueron pensados para trabajar sobre ambientes distribuidos de supercomputación, dónde los entornos son homogéneos y dónde todas las comunicaciones tienen el mismo coste. Sin embargo, cuando estos particionadores son utilizados sobre entornos distribuidos, el rendimiento de las aplicaciones se degrada. De esta forma, si decidimos utilizarlos para repartir la carga de datos en entornos Grid, estos particionadores resultarían ineficientes, debido a que no consideran los efectos en el rendimiento producidos por las altas latencias y los bajos anchos de banda, propios de éstos entornos de programación [12] [10] [19].

De esta manera, para realizar la partición de los datos en entornos Grid necesitamos utilizar herramientas de particionamiento que consideren los efectos que producen la latencia y el ancho de banda en el rendimiento de la aplicación.

Este problema ya ha comenzado a estudiarse, de manera que, para determinar cómo adaptaremos los esquemas de balanceo de carga es necesario conocer las etapas que caracterizan este proceso de distribución de los datos [27]. Básicamente, un esquema de balanceo de carga esta formado por tres fases:

1. **La colección de los datos:** En esta fase se obtiene información sobre la distribución de la carga de trabajo y el estado del sistema. De esta manera, es posible detectar en qué procesadores se produce el desbalanceo de la carga de los datos.
2. **El particionamiento de los datos:** Esta fase determina una distribución óptima de los datos, minimizando el número de particiones (vecinos).

3. **La distribución de los datos:** Esta fase se encarga de reorganizar la cantidad de trabajo adicional de los procesadores sobrecargados, y transferirla a los procesadores con menos carga de trabajo.

En cada una de las fases anteriores, se pueden utilizar diferentes esquemas de balanceo de carga, de manera que dependiendo del esquema utilizado, los esquemas de balanceo de carga de los datos se pueden clasificar en:

Esquemas estáticos y esquemas dinámicos

Los esquemas estáticos [11] necesitan conocer *a priori* información, tanto de la aplicación como del sistema. Estos esquemas resultan ventajosos porque tiene bajo coste. Principalmente, esto se debe a que el particionamiento de los datos divide una sola vez los datos antes de iniciar el cálculo. Sin embargo, este esquema no puede adaptarse a las fluctuaciones de los requerimientos de cálculo de la aplicación y del estado del sistema.

El balanceo de carga dinámico solventa este problema, ya que permite realizar el particionamiento de los datos dependiendo del comportamiento de la aplicación y del sistema. Esto genera costos adicionales que pueden llegar a influir en los tiempos de ejecución de la aplicación.

Esquemas centralizados y esquemas distribuidos

En los esquemas centralizados, un gestor central requiere información de la carga de datos para realizar el particionamiento, basado en el conocimiento global de la aplicación. Estos esquemas, exigen una sincronización global para obtener la información. Esto genera costes de sincronización elevados.

Por otra parte, los esquemas de balanceo distribuidos exigen un conocimiento local de la distribución de carga. Estos esquemas realizan un particionamiento de los datos fundamentado en el conocimiento parcial de la distribución de los mismos [6]. Sin embargo, el desconocimiento global de la carga de trabajo retrasa el balanceo de la carga de los datos.

Esquemas a nivel de aplicación y esquemas a nivel de sistema

En los esquemas a nivel de aplicación, el balanceo de la carga está orientado a minimizar los tiempos de ejecución de la aplicación. Sin embargo, en los esquemas a nivel de sistema, también conocido como planificación distribuida, el balanceo de la carga esta orientado a maximizar el rendimiento/utilización de las máquinas.

4.1.2 Balanceo de carga en entornos Grid

La clasificación anterior realiza una categorización de los esquemas de balanceo de carga en entornos distribuidos. Para entornos Grid, existe también una categorización que toma en cuenta los desafíos del Grid, como por ejemplo, la heterogeneidad de los sistemas, las elevadas latencias y los estados dinámicos del sistema [27]. La clasificación de estos esquemas se resume en tres categorías:

Particiones estáticas para problemas acoplados

Estos esquemas de particionamiento modelan el dominio del problema utilizando un grafo compuesto de nodos y fronteras de comunicación [36]. En estos esquemas, el problema del balanceo de carga tiene como objetivo reducir el desbalanceo de la carga en los subdominios y minimizar las comunicaciones entre ellos. De esta forma, se necesita conocer *a priori,* la carga de trabajo y el estado del sistema, lo que impide que este esquema pueda adaptarse a los cambios del sistema y a la heterogeneidad del Grid. Sin embargo, existen algunos particionadores que consideran este aspecto, pero no toman en cuenta el costo de movilidad de los datos [9] [41]. Este costo influye, de forma significativa, en el rendimiento de la aplicación, debido a las altas latencias presentes en la comunicación remota. Existen otros particionadores [7] que reducen al mínimo el movimiento de los datos derivado del balanceo de la carga.

Particiones estáticas para problemas desacoplados

Este esquema supone que tanto la carga de trabajo como las

comunicaciones pueden dividirse de forma desacoplada [11], es decir, que no existe una relación especial entre los datos, lo que permite que la carga se asigne de forma independiente. Este método toma en cuenta la latencia de comunicación presente en el Grid. Además, optimiza la distribución de la carga de datos. Sin embargo, no toma en consideración los cambios dinámicos de la utilización del procesador y de la red. Esto limita el rendimiento de la aplicación.

Particiones dinámicas

Estos esquemas realizan estimaciones/predicciones de la carga futura y del costo de comunicación para evaluar el rendimiento. Sin embargo, para calcular esta estimación, no es suficiente utilizar expresiones lineales, ya que, el estado del sistema se encuentra en constante cambio. Además, estas expresiones no toman en cuenta la contención de los recursos en el Grid. Algunos experimentos demuestran que este esquema de balanceo de carga funciona bien para tareas a corto plazo [47].

Igual que en los entornos distribuidos y considerando la categorización anterior, no existe un esquema de balanceo de carga que sea efectivo para todas las aplicaciones y que considere todas las características del Grid.

Esto ha motivado la implementación de herramientas de balanceo de carga que resultan adecuadas para determinadas aplicaciones [13], sin dejar de ser aplicables al resto de ellas. En general, las herramientas que mas destacan por su rendimiento en estos entornos son: METIS y MinEX [7].

METIS es una herramienta para realizar el particionamiento de los datos que ha sido comentada anteriormente (sección 4.1.1). MinEX es un particionador de datos que toma en cuenta los efectos de la latencia y de los bajos anchos de banda a la hora de generar las particiones de datos. Sin embargo, este particionador sólo ha sido estudiado en problemas de movimiento de partículas y considerando redes de interconexión dedicadas. Se continúan realizando trabajos para mejorar este particionador, como por ejemplo, implementación de versiones paralelas, simulación

de aplicaciones científicas en entornos Grid considerando valores particulares del ancho de banda.

4.2 Geometría de las mallas

Como lo comentamos anteriormente, al realizar la descomposición en dominios es necesario tomar en cuenta la geometría de la malla de elementos y la cantidad de dominios requerida. Estos factores influyen en el particionamiento de los datos. Por ejemplo, si la cantidad de dominios es pequeña, la partición de la malla de elementos se realiza en una dirección obteniendo particiones en forma de tiras. Esta partición es óptima, ya que, cada dominio de datos está relacionado, como máximo, con dos dominios. Sin embargo, si se incrementa considerablemente el número de dominios, la partición en tiras no resulta óptima, ya que la cantidad de datos obtenida para cada dominio es pequeña. De esta forma, cuando esto ocurre, el siguiente paso es particionar la malla de elementos considerando dos o más direcciones de particionamiento. Sin embargo, igual que en el caso anterior, la cantidad total de dominios máxima está limitada. En estos casos, se considera una nueva dirección de particionamiento.

Las direcciones de particionamiento, también están relacionadas con la geometría de la malla. Si la malla de elementos tiene forma alargada, el particionamiento se realiza en tiras, mientras que si no tiene una dimensión de longitud privilegiada, el particionamiento se realiza en forma de cajas. De esta forma, la decisión de cuando se realiza el particionamiento en forma de tiras o cuando se realiza en forma de cajas, está relacionada con la geometría de la malla de elementos y el número de dominios. Estos factores fueron tomados en cuenta en nuestras estrategias de particionamiento.

De esta forma y dado que queremos estudiar casos muy concretos, nuestras aplicaciones se caracterizan por tener dos tipos de particiones: partición en tiras y partición en cajas. Estos tipos de particiones caracterizan la geometría de las mallas de elementos y generalizan la mayoría de los casos industriales. Las particiones alargadas

son comunes en muchas piezas de la industria automovilística. La mayoría de estas piezas tienen asociadas particiones en forma de tiras. El resto de casos, pueden fácilmente incluirse dentro del caso genérico sin restricciones que caracteriza a las particiones en forma de caja. A continuación describimos estos tipos de particiones.

Partición en tiras: En este tipo de partición la malla de elementos puede descomponerse en tiras (figura 4.1). Esta descomposición garantiza que, cada proceso paralelo (dominio) tiene como máximo dos comunicaciones remotas con otras máquinas vecinas. Esta situación ocurre cuando la geometría de la malla de elementos tiene una dirección principal, como por ejemplo, cuando simulamos el estampado del parachoques o el chasis de un automóvil, o cuando la cantidad de dominios es pequeña. Por ejemplo, si consideramos una malla compuesta por 1.000.000 de elementos cuyas dimensiones son 10^4x10x10 nodos, la forma de la malla sería la siguiente:

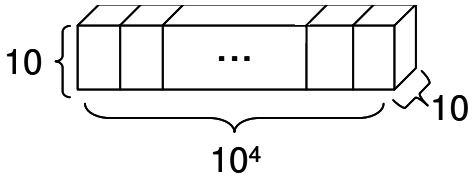


Fig. 4.1 Representación gráfica de la malla: partición en tiras

Partición en caja: Estas particiones son comunes en mallas de elementos cuyas geometrías no tienen dimensiones privilegiadas, como por ejemplo la estructura completa de un vehículo. Además, estas particiones se realizan cuando la cantidad de dominios es grande. En ambos casos, la malla de elementos no puede ser descompuesta en tiras, por lo que el número de dominios vecinos para esta partición es mayor que para el particionamiento en tiras (figura 4.2). La cantidad de dominios vecinos es variable y depende del particionamiento concreto que se realice. Igual que en el caso anterior, si obtenemos una malla de elementos cuyas dimensiones son

10^2x10^2x10^2 nodos, la forma de la malla es:

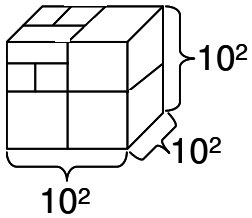


Fig. 4.2 Representación gráfica de la malla: partición en cajas

Conocida la geometría de la malla de elementos y considerando una distribución no balanceada de los datos realizamos el particionamiento proponiendo nuevas estrategias de distribución. Antes de describir estas estrategias, comentamos la clásica distribución de datos, la distribución balanceada.

4.3 Distribución balanceada

Esta distribución divide la malla de elementos finitos en particiones de datos uniformes. El número de partes/dominios es equivalente al número de procesadores totales que forman el Grid. Este particionamiento de datos garantiza que la carga computacional esta balanceada en todo el entorno Grid.

La distribución balanceada es muy eficiente en ambientes homogéneos, donde todas las comunicaciones tienen el mismo costo. Sin embargo, en ambientes heterogéneos esta estrategia puede no ser la más adecuada.

En nuestros experimentos, comparamos los resultados de rendimiento de nuestras aplicaciones con los obtenidos al utilizar la distribución balanceada para particionar los datos. Para realizar el particionamiento balanceado de los datos utilizamos METIS, particularmente, las rutinas de: *METIS_PartGraphRecursive* y *METIS_PartGraphKway*. *METIS_PartGraphRecursive* utiliza algoritmos recursivos de bisección multinivel, mientras que *METIS_PartGraphKway*, emplea algoritmos de particionamiento multinivel *K-way*. Ambas rutinas particionan la malla de datos en *k* partes iguales. La elección de utilizar una determinada rutina, está relacionada con el número de máquinas remotas que

forman el Grid. De esta forma, cuando el número de máquinas remotas es menor o igual a ocho utilizamos *METIS_PartGraphRecursive*, en caso contrario, utilizamos *METIS_PartGraphKway*. La razón de esta elección es que *METIS_PartGraphKway* genera particiones con menos vecinos.

Las rutinas de METIS necesitan como entrada el grafo de elementos y las funciones de peso asignadas a cada nodo. El grafo de elementos describe la lista de adyacencia de cada nodo (vecinos). Esta estructura es almacenada utilizando un formato de almacenamiento comprimido. En este formato, un grafo de *n* vértices es representado utilizando tres *arrays*, denotados por *xadj*, *adj* y *edwgt*, donde:

xadj: contiene el índice en *adj,* del primer vecino de cada nodo,

adj: contiene la lista de vecinos de cada nodo, y

edwgt: contiene el peso asignado a cada nodo.

Por ejemplo, la figura 4.3 muestra el grafo de elementos finitos con 9 grados de libertad y su correspondiente formato de almacenamiento, considerando los arrays *xadj*, *ady* y *edwgt*.

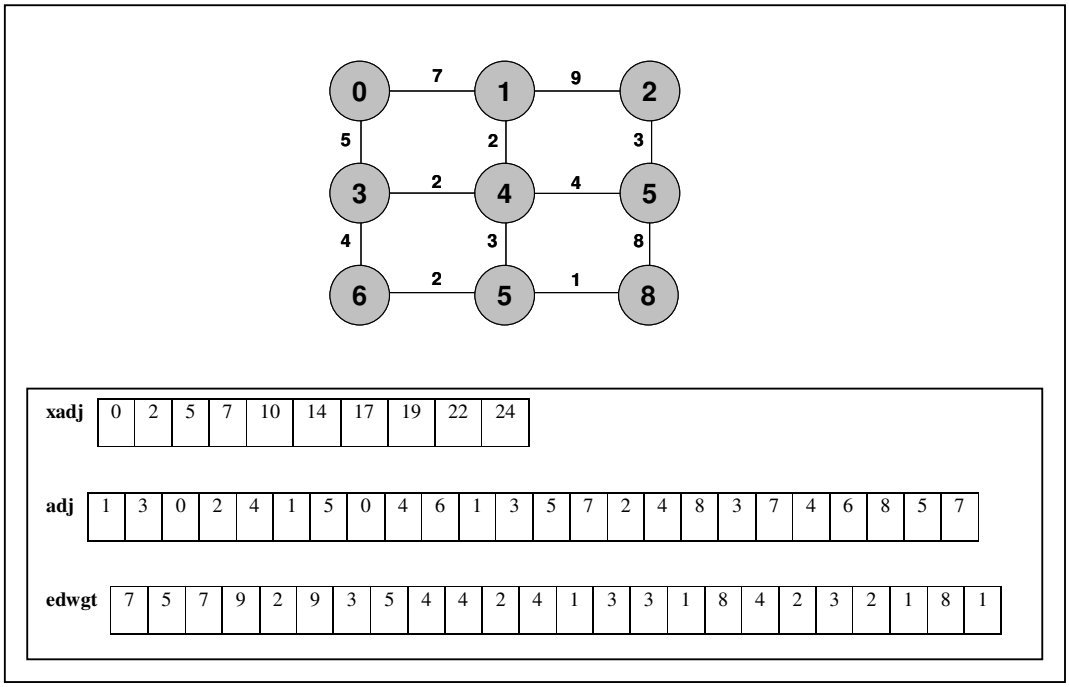


Fig. 4.3 Formato de almacenamiento del grafo de elementos

De esta forma, el algoritmo de generación de particiones balanceadas es el siguiente:

ENTRADA: Matriz en formato comprimido, número de máquinas remotas (*nhosts*).

PROGRAMA

1. **Generación *nhosts* partes**

   Si (nhosts <=8) **entonces**

   METIS_PartGraphRecursive

   sino

   METIS_PartGraphKway

   fsi

2. **Marcar nodos frontera**

3. **Ordenar nodos**

   Nodos internos de todos los dominios

   Nodos fronteras de todos los dominios

4. **Permutar matriz de datos y vector del sistema**

   Para *nhosts* dominios **hacer**

   Obtener partición del subgrafo

   Reordenar (nueva numeración nodos)

   Numerar nodos internos

   Numerar nodos frontera

   Actualizar vector de permutación

   Fpara

5. **Obtener matriz y vector con nueva numeración (permutados)**

SALIDA: ficheros con subgrafos en formato comprimido

Para ilustrar el funcionamiento del algoritmo de generación de particiones balanceadas consideramos una malla de elementos finitos en 2D con 64 grados de libertad (figura 4.4) y un entorno Grid compuesto de dos máquinas con cuatro procesadores cada una.

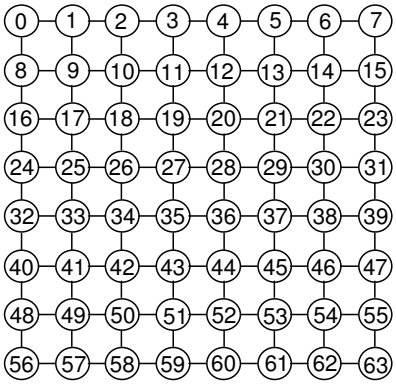


Fig. 4.4 Grafo asociado a la malla de elementos finitos

Inicialmente, se realiza una partición por máquinas, en este caso se obtienen dos particiones. En la figura 4.5, la línea discontinua ilustra estas particiones. En cada partición se efectúa el particionamiento de la malla, considerando el número de procesadores en cada máquina. Luego se numeran los nodos, comenzando por los nodos internos de cada partición de datos y continuando con sus nodos fronteras (figura 4.6).

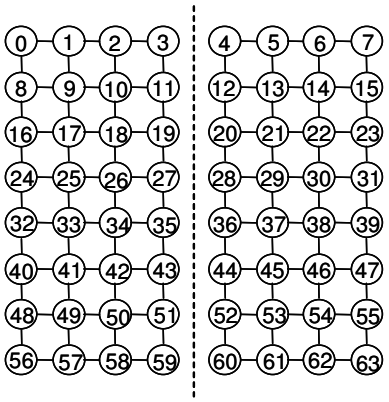


Fig. 4.5 Partición por máquinas

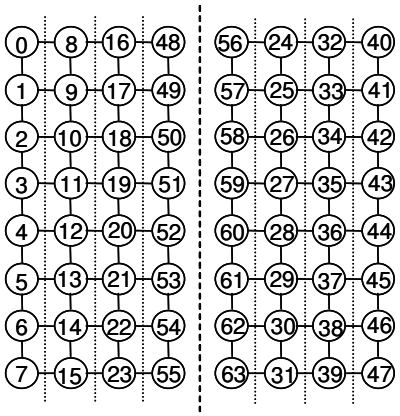


Fig. 4.6 Numeración de los nodos: Distribución balanceada

La figura 4.7 muestra los dominios generados para la arquitectura considerada utilizando la distribución balanceada. En este caso, el grafo de elementos tiene asociado una malla de forma alargada, y dado que la cantidad de procesos es relativamente pequeña, la partición generada se realiza en una sola dirección obteniendo particiones en forma de tiras.

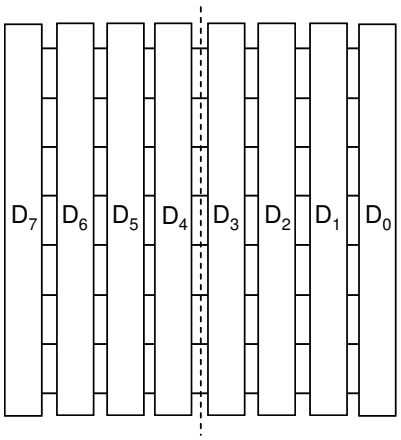


Fig. 4.7 Dominios distribución balanceada: Partición en tiras

Por otra parte, para grafos de elementos cuya malla de elementos tiene forma cuadrada, la partición generada se realizada en varias direcciones, obteniendo particiones en forma de caja. La figura 4.8 muestra un ejemplo para este tipo de mallas considerando una arquitectura de dos máquinas y 16 procesadores por máquina.

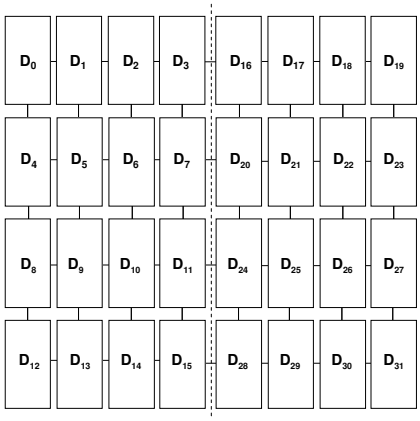


Fig. 4.8 Dominios distribución balanceada: Partición en cajas

4.4 Distribuciones no balanceadas

Para mejorar el rendimiento de las aplicaciones sobre entornos de programación Grid, proponemos nuevas estrategias de distribución no balanceadas de los datos para éstos entornos. Estas distribuciones de datos toman en cuenta el costo de las comunicaciones entre máquinas y entre procesadores.

El algoritmo para realizar el particionamiento no balanceado de los datos es el siguiente:

ENTRADA: Matriz y número de máquinas remotas (*nhosts*).

PROGRAMA

Leer matriz

Generación *nhosts* partes

Si (*nhosts*>1) **entonces**

Si (*nhosts* <=8) **entonces**

METIS_PartGraphRecursive

sino

METIS_PartGraphKway

fsi

Numeración de los dominios

Marcar nodos frontera de los dominios

Obtener permutaciones

sino

Generar única partición

fsi

Generar vectores de permutación de filas y columnas

Generar vecinos de cada dominio

Permutar matriz de datos y vector del sistema

Para *nhosts* dominios **hacer**

Extraer nodos internos

Obtener partición del subgrafo

Generar partición final con numeración global

Nueva numeración de los nodos en la máquina

Marcar nodos frontera del dominio

Obtener permutaciones

fpara

Permutación de la matriz

Permutación del vector independiente

Generación de los bloques de la matriz

SALIDA: Subgrafos de la matriz y del vector independiente en formato CSR. Estructuras de distribución adicionales.

A continuación, describimos en qué consisten las distribuciones de datos propuestas [29] [30] [31] [32].

4.4.1 Propuesta 1: Distribución *U-1domains*

Esta estrategia es denominada distribución *U-1domains*, ya que todas las máquinas, además de los dominios estándares, tienen un dominio especial. Este dominio especial contiene una partición de los nodos que son frontera entre los dominios asignados a máquinas remotas.

Al realizar la operación producto matriz vector, la información de los nodos frontera es enviada a los procesadores remotos en otras máquinas. Estas comunicaciones son lentas y generan grandes costes que perjudican el rendimiento de la aplicación. La distribución no balanceada, asocia los dominios especiales con las comunicaciones remotas. El resto de dominios que no contienen particiones de los nodos frontera son llamados dominios computacionales. Estos dominios son asignados, respectivamente, a procesadores dentro de la máquina. De esta manera, la distribución no balanceada esta formada por dos tipos de dominios:

- **Dominios especiales:** Dominios que contienen únicamente la frontera de datos que es común entre los dominios de las máquinas que comunican remotamente.
- **Dominios computacionales:** Dominios que contienen particiones/subgrafos de la malla de datos asociada al problema. Estos dominios no contienen nodos frontera. Los procesadores a los que se les asignan dominios computacionales realizan más cálculo que los procesadores que tienen únicamente dominios especiales.

Este tipo de dominios, es común entre las diferentes propuestas de distribuciones no balanceadas.

A continuación describimos cómo particionamos los datos utilizando la distribución *U-1domains*. Básicamente, la distribución *U-1domains* se realiza en dos fases:

1. **Fase 1. Particionamiento a nivel de máquinas**

   En esta primera fase, dividimos la malla de elementos finitos en tantas partes como máquinas remotas tenga el Grid. En este caso, se realiza un particionamiento proporcional de los datos, dependiendo de la velocidad relativa de cada máquina. Esto garantiza que la carga de trabajo computacional para cada máquina remota es aproximadamente la misma.

Para realizar este primer nivel de particionamiento, usamos METIS.

2. **Fase 2. <u>Particionamiento a nivel de procesadores</u>**

   En esta segunda fase, trabajamos con cada subgrafo obtenido en la fase anterior. Para cada máquina se obtiene el dominio especial, subgrafo integrado por todos los nodos fronteras. Posteriormente, los nodos restantes, es decir los nodos que no han sido incluidos como dominios especiales forman un nuevo subgrafo. Este subgrafo es particionado en *nprocs*-1 partes, donde *nprocs* representa la cantidad total de procesadores en la máquina. Este particionamiento se considerando las velocidades relativas de cada procesador en la máquina.

Para ilustrar la generación de las particiones de los datos utilizando la distribución *U-1domains*, consideramos el grafo asociado a la malla de elementos finitos de la figura 4.4. Igual que en el ejemplo de la distribución balanceada, supondremos un entorno Grid compuesto por dos máquinas y cuatro procesadores por máquina.

La primera fase de la estrategia es similar a la planteada para la distribución balanceada (figura 4.5). En este caso, para cada subgrafo, se identifican los nodos frontera que forman parte de los dominios especiales. Estos nodos no son incluidos en el particionamiento de la segunda fase. De esta forma, cada dominio especial es asignado a un procesador dentro de la máquina, por lo que el subgrafo restante es dividido, para cada máquina, en tres partes.

La figura 4.9 ilustra los nodos fronteras para este ejemplo utilizando círculos rellenos de color gris. Además, esta figura muestra la nueva numeración de los nodos utilizando la distribución *U-1domains*. Las figuras 4.10 y 4.11 describen, respectivamente, los dominios obtenidos al utilizar la distribución *U-1domains* considerando particiones en forma de tiras y de cajas. En cada caso, los dominios denotados como D_3 y D_4 representan los dominios especiales.

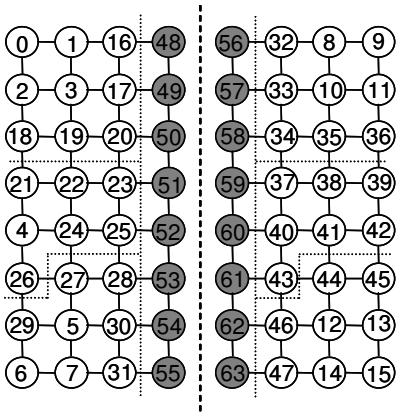


Fig. 4.9 Numeración de los nodos: Distribución *U-1domains*

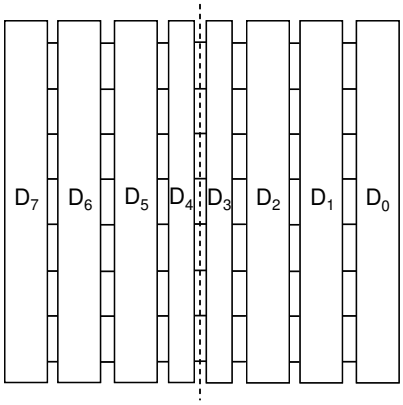


Fig. 4.10 Dominios distribución *U-1domains*: Partición en tiras

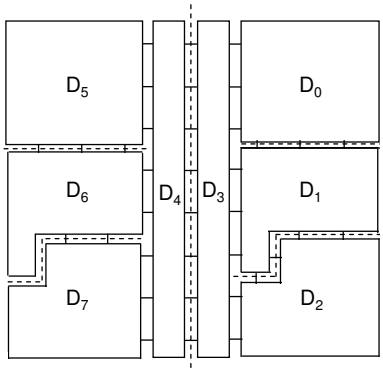


Fig. 4.11 Dominios distribución *U-1domains*: Partición en cajas

Como veremos, los patrones de comunicación de la aplicación dependen de la arquitectura y de la geometría de la malla de elementos. La figura 4.12 ilustra los patrones de comunicación para una iteración del método explícito considerando las distribuciones balanceada y no balanceada del ejemplo anterior. Las flechas en la figura, identifican los procesadores que intercambian datos. Las flechas de longitud menor, representan las comunicaciones locales, mientras que las flechas de longitud mayor y con trazado discontinuo representan las comunicaciones remotas.

Las figuras 4.12.a y 4.12.b muestran como las comunicaciones remotas se solapan con el cálculo. Sin embargo, en la figura 4.12.c todos los procesadores se encuentran ocupados y las comunicaciones remotas se realizan al final de cada iteración.

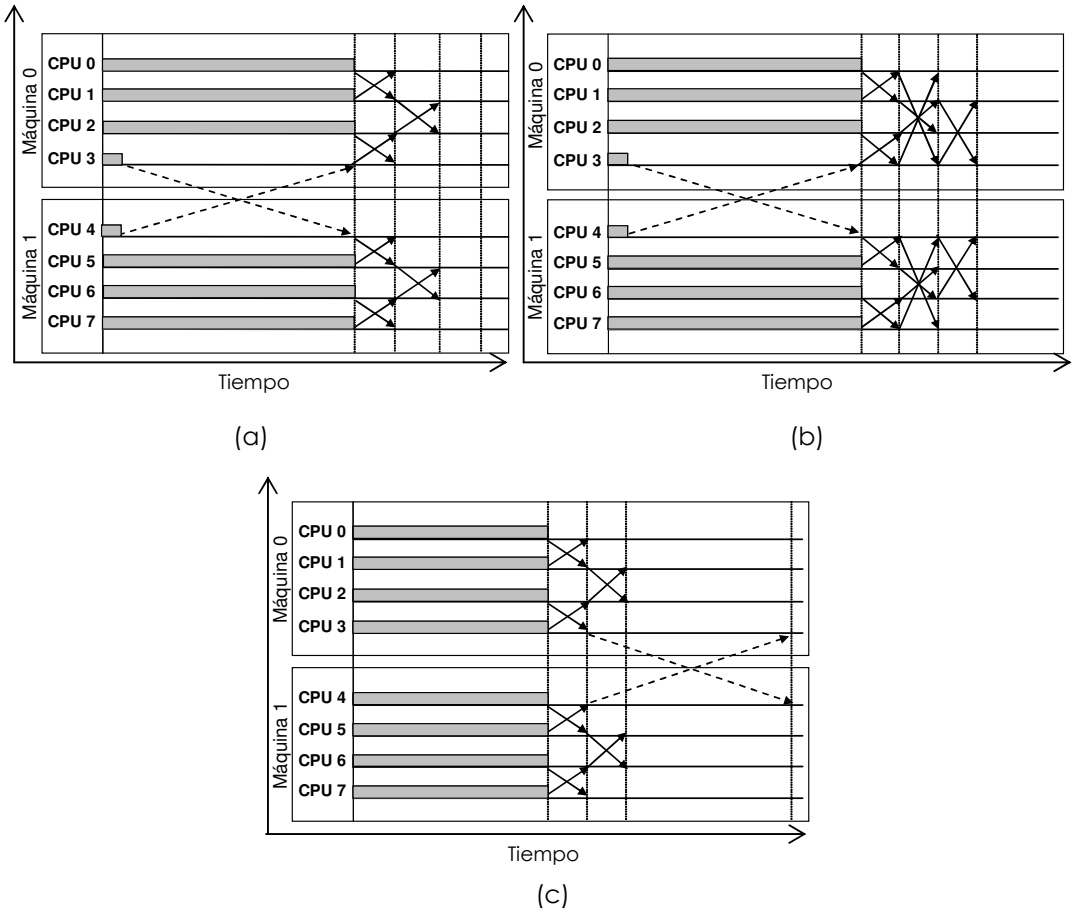


Fig. 4.12 Diagrama de comunicaciones para una iteración del método explícito: (a) Distribución *U-1domains*: partición en tiras (b) Distribución *U-1domains*: partición en cajas (c) Distribución balanceada: partición en tiras

La figura 4.12.c muestra el patrón de comunicaciones para una partición en tiras de la malla de elementos finitos, utilizando la distribución balanceada. En este caso, el número de procesos requerido es pequeño, por lo que la partición en cajas no es posible. Sin embargo, cuando la partición de la malla se realiza en cajas, el patrón de comunicaciones para esta distribución incorpora un mayor número de comunicaciones remotas por iteración. Por ejemplo, si consideramos el grafo de la figura 4.8 obtenemos el diagrama de comunicaciones de la figura 4.13.

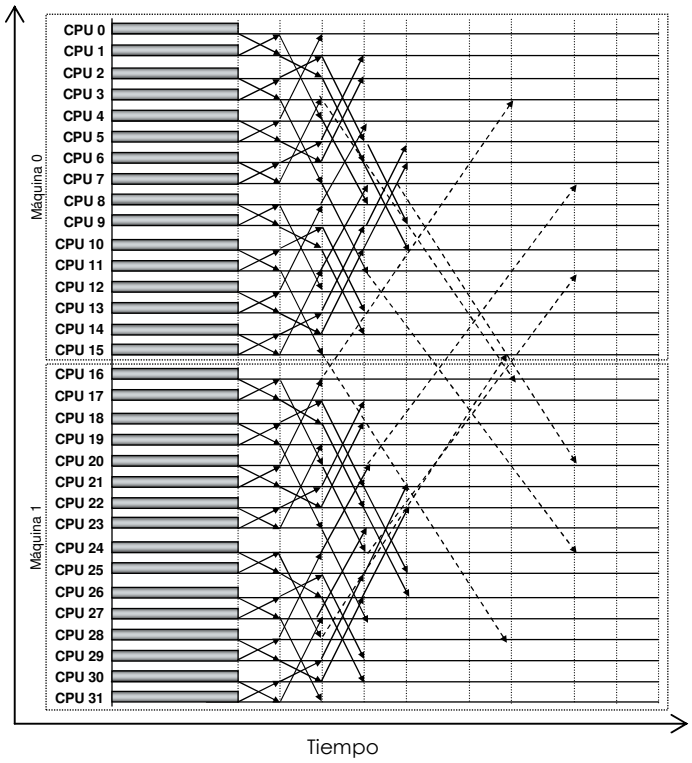


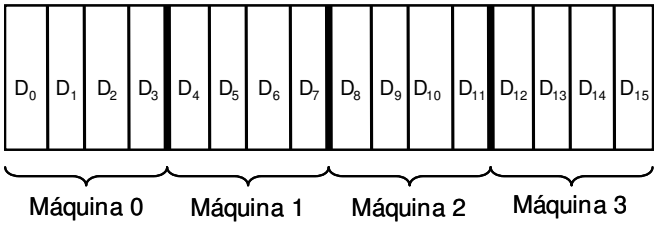
Fig. 4.13 Diagrama de comunicaciones: Distribución balanceada (partición en cajas)

Por otra parte, si realizamos el diagrama de comunicaciones para este ejemplo utilizando la distribución *U-1domains*, tendríamos una sola comunicación remota por iteración. Esta comunicación remota se solaparía con el cálculo.

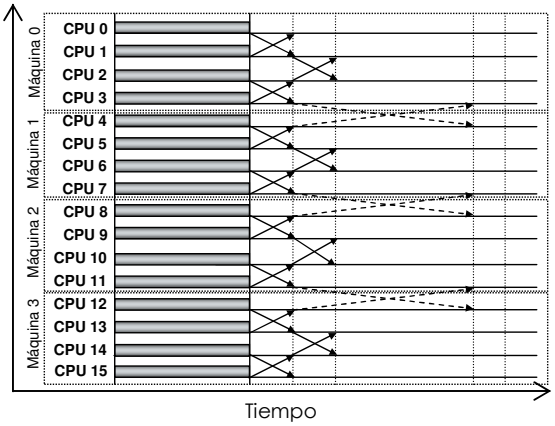
De esta forma, para entornos Grid compuestos por dos máquinas y ocho o más procesadores, la distribución *U-1domains* toma ventaja con respecto a la distribución balanceada, independientemente de la partición que se realice (tiras

o cajas). Estos resultados pueden observarse en el anexo A, figuras A.2 a A.5 y A.12 a A.15.

Sin embargo, para entornos Grid que consideran un mayor número de máquinas, la distribución *U-1domains* no puede competir con la distribución balanceada. Básicamente, el problema se debe a que el número de pasos para completar las comunicaciones de una iteración del método explícito, tiene más de una comunicación remota. Estas comunicaciones se realizan en secuencia, lo que aumenta considerablemente los tiempos de ejecución de la aplicación. Por ejemplo, si consideramos un entorno Grid formado por 4 máquinas y 4 procesadores por máquina y una partición en tiras, obtendríamos, respectivamente para cada distribución, los esquemas de particionamiento y de comunicación asociados a las figuras 4.14 y 4.15.



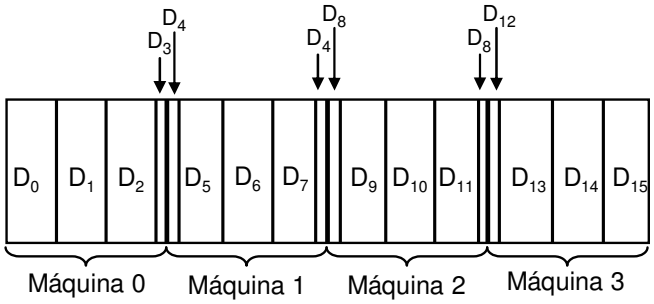
(a)



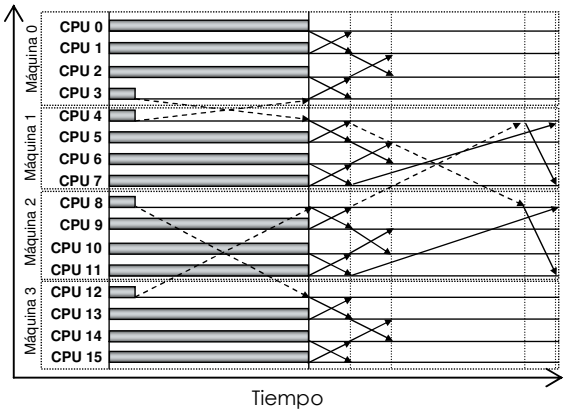
(b)

Fig. 4.14 Distribución balanceada: (a) Partición en tiras de la malla (b) Diagrama de comunicaciones

En la figura 4.14.a, la línea de trazado mas grueso delimita la porción de datos y la cantidad de dominios asignada a cada máquina. Cada dominio de datos obtenido es asignado al procesador con notación correspondiente, es decir, el dominio D_0 es asignado al procesador P_0 y así sucesivamente. De esta forma, el diagrama de comunicación para la figura 4.14.b indica que el número de pasos totales para completar una iteración esta formado por 1 comunicación remota y 1 comunicación local. Sin embargo, para la distribución *U-1domains* el esquema de particionamiento indica, por ejemplo, que la máquina 1 tiene comunicación remota con la máquina 0 y con la máquina 2 (figura 4.15.a). De esta forma, el diagrama de comunicaciones para el mismo tipo de partición (figura 4.15.b) muestra que son necesarias 2 comunicaciones remotas y 2 comunicaciones locales para completar la comunicación de una iteración.



(a)



(b)

Fig. 4.15 Distribución *U-1domains* (a) Partición en tiras de la malla (b) Diagrama de comunicaciones

Estas comunicaciones se realizan en secuencia, ya que sólo hay un único procesador que gestiona todas las comunicaciones externas de la máquina. Sin embargo, en la distribución balanceada cada comunicación remota es gestionada por diferentes procesadores. Las tablas 4.1 y 4.2 muestran la cantidad total de pasos de comunicación requeridos para completar una iteración del método explícito considerando diferentes topologías y particiones. De esta forma, el particionamiento del grafo de elementos depende de la geometría de la malla y del número de procesos que forman el entorno Grid.

Tabla 4.1. Número de pasos de comunicación de una iteración del método explícito: Distribución balanceada y distribución *U-1domains* (partición en tiras)

	Partición en tiras			
Máquinas x CPUs	**Distribución balanceada**		**Distribución *U-1domains***	
	Comunicaciones		**Comunicaciones**	
	remotas	**locales**	**remotas**	**locales**
2x4	1	1	1	1
2x8	1	1	1	1
2x16	1	1	1	1
2x32	1	1	1	1
2x64	1	1	1	1
4x4	1	1	2	2
4x8	1	1	2	2
4x16	1	1	2	2
4x32	1	1	2	2
8x8	1	1	2	2
8x16	1	1	2	2

Tabla 4.2. Número de pasos de comunicación de una iteración del método explícito: Distribución balanceada y distribución *U-1domains* (partición en cajas)

	Partición en cajas			
Máquinas x CPUs	**Distribución balanceada**		**Distribución *U-1domains***	
	Comunicaciones		**Comunicaciones**	
	remotas	**locales**	**remotas**	**locales**
2x4	2	3	1	3
2x8	4	5	1	6
2x16	5	8	1	7
2x32	6	7	1	15
2x64	7	8	1	25
4x8	7	5	3	6
4x16	10	9	3	11
4x32	9	8	3	22
8x8	13	5	6	7
8x16	13	4	6	13

4.4.2 Propuesta 2: Distribución *U-Bdomains*

Esta propuesta surge como una alternativa para solucionar el problema de la propuesta anterior al realizar comunicaciones remotas en la misma máquina con diferentes máquinas. Este problema origina cuellos de botella, ya que únicamente se dispone de un procesador en la máquina para gestionar todas las comunicaciones remotas. La formación de estos cuellos de botella hace que el rendimiento de la aplicación sea peor, para ciertas topologías, que cuando se utiliza la distribución balanceada.

Para solucionar este problema, proponemos generar tantos dominios especiales como comunicaciones remotas tenga la máquina con otras máquinas. Cada dominio especial gestionará la comunicación remota con una determinada máquina. Este dominio especial se asignará a un procesador diferente dentro de la máquina. De esta manera, denotaremos por *Bdomains,* el número de dominios especiales requeridos en una máquina. Igual que para la distribución *U-1domains* esta distribución se realiza en dos fases:

1. **Fase 1: <u>Particionamiento a nivel de máquinas</u>**

   Representa el primer nivel de particionamiento y se encarga de balancear la carga de los datos entre las máquinas. Igual que en la propuesta 1, en esta fase se divide la malla de elementos en tantas partes como máquinas tenga el Grid. La carga de datos asignada a cada máquina es proporcional a la velocidad relativa de esa máquina con respecto a la máquina más potente.

2. **Fase 2: <u>Particionamiento a nivel de procesadores</u>**

   A diferencia de la distribución *U-1domains*, en esta fase se divide el dominio especial obtenido para cada máquina. Este particionamiento del dominio especial divide el dominio en *B* subdominios especiales. De esta forma, *B* denota el número de comunicaciones remotas que tiene la máquina con otras máquinas. Cada subgrafo representa una comunicación remota con un procesador diferente en otra máquina. Esto ocasiona que la cantidad de

dominios especiales puede ser diferente para cada máquina. El resto de los nodos en el subgrafo se divide en *nprocs-B* partes proporcionales a las velocidades de cada procesador. La cantidad total de *nprocs-B* dominios es llamada *Cdomains*, ya que representan los dominios computacionales. Entonces, el número total de dominios en cada máquina es igual a la suma de la cantidad de dominios *Bdomains* y *Cdomains*. La numeración de los nodos para esta distribución es la misma forma que para la distribución *U-1domains*. Primero, se numeran los nodos internos de todos los dominios, luego, se numeran los nodos frontera de cada dominio computacional. Por último, se numeran los nodos frontera de los dominios especiales.

La figura 4.16 muestra el particionamiento a nivel de máquinas (fase 1) para esta propuesta, considerando una malla de 2D con 256 grados de libertad y un entorno Grid compuesto por cuatro máquinas y ocho procesadores por máquina.

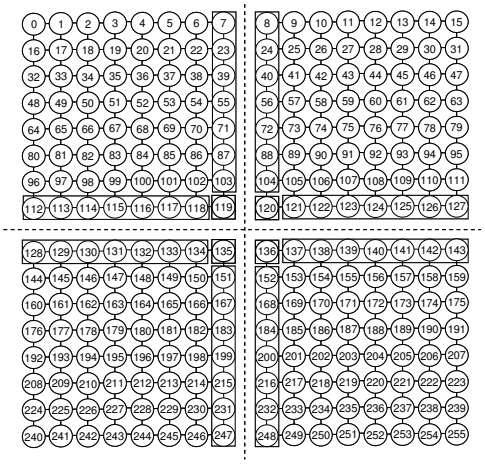


Fig. 4.16 Nodos frontera por máquina

La figura 4.17 ilustra el particionamiento a nivel de procesadores. En cada máquina, se identifican los dominios especiales. Cada dominio especial es asignado a un procesador. El resto de los procesadores disponibles en la máquina determina la cantidad de partes en las que se dividirá el subgrafo para obtener los *Cdomains*.

Por ejemplo, supongamos que los dominios D_0 a D_7 en la figura 4.17 son asignados a la máquina 0. Esta máquina tendrá tres dominios especiales (*B*=3: D_5, D_6 y D_7); y cinco dominios computacionales (D_0, D_1, D_2, D_3 y D_4). Las figuras 4.18 y 4.19 muestran respectivamente, el particionamiento de los datos para este ejemplo, utilizando las estrategias de distribución balanceada y *U-1domains*.

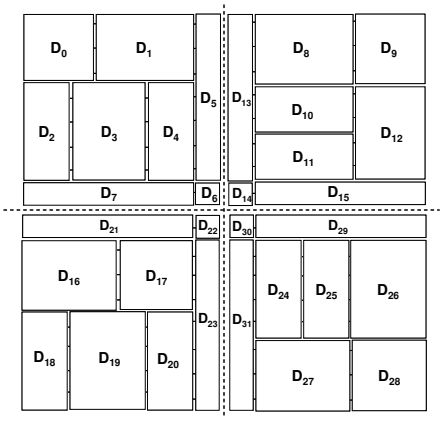


Fig. 4.17 Distribución *U-Bdomains*: partición en cajas

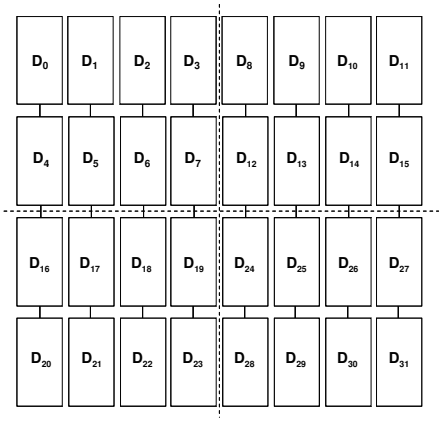


Fig. 4.18 Distribución balanceada: partición en cajas

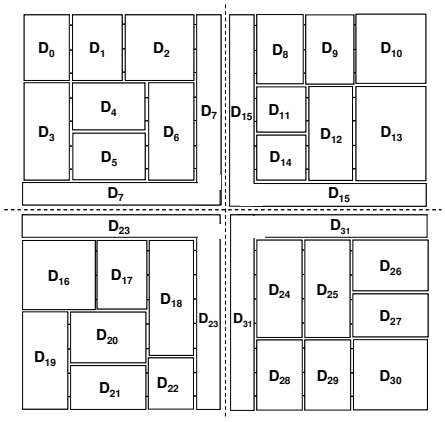


Fig. 4.19 Distribución *U-1domains*: partición en cajas

Las figuras 4.20, 4.21 y 4.22 muestran, respectivamente, los patrones de comunicación para la distribución balanceada y las distribuciones no balanceadas: *U-1domains* y *U-Bdomains*. Para cada distribución el número de vecinos es diferente.

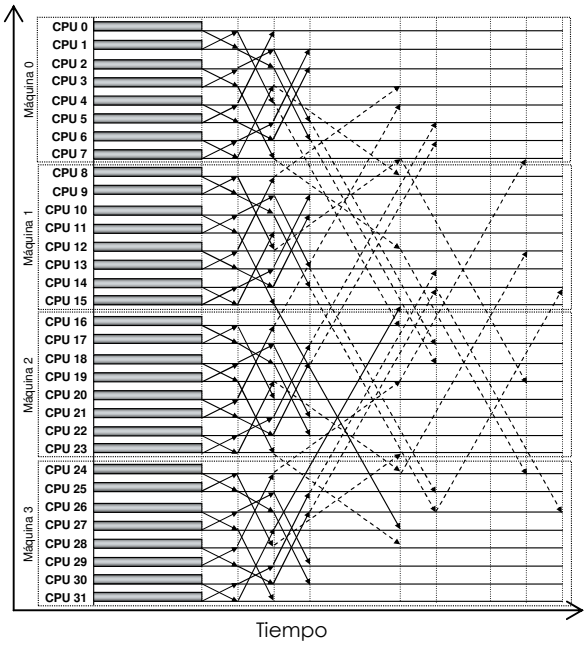


Fig. 4.20 Distribución balanceada: Diagrama de comunicaciones (4 máquinas y 8 procesadores por máquina)

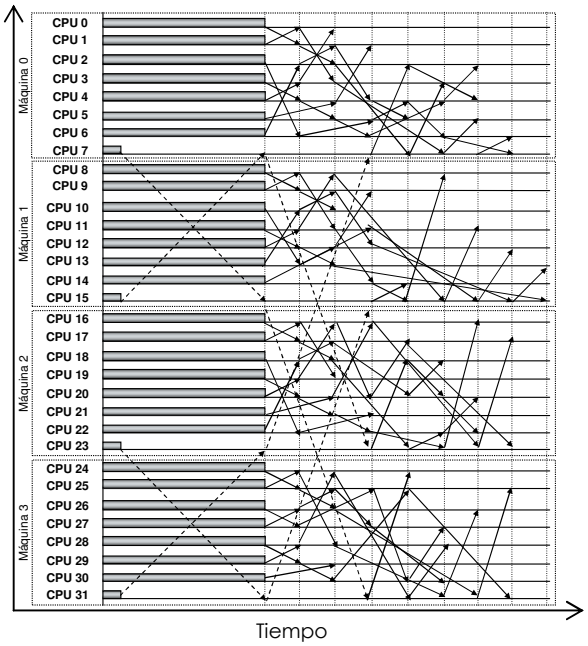


Fig. 4.21 Distribución *U-1domains*: Diagrama de comunicaciones (4 máquinas y 8 procesadores por máquina)

En la distribución balanceada (figura 4.20) todos los procesadores están ocupados y las comunicaciones remotas se realizan al final de la iteración. Sin embargo, en las distribuciones no balanceadas (figuras 4.21 y 4.22), las comunicaciones remotas se solapan con el cálculo.

A pesar de que las distribuciones no balanceadas solapan cálculo con comunicación, observamos que la distribución *U-1domains* únicamente realiza el solapamiento de comunicación y cálculo para la primera comunicación remota (figura 4.21). Sin embargo, la distribución *U-Bdomains* permite que todas las comunicaciones remotas se puedan solapar con cálculo (figura 4.22).

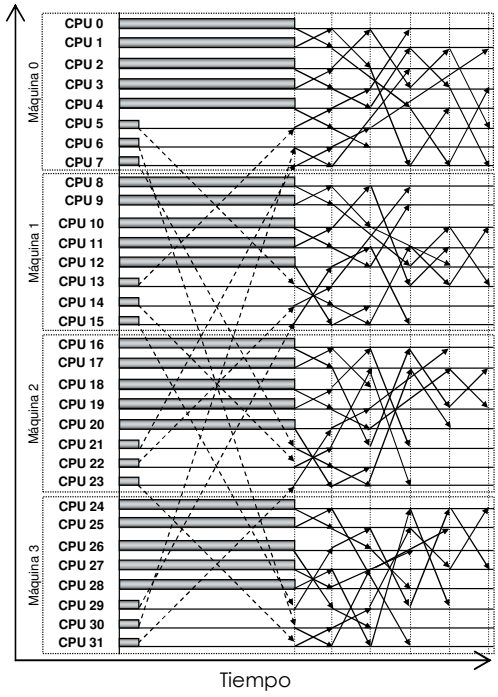


Fig. 4.22 Distribución *U-Bdomains*: Diagrama de comunicaciones (4 máquinas y 8 procesadores por máquina)

Las tablas 4.3 y 4.4 muestran el número total de pasos de comunicación que se necesitan para completar una iteración del método utilizando diferentes configuraciones y tipos de particiones. Estos resultados parecen indicar que la distribución *U-Bdomains* contribuirá a mejorar más el rendimiento de las simulaciones. Sin embargo, y como lo veremos a continuación, la escalabilidad de

ésta distribución es moderada, ya que es necesario utilizar varios procesadores para gestionar las comunicaciones remotas en una máquina.

Tabla 4.3. Número de pasos de comunicación de una iteración del método explícito: Distribuciones balanceada, *U-1domains* y *U-Bdomains* (partición en tiras)

	Partición en tiras					
Máquinas x CPUs	**Distribución balanceada**		**Distribución *U-1domains***		**Distribución *U-Bdomains***	
	Comunicaciones		**Comunicaciones**		**Comunicaciones**	
	remotas	**locales**	**remotas**	**locales**	**remotas**	**locales**
2x4	1	1	1	1	1	1
2x8	1	1	1	1	1	1
2x16	1	1	1	1	1	1
2x32	1	1	1	1	1	1
2x64	1	1	1	1	1	1
4x4	1	1	2	2	1	3
4x8	1	1	2	2	1	3
4x16	1	1	2	2	1	3
4x32	1	1	2	2	1	3
8x8	1	1	2	2	1	3
8x16	1	1	2	2	1	3

Tabla 4.4. Número de pasos de comunicación de una iteración del método explícito: Distribuciones balanceada, *U-1domains* y *U-Bdomains* (partición en cajas)

	Partición en cajas					
Máquinas x CPUs	**Distribución balanceada**		**Distribución *U-1domains***		**Distribución *U-Bdomains***	
	Comunicaciones		**Comunicaciones**		**Comunicaciones**	
	remotas	**locales**	**remotas**	**locales**	**remotas**	**locales**
2x4	2	3	1	3	1	3
2x8	4	5	1	6	1	6
2x16	5	8	1	7	1	8
2x32	6	7	1	15	1	14
2x64	7	8	1	25	1	24
4x8	7	5	3	6	4	6
4x16	10	9	3	11	4	9
4x32	9	8	3	22	4	14
8x8	13	5	6	7	13	7
8x16	13	4	6	13	13	11

4.4.3 Propuesta 3: Distribución *U-CBdomains*

En la segunda propuesta, cada dominio *Bdomains* es asignado a un procesador en la máquina. Sin embargo, estos procesadores permanecen inactivos una vez que terminan de realizar las comunicaciones remotas, debido a la escasa carga computacional que tienen asignada.

Esta ineficiencia podría ser resuelta asignando todos los dominios especiales (*Bdomains*) al mismo procesador en la máquina. De esta manera, cada

comunicación remota se ejecutaría concurrentemente en la máquina.

Igual que en las propuestas anteriores, el particionamiento de los datos para esta propuesta se realiza en dos fases. La primera fase, es similar a las propuestas anteriores. La segunda fase, divide el subgrafo de nodos frontera, obtenido en la primera fase, en tantas partes como comunicaciones remotas tenga la máquina con diferentes máquinas. Sin embargo, a diferencia de la propuesta anterior, en este caso, todos los dominios especiales son asignados a un solo procesador en la máquina. Además, la partición restante de los datos en el subgrafo, es descompuesta en *nprocs-1* dominios. La cardinalidad de cada dominio de datos es proporcional a la velocidad relativa que tenga el procesador en la máquina con respecto al procesador de mayor velocidad. La numeración de los nodos se realiza igual que en las propuestas anteriores.

La figura 4.23 ilustra el particionamiento de los datos para la distribución *U-CBdomains*, considerando el grafo asociado a la malla de elementos de la figura 4.16 y una topología compuesta de cuatro máquinas y ocho procesadores por máquina. La figura 4.24 muestra el patrón de comunicaciones de la distribución *U-CBdomains* para este ejemplo.

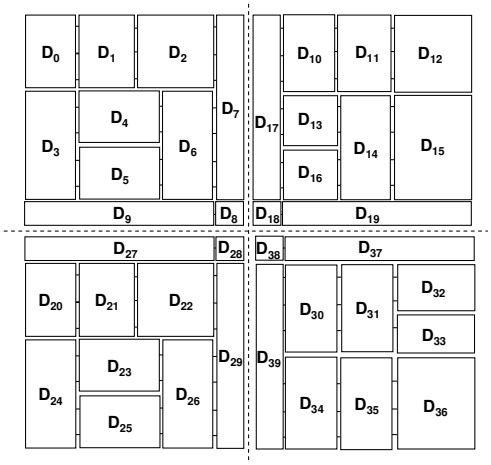


Fig. 4.23 Distribución *U-CBdomains*

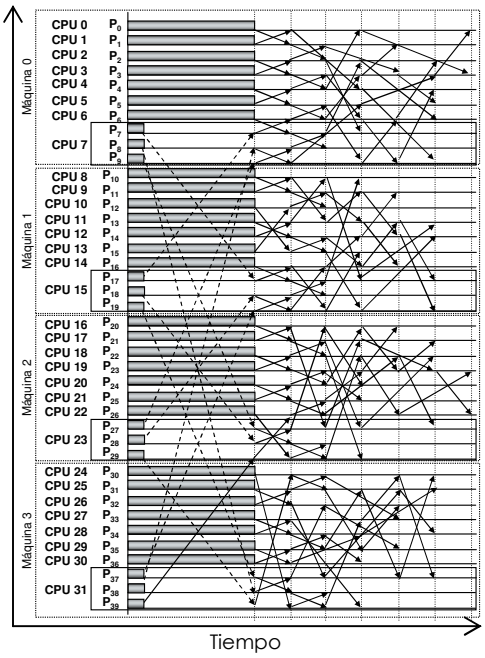


Fig. 4.24 Distribución *U-CBdomains*: Diagrama de comunicaciones (4 máquinas y 8 procesadores por máquina)

En este caso, más de un proceso es asignado a un procesador. Por ejemplo, en la figura 4.24 el procesador denotado por CPU 31 en la máquina 3 tiene asignado los procesos P_{37}, P_{38} y P_{39}. En este caso, los procesos se ejecutan concurrentemente dentro de este procesador.

Sin embargo la aplicabilidad real de esta propuesta requiere librerías de MPI *multithreading*. Actualmente, no existen versiones de MPI que permitan esto.

4.5 Resultados

En esta sección comentamos las características de los experimentos realizados y los resultados obtenidos para cada caso. Inicialmente, describimos las mallas utilizadas y el ambiente de simulación. Luego, mostramos los resultados obtenidos para cada propuesta de distribución de datos.

4.5.1 Tamaño de las mallas de elementos

En nuestras simulaciones las mallas tienen 1.000.000 de elementos. Esta cantidad de elementos, es muy común en las simulaciones del estampado de chapa y en

simulaciones que estudian el efecto que producen las colisiones de coches sobre las estructuras. Sin embargo, la cantidad de elementos de las mallas que caracterizan estas aplicaciones aumenta debido a la complejidad de los modelos. Hoy en día, los problemas industriales del estampado de chapa y las colisiones de coches usan mallas que tienen entre $5x10^5$ y $20x10^6$ elementos.

4.5.2 Ambiente de simulación

Los experimentos realizados para evaluar el rendimiento de las aplicaciones utilizando las diferentes distribuciones de datos (balanceada, *U-1domains, U-Bdomains* y *U-CBdomains*) consideran un ambiente donde:

- El tiempo de contención de los recursos es despreciable, ya que consideramos que todos los recursos están disponibles.
- El número de enlaces en cada máquina es equivalente al número de comunicaciones remotas en la misma máquina con diferentes máquinas.
- El tiempo de contención de la WAN utiliza un modelo para estimar el tráfico en la red. Consideramos una función de tráfico con un 1% de influencia para el tráfico interno y un 99% de influencia del tráfico externo.
- Conexiones *full-dúplex* entre máquinas.
- Los valores del ancho de banda y de la latencia fueron definidos y justificados en el capítulo 3 (tabla 3.3).

4.5.3 Topologías

Para realizar nuestras simulaciones necesitamos generar una traza que describa el comportamiento de la aplicación. Esta traza es obtenida ejecutando la aplicación en una máquina paralela. En nuestro caso, la máquina paralela disponible, para realizar los experimentos, tenía 128 procesadores. De esta manera, la cantidad total de procesadores del entorno Grid debe ser menor o igual a 128.

Para definir la topología del entorno Grid, es decir la cantidad de máquinas y el número de procesadores en cada máquina es necesario considerar dos factores.

El primero de ellos, está relacionado con el número mínimo de procesadores requerido en cada máquina para realizar las comunicaciones remotas con otras máquinas. Por ejemplo, para particiones en tiras, este número es dos. Sin embargo, para particiones en cajas, este número no es fácil de determinar.

El segundo factor a considerar, es el número de procesadores que realizan cálculo. Este número no debe ser inferior al 50% del total de procesadores de la máquina, ya que sino la eficiencia paralela se reduciría a la mitad.

De esta forma, si consideramos como mínimo dos procesadores para realizar las comunicaciones remotas y dos procesadores para realizar el cálculo, la cantidad total mínima de procesadores por máquina sería cuatro.

Además, la cantidad de procesadores en cada máquina, aproximadamente, no debe ser inferior al número total de máquinas que definen el entorno Grid, ya que es posible que una máquina comunique con el resto de máquinas del entorno.

La tabla 4.5 define las topologías de nuestros experimentos considerando todas las restricciones anteriores. El modelo general de interconexión para estas configuraciones se ilustra en la figura 4.25.

Tabla 4.5. Número de máquinas y procesadores por máquina

N° Máquinas	N° de procesadores por máquina
2	4, 8, 16, 32, 64
4	4, 8, 16, 32
8	8, 16

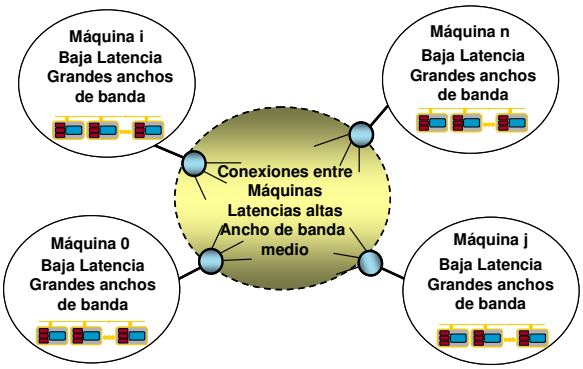


Fig. 4.25 Modelo general de interconexión entre máquinas

Además, en nuestras simulaciones:

- cada máquina esta formada por un conjunto de procesadores,
- para las distribuciones balanceada, *U-1domains* y *U-Bdomains*, cada dominio de datos es asignado a un procesador,
- en la distribución *U-CBdomains*, todos los dominios especiales son asignados a un solo procesador dentro de la máquina,
- para las distribuciones no balanceadas, las comunicaciones remotas las realizan los procesadores que tienen asignados dominios especiales. En cada comunicación remota el tamaño de los mensajes es igual a la cantidad de nodos que definen los dominios especiales (alrededor de 100 nodos),
- para la distribución balanceada, las comunicaciones remotas la realizan los procesadores que tienen particiones de datos que contienen nodos frontera,
- la métrica utilizada para comparar el rendimiento de las distribuciones no balanceadas es el porcentaje de reducción de los tiempos de ejecución con respecto a la distribución balanceada.
- para particiones en forma de caja existen algunas configuraciones que no se pueden estudiar, ya que al realizar el particionamiento de los datos la cantidad de procesadores con dominios especiales supera la

cantidad de procesadores que realizan cálculo en esa máquina. Ejemplos de estos casos, son las configuraciones 2x4 y 4x4 para la distribución *U-Bdomains*.

4.5.4 Resultados: Distribución *U-1domains*

A continuación mostramos los resultados obtenidos para diferentes configuraciones del entorno Grid para la distribución *U-1domains*.

Las figuras 4.26 a 4.28 muestran los porcentajes de reducción del tiempo de ejecución de las simulaciones considerando particiones en tiras, mientras que las figuras 4.29 a 4.31 muestran los resultados para particiones en caja.

Para particiones en tiras, topologías formadas por dos máquinas y mas de cuatro procesadores, la distribución *U-1domains,* (figura 4.26 y figuras A.2 a A.4 del anexo A), toma ventaja sobre la distribución balanceada, ya que el número de comunicaciones remotas puede solaparse perfectamente con el cálculo y este número de comunicaciones no supera la cantidad total de comunicaciones remotas que necesita realizar la distribución balanceada (ver tabla 4.1).

Sin embargo, para la configuración de dos máquinas y cuatro procesadores, esta distribución no mejora los porcentajes de reducción del tiempo de ejecución con respecto a los obtenidos por la distribución balanceada. Esto se debe fundamentalmente a que en la distribución *U-1domains* únicamente realiza cálculo el 75% de los procesadores.

En el resto de configuraciones con mas de dos máquinas, (figuras 4.27 a 4.28 y A.6 a A.11) no se mejora significativamente el rendimiento de la aplicación utilizando esta distribución, ya que en estos casos la distribución *U-1domains* tiene que realizar en cada iteración dos comunicaciones remotas (tabla 4.1), de las cuales una sola de ellas se solapa con cálculo. Mientras, la distribución balanceada únicamente debe realizar una comunicación remota. De esta manera, en estos casos empeoramos el tiempo de ejecución para la distribución *U-1domains*.

Por otra parte, para particiones tipo caja y configuraciones del entorno Grid formadas por dos máquinas (figuras 4.29 y A.12 a A.15), el porcentaje de reducción es enorme, ya que el número de comunicaciones remotas para la distribución *U-1domains* es menor que las obtenidas al utilizar la distribución balanceada (tabla 4.2). Particularmente, para la configuración de dos máquinas y 64 procesadores por máquina, la cantidad de comunicaciones remotas se reducen en un 86%.

Para particiones tipo caja y topologías compuestas de 4 y 8 máquinas (figuras 4.30, 4.31 y A.16 a A.21), la distribución *U-1domains* reduce el tiempo de ejecución de las simulaciones entre 2% y 65%. Esto se debe a que la cantidad de comunicaciones remotas para esta distribución disminuye hasta un 50% en comparación con la distribución balanceada (ver tabla 4.2).

En general, podemos observar que en la medida en que aumenta la relación entre el tiempo de comunicación y el tiempo de cálculo, el porcentaje de reducción del tiempo de ejecución aumenta.

Además, en la medida en que aumenta la cantidad de procesadores en una máquina el porcentaje de reducción también aumenta. Esto se debe principalmente, a que se dedican más procesadores a realizar cálculo.

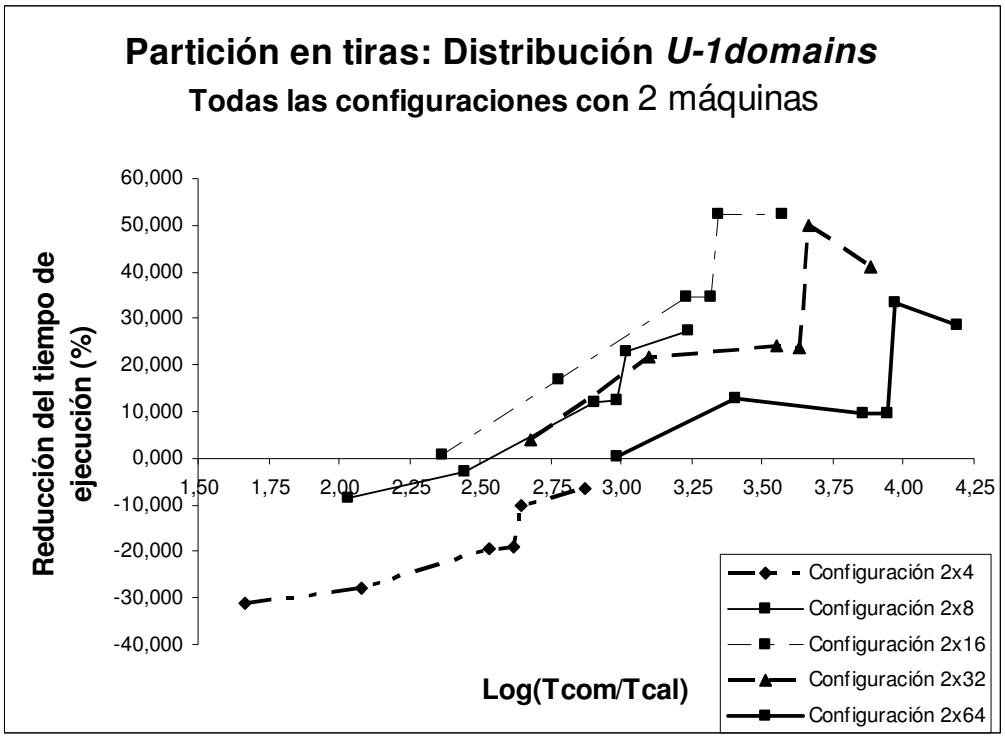


Fig. 4.26 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en tiras (todas las configuraciones con 2 máquinas)

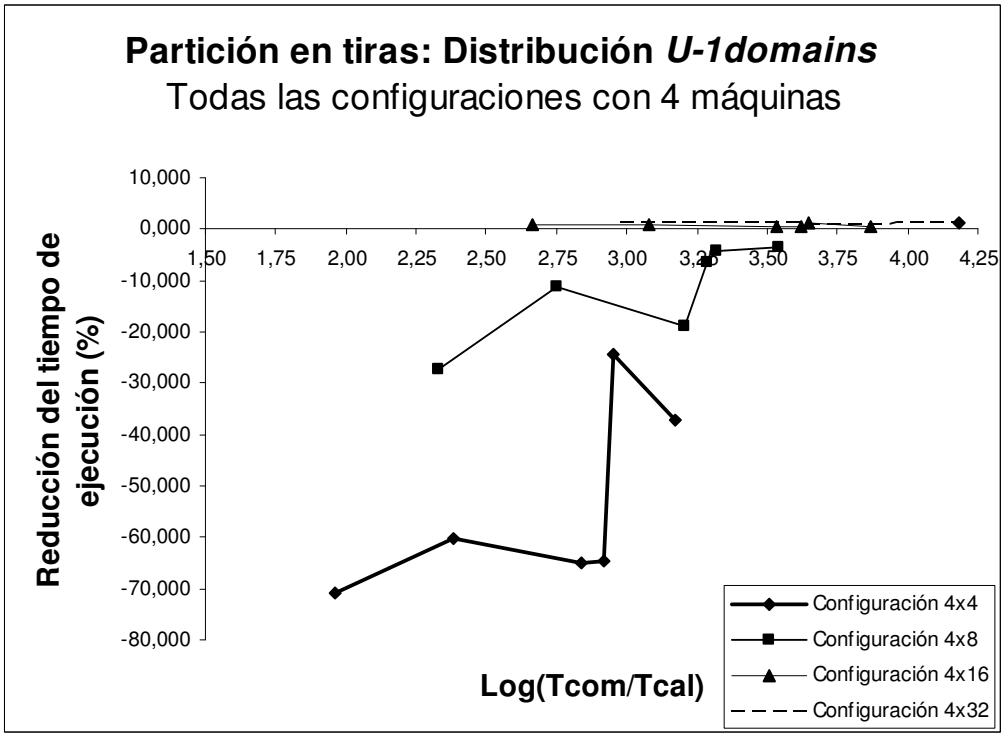


Fig. 4.27 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en tiras (todas las configuraciones con 4 máquinas)

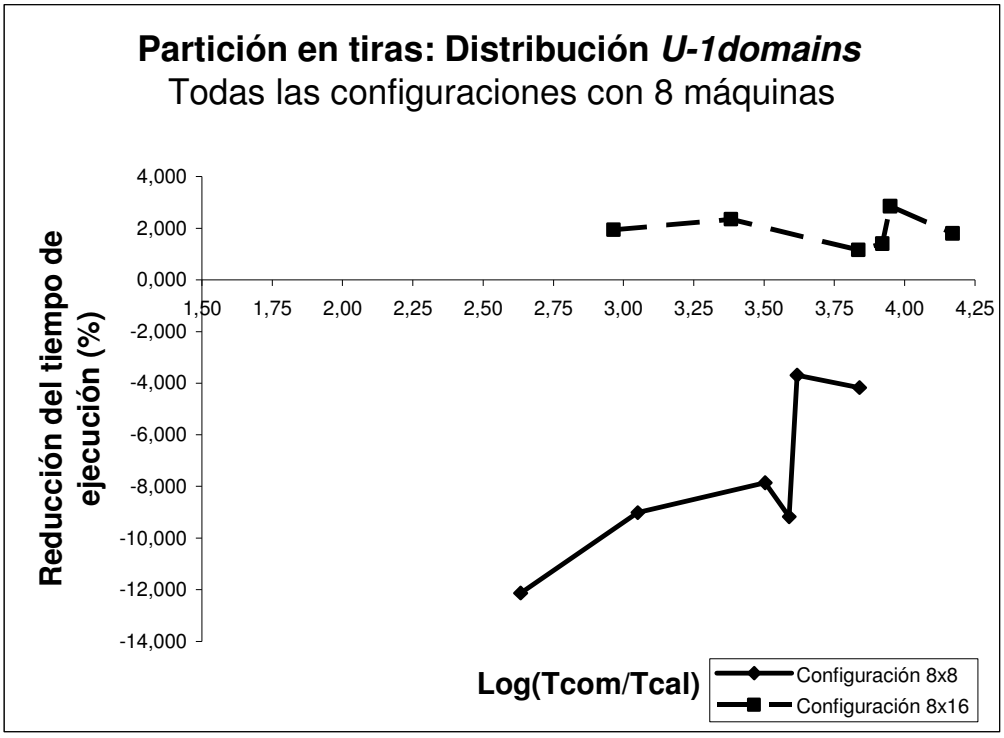


Fig. 4.28 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en tiras (todas las configuraciones con 8 máquinas)

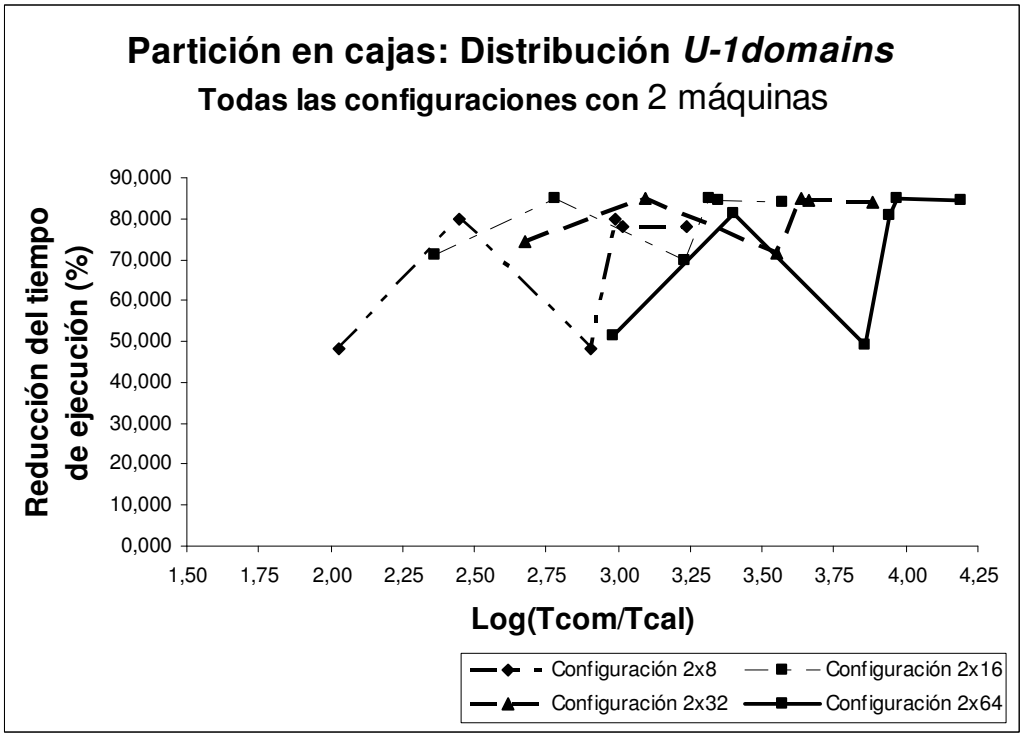


Fig. 4.29 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en cajas (todas las configuraciones con 2 máquinas)

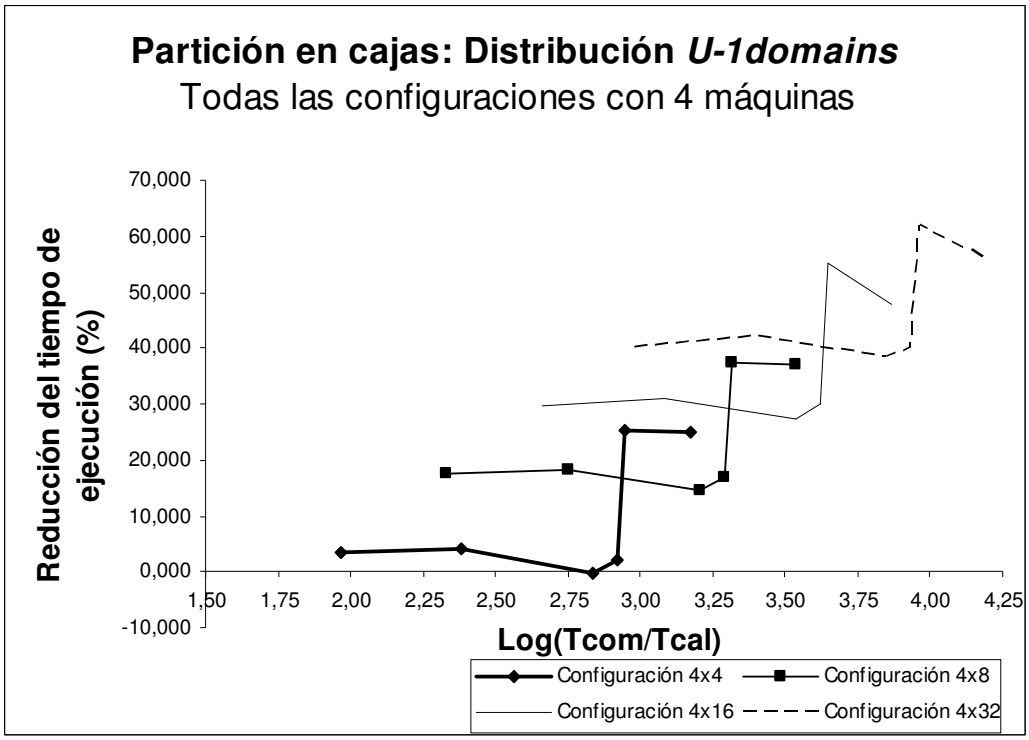


Fig. 4.30 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en cajas (todas las configuraciones con 4 máquinas)

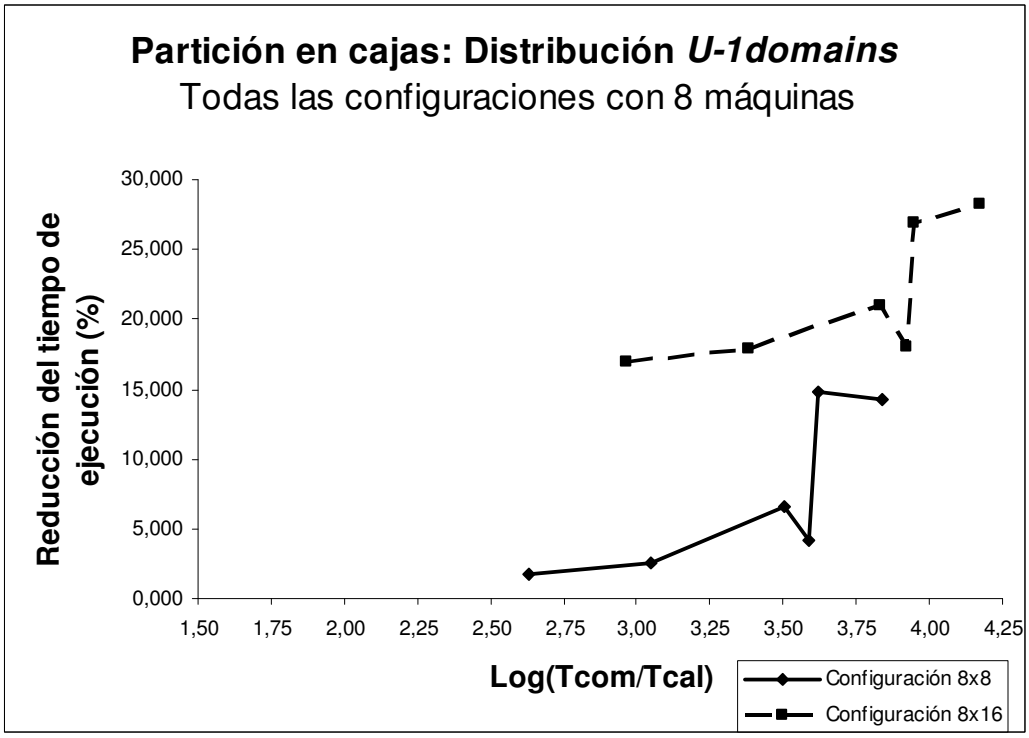


Fig. 4.31 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en cajas (todas las configuraciones con 8 máquinas)

4.5.5 Resultados: Distribución *U-Bdomains*

A continuación mostramos los resultados obtenidos para diferentes configuraciones del entorno Grid utilizando la distribución *U-Bdomains*. Las figuras 4.32 a 4.34 muestran los porcentajes de reducción del tiempo de ejecución de las aplicaciones, considerando particiones en forma de tiras, mientras que las figuras 4.35 a 4.37 muestran los resultados obtenidos para las particiones en forma de caja.

Igual que en la distribución *U-1domains*, la distribución *U-Bdomains,* para topologías compuestas de dos máquinas (figuras 4.32, A.22 a A.26, 4.35 y A.33 a A.36), toma ventaja sobre la distribución balanceada independientemente del tipo de partición. En estos casos, el número de comunicaciones remotas puede solaparse perfectamente con el cálculo y este número de comunicaciones es inferior a las que debe realizar la distribución balanceada. Para ambas distribuciones, *U-1domains* y *U-Bdomains*, el número total de pasos requeridos para finalizar las comunicaciones de una iteración el mismo (ver tablas 4.3 y 4.4).

Sin embargo, a diferencia del caso anterior para particiones en forma de tiras, y topologías compuestas por más de dos máquinas (figuras 4.33, 4.34 y A.27 a A.32), la distribución *U-Bdomains* aumenta el porcentaje de reducción de los tiempos de ejecución de las simulaciones. Esto se debe principalmente a que la cantidad de comunicaciones remotas que pueden realizarse de forma simultánea aumenta (ver tabla 4.3). De esta forma, se pueden obtener porcentajes de reducción del tiempo ejecución de hasta el 80% donde antes había pérdidas (figuras 4.27 y 4.28). Sin embargo, si la cantidad de procesadores por máquina para estas configuraciones es menor a ocho, se produce una pérdida significativa en el porcentaje de reducción, debido a que, se utiliza aproximadamente el 50% de los procesadores para gestionar las comunicaciones remotas.

Para particiones en forma de cajas y topologías que tienen más de dos máquinas (figuras 4.36, 4.37 y A.37 a A.41), el comportamiento es muy similar al esperado para la distribución *U-1domains* y puede alcanzar el 60% de reducción

del tiempo de ejecución de las simulaciones. Sin embargo, para topologías con ocho máquinas, estos valores disminuyen, ya que la cantidad de procesadores que realizan cálculo puede superar el 50% del total de procesadores de la máquina. La tabla 4.6 muestra la cantidad total de dominios especiales por máquina. A pesar de esto, los porcentajes de reducción del tiempo de ejecución pueden llegar hasta el 15% (figuras 4.37 y A40 a A.41).

En general, para esta distribución se mejoran los porcentajes de reducción de los tiempos de ejecución cuando el valor de la relación entre el tiempo de comunicación y el tiempo de cálculo es superior a 3, independientemente de la partición y del tipo de configuración del entorno Grid.

Tabla 4.6. Cantidad máxima de dominios especiales por máquina

	Procesos *B-domains*	
Máquinas	**Malla barra**	**Malla caja**
2	1	1
4	1 ó 2	2 ó 3
8	1 ó 2	3, 5 ó 6

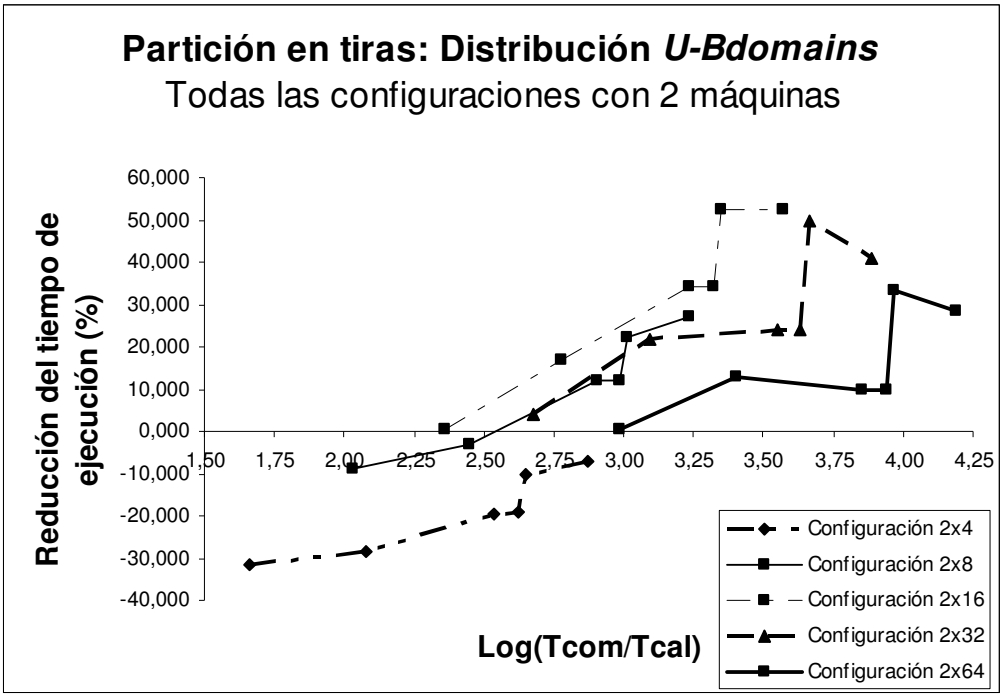


Fig. 4.32 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en tiras (todas las configuraciones con 2 máquinas)

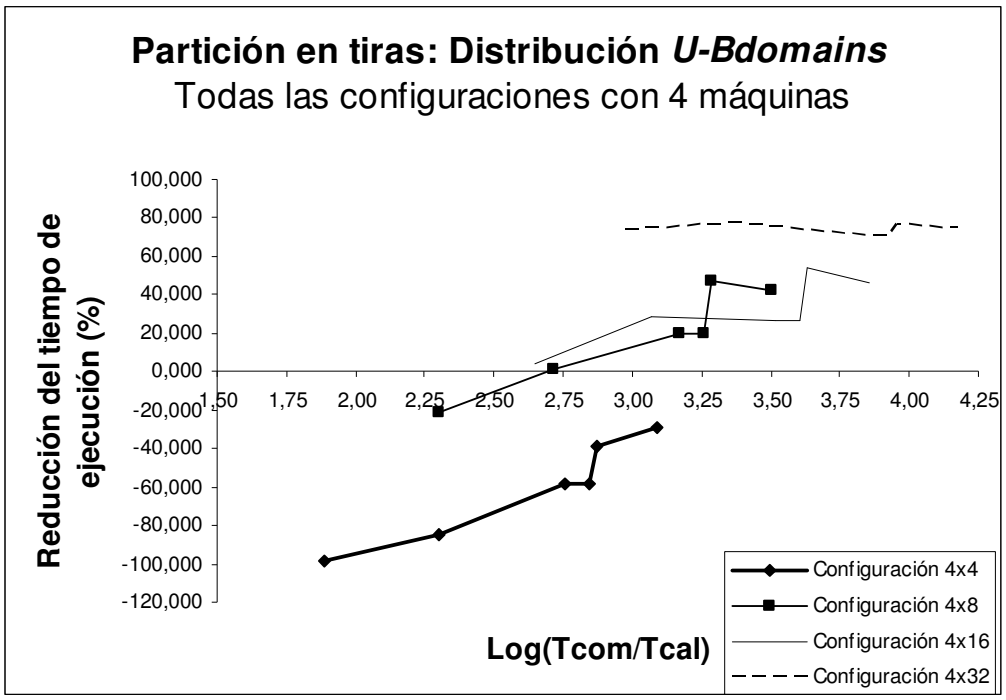


Fig. 4.33 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en tiras (todas las configuraciones con 4 máquinas)

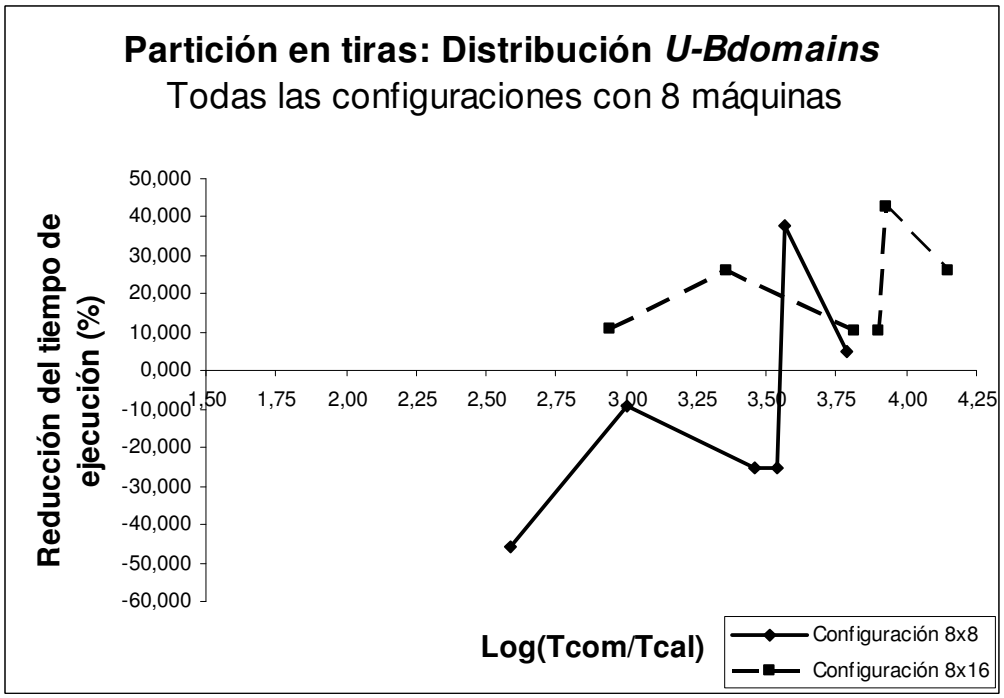


Fig. 4.34 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en tiras (todas las configuraciones con 8 máquinas)

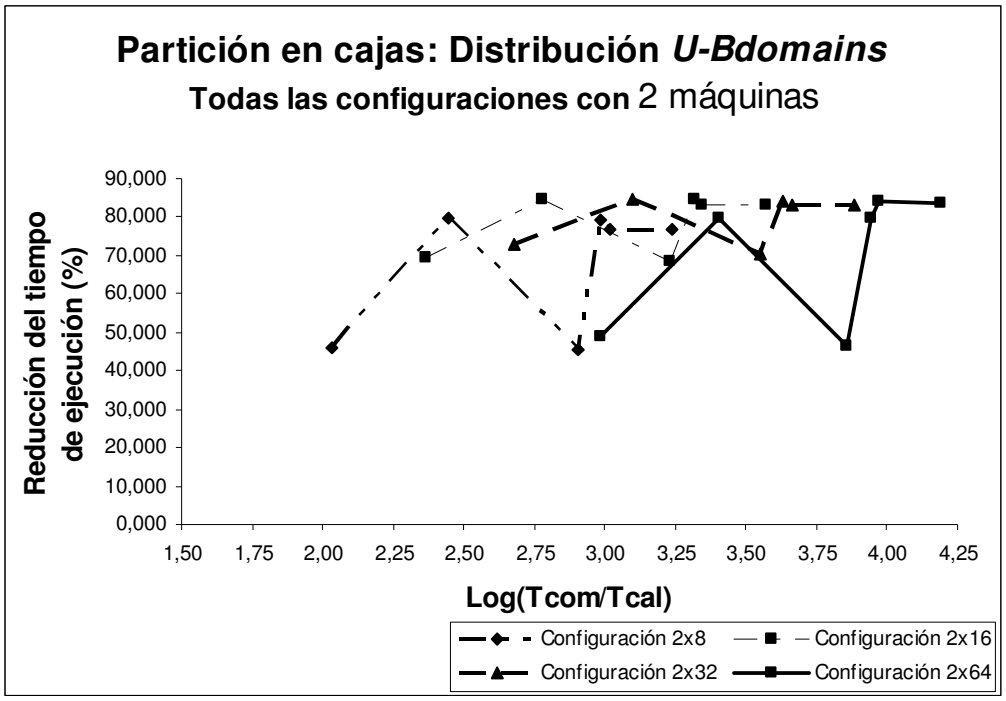


Fig. 4.35 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en cajas (todas las configuraciones con 2 máquinas)

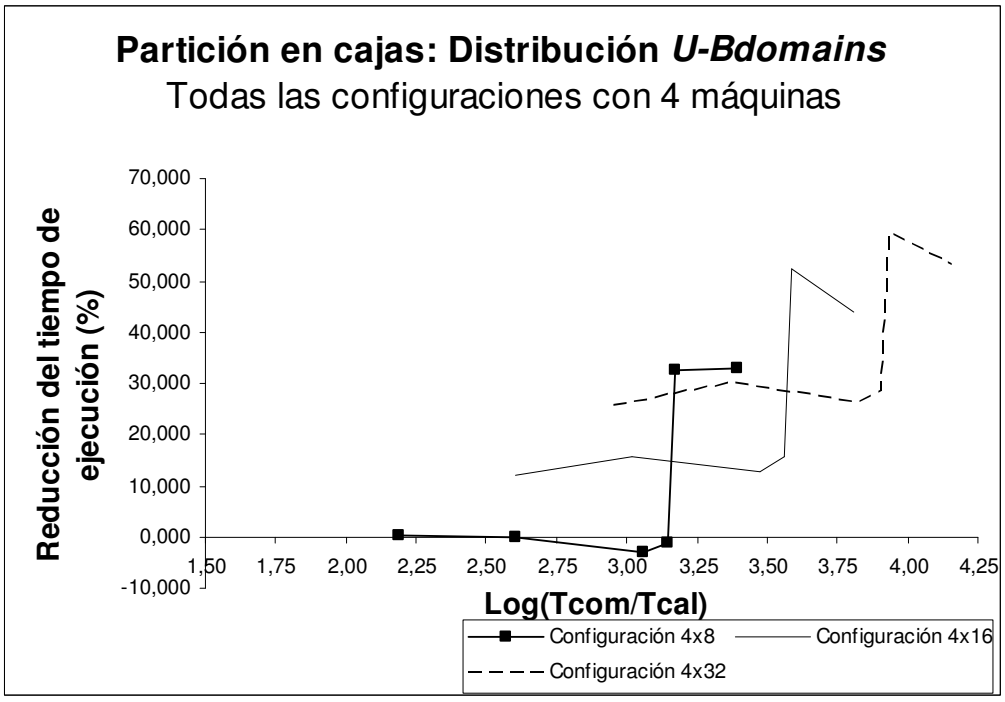


Fig. 4.36 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en cajas (todas las configuraciones con 4 máquinas)

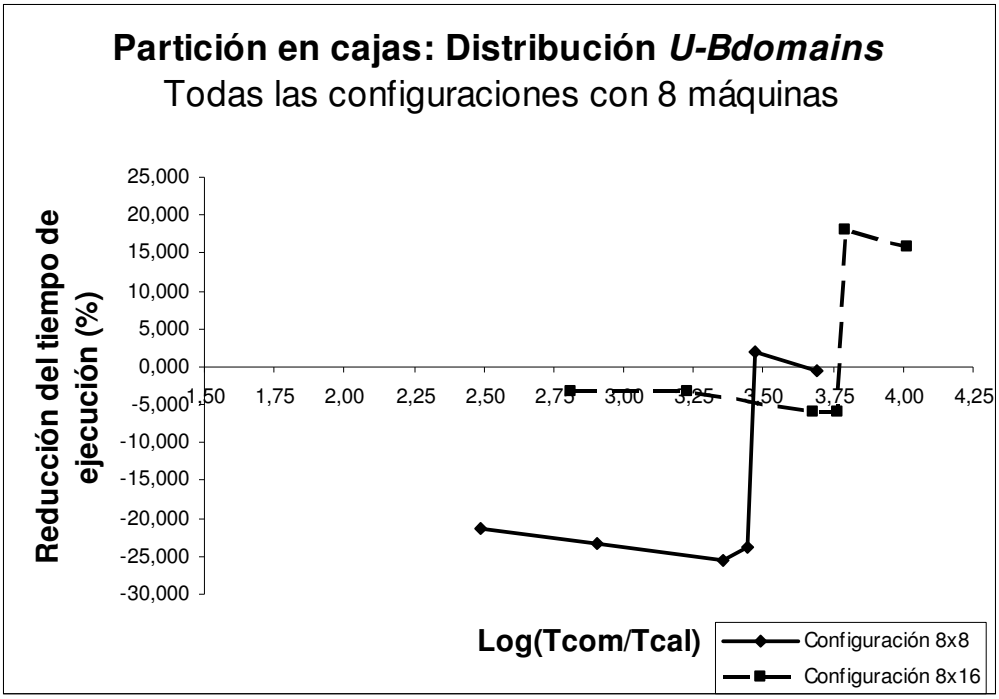


Fig. 4.37 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en cajas (todas las configuraciones con 8 máquinas)

4.5.6 Resultados: Distribución *U-CBdomains*

A continuación mostramos los resultados obtenidos para diferentes configuraciones del entorno Grid utilizando la distribución *U-Bdomains*. Las figuras 4.38 a 4.40 muestran los porcentajes de reducción del tiempo de ejecución de las aplicaciones, considerando particiones en tiras, mientras que las figuras 4.41 a 4.43 muestran los resultados obtenidos para las particiones en forma de caja.

Para topologías compuestas por dos máquinas, tanto para particiones en forma de tiras (figuras 4.38 y A.42 a A.46) como para particiones en forma de caja (figuras 4.41 y A.53 a A.56), los resultados son los mismos que los obtenidos para la distribución *U-Bdomains*. Esto se debe a que la cantidad de dominios especiales es uno, por lo implica que sólo se requiere un procesador por máquina para gestionar las comunicaciones remotas (ver tabla 4.6).

Sin embargo, cuando consideramos particiones en forma de tiras y topologías con más de dos máquinas (figuras 4.39, 4.40 y A.47 a A.52), la distribución *U-*

CBdomains hace que los porcentajes de reducción de los tiempos de ejecución pueden llegar a alcanzar el 72% de reducción (figuras 4.39 y A.50), cuando en la distribución U-Bdomains para este mismo caso se podían obtener porcentajes de hasta 78% (figuras 4.33 y A.30). En general, los porcentajes de reducción, para este caso, descienden hasta un 10%, si los comparamos con los obtenidos al utilizar la distribución *U-Bdomains*. Similarmente, para particiones en forma de caja (figuras 4.42, 4.43 y A.57 a A.61), estos porcentajes descienden hasta 30%.

Este empeoramiento en los porcentajes de reducción de los tiempos de ejecución, se debe principalmente a dos razones. La primera es que ahora las diferentes comunicaciones remotas son gestionadas concurrentemente por un solo procesador en la máquina. Y la segunda razón, es que la gestión concurrente de muchas de estas comunicaciones no es lo suficientemente eficaz.

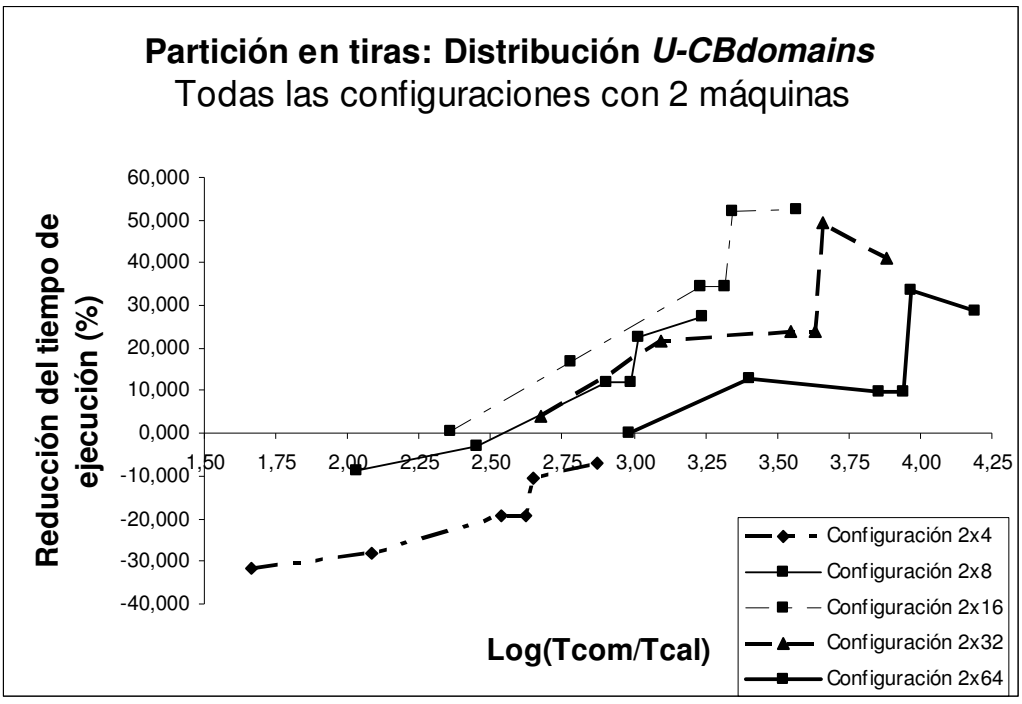


Fig. 4.38 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en tiras (todas las configuraciones con 2 máquinas)

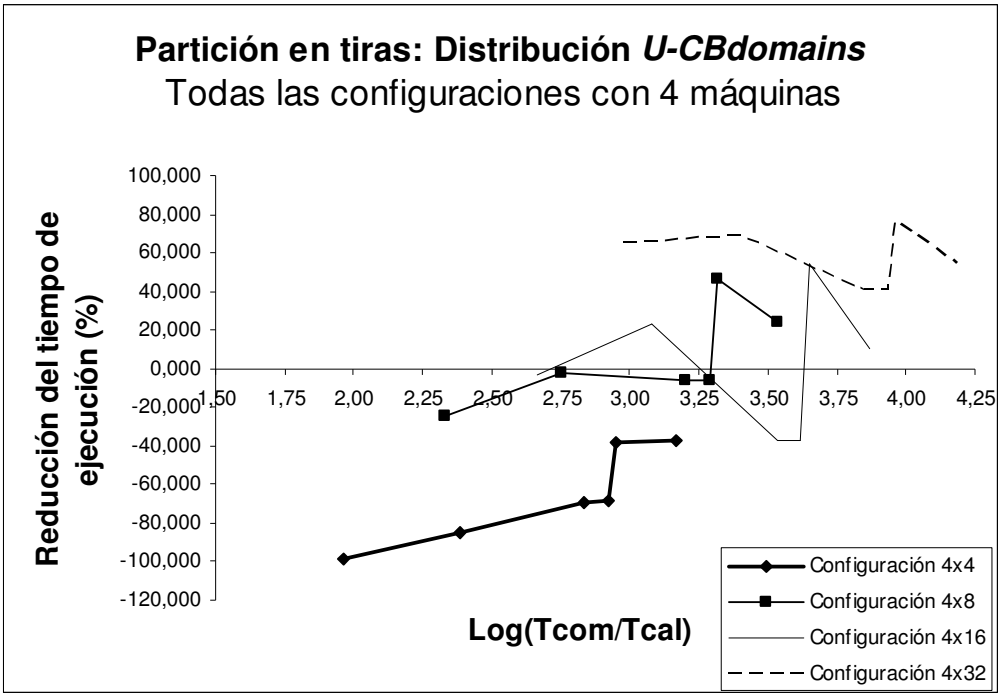


Fig. 4.39 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en tiras (todas las configuraciones con 4 máquinas)

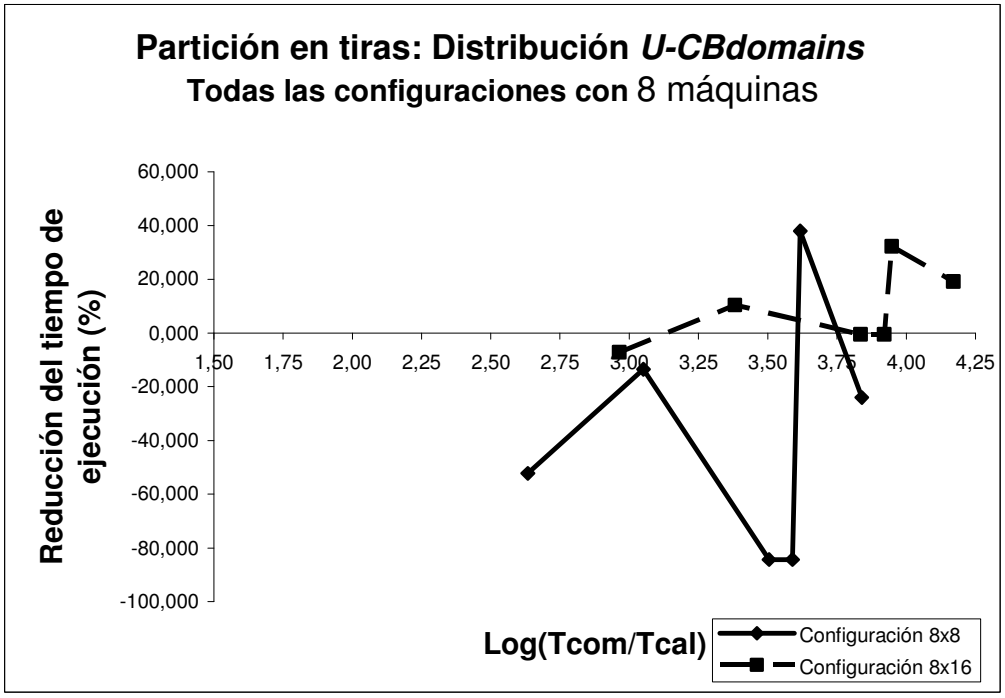


Fig. 4.40 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en tiras (todas las configuraciones con 8 máquinas)

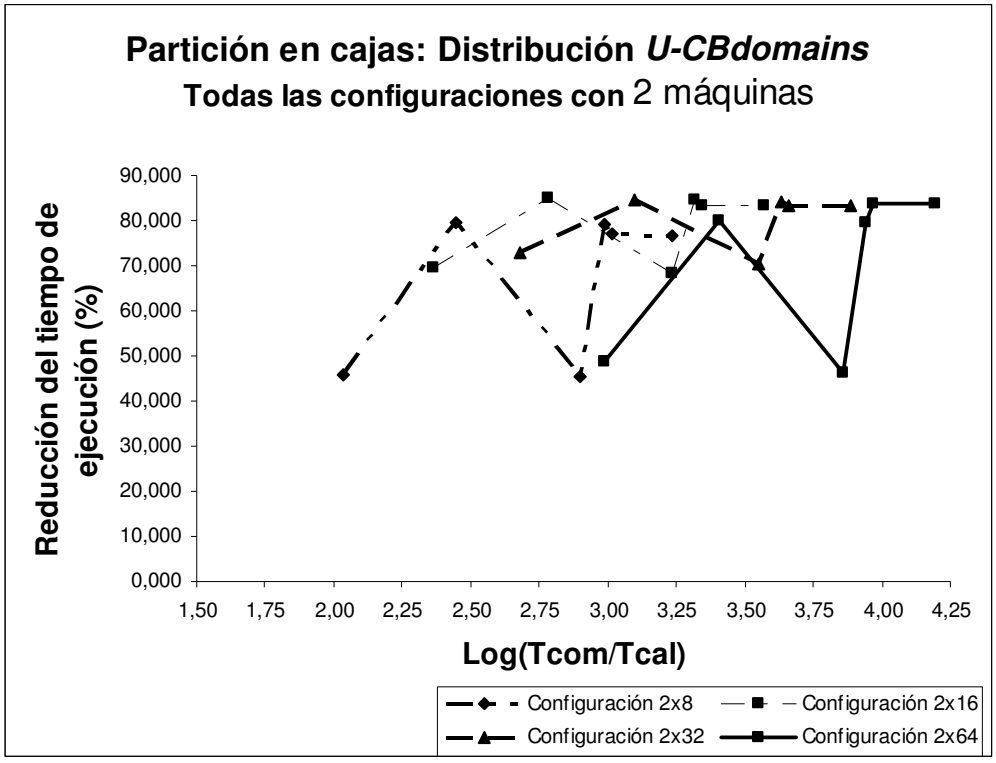


Fig. 4.41 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en cajas (todas las configuraciones con 2 máquinas)

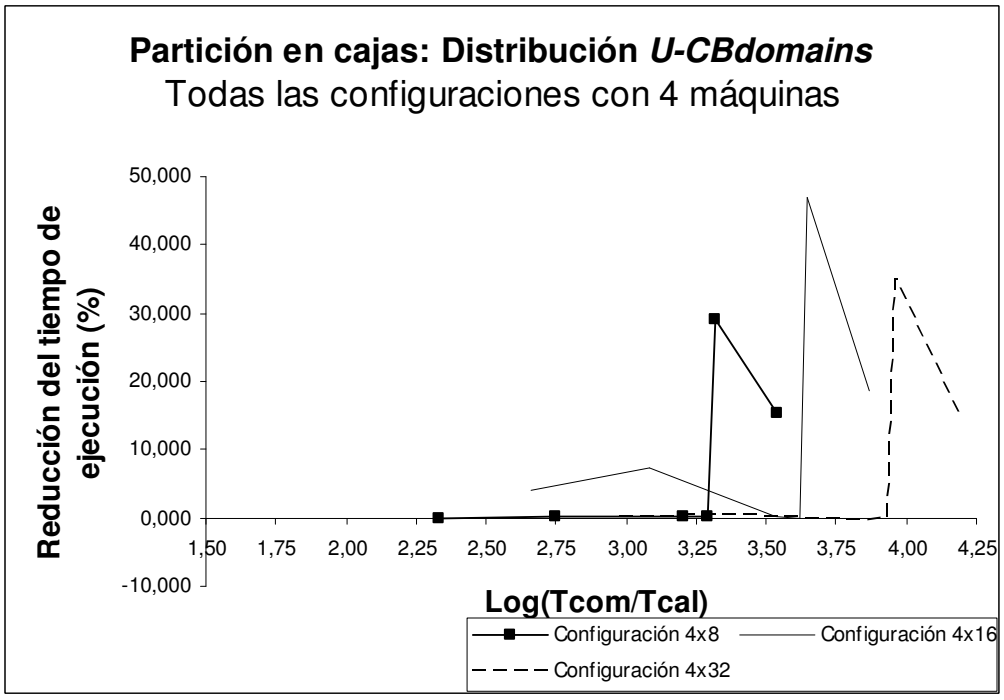


Fig. 4.42 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en cajas (todas las configuraciones con 4 máquinas)

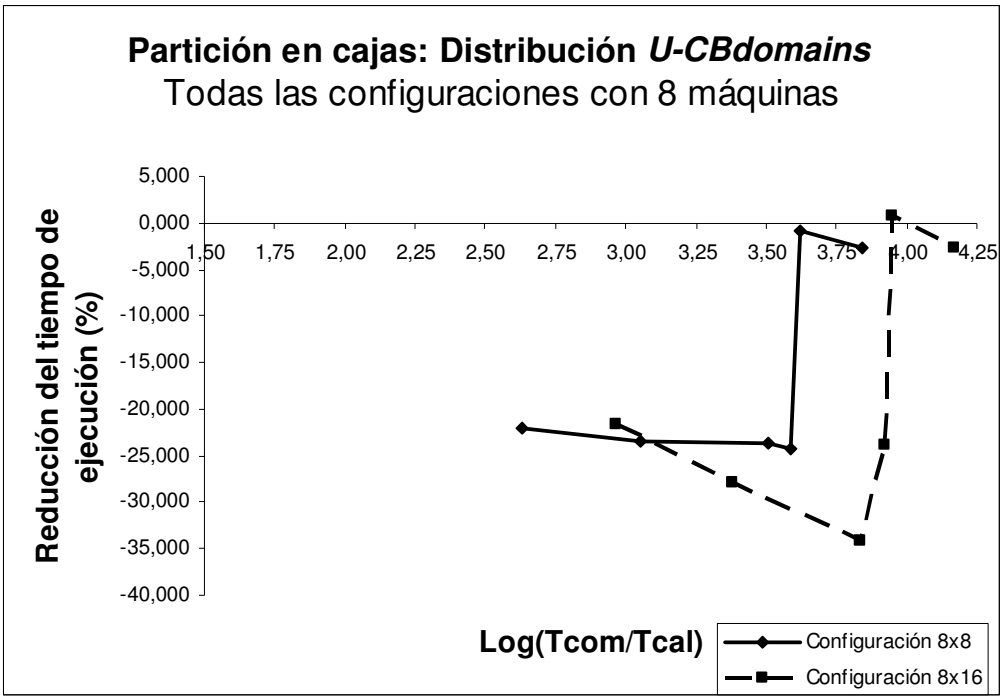


Fig. 4.43 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en cajas (todas las configuraciones con 8 máquinas)

En general, para obtener un buen rendimiento de las estrategias de distribución de los datos propuestas, es necesario que:

1. El número de procesadores por máquina sea mayor a ocho.

2. La relación entre el tiempo de comunicación y el tiempo de cálculo sea superior a 1000.

3. Para particiones en forma de tiras se utilice la distribución *U-Bdomains*, ya que mostró los mejores resultados para este tipo de partición.

4. Para particiones tipo caja se utilice la distribución *U-Bdomains*, siempre y cuando el número de procesadores utilizados para realizar la comunicación entre máquinas sea inferior al 50%. En el caso contrario, se debe utilizar la distribución *U-1domains*.

Conclusiones

El objetivo principal de la tesis es proponer estrategias de distribuciones de datos que permitan ejecutar simulaciones numéricas explícitas de elementos finitos con bajo coste y eficiencia moderada en entornos Grid.

Hasta ahora, la ejecución de estas simulaciones de elementos finitos estaba pensada para ser realizada sobre costosos supercomputadores. Por ello, el particionamiento de los datos se realizaba de forma balanceada. Sin embargo, debido a lo costoso que resulta ejecutar las simulaciones explícitas utilizando supercomputadores, pocas empresas pueden llegar a disponer de esta tecnología.

En este trabajo, proponemos reutilizar los equipos que tiene la pequeña empresa para ejecutar las simulaciones numéricas de aplicaciones industriales tales como: el estampado de chapa y las deformaciones producidas sobre la estructura de un coche cuando es sometido a fuerzas de impacto.

Sin embargo, si ejecutamos las simulaciones explícitas para estas aplicaciones en entornos Grid, sin realizar ninguna modificación en el código fuente, su rendimiento puede sufrir una degradación importante, debido a que ahora tenemos comunicaciones entre máquinas (remotas) que son mucho más lentas que las comunicaciones internas (locales) en una máquina. Para solventar este problema, planteamos diferentes estrategias de particionamiento no balanceado de los datos en los procesadores. Estas estrategias de particionamiento utilizan algunos procesadores para gestionar las comunicaciones lentas, a cambio de solapar, en el tiempo, dichas comunicaciones con el cálculo.

Nuestras propuestas demuestran que es posible paralelizar y ejecutar aplicaciones explícitas de elementos finitos sobre entornos Grid, con una eficiencia razonable.

Las estrategias de descomposición de los datos propuestas están basadas en dos niveles de descomposición en dominios. Inicialmente, se realiza un particionamiento balanceado de los datos entre diferentes máquinas. La cantidad de datos asignados a cada máquina es proporcional a la velocidad relativa de la máquina, con respecto, a la máquina que realiza más operaciones por unidad de

tiempo. En el segundo nivel de descomposición, para cada máquina se realiza el particionamiento no balanceado de los datos, de forma que, parte de los procesadores de la máquina están dedicados a gestionar las comunicaciones remotas. El resto de procesadores realiza cálculo. La asignación de la carga computacional para los procesadores en la máquina, es proporcional a la velocidad relativa que cada uno de ellos tenga, con respecto al procesador que realiza más operaciones por unidad de tiempo. De esta manera, las estrategias de distribución desarrolladas son: *U-1domains*, *U-Bdomains* y *U-CBdomains*.

En la distribución *U-1domains* se utiliza un solo procesador por máquina para gestionar las comunicaciones remotas, el resto de procesadores realizan cálculo. En la distribución *U-Bdomains* se dedican tantos procesadores a gestionar las comunicaciones remotas como dominios especiales se definan para esa máquina. En la distribución *U-CBdomains*, todos los dominios especiales asociados a una máquina son mapeados en una única CPU. La gestión de las comunicaciones remotas se realiza concurrentemente mediante *threads*.

El rendimiento de estas distribuciones es comparado con la clásica distribución balanceada y es aplicado a diferentes tipos de mallas: particiones en tiras y particiones en cajas. Los resultados muestran que:

- La cantidad de procesadores para cada máquina debe ser superior a ocho, ya que en caso contrario, la cantidad de procesadores por máquina disponibles para realizar cálculo no es suficiente para solapar completamente el cálculo con las comunicaciones remotas.
- La relación entre el tiempo de comunicación y el tiempo de cálculo debe ser mayor a 1000. En general, se obtienen reducciones de los porcentajes de ejecución cuando el costo de realizar las comunicaciones remotas es tres veces superior al tiempo invertido para realizar cálculo en la máquina.
- La estrategia de distribución *U-1domains* es la más apropiada cuando la partición de la malla es en forma de caja y la cantidad de procesadores que

se dedican a realizar las comunicaciones remotas, es superior al 50% del total de procesadores en la máquina.

- La distribución *U-Bdomains* es la más adecuada cuando:
    - la partición de la malla es en forma de tiras, ya que sólo se dedican como máximo dos procesadores a gestionar las comunicaciones remotas, mientras que el resto de procesadores en la máquina realizan cálculo. Esta estrategia disminuye la cantidad de pasos de comunicación que deben realizar los procesadores que gestionan las comunicaciones remotas en la máquina.
    - la partición de la malla es en forma de cajas, siempre que la cantidad de procesadores dedicados a realizar las comunicaciones remotas en la máquina no supere el 50% del total de procesadores en esa máquina.
- La distribución *U-CBdomains* parece ser competitiva cuando el número de procesadores por máquina es inferior a ocho, ya que puede llegar a mejorar el porcentaje de ocupación de los procesadores en la máquina de un 25% a un 75%. Sin embargo, en nuestros experimentos esta estrategia de distribución no mejora los porcentajes de reducción obtenidos por la distribución *U-Bdomains*, esto se debe principalmente a problemas en el simulador para realizar la gestión concurrente de las comunicaciones remotas.

Los trabajos futuros que se proponen a raíz de los resultados obtenidos en esta tesis son:

- Integrar otros parámetros al modelo de comunicación de Dimemas y estudiar la influencia que éstos ejercen en el rendimiento de las aplicaciones cuando son ejecutadas en entornos Grid.
- Rediseñar los ficheros de entrada que contienen trazas de la aplicación, para obtener ficheros de menor tamaño. Esta reducción en el tamaño del fichero de entrada permitirá trabajar con entornos Grid formados por más procesadores.

- Definir heurísticas que determinen la cantidad de elementos de las mallas que garantizan el solapamiento eficiente de las comunicaciones remotas con el cálculo, para una determinada configuración del Grid.
- Implementar un generador de mallas que permita obtener más de 1.000.000 de elementos.
- Evaluar el rendimiento de las estrategias de distribución en entornos Grid reales.

Anexo A

Rendimiento de las estrategias propuestas para diferentes configuraciones del Grid

A.1 Distribución *U-1domains*: Partición en tiras

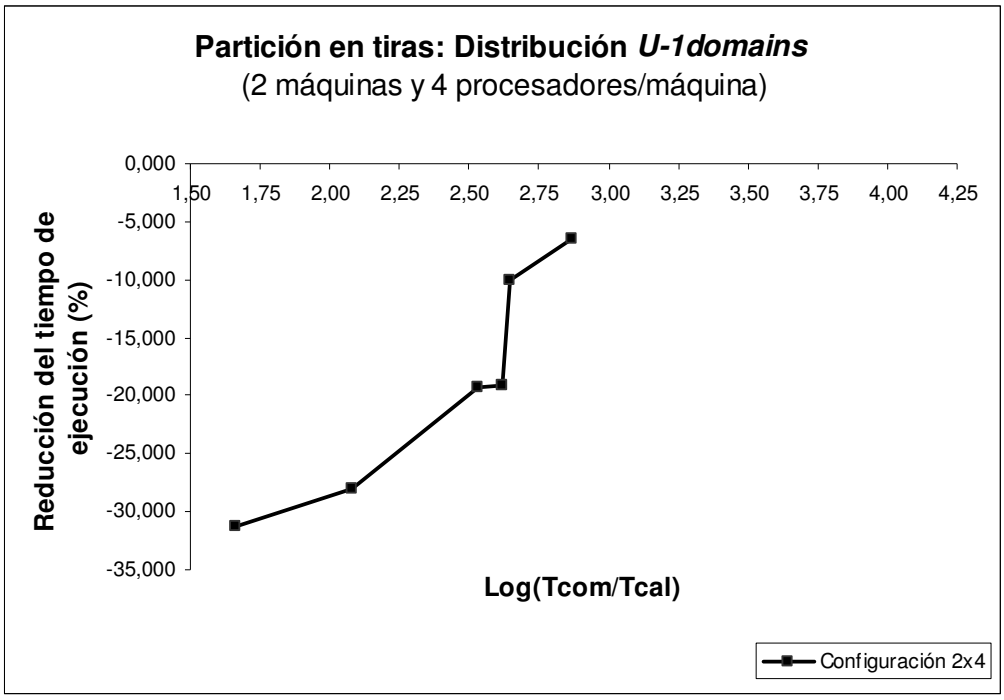


Fig. A.1 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en tiras (2 máquinas y 4 procesadores por máquina)

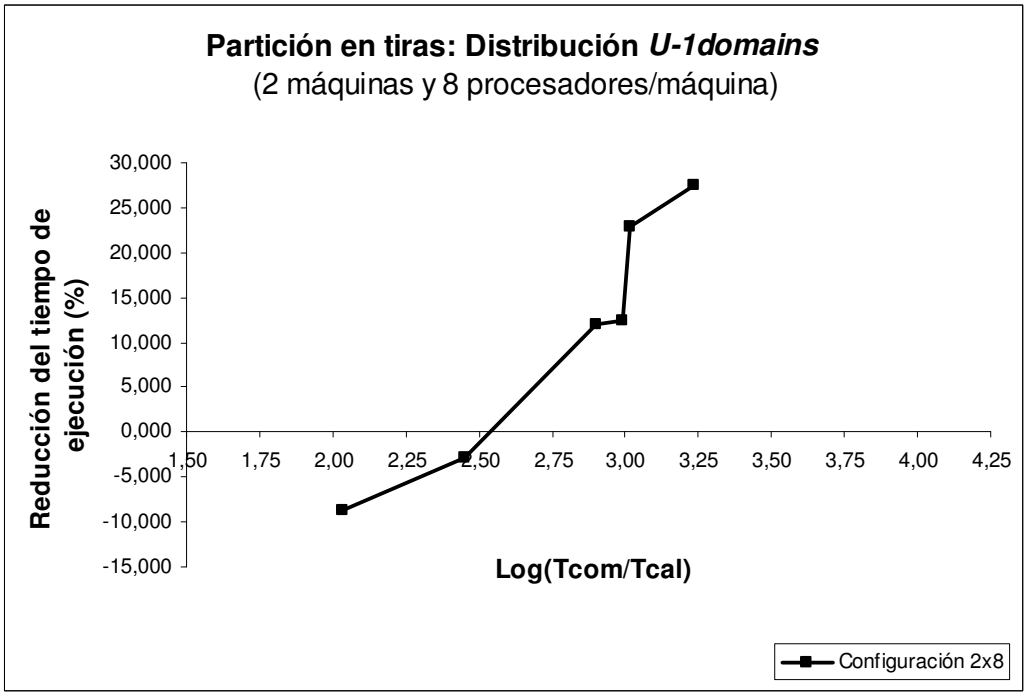


Fig. A.2 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en tiras (2 máquinas y 8 procesadores por máquina)

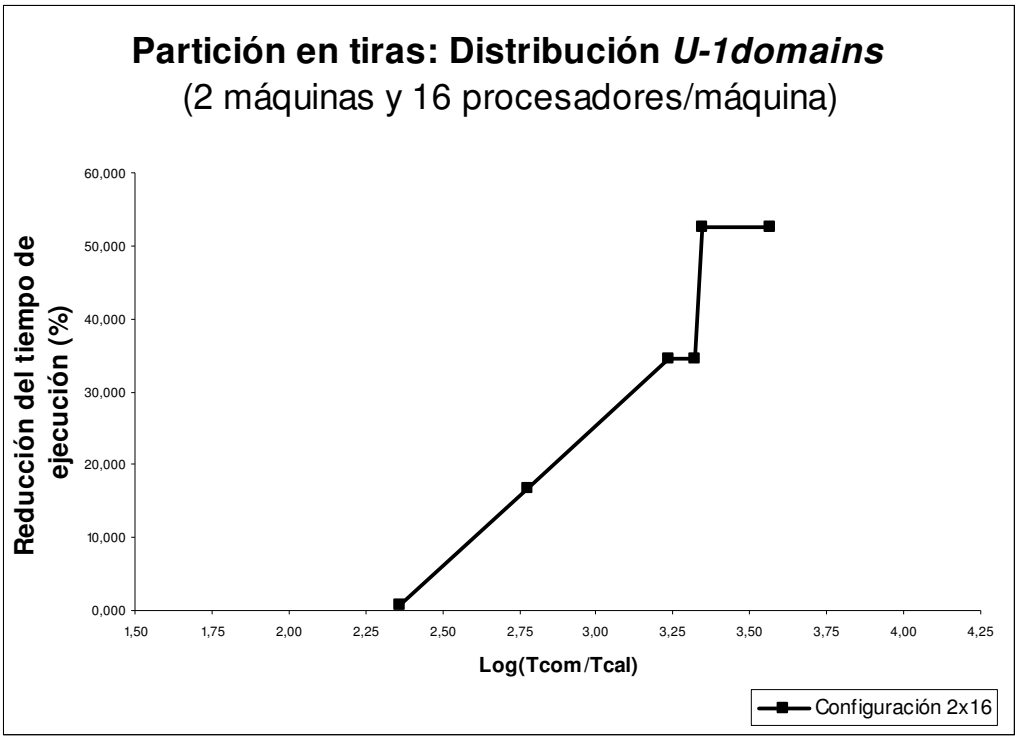


Fig. A.3 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en tiras (2 máquinas y 16 procesadores por máquina)

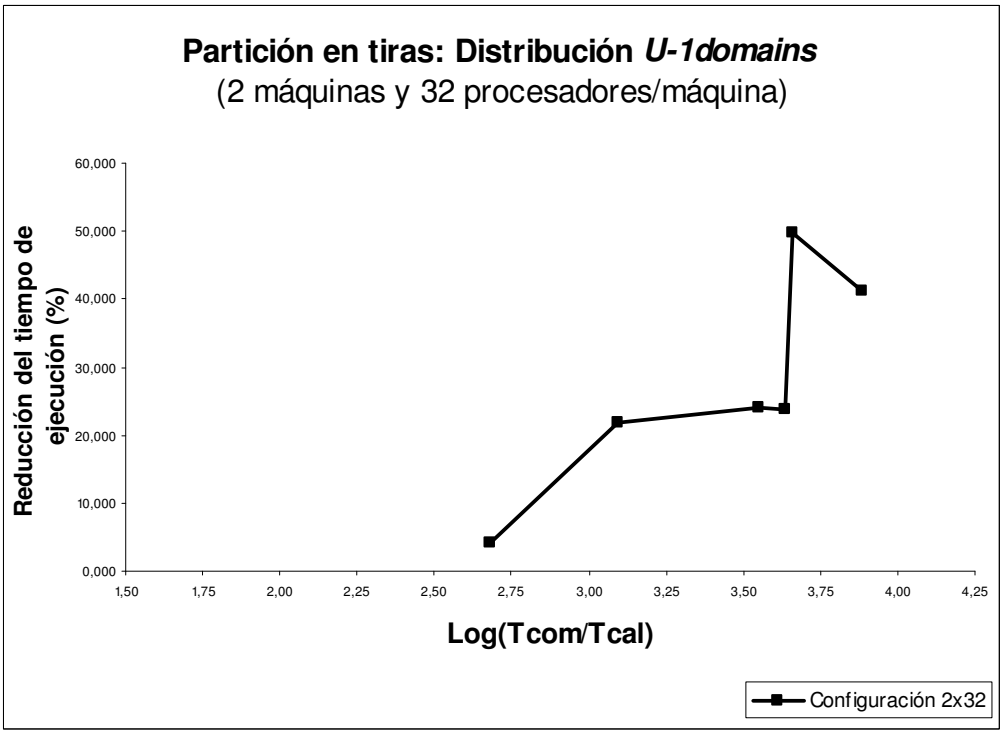


Fig. A.4 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en tiras (2 máquinas y 32 procesadores por máquina)

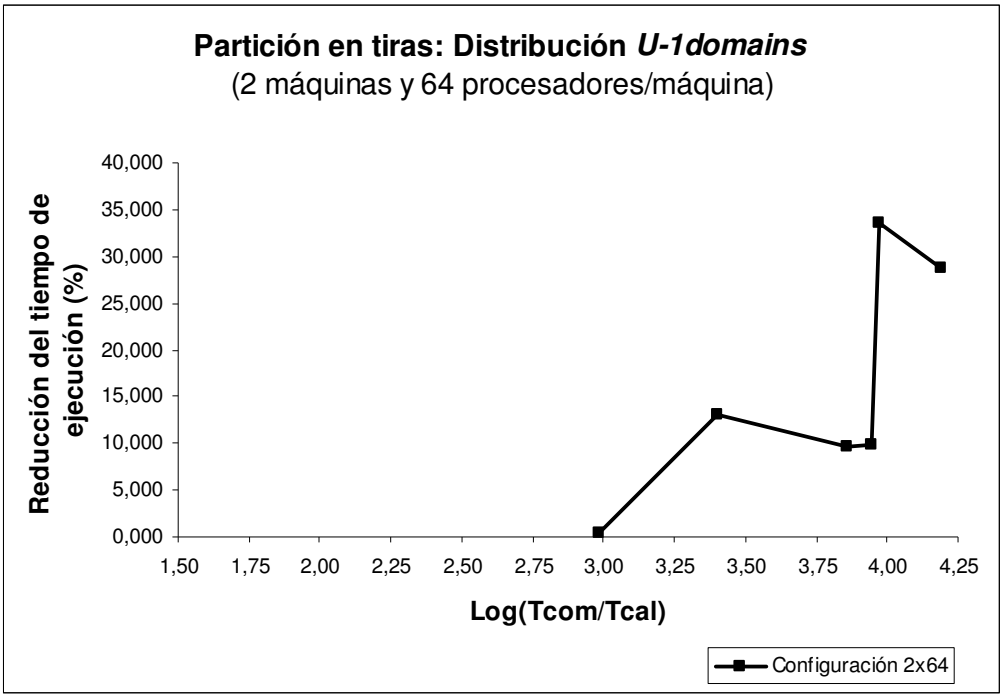


Fig. A.5 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en tiras (2 máquinas y 64 procesadores por máquina)

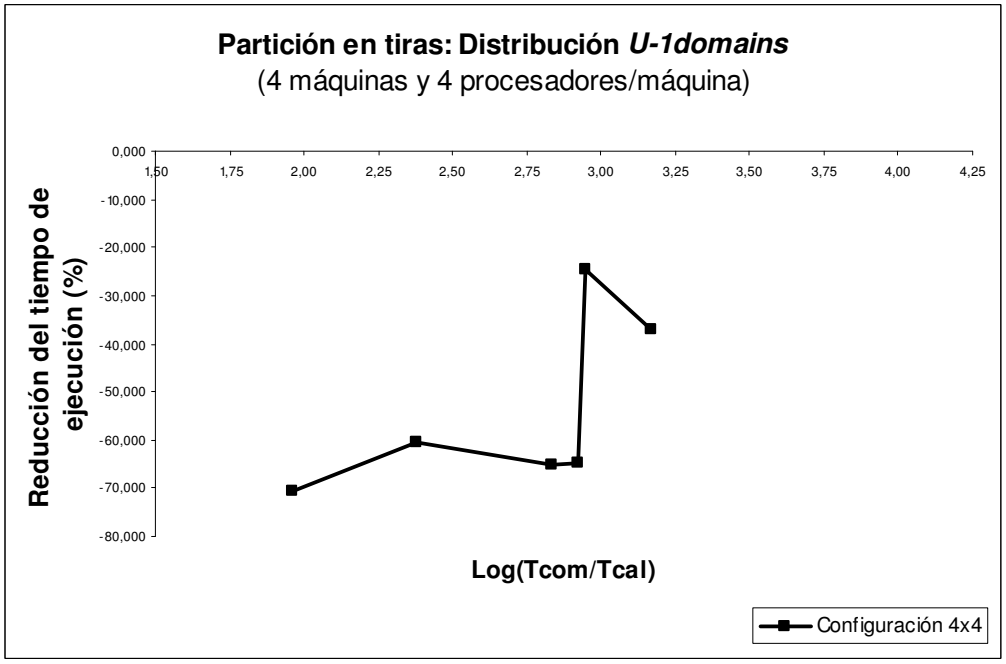


Fig. A.6 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en tiras (4 máquinas y 4 procesadores por máquina)

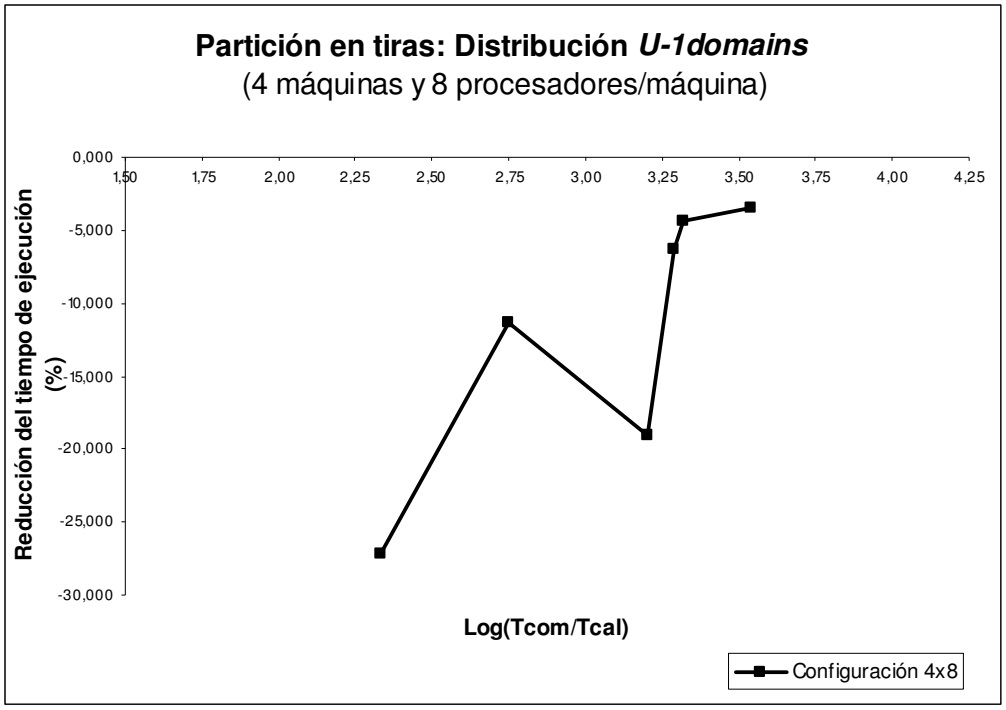


Fig. A.7 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en tiras (4 máquinas y 8 procesadores por máquina)

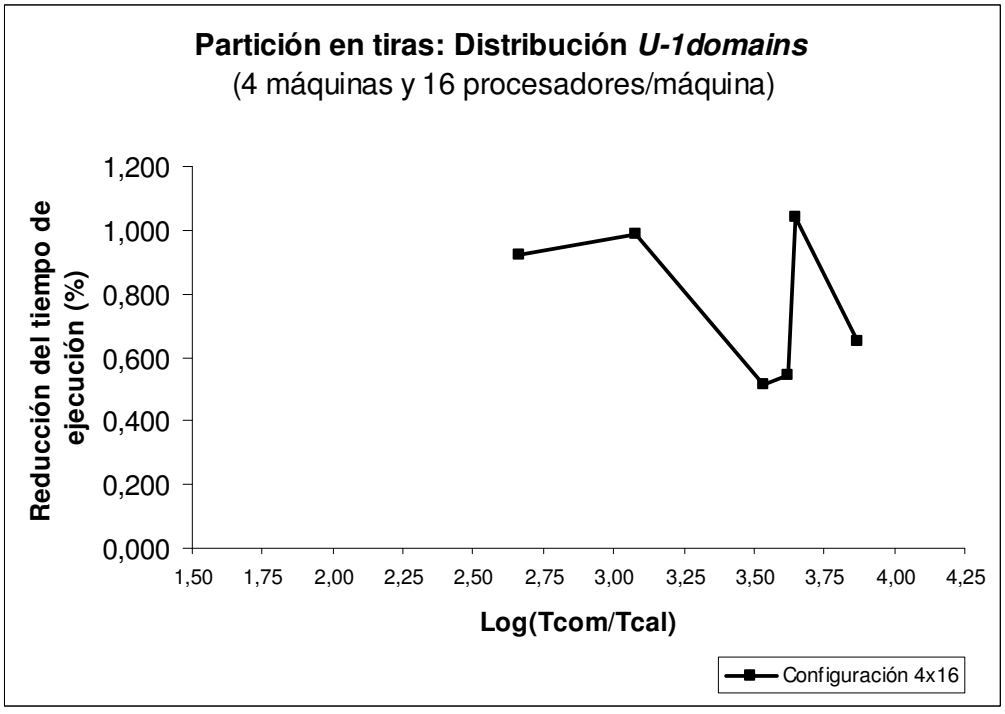


Fig. A.8 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en tiras (4 máquinas y 16 procesadores por máquina)

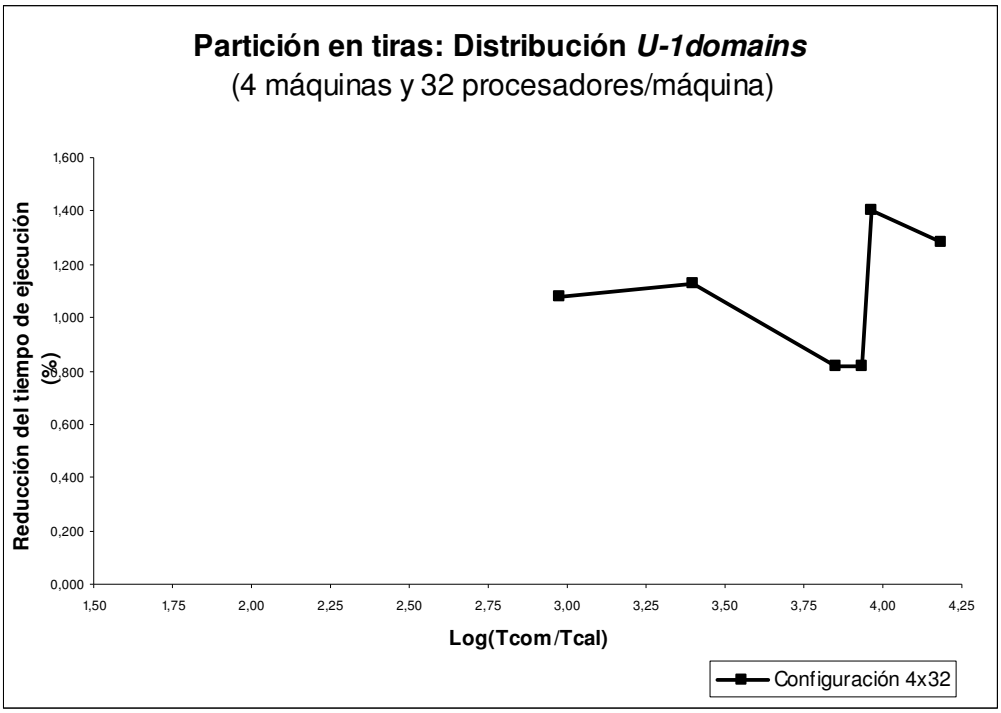


Fig. A.9 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en tiras (4 máquinas y 32 procesadores por máquina)

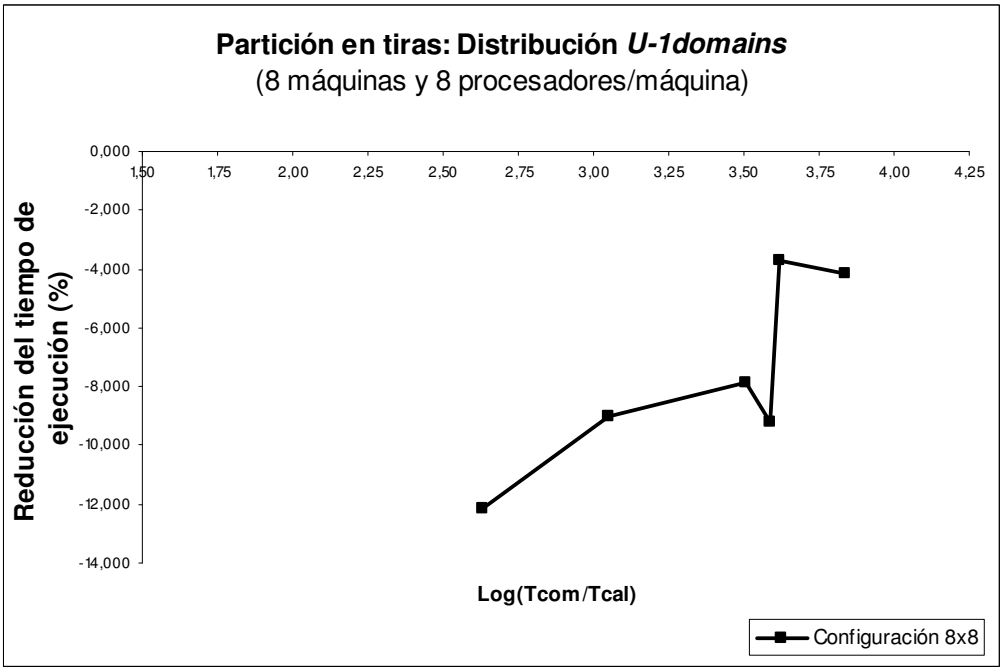


Fig. A.10 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en tiras (8 máquinas y 8 procesadores por máquina)

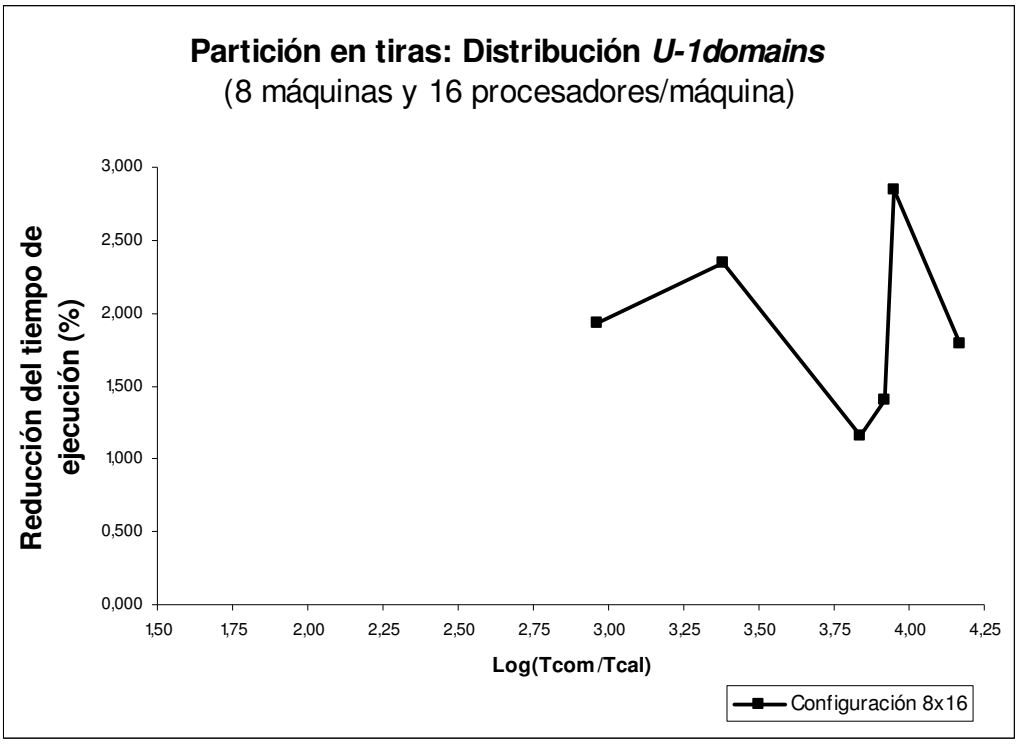


Fig. A.11 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en tiras (8 máquinas y 16 procesadores por máquina)

A.2 Distribución *U-1domains*: Partición en cajas

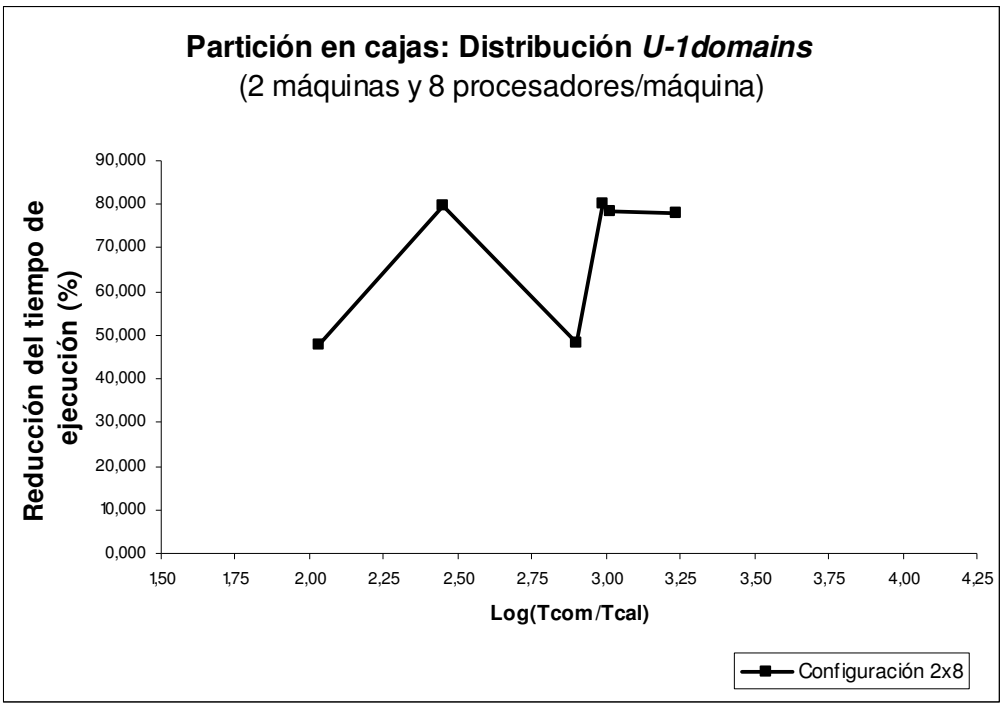


Fig. A.12 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en cajas (2 máquinas y 8 procesadores por máquina)

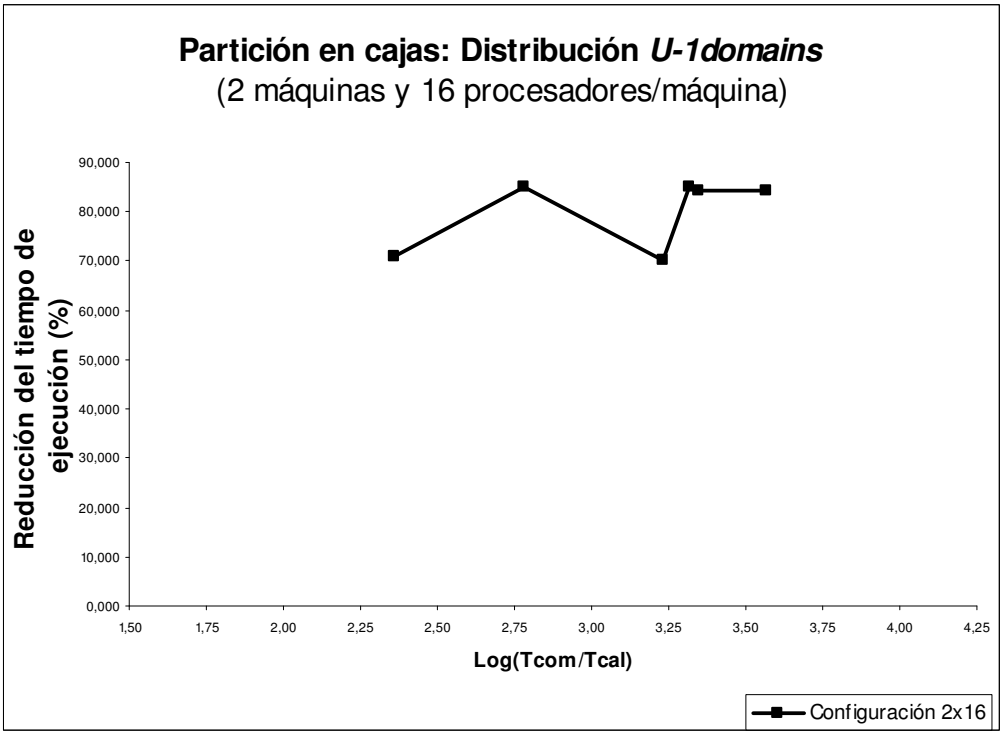


Fig. A.13 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en cajas (2 máquinas y 16 procesadores por máquina)

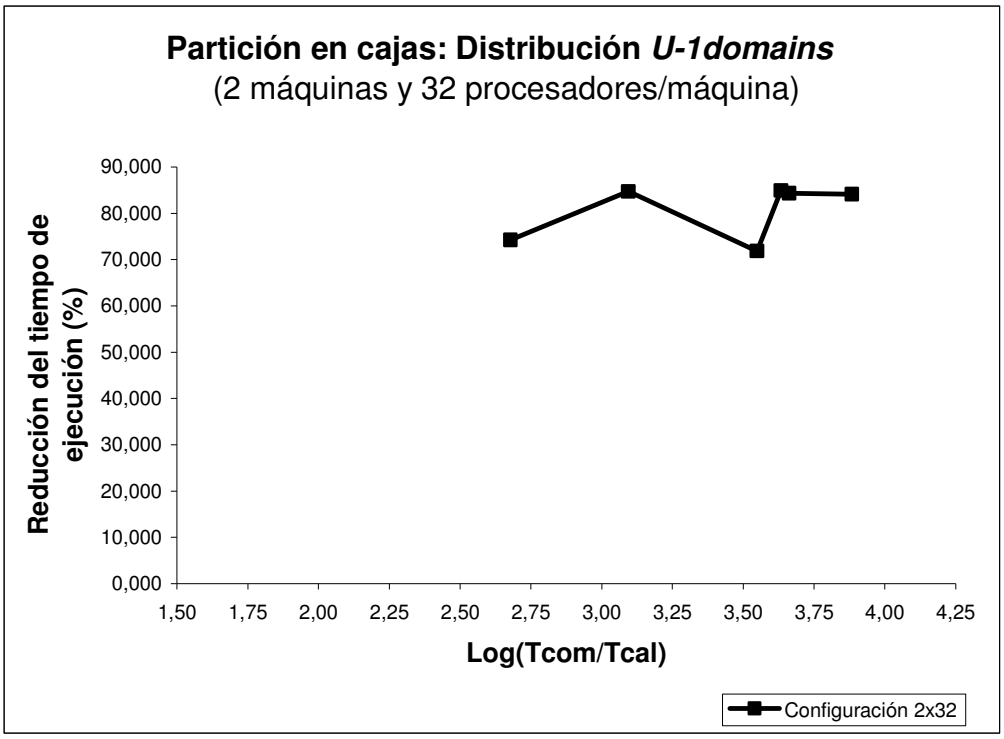


Fig. A.14 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en cajas (2 máquinas y 32 procesadores por máquina)

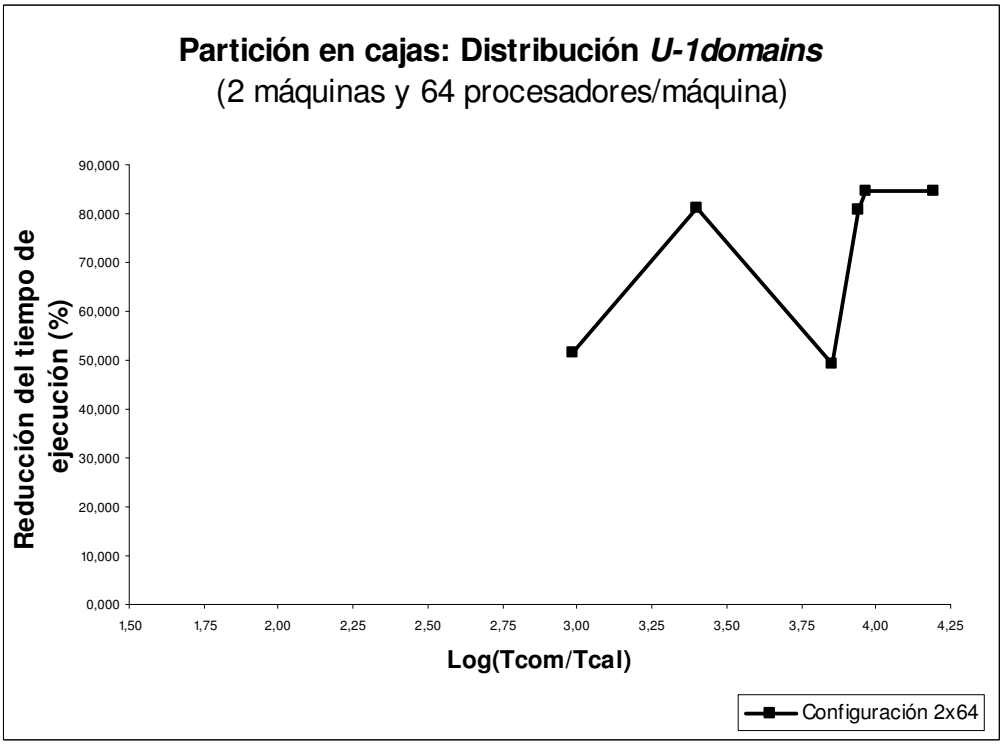


Fig. A.15 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en cajas (2 máquinas y 64 procesadores por máquina)

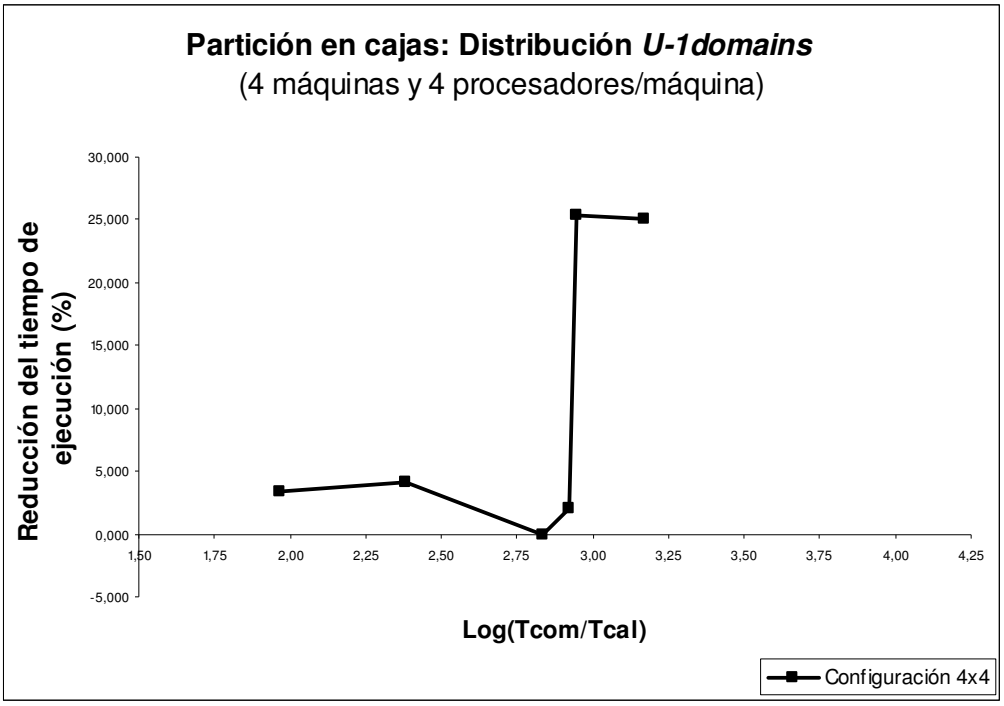


Fig. A.16 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en cajas (4 máquinas y 4 procesadores por máquina)

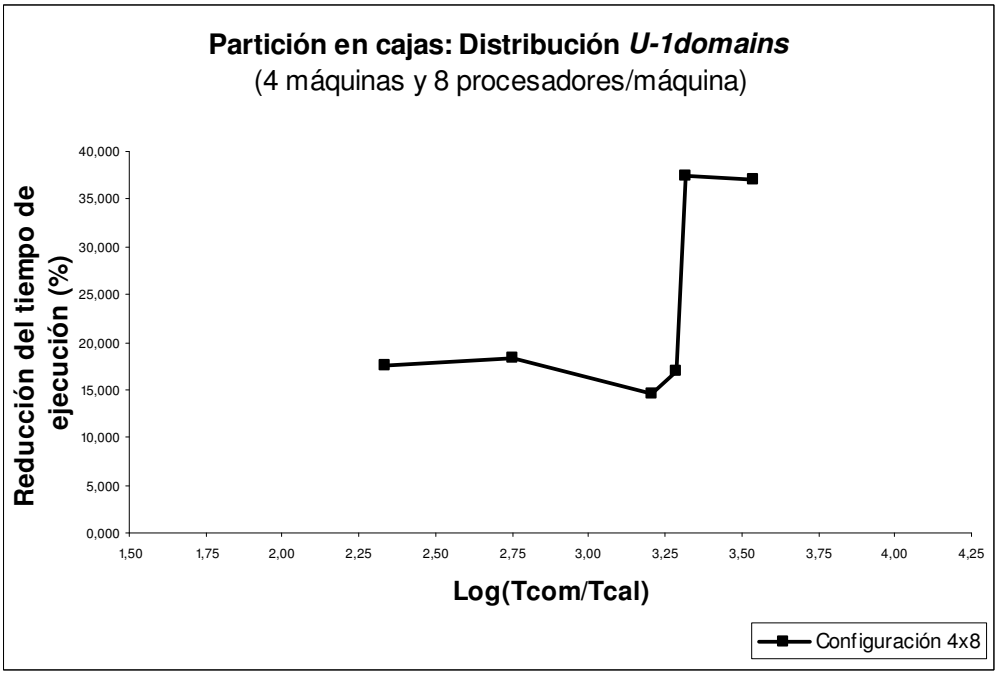


Fig. A.17 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en cajas (4 máquinas y 8 procesadores por máquina)

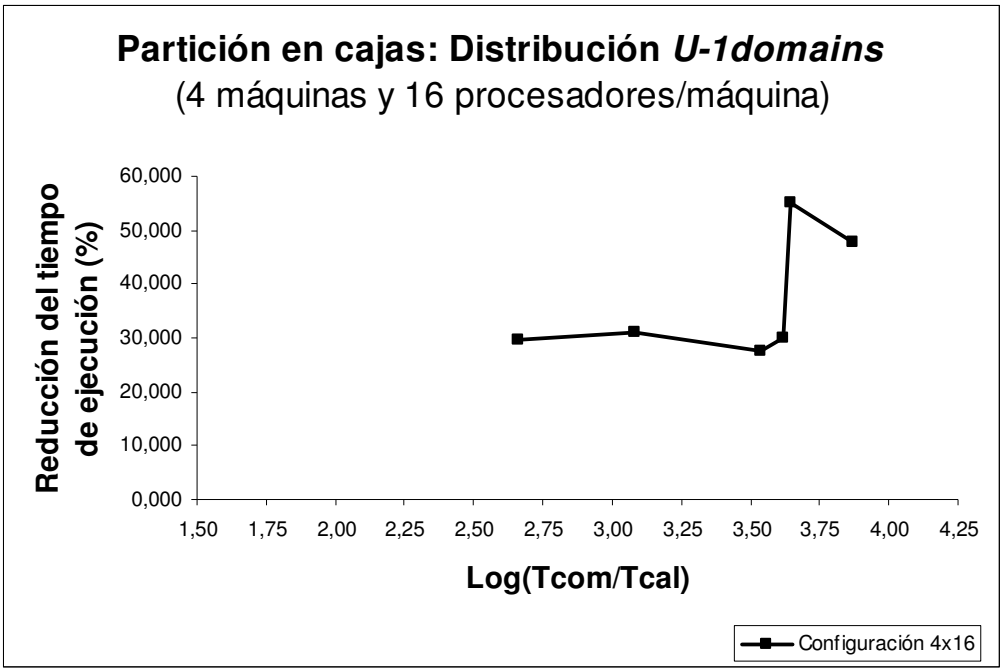


Fig. A.18 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en cajas (4 máquinas y 16 procesadores por máquina)

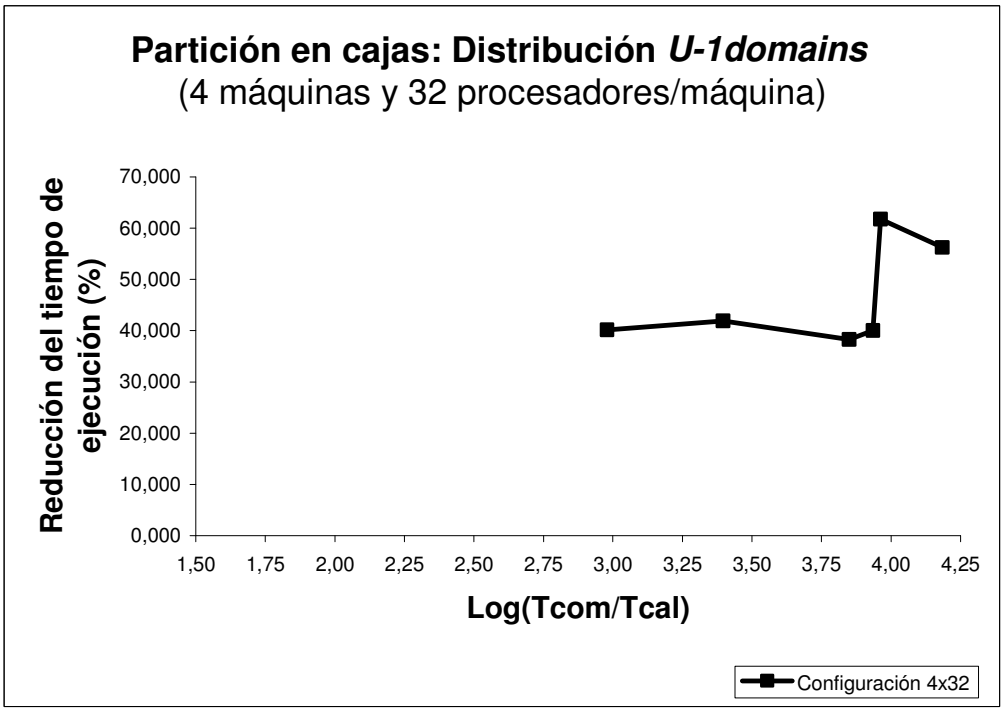


Fig. A.19 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en cajas (4 máquinas y 32 procesadores por máquina)

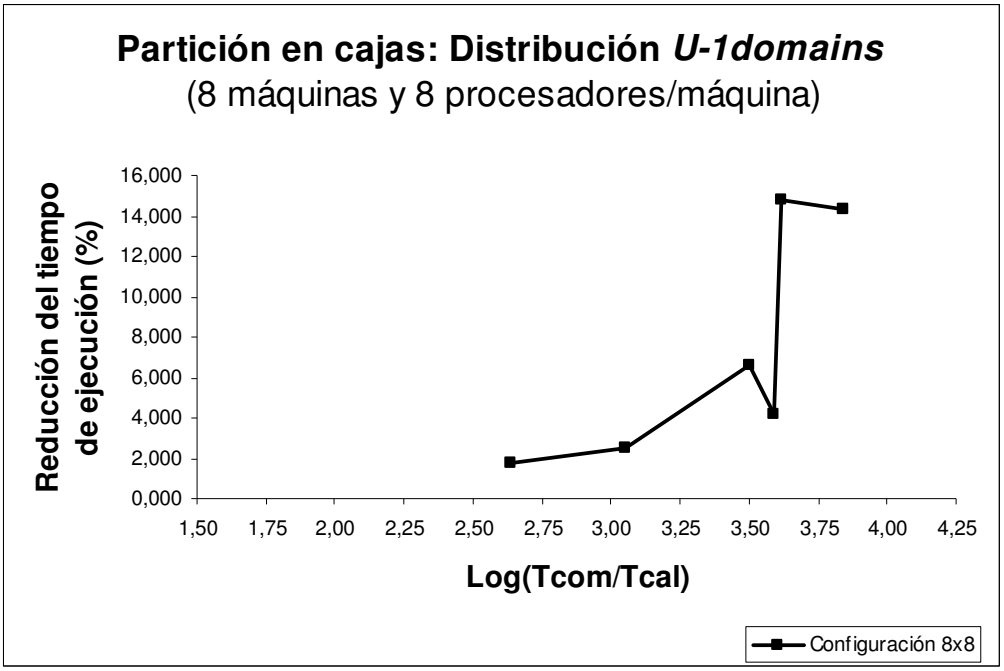


Fig. A.20 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en cajas (8 máquinas y 8 procesadores por máquina)

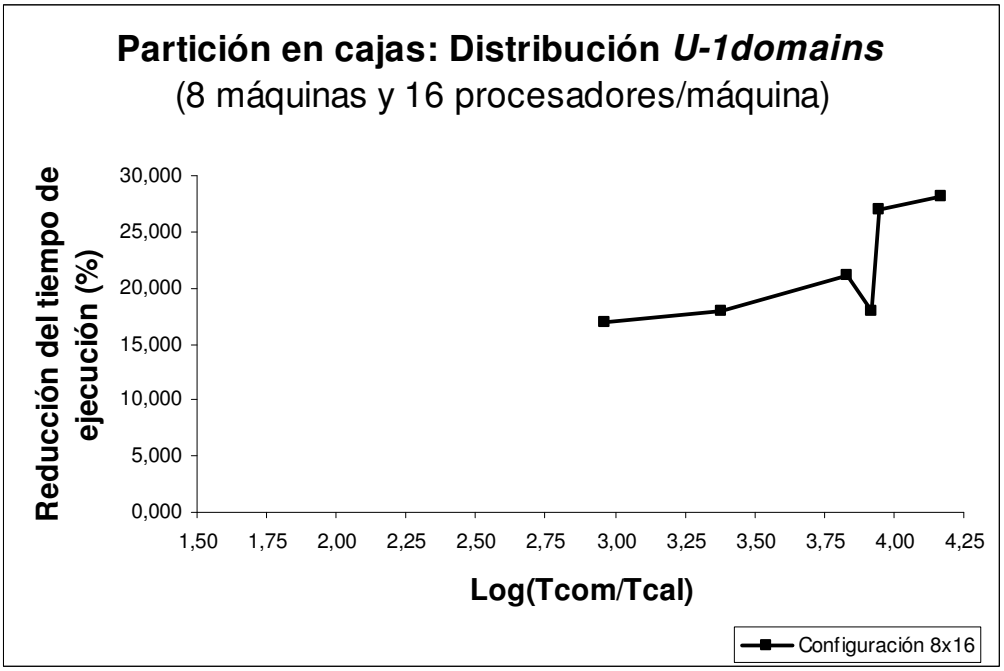


Fig. A.21 Reducción del tiempo de ejecución utilizando la distribución *U-1domains* para particiones en cajas (8 máquinas y 16 procesadores por máquina)

A.3 Distribución *U-Bdomains*: Partición en tiras

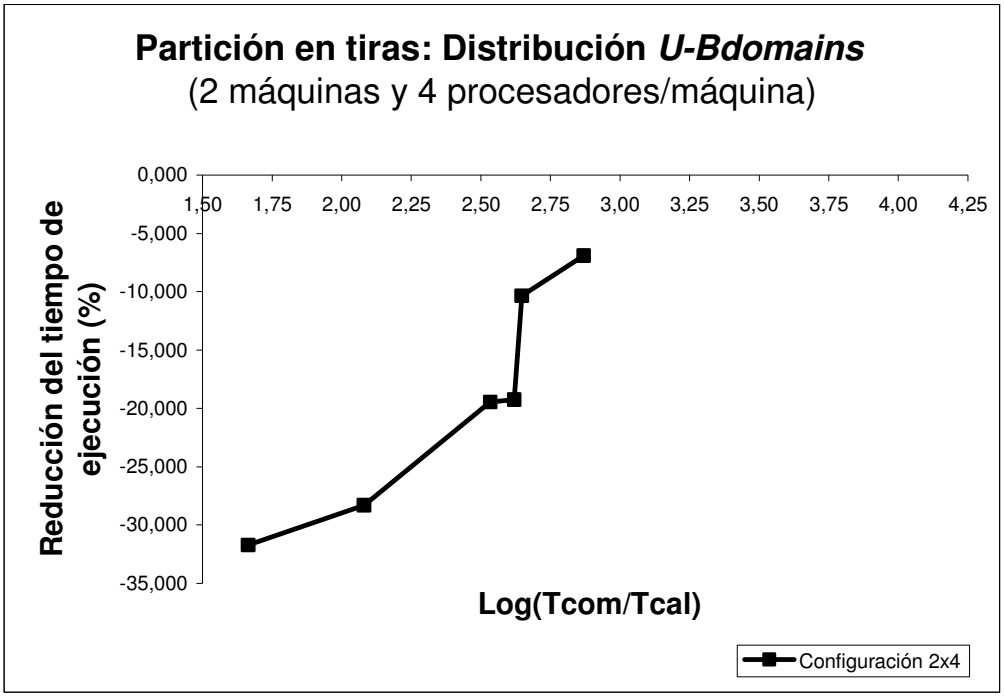


Fig. A.22 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en tiras (2 máquinas y 4 procesadores por máquina)

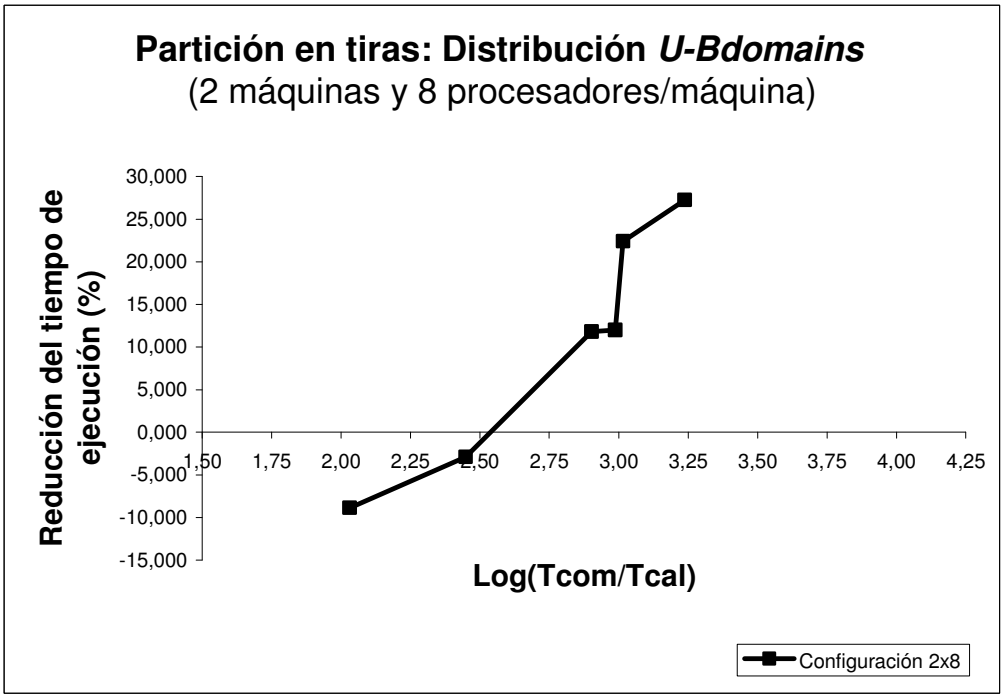


Fig. A.23 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en tiras (2 máquinas y 8 procesadores por máquina)

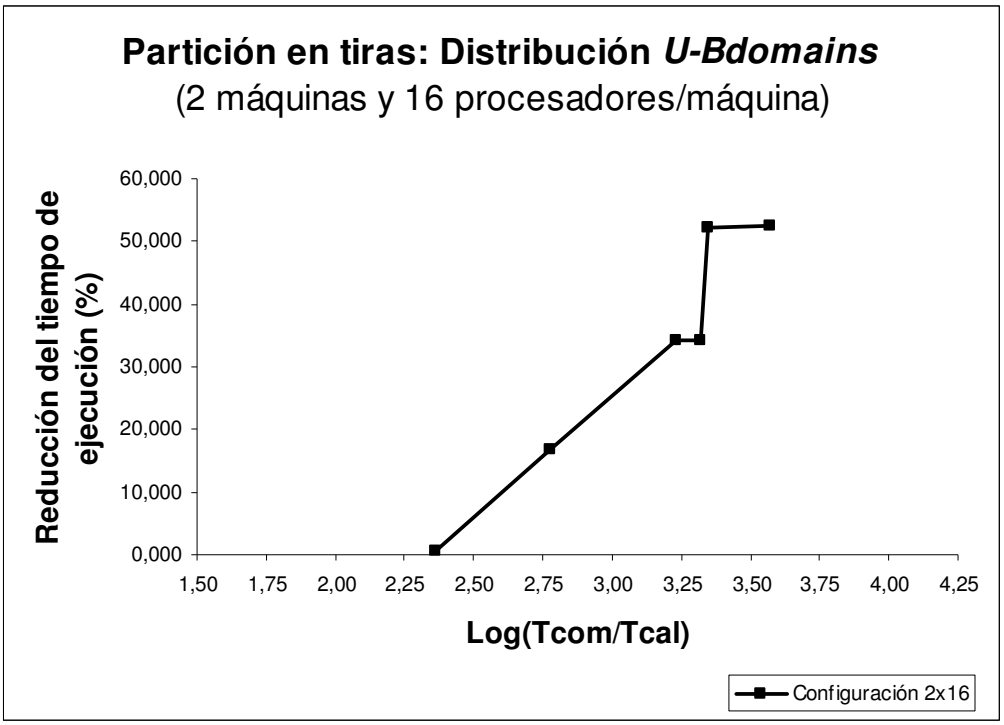


Fig. A.24 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en tiras (2 máquinas y 16 procesadores por máquina)

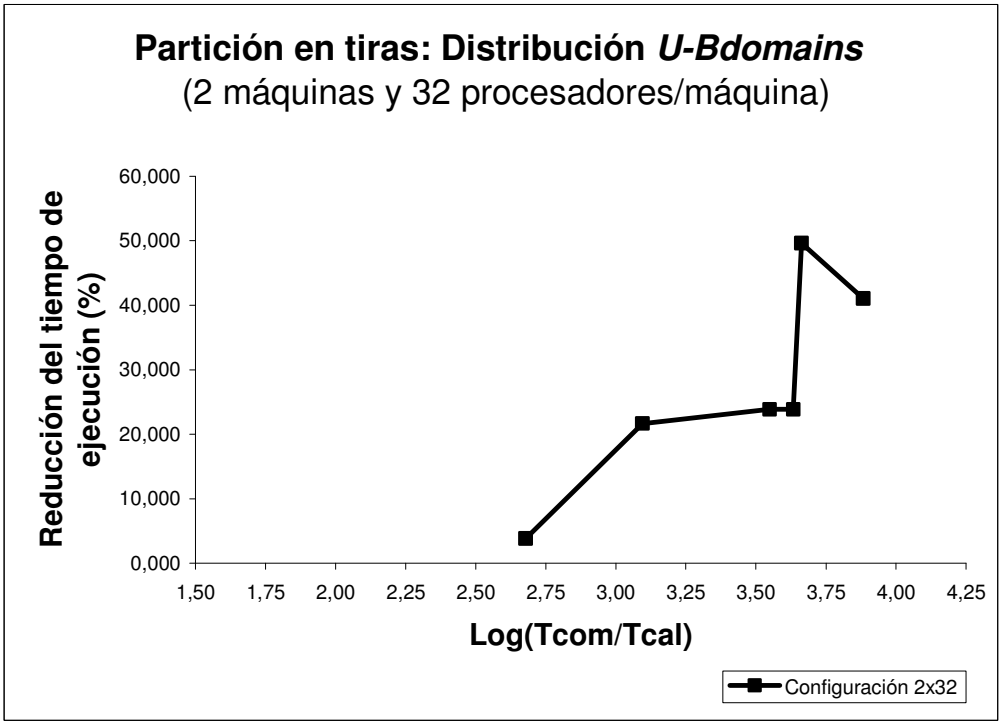


Fig. A.25 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en tiras (2 máquinas y 32 procesadores por máquina)

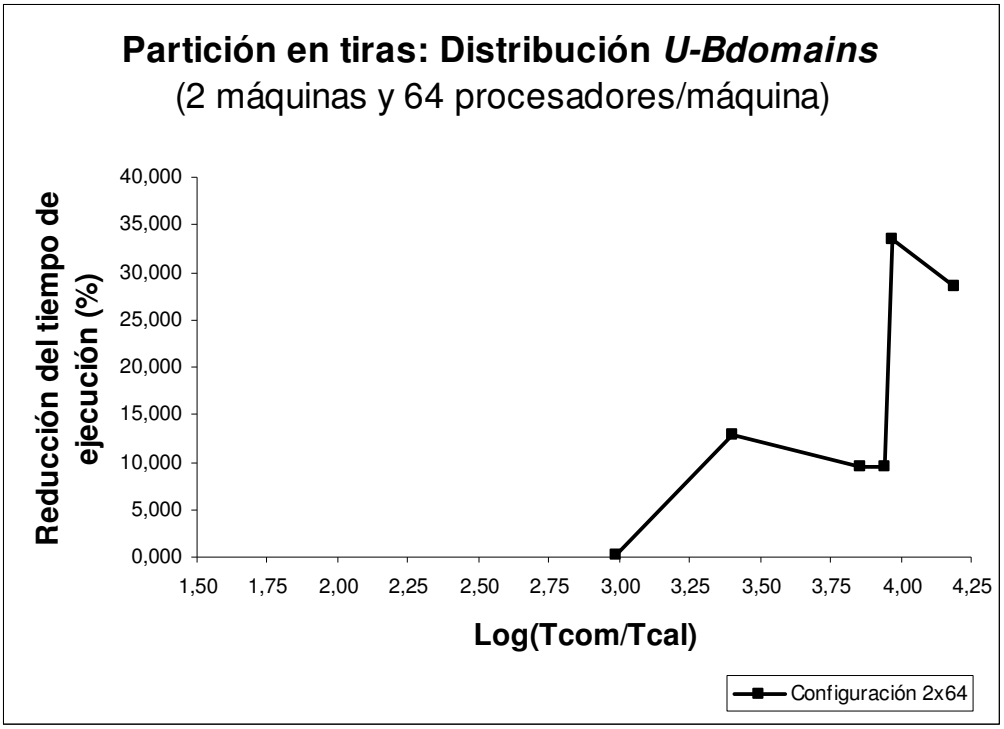


Fig. A.26 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en tiras (2 máquinas y 64 procesadores por máquina)

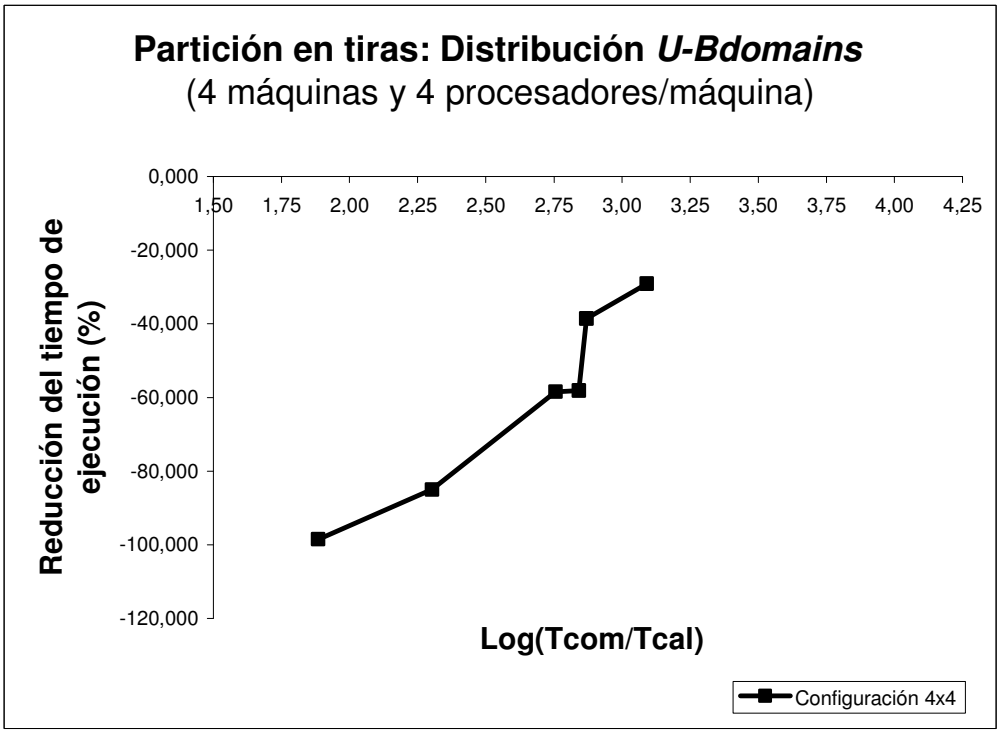


Fig. A.27 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en tiras (4 máquinas y 4 procesadores por máquina)

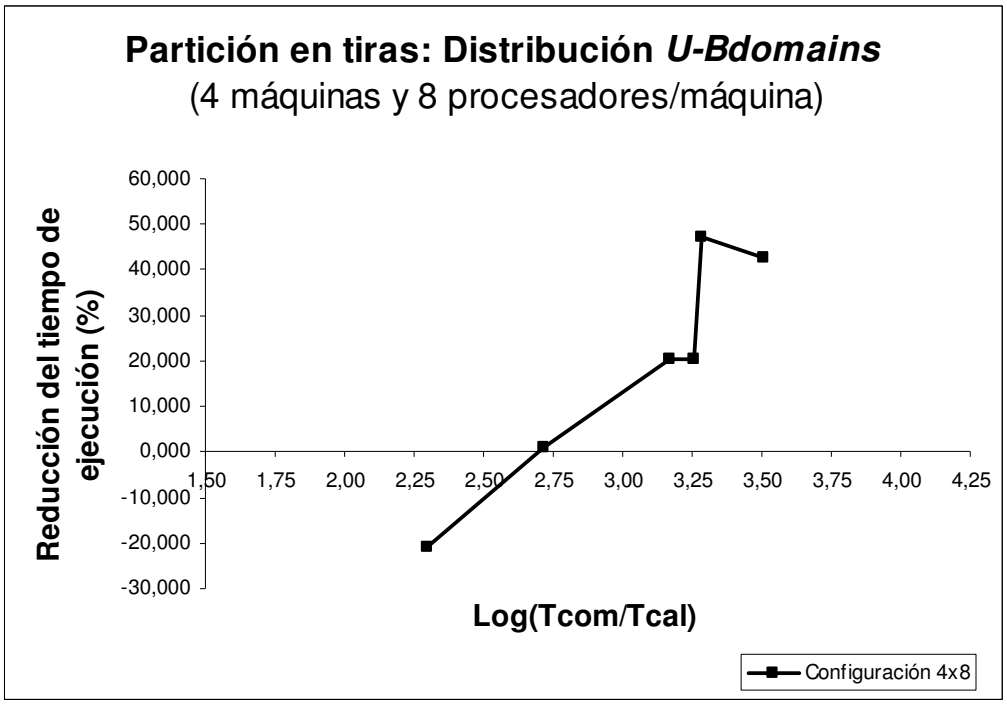


Fig. A.28 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en tiras (4 máquinas y 8 procesadores por máquina)

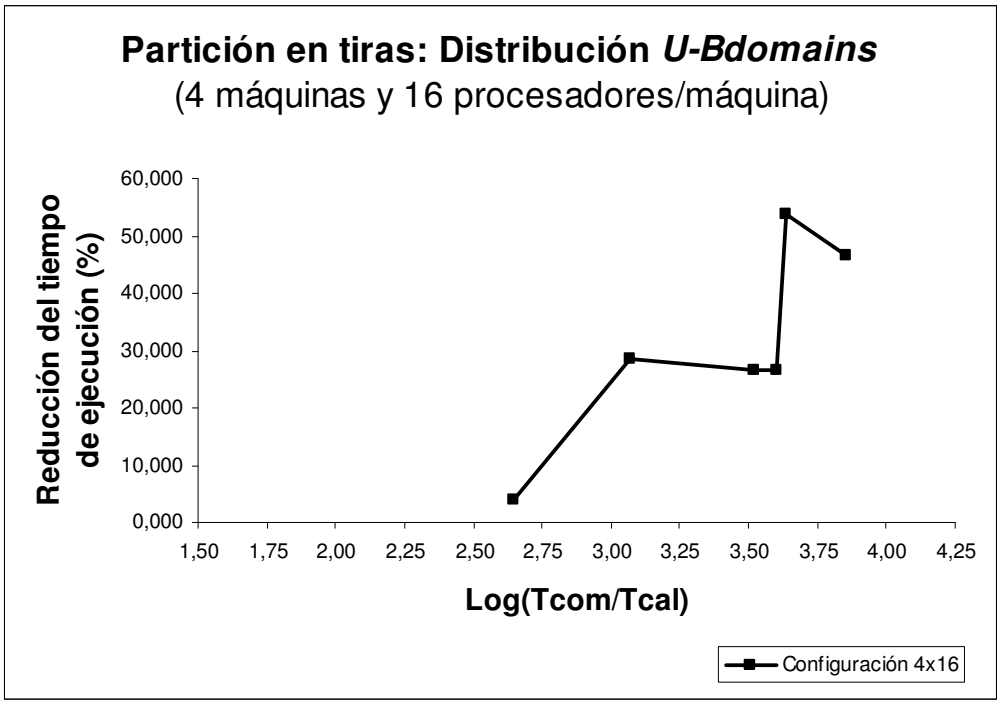


Fig. A.29 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en tiras (4 máquinas y 16 procesadores por máquina)

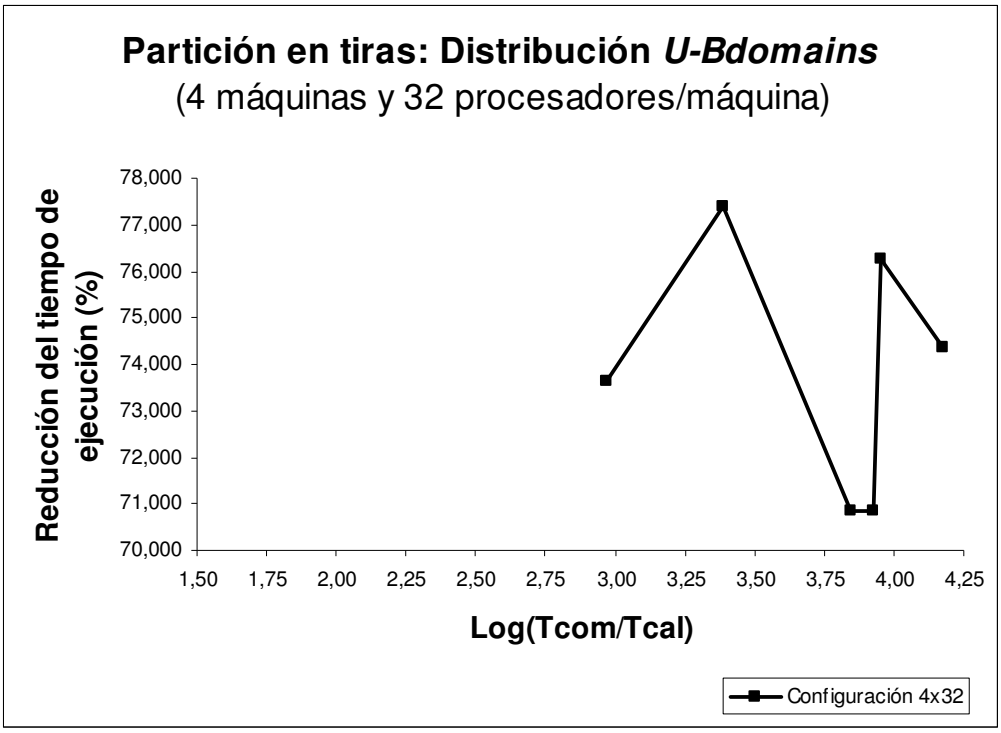


Fig. A.30 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en tiras (4 máquinas y 32 procesadores por máquina)

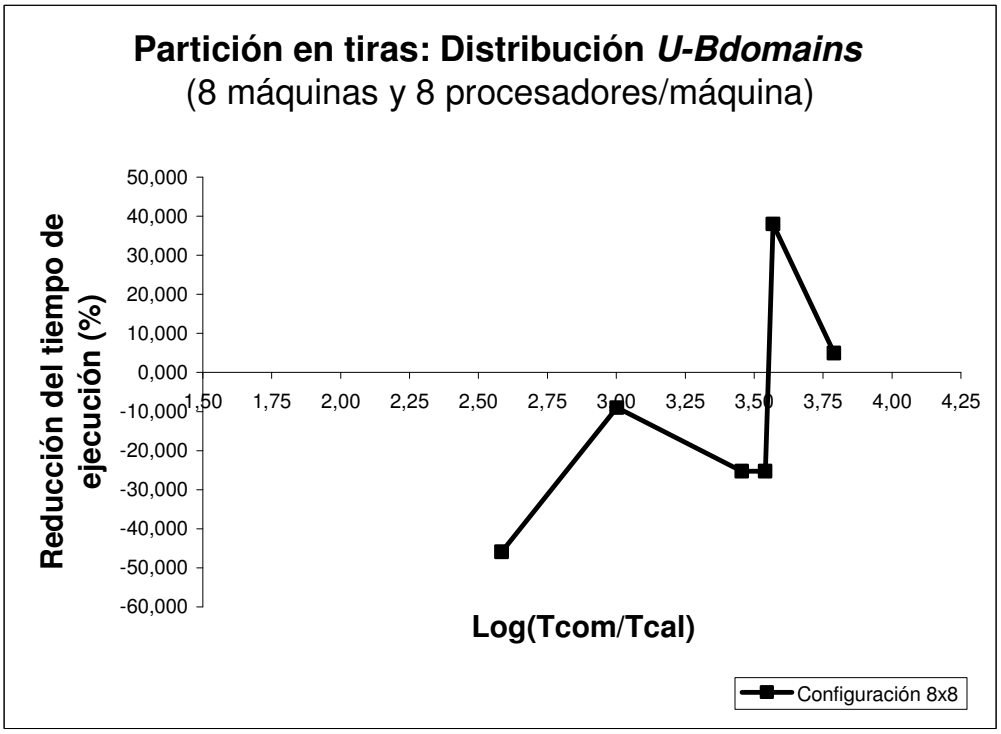


Fig. A.31 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en tiras (8 máquinas y 8 procesadores por máquina)

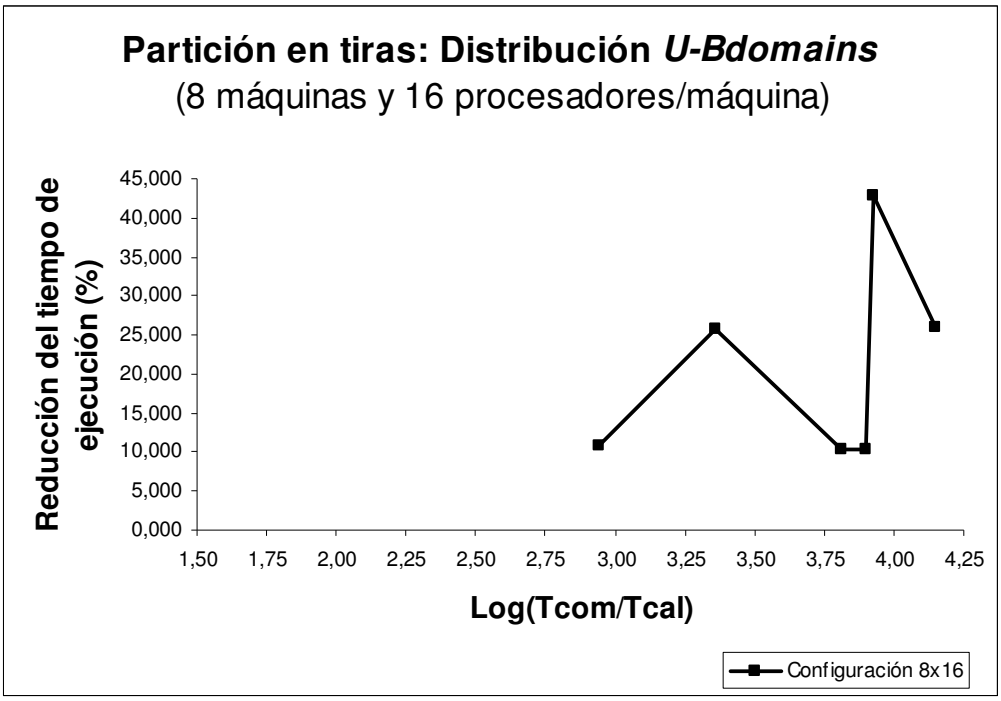


Fig. A.32 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en tiras (8 máquinas y 16 procesadores por máquina)

A.4 Distribución *U-Bdomains*: Partición en cajas

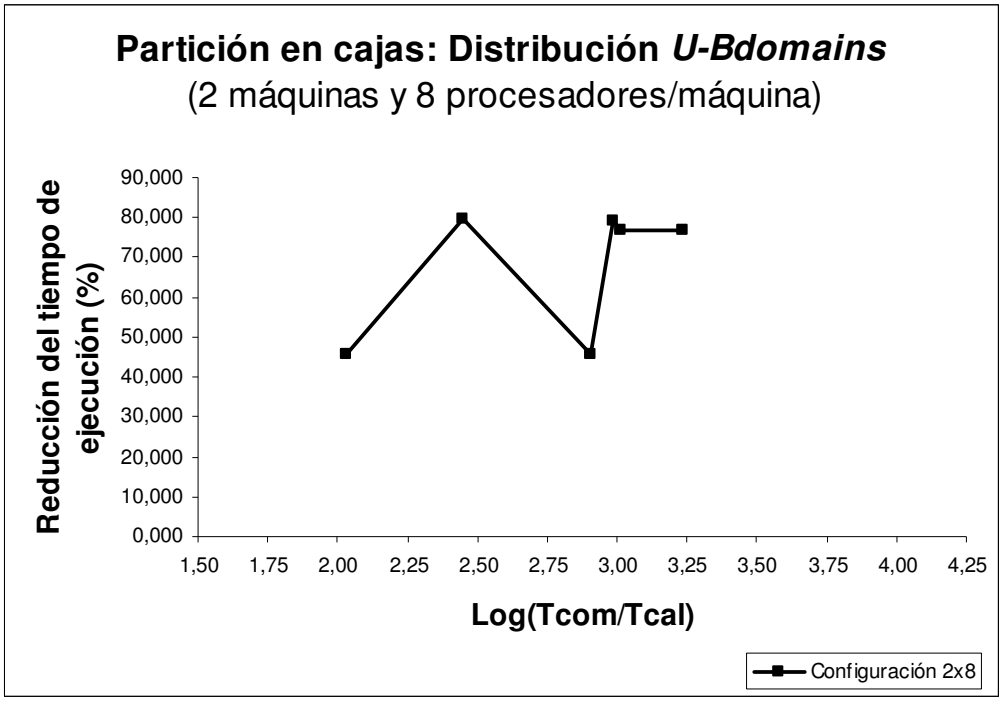


Fig. A.33 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en cajas (2 máquinas y 8 procesadores por máquina)

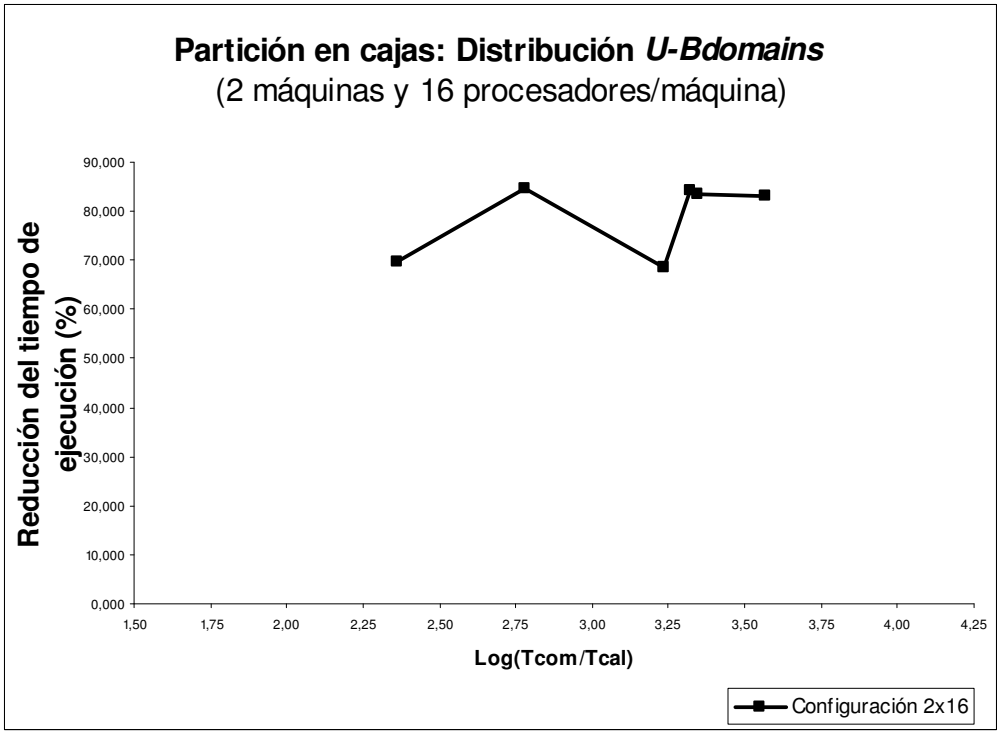


Fig. A.34 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en cajas (2 máquinas y 16 procesadores por máquina)

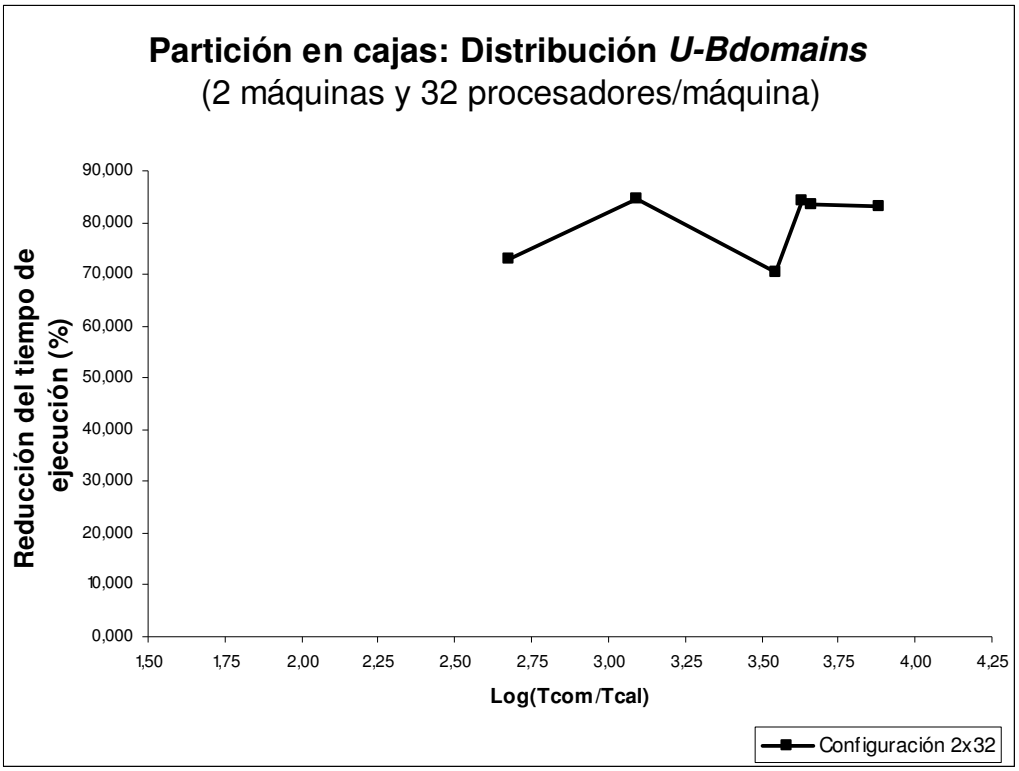


Fig. A.35 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en cajas (2 máquinas y 32 procesadores por máquina)

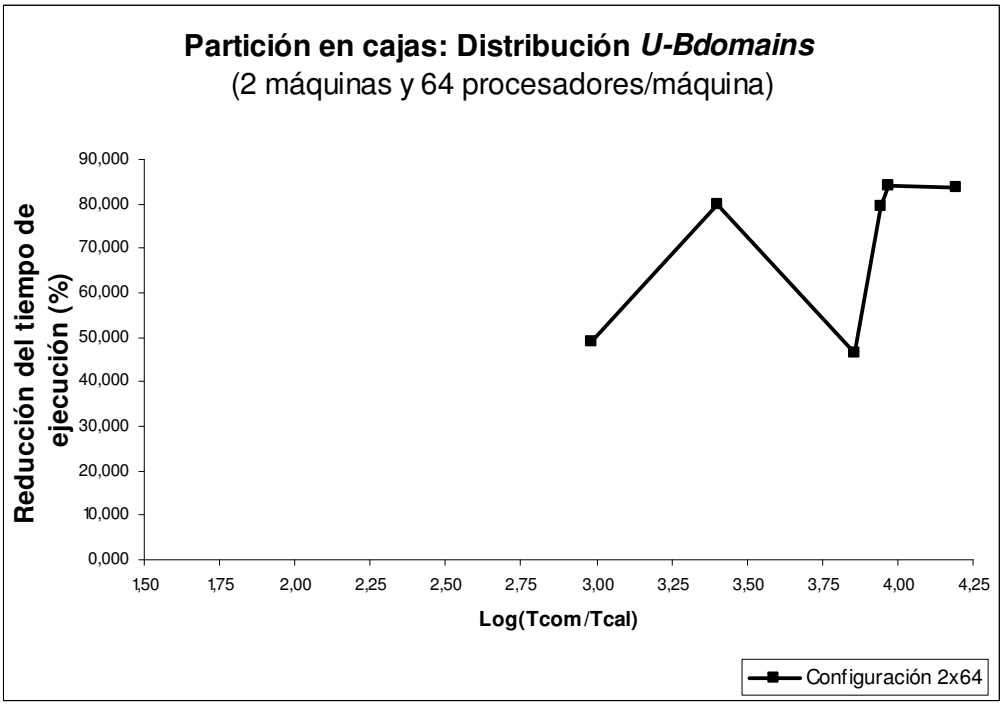


Fig. A.36 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en cajas (2 máquinas y 64 procesadores por máquina)

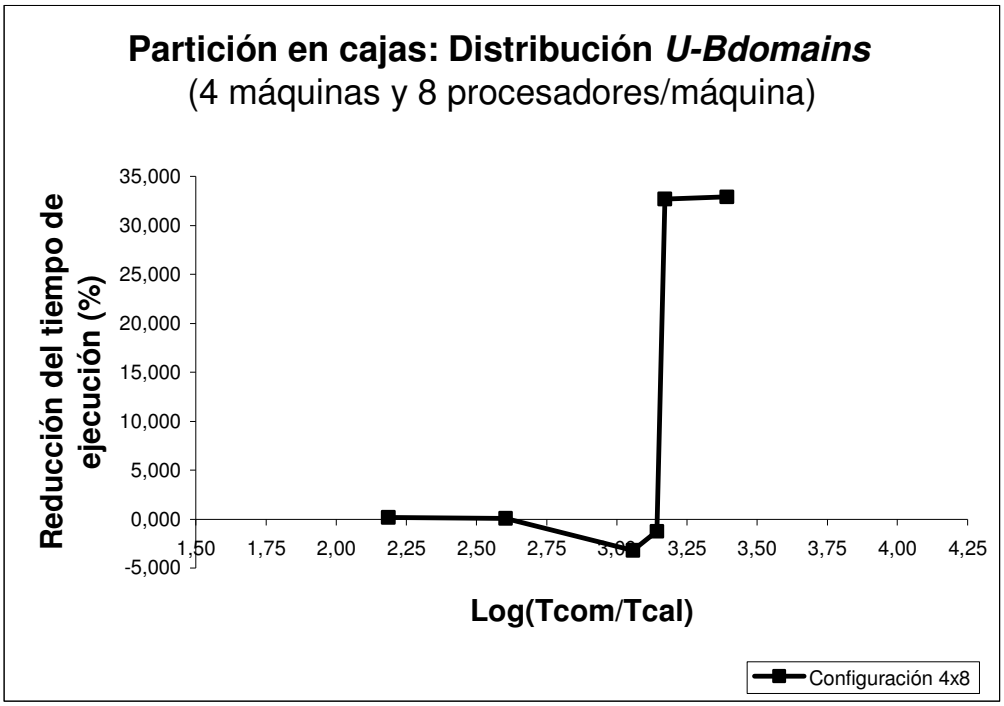


Fig. A.37 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en cajas (4 máquinas y 8 procesadores por máquina)

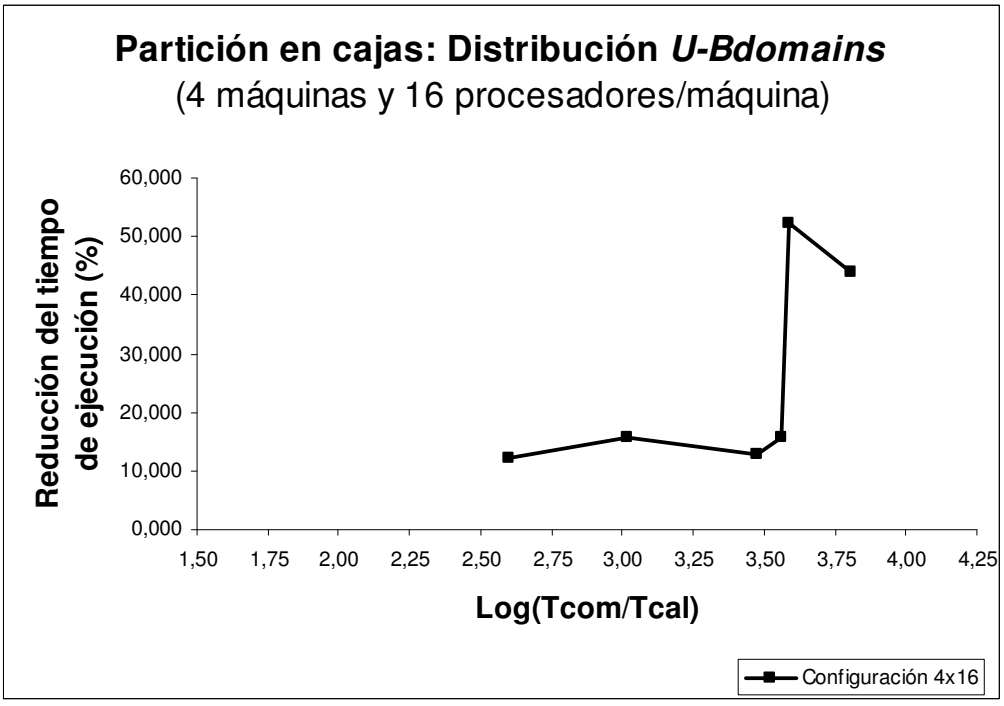


Fig. A.38 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en cajas (4 máquinas y 16 procesadores por máquina)

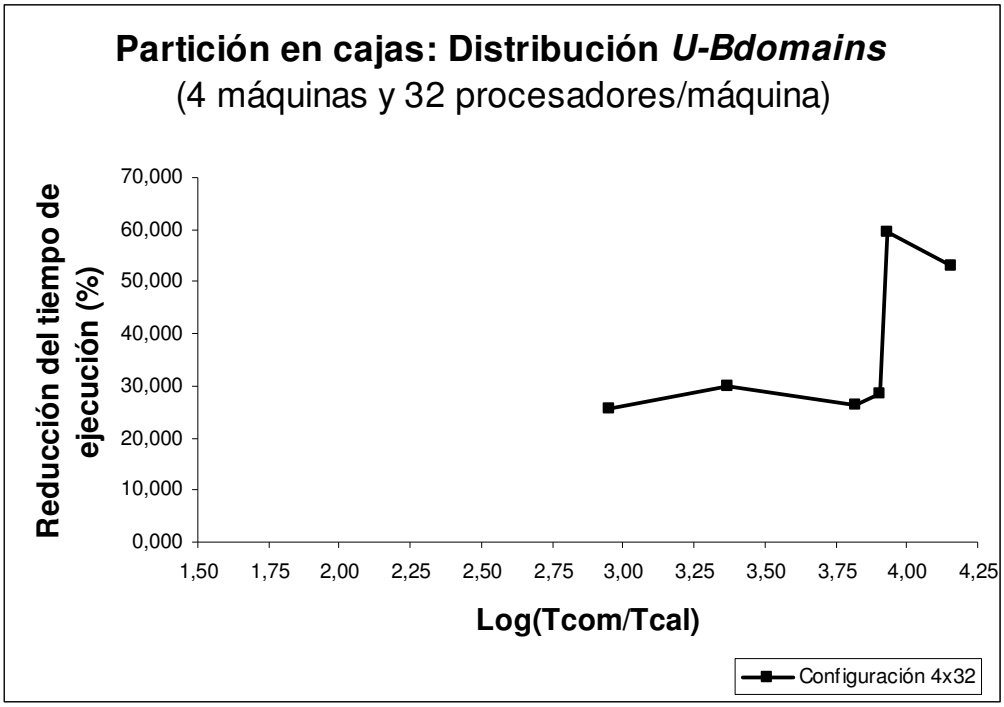


Fig. A.39 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en cajas (4 máquinas y 32 procesadores por máquina)

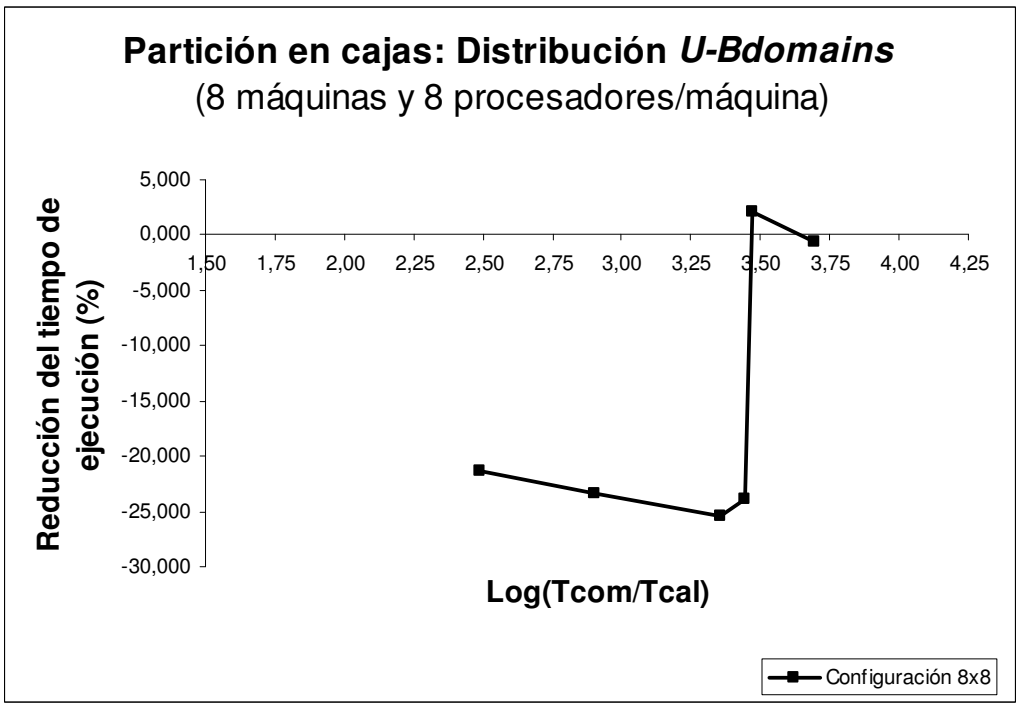


Fig. A.40 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en cajas (8 máquinas y 8 procesadores por máquina)

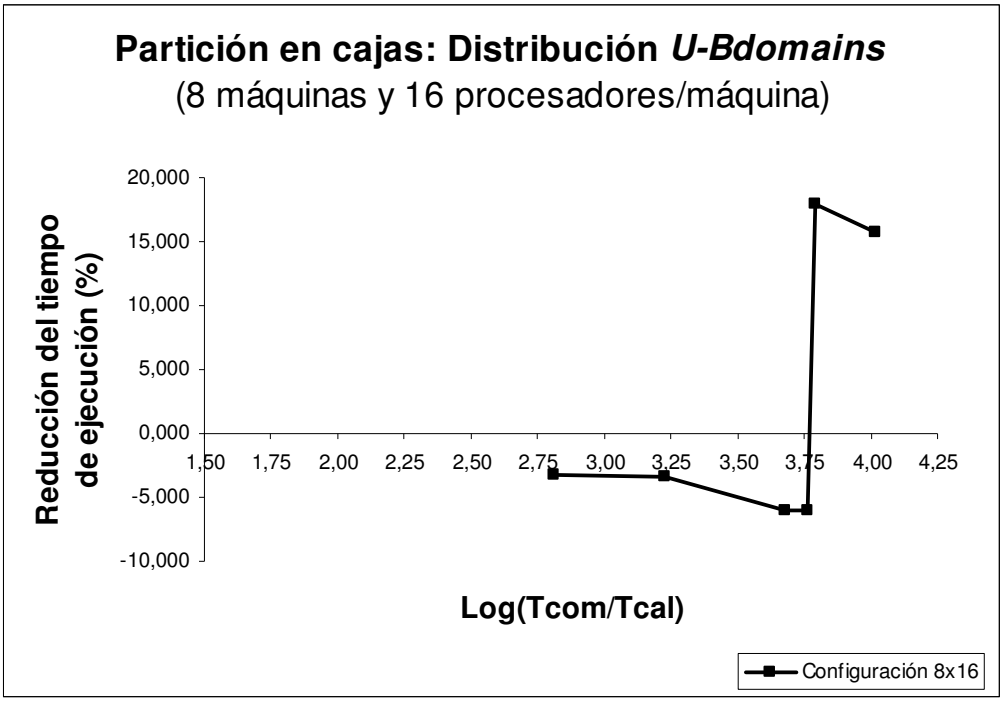


Fig. A.41 Reducción del tiempo de ejecución utilizando la distribución *U-Bdomains* para particiones en cajas (8 máquinas y 16 procesadores por máquina)

A.5 Distribución *U-CBdomains*: Partición en tiras

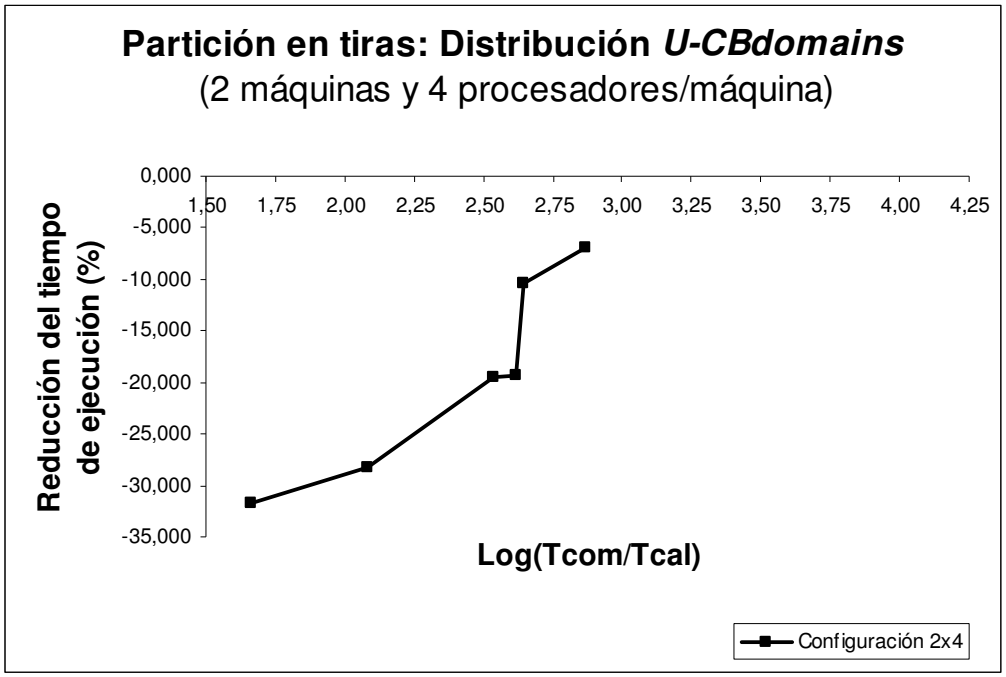


Fig. A.42 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en tiras (2 máquinas y 4 procesadores por máquina)

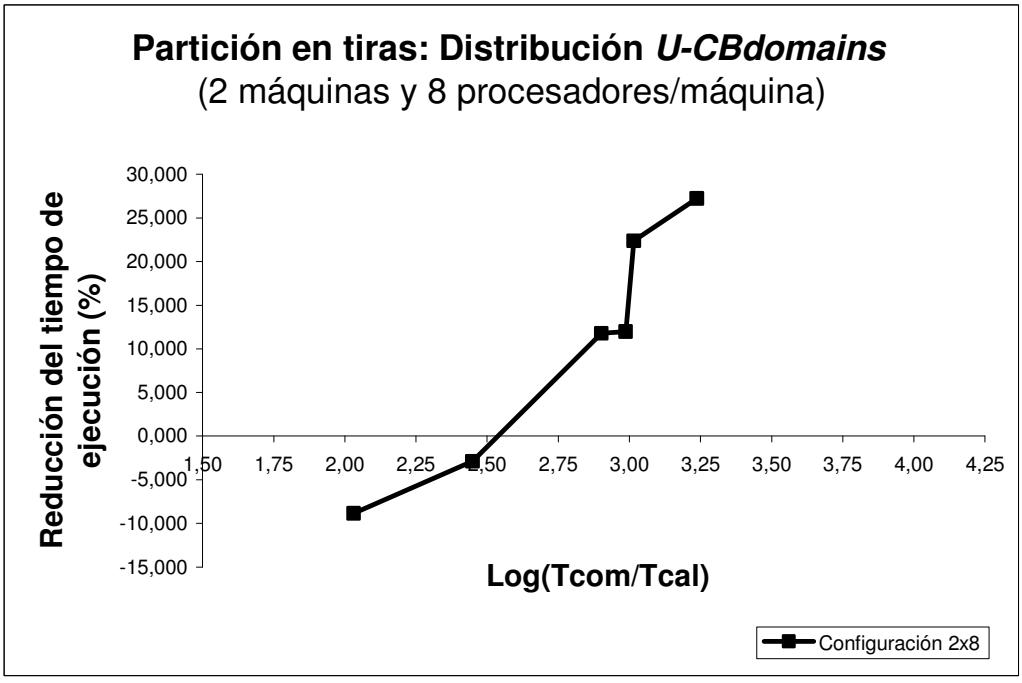


Fig. A.43 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en tiras (2 máquinas y 8 procesadores por máquina)

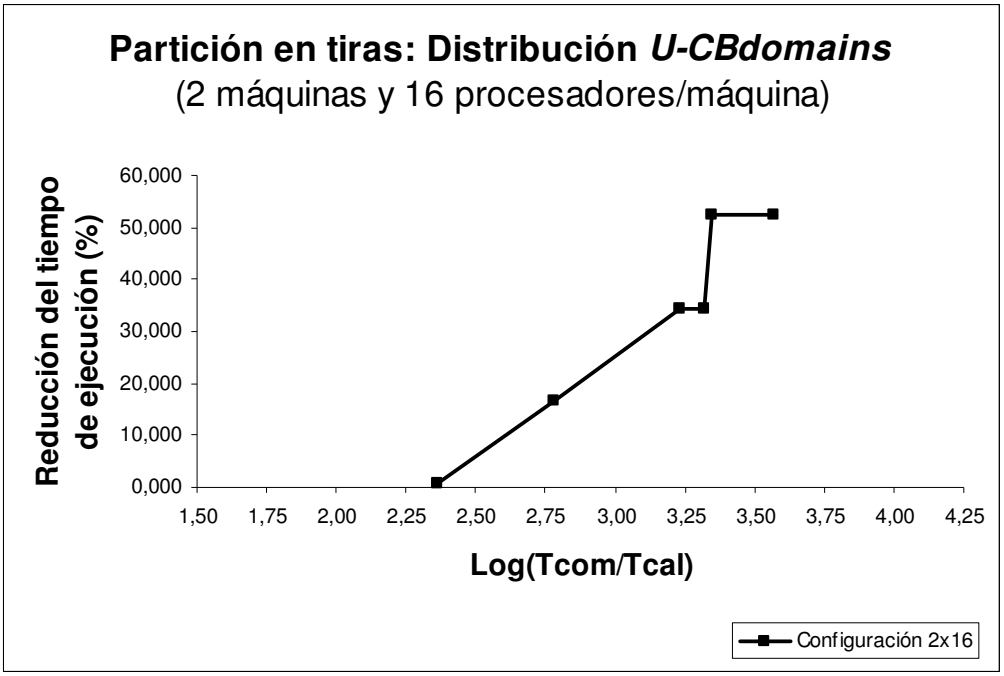


Fig. A.44 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en tiras (2 máquinas y 16 procesadores por máquina)

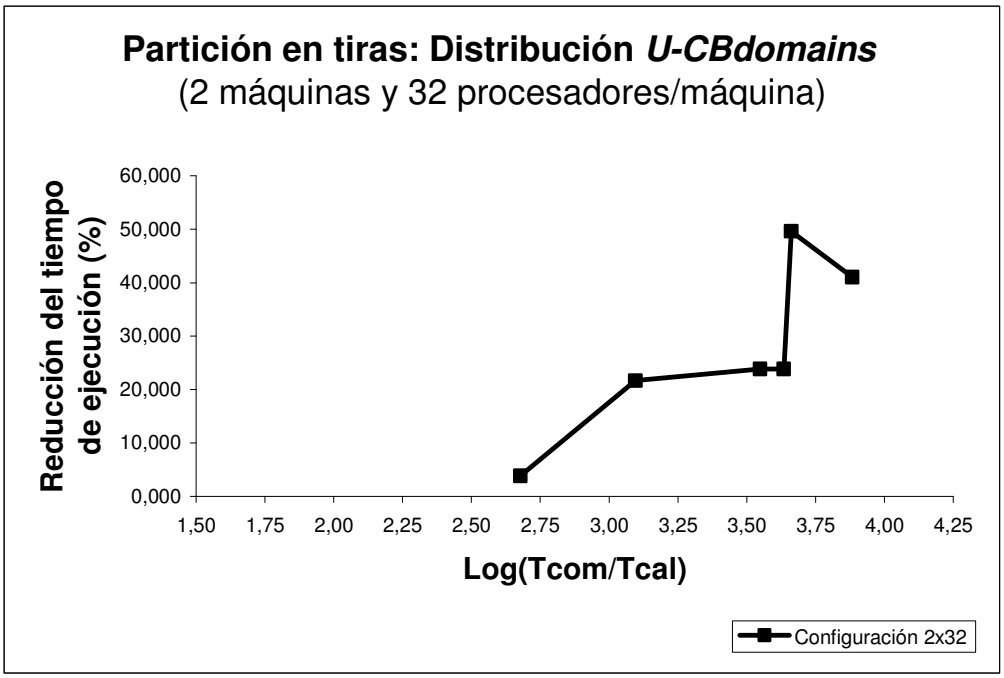


Fig. A.45 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en tiras (2 máquinas y 32 procesadores por máquina)

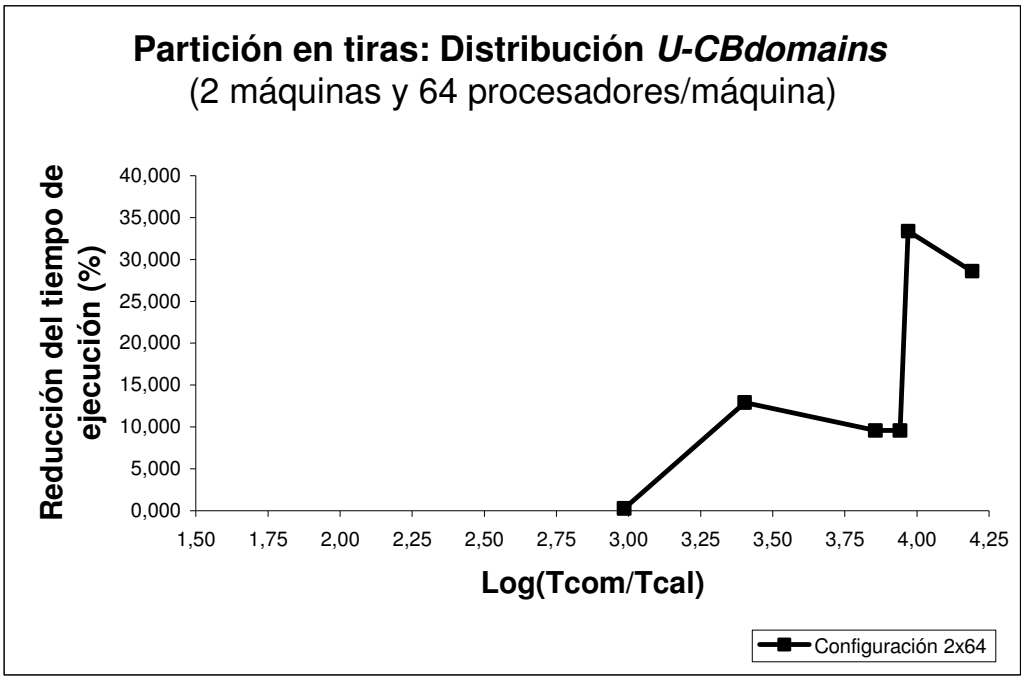


Fig. A.46 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en tiras (2 máquinas y 64 procesadores por máquina)

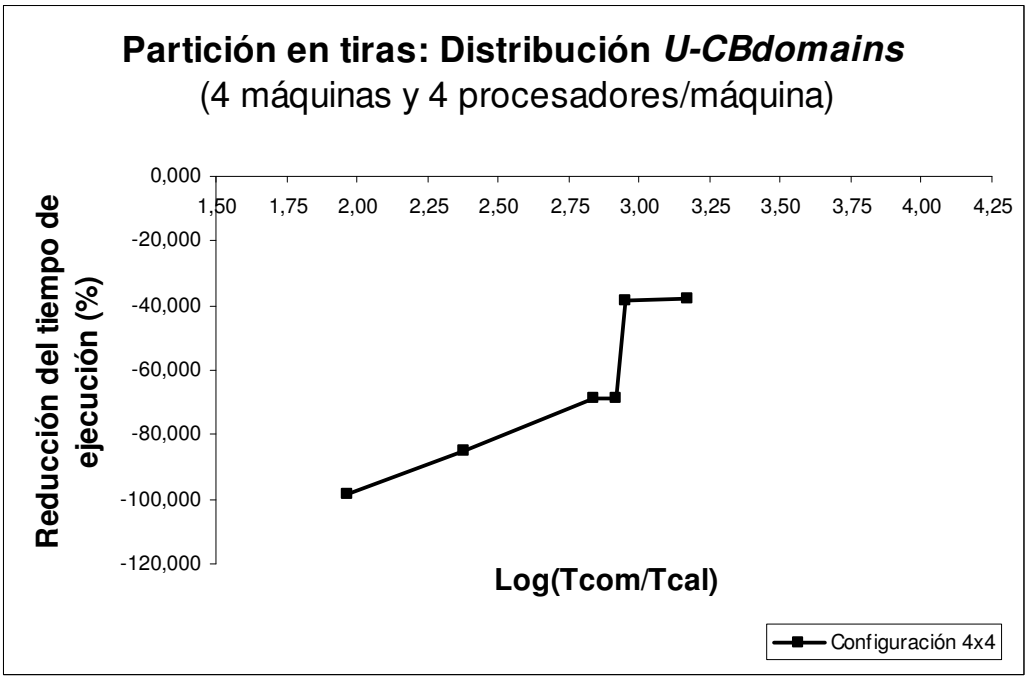


Fig. A.47 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en tiras (4 máquinas y 4 procesadores por máquina)

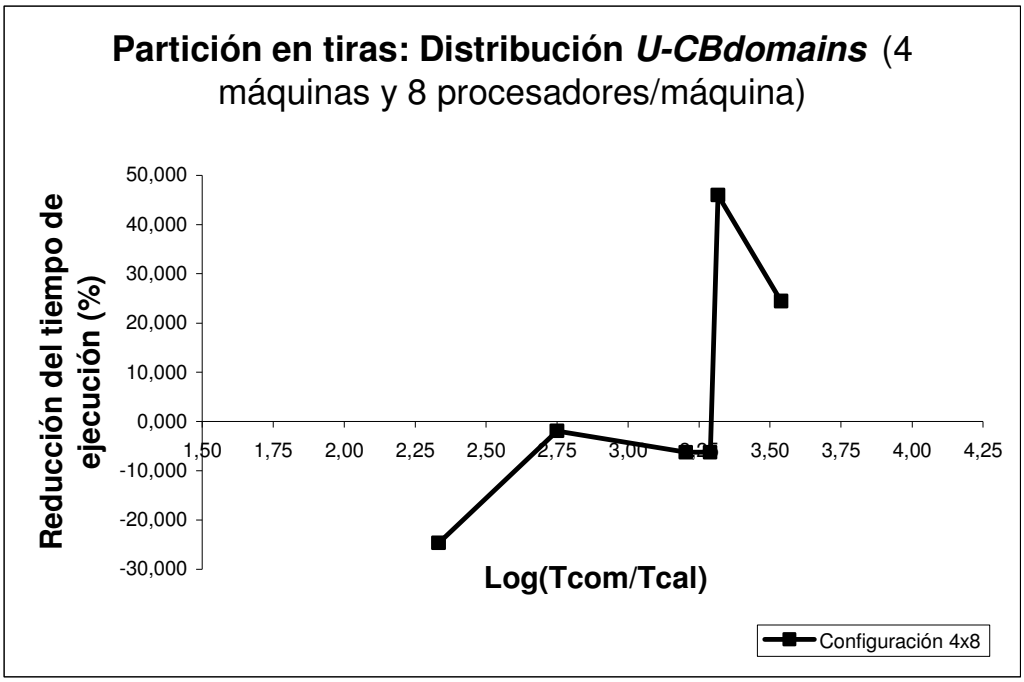


Fig. A.48 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en tiras (4 máquinas y 8 procesadores por máquina)

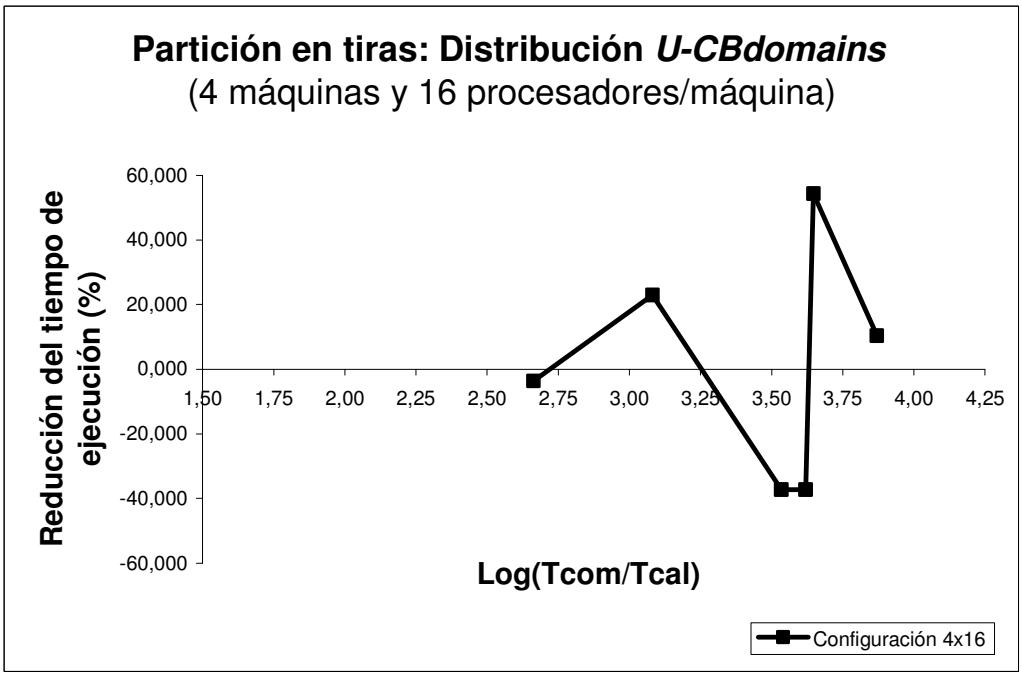


Fig. A.49 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en tiras (4 máquinas y 16 procesadores por máquina)

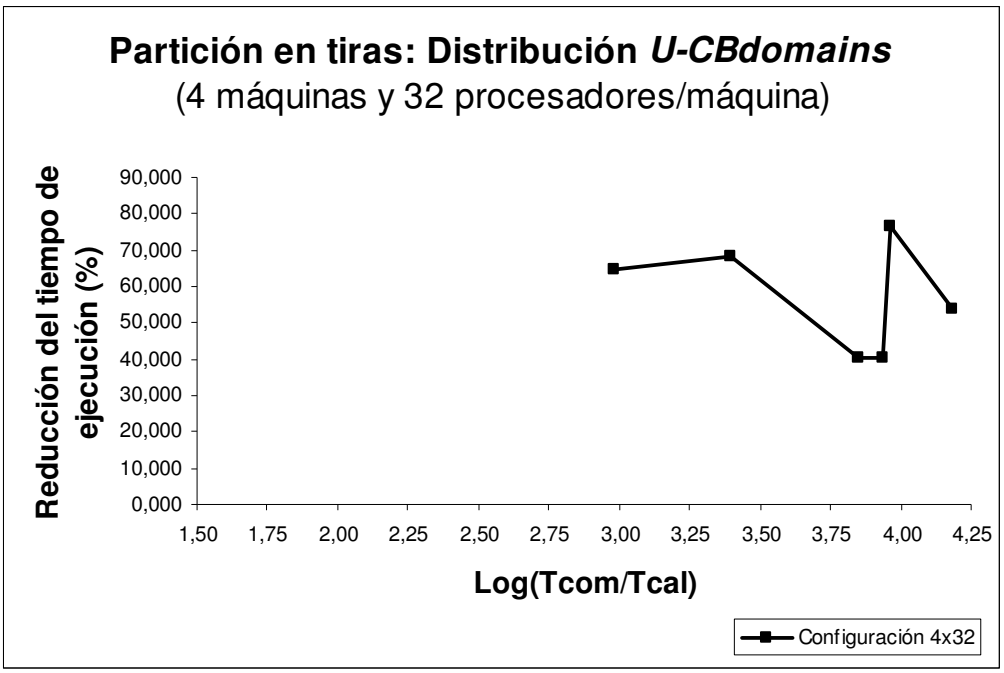


Fig. A.50 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en tiras (4 máquinas y 32 procesadores por máquina)

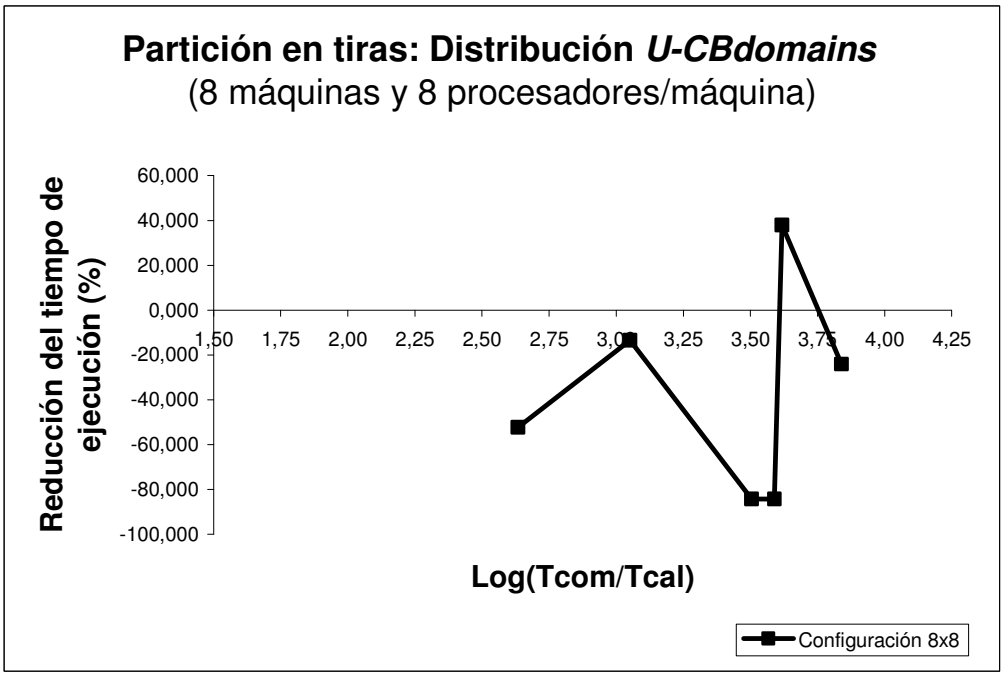


Fig. A.51 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en tiras (8 máquinas y 8 procesadores por máquina)

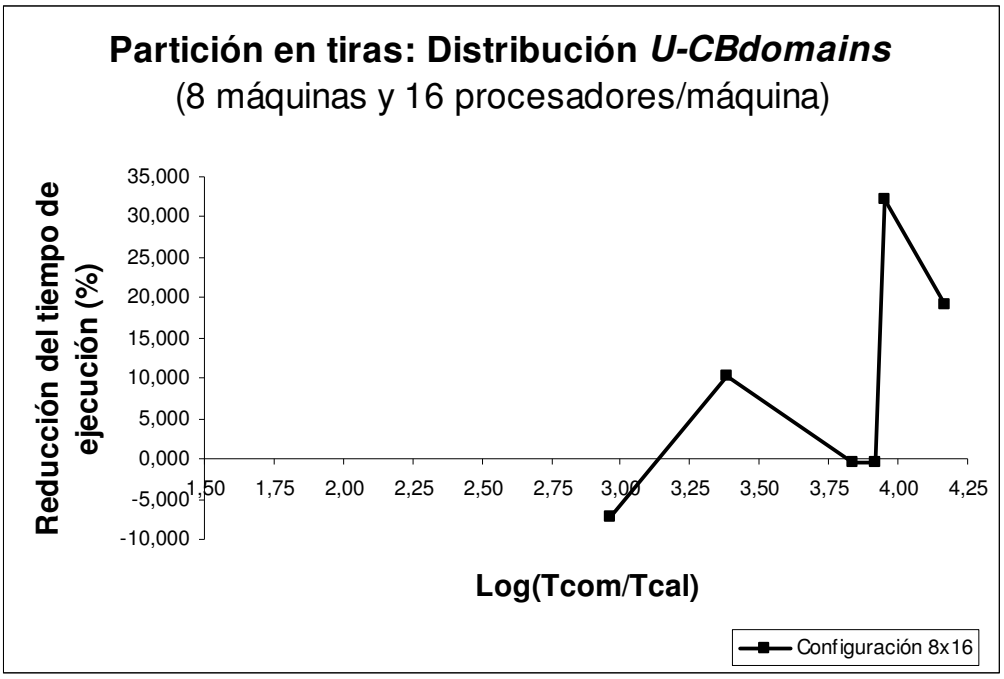


Fig. A.52 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en tiras (8 máquinas y 16 procesadores por máquina)

A.6 Distribución *U-CBdomains*: Partición en cajas

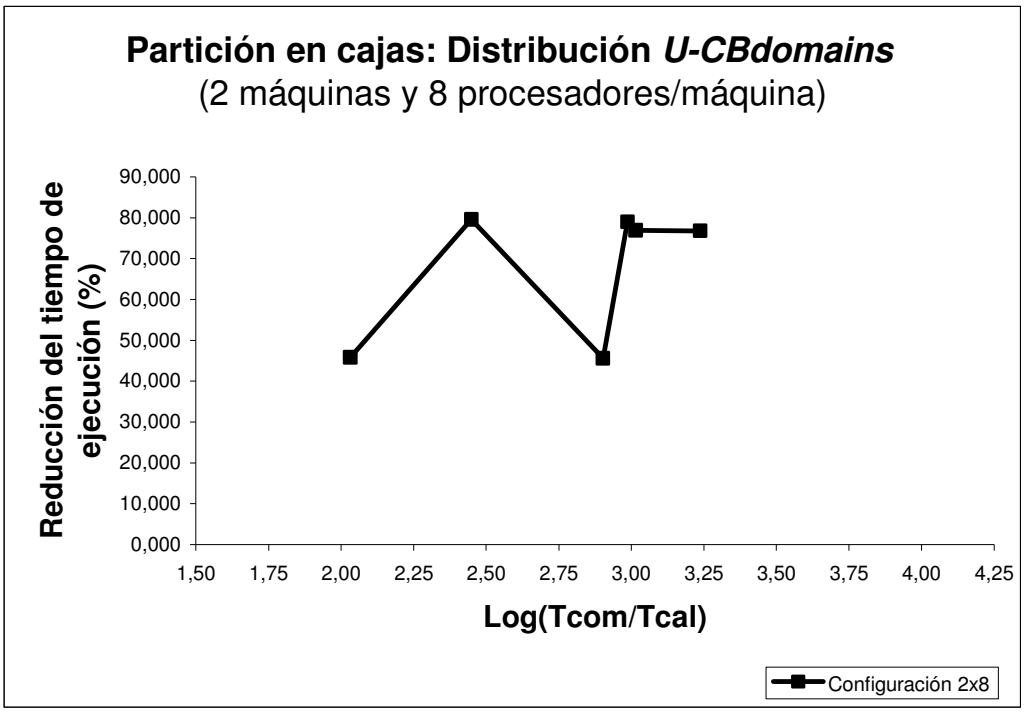


Fig. A.53 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en cajas (2 máquinas y 8 procesadores por máquina)

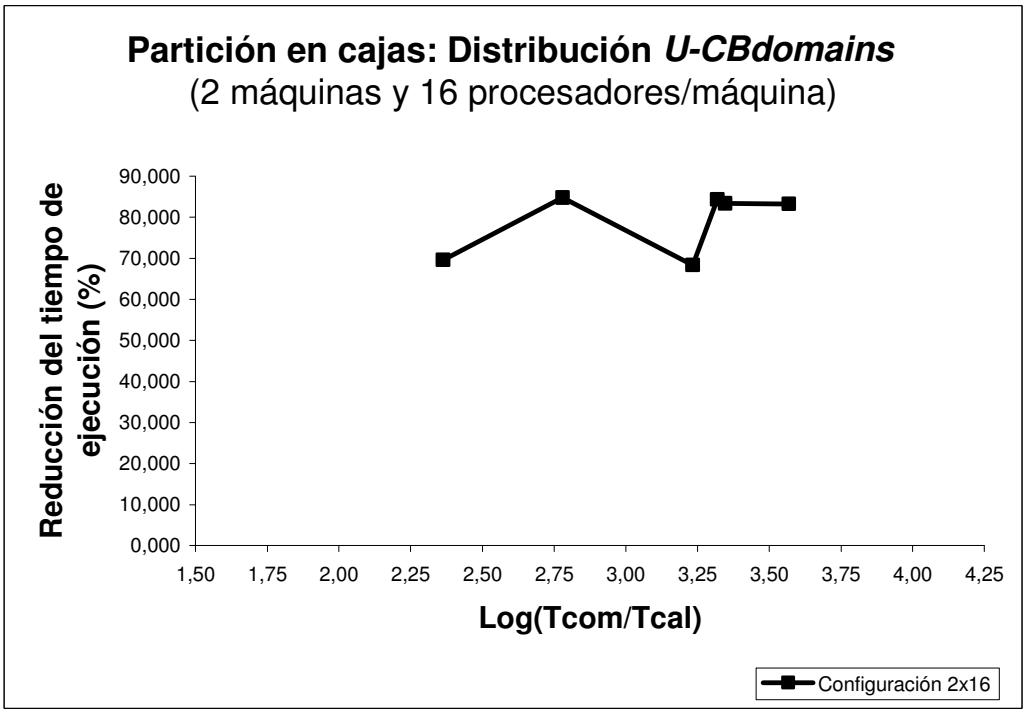


Fig. A.54 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en cajas (2 máquinas y 16 procesadores por máquina)

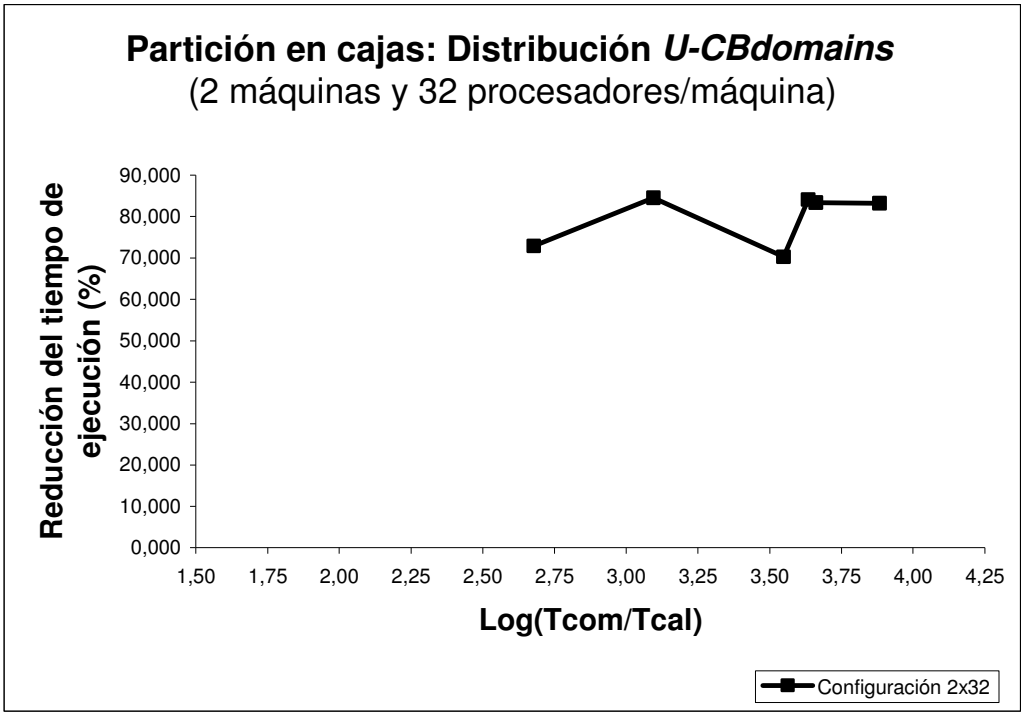


Fig. A.55 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en cajas (2 máquinas y 32 procesadores por máquina)

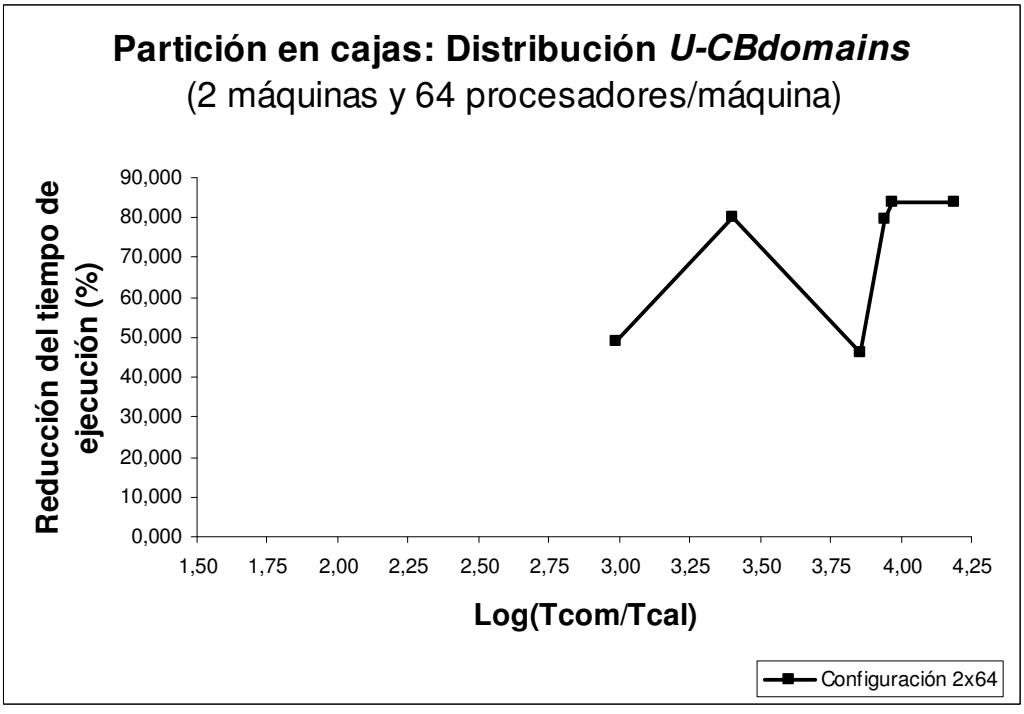


Fig. A.56 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en cajas (2 máquinas y 64 procesadores por máquina)

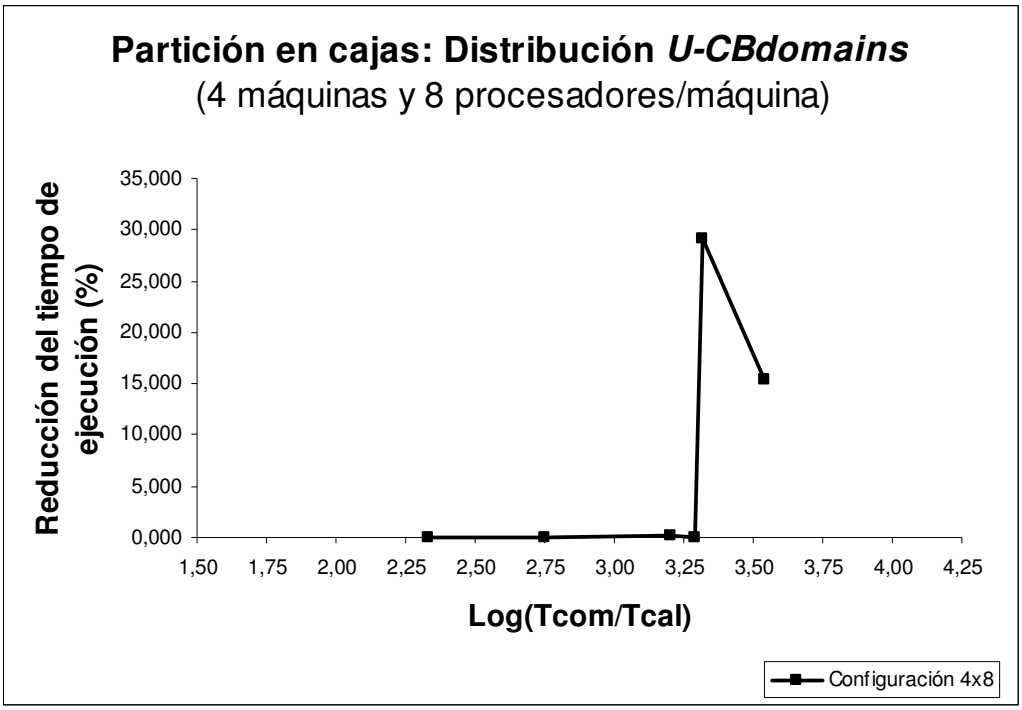


Fig. A.57 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en cajas (4 máquinas y 8 procesadores por máquina)

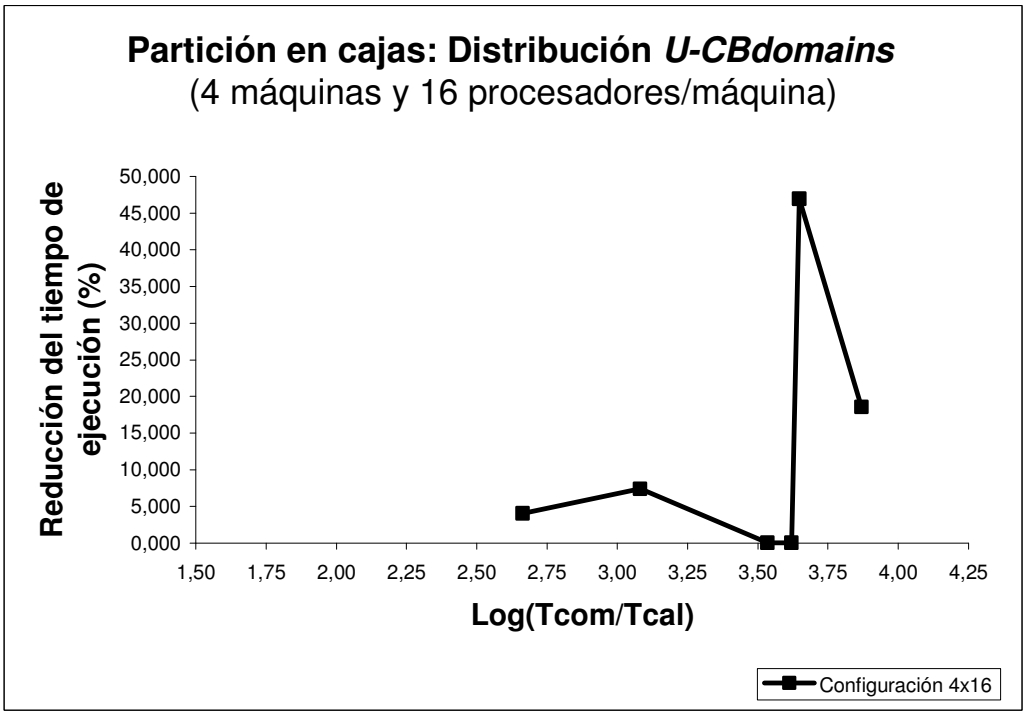


Fig. A.58 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en cajas (4 máquinas y 16 procesadores por máquina)

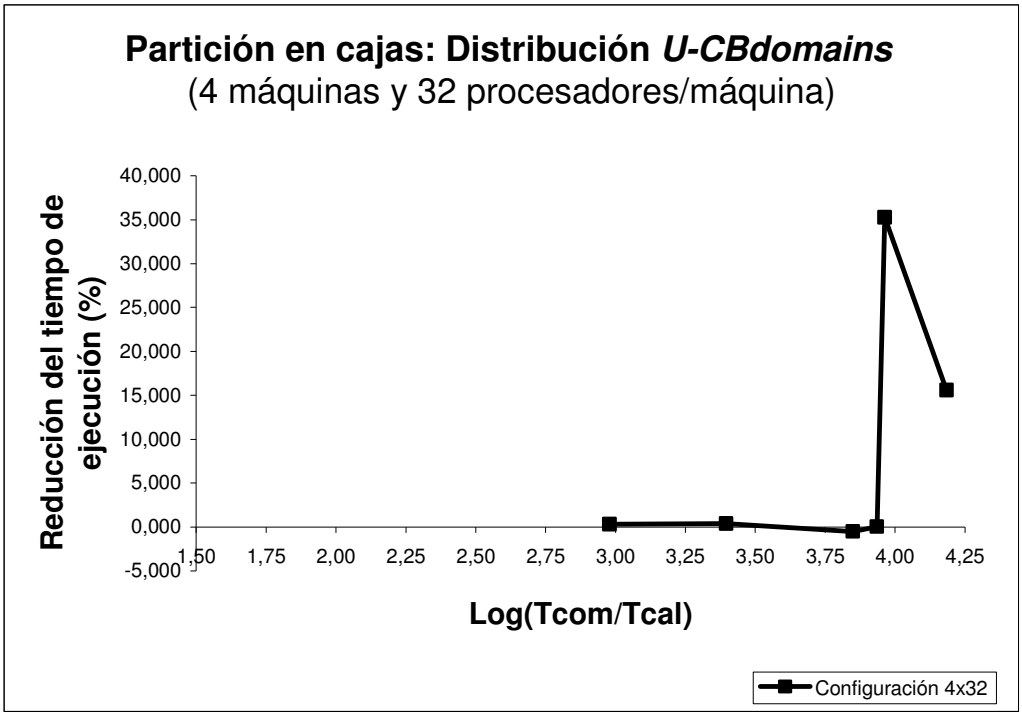


Fig. A.59 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en cajas (4 máquinas y 32 procesadores por máquina)

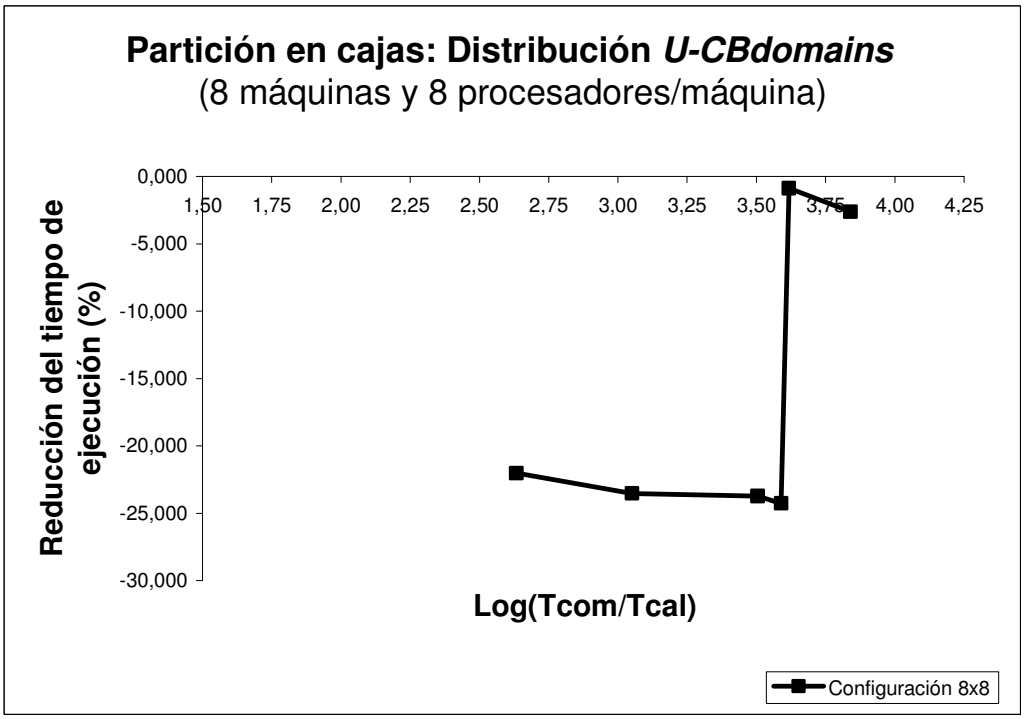


Fig. A.60 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en cajas (8 máquinas y 8 procesadores por máquina)

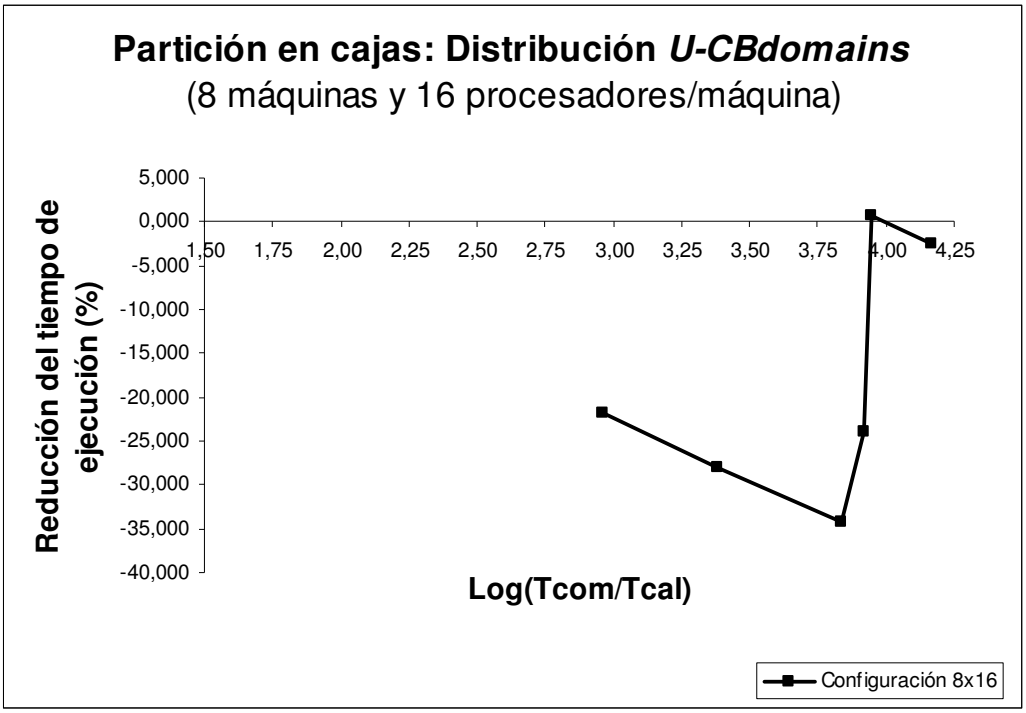


Fig. A.61 Reducción del tiempo de ejecución utilizando la distribución *U-CBdomains* para particiones en cajas (8 máquinas y 16 procesadores por máquina)

Bibliografía

Capítulo 1

[1] Bull M. and Kambites M. "JOMP: An OpenMP-like Interface for Java". Proceedings of ACM 2000 Java Grande Conference. ACM, pp. 44-53, 2000.

[2] Butenhoff D. Programming with POSIX Threads. 1st edition, Addison Wesley, 1997. ISBN 0-20-163392-2.

[3] Buyya R. High Performance Cluster Computing: Systems and Architectures. Vol. 1, Prentice Hall PTR, NJ, 1999.

[4] Corporate the MPI Forum. "MPI: A Message Passing Interface". Proceedings of the Conference on Supercomputing'93, pp. 878-883, 1993.

[5] Foster I. and Karonis N. "A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems". Proceedings of SuperComputing'98, Orlando, 1998.

[6] Foster I. and Kesselman C. The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers, Inc. San Francisco California, USA, 1999. ISBN 1-55-860475-8.

[7] Foster I. T. and Mani Chandy K. "Fortran M: A Language for Modular Parallel Programming". Journal of Parallel and Distributed Computing, Vol. 26, N° 1, pp. 24-35, 1995.

[8] Fox G., Johnson M., Lyzenga G., Otto S., Salmon J. and Walker D. "Solving Problems on Concurrent Processors". Englewood Cliffs, NJ: Prentice Hall, Vol. 1, 1988. ISBN 0-13-823022-6.

[9] Fox G., Williams R. D. and Messina P. C. Parallel Computing Works. San Francisco: Morgan Kaufmann Publishers, 1994. ISBN 1-55-860253-4.

[10] García F., Calderón A., Carretero J. "MiMPI: A Multithread-Safe Implementation of MPI". Proceedings of PVM/MPI 99, Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting, Lectures Notes on Computer Science, Vol. 1697, pp. 207-214, Springer Verlag, Barcelona, 1999.

[11] Geist A., Beguelin A., Dongarra J., Jiang W., Manchek B. and Sunderam V. "PVM: Parallel Virtual Machine". A User's Guide and Tutorial for Network Parallel Computing. Cambridge, MA: MIT Press, 1994.

[12] Grama A., Gupta A., Karypis G. and Kumar V. Introduction to Parallel Computing. Second Edition. Addison-Wesley, 2003. ISBN 0-20-164865-2.

[13] Grand Challenging Applications. http://www.mcs.anl.gov/Projects/grand-challenges/

[14] Gropp W., Lusk E. and Skjellum A. "Using MPI: Portable Parallel Programming with the Message Passing Interface". Cambridge, MA: MIT, Press, 1994.

[15] Gropp W., Lusk E., Doss N. and Skjellum A. "A High-Performance, Portable

Implementation of the MPI Message Passing Interface Standard". Parallel Computing, Vol. 22, N° 6, pp. 789-828, 1996.

[16] Hockney R. W. and Jesshope C. R. Parallel Computers: Architecture, Programming and Algorithms. Adam Hilger Ltd., Bristoll, UK, 2nd edition, 1988. ISBN 0-85274-811-6.

[17] http://www.mhpcc.hpc.mil/

[18] Karonis N., Toonen B., and Foster I. "MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface". Journal of Parallel and Distributed Computing (JPDC), Vol. 63, N° 5, pp. 551-563, 2003.

[19] Klaiber A. C. and Levy H. M. "A Comparison of Message Passing and Shared Memory Architectures for Data Parallel Programs". Proceedings of 21st Annual International Symposium on Computer Architecture, Chicago, US, pp. 94-105, 1994.

[20] Kleiman S. et al. Programming with Threads. 1st edition, Prentice Hall, 1996. ISBN 0-13-172389-8.

[21] Lewis B. and Berg D. Threads Primer. 1st edition, Prentice Hall, 1995. ISBN 0-13-443698-9.

[22] Li K. and Hudak P. "Memory Coherence in Shared Virtual Memory Systems". ACM Transactions on Computer Systems, Vol. 7, N° 4, pp. 321-359, 1989.

[23] Loveman D. "High-Performance Fortran". IEEE Computer Science Parallel and Distributed Technology, Vol. 1, N° 1, 1993.

[24] MPI 1.1 Standard. http://www-unix.mcs.anl.gov/mpi/mpich.

[25] MPI–2: Extensions to the MPI Interface. http://www-unix.mcs.anl.gov/mpi/mpich.

[26] Nichols B. et al. Pthreads Programming. 1st edition, O'Reilly and Associates, 1996. ISBN 1-56-592115-1.

[27] Nitzberg B. and Lo V. "Distributed Shared Memory: A Survey of Issues and Algorithms". IEEE Computer Science, Vol. 24, N° 8, pp. 52-60, 1991.

[28] OpenMP Architecture Review Board. "OpenMP C and C++ Application Program Interface, Version 1.0". 1998. http://www.openmp.org.

[29] Pittsburgh Supercomputer Center. "Parallel Programming Techniques: OpenMP". http://www.mhpcc.edu/training/workshop2/openMP/MAIN.html.

[30] Singh J. P., Joe T., Hennessy J. L. and Gupta A. "An Empirical Comparison of the Kendall Square Research KSR-1 and the Stanford DASH Multiprocessors". Proceedings of Supercomputing'93, pp. 214-225, Portland, OR, Nov. 1993.

[31] Sunderam V. S. "PVM: A Framework for Parallel Distributed Computing", Concurrency: Practice and Experience, Vol. 2, N° 4, pp. 315-339, 1990.

[32] Wilson G. V. "A Glossary of Parallel Computing Terminology". IEEE Parallel and Distributed Technology: Systems and Applications, Vol. 1, N° 1, pp. 52-67, 1993.

[33] Zomaya A. Y. "Parallel and Distributed computing: The Scene, The Props, The Players". Parallel and Distributed Computing Handbook Chapter 1, pp. 5-23. ISBN 0-07-073020-2.

Capítulo 2

[1] Abramson D., Giddy J. and Kotler L. "High Performance Parametric Modelling with Nimrod/G: Killer Application for The Global Grid?". Proceedings of 14th International Parallel and Distributed Processing Symposium (IPDPS). IEEE Computer Society Press, pp. 520-528, 2000.

[2] Abramson D., Sosic R., Giddy J. and Hall B. "Nimrod: A Tool for Performing Parameterised Simulations Using Distributed Workstations". Proceedings of 4th IEEE Symposium on High Performance Distributed Computing (HPDC). IEEE Computer Society Press, Washington, DC, USA, pp. 112-121, 1995.

[3] Akarsu E., Fox G. C., Furmanski W. and Haupt T. "WebFlow: High-Level Programming Environment and Visual Authoring Toolkit for High Performance Distributed Computing". Proceedings of SC98: High Performance Networking and Computing, Orlando, FL, 1998.

[4] Allen G., Dramlitsch T., Foster I., Karonis N., Ripeanu M., Ed. Seidel and B. Toonen. "Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus, Winning Paper for Gordon Bell Prize". Special Category, Proceedings of Supercomputing'01. Denver, 2001.

[5] Almond J. and Snelling D. "UNICORE: Uniform Access to Supercomputing as an Element of Electronic Commerce". Future Generation Computer Systems, N° 15, pp. 539-548, 1999.

[6] Armstrong R. et al. "Toward a Common Component Architecture for High Performance Scientific Computing". Proceedings of 8th High Performance Distributed Computing (HPDC-99), pp. 115-124, 1999.

[7] Baldeschwieler J., Blumofe R. and Brewer E. "ATLAS: An Infrastructure for Global Computing". Proceedings of 7th ACM SIGOPS European Workshop on System Support for Worldwide Applications. Ireland, pp. 165-172, 1996.

[8] Baratloo A., Karaul M., Kedem Z. and Wyckoff P. "Charlotte: Metacomputing on the Web". Proceedings of 9th Conference on Parallel and Distributed Computing Systems (PDCS'96). France, pp. 181-188, 1996.

[9] Berman F., Fox G. and Hey T. "The Grid: Past, Present, Future". Grid Computing: Making the Global Infrastructure a Reality. John Wiley & Sons, Chapter 1, pp. 09-50, ISBN: 0-470-85319-0, 2003.

[10] Booth S. and Mourao E. "Single Sided MPI Implementations for SUN MPI". Proceedings of Supercomputing'00. Dallas, 2000.

[11] Bote-Lorenzo M. L., Dimitriadis Y. A. and Gómez-Sánchez E. "Grid Characteristics and Uses: a Grid Definition". Postproceedings extended and revised version. 1st

European Across Grids Conference, ACG'03, Springer-Verlag, Lecture Notes Computer Science N° 2970. Santiago de Compostela, Spain, pp. 291-298, 2004.

[12] Browne J. "Software Engineering of Parallel Programs in a Computationally Oriented Display Environment". In D. Gelernter, A. Nicolau and D. Padua, eds. Languages and Compilers for Parallel Computing. Cambridge, MA: MIT Press, pp. 75-94, 1990.

[13] Buyya R., Abramson D. and Giddy J. "Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid". The 4th International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2000), Beijing, China, pp. 283-289, 2000.

[14] Cactus Webmeister, The Cactus Code Website, 2000. http://www.CactusCode.org/

[15] Camiel N., London S., Nisan N. and Regev O. "The POPCORN Project: Distributed Computation over The Internet in Java". Proceedings of 6th International World Wide Web Conference, 1997.

[16] Carpenter B. et al. "MPJ: MPI-Like Message-Passing for Java". Concurrency: Practice and Experience, Vol. 12, N° 11, pp. 1019-1038, 2000.

[17] Catlett C. and Smarr L. "Metacomputing". Communications of the ACM, Vol. 35, N° 6, pp. 44-52, 1992.

[18] Cheng G., Fox G. and Mills K. "Integrating Multiple Programming Paradigms on Connection Machine CM-5 in a Dataflow-Based Software Environment". Technical Report, Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY, 1993.

[19] Clark D. "Face-to-Face with Peer-to-Peer networking". Computer, Vol. 34, N° 1, pp. 18-21, 2001.

[20] CORBA 3 Release Information, The Object Management Group, 2000. http://www.omg.org/technology/corba/corba3releaseinfo.htm

[21] De Roure D., Baker M. A., Jennings N. R. and Shadbolt N. R. "The Evolution of The Grid". Grid Computing: Making the Global Infrastructure a Reality. John Wiley & Sons. Chapter 3, pp. 65-100. ISBN: 0-470-85319-0, 2003.

[22] EU DataGrid Project. http://www.eu-datagrid.org/

[23] FAFNER. http://www.npac.syr.edu/factoring.html

[24] Fischer L. The Workflow Handbook. Lighthouse Point, FL: Future Strategies, Inc., 2002.

[25] Foster I. and Karonis N. T. "A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems". Proceedings of Supercomputing'98. IEEE, Orlando, FL, 1998. http://www.supercomp.org/sc98

[26] Foster I. and Kesselman C. "Globus: A Metacomputing Infrastructure Toolkit". International Journal of Supercomputer Applications, Vol. 11, N° 2, pp. 115-128, 1997.

[27] Foster I. and Kesselman C. The Grid: Blueprint for a New Computing Infrastructure. San Francisco, CA: Morgan Kaufmann Publishers, ISBN 1-55860-475-8, 1998.

[28] Foster I. and Kesselman C. "Computational Grids". Chapter 2 of the Grid: Blueprinf for a New Computing Infrastructure, pp. 15-51, Morgan-Kaufmann, 1989.

[29] Foster I., Geisler J., Gropp W., Karonis N., Lusk E., Thiruvathukal G. and Tuecke S. "A Wide-Area Implementation of the Message Passing Interface". Parallel Computing, 1998.

[30] Foster I., Geisler J., Kesselman C. and Tuecke S. "Managing Multiple Communication Methods in High-Performance Networked Computing Systems". Journal of Parallel and Distributed Computing, Vol. 40, pp. 35-48, 1997.

[31] Foster I., Geisler J., Nickless W., Smith W. and Tuecke S. "Software Infrastructure for the I-WAY High Performance Distributed Computing Experiment". Proceedings 5th IEEE Symposium on High Performance Distributed Computing, pp. 562-571, 1997.

[32] Foster I., Kesselman C. and Tuecke S. "The Nexus Approach to Integrating Multithreading and Communication". Journal of Parallel and Distributed Computing, Vol. 37, pp. 70-82, 1996.

[33] Foster I., Kesselman C. and Tuecke S. "The Anatomy of the Grid: Enabling Scalable Virtual Organizations". International Journal of Supercomputer Applications and High Performance Computing, Vol. 15, N° 3, 2001.

[34] Foster I., Kesselman C., Nick J. and Tuecke S. "The Physiology of the Grid: Open Grid Services Architecture for Distributed Systems Integration". IEEE Computer, pp. 37-46, 2002.

[35] Frisch N., Rose D., Sommer O., and Ertl Th. "Visualization and Pre-Processing of Independent Finite Element Meshes for Car Crash Simulations". The Visual Computer, Vol. 18, N° 4, pp. 236-249, 2002.

[36] Furmento N., Mayer A., McGough S., Newhouse S., Field T. and Darlington J. "An Integrated Grid Environment for Component Applications". Proceedings of 2nd International Workshop on Grid Computing (Grid 2001), Lecture Notes Computer Science, Vol. 2242, pp. 26-37, 2001.

[37] Gabriel E., Resch M., Beisel T. and Keller R. "Distributed Computing in a Heterogeneous Computing Environment". Recent Advances in Parallel Virtual

Machine and Message Passing Interface. Proceedings of 5th European PVM/MPI Users' Group Meeting, Springer, pp. 180-188, Liverpool, UK, 1998.

[38] Getov V., von Laszewski G., Philippsen M. and Foster I. "Multi-Paradigm Communications in Java for Grid Computing". Communications of the ACM Vol. 44, N° 10, pp. 118-125, 2001.

[39] Gong L. "JXTA: A Network Programming Environment". IEEE Internet Computing, Vol. 5, N° 3, pp. 88-95, 2001.

[40] Gong L. "Introduction: Peer-to-Peer Networks in Action". IEEE Internet Computing, Vol. 6, N° 1, pp. 37-39, 2002.

[41] http://dsonline.computer.org/0201/ic/w102gei.htm

[42] Grimshaw A. et al. "The Legion Vision of a Worldwide Virtual Computer". Communications of the ACM, Vol. 40, N° 1, pp. 39-45, 1997.

[43] Grimshaw A. S., Ferrari A. J., Lindahl G. and Holcomb K. "Metasystems". Communications of the ACM, Vol. 41, N° 11, pp. 46-55, 1998.

[44] Haupt T., Akarsu E., Fox G. and Furmanski W. "Web Based Metacomputing". Special Issue on Metacomputing. Future Generation Computer Systems. New York: North Holland, 1999.

[45] Hempel R. "Advanced Environments and Tools for High Performance Computing-Review of Progress since Last Meeting". EURESCO Conference 2003-139, 2003.

[46] HotPage. https://hotpage.npaci.edu/
http://www-4.ibm.com/software/solutions/Webservices/pdf/WSFL.pdf

[47] Imamura T., Tsujita Y., Koide H. and Takemiya H. "Architecture of Stampi: MPI Library on a Cluster of Parallel Computers". Recent Advances in Parallel Virtual Machine and Message Passing Interface. Lecture Notes in Computer Science, Vol. 1908. Proceedings of 7th European PVM/MPI Users' Group Meeting, Lake Balaton, Hungary, pp. 200-207, 2000.

[48] Jacobsen, H. A. et al. High Performance CORBA Working Group, 2001. www.omg.org/realtime/working_groups/high_performance_corba.html

[49] JINI. http://www.jini.org

[50] JXTA. http://www.jxta.org/

[51] Kielmann T., Hofman R. F. H., Bal H. E., Plaat A. and Bhoedjang R. A. F. "MagPle: MPI's Collective Communication Operations for Clustered Wide Area Systems". Proceedings of 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), Atlanta, GA, pp. 131-140, 1999.

[52] Lee C. and Talia D. "Grid Programming Models: Current Tools, Issues and Directions". Grid Computing: Making the Global Infrastructure a Reality. John Wiley & Sons. Chapter 1, pp. 555-578. ISBN: 0-470-85319-0, 2003.

[53] Lee C., Mtsuoka S., Talia D., Sussman A., Mueller M., Allen G. and Saltz J. "A Grid Programming Primer". Global Grid Forum. 2001.

[54] http://www.gridforum.org/7_APM/APS.htm

[55] Legion. http://legion.virginia.edu/

[56] Lewis M. and Grimshaw A. "The Core Legion Object Model". Proceedings of the 5th International Symposium of High Performance Distributed Computing (HPDC'96). IEEE Computer Society, Washington DC, USA, pp. 89-99, 1996.

[57] Litzkow M., Livny M. and Mutka M. W. "Condor a Hunter of Idle Workstations". Proceedings of 8th International Conference of Distributed Computing Systems (ICDCS), San José, California, pp. 104-111, 1988.

[58] Marinescu D. and Lee C. Process Coordination and Ubiquitous Computing. Boca Raton, FL: CRC Press, 2002.

[59] Mechoso C., Ma C. C., Farrara J., Spahr J. and Moore R. "Parallelization and Distribution of a Coupled Atmosphere-Ocean General Circulation Model". Monthly Weather Review, Vol. 121, N° 7, pp. 2062, 1993.

[60] Message Passing Interface Forum, MPI: A Message Passing Interface Standard, 1995. http://www.mpi-forum.org/

[61] Message Passing Interface Forum, MPI-2: Extensions to the Message Passing Interface, 1997. http://www.mpi-forum.org/

[62] Nakada H., Matsuoka S., Seymour K., Dongarra J., Lee C. and Casanova H. "Overview of GridRPC: A Remote Procedure Call API for Grid Computing". 3rd International Workshop on Grid Computing. Lecture Notes Computer Science, Vol. 2536, pp. 274-278, 2002.

[63] NLANR Grid Portal Development Kit. http://www.nlanr.net

[64] Norman M., Beckman P., Bryan G., Dubinski J., Gannon D., Hernquist L., Keahey K., Ostriker J., Shalf J., Welling J. and Yang S. "Galaxies Collide on the I-WAY: An Example of Heterogeneous Wide Area Collaborative Supercomputing". International Journal of Supercomputer Applications, Vol. 10, N° 2, pp. 131-140, 1996.

[65] NSF Grand Challenge as part of 1997 HPCC Implementation Plan.

[66] OpenMP Consortium. OpenMP C and C++ Application Program Interface, Version 1.0, 1997.

[67] Platform Computing. http://www.platform.com

[68] Portable Batch System. http://www.openpbs.org/

[69] Rajasekar A. K. and Moore R. W. "Data and Metadata Collections for Scientific Applications". Proceedings of 9th International Conference of High Performance Computing Europe. Lecture Notes Computer Science, Vol. 2110, Amsterdam, Holland, pp. 72-80, 2001.

[70] Rana O. F., Getov V. S., Newhouse E. and Allan R. "Building Grid Services with Jini and JXTA". GGF2 Working Document, 2002.

[71] Rasure J. R. and Kubica S. "The Khoros Application Development Environment". Experimental Environments for Computer Vision and Image Processing. Singapore: World Scientific, pp. 1-32, 1994.

[72] Saelee D. and Rana O. F. "Implementing Services in a Computational Grid with Jini and Globus". Proceedings of 1st EuroGlobus workshop, 2001.

[73] Sato M., Hirono M., Tanaka Y. and Sekiguchi S. "OmniRPC: a Grid RPC Facility for Cluster and Global Computing in OpenMP, WOMPAT". Lecture Notes Computer Science, Vol. 2104. West Lafayette, IN: Springer Verlag, pp. 130-136, 2001.

[74] SDSC GridPort Toolkit. http://gridport.npaci.edu/

[75] SDSC Storage Resource Broker. http://www.npaci.edu/DICE/SRB/

[76] Skillicorn D. and Talia D. "Models and Languages for Parallel Computation". ACM Computing Surveys, Vol. 30, N° 2, pp. 123-169, 1998.

[77] Skillicorn D. B. "Motivating Computational Grids". Proceedings of 2nd IEEE/ACM International Symposium on Cluster Computing and Grid, pp. 371-376, 2002.

[78] Smith L. and Bull M. "Development of Mixed Mode MPI/OpenMP Applications". Scientific Programming, Vol. 9, N° 2-3, pp. 83-96, 2001.

[79] Sosnowski W. "Flow Approach-Finite Element Model for Stamping Processes Versus Experiment". Computer Assisted Mechanics and Engineering Sciences, Vol. 1, pp. 49-75, 1994.

[80] Sun Grid Engine. http://www.sun.com/software/Gridware/

[81] The Condor Project. http://www.cs.wisc.edu/condor

[82] The Foundation for Physical Agents. http://www.fipa.org

[83] The Globus Project Web Site. http://www.globus.org

[84] Tolksdorf R. Models of Coordination and Web-Based Systems, in Marinescu, D. and Lee, C. (Eds.). Process Coordination and Ubiquitous. Boca Raton, FL: CRC Press, 2002.

[85] UNICORE. http://www.unicore.de/

[86] Vahdat A., Eastham P. and Anderson T. "WebFS: A Global Cache Coherent Filesystem". Technical Report, Department of Computer Science, UC Berkeley, 1996.

[87] Web Services Flow Language (WSFL), version 1.0.

[88] Web Services for Business Process Design. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

Capítulo 3

[1] Aparicio Sánchez C. A. "Estudo de Impacto Usando Elementos Finitos e Análise Não Linear". Tesis de Mestre em Engenharia Mecânica. Universidade de São Paulo. Escola de Engenharia de São Carlos. Departamento de Engenharia Mecânica. 143 pp., 2001.

[2] Badía R. M., Escalé F., Gabriel E., Giménez J., Keller R., Labarta J., Müller M. S. "Performance Prediction in a Grid Environment". Proceedings of 1st European Across Grid Conference, Santiago de Compostela, Spain 2003.

[3] Badía R. M., Escalé F., Giménez J. and Labarta J. "DAMIEN: Distributed Applications and Middleware for Industrial Use of European Networks D5.3/CEPBA". IST-2000-25406.

[4] Badía R. M., Giménez J., Labarta J., Escalé F. and Keller R. "DAMIEN: Distributed Applications and Middleware for Industrial Use of European Networks D5.2/CEPBA". IST-2000-25406.

[5] Badía R. M., Labarta J., Giménez J., Escalé F. "Dimemas: Predicting MPI Applications Behavior in Grid Environments". Workshop on Grid Applications and Programming Tools (GCF8), 2003.

[6] Bardillo O. and Badía R. M. "DIMEMAS Graphic User Interface Manual". http://www.cepba.upc.es/dimemas/

[7] Barnard S. T. and Simon H. D. "A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems". Concurrency: Practice and Experience, Vol. 6, N° 2, pp. 101-117, 1994.

[8] Bathe K. J. Finite Element Procedures. Prentice Hall, New Jersey, 1996.

[9] Bathe, K. J. Finite Element Procedures in Engineering Analysis. Prentice-Hall, Englewood Cliffs, N. J., 1982, ISBN: 0-13317-305-4.

[10] Belytschko T. and Hughes T. J. Computational Methods for Transient Analysis. North-Holland, Amsterdam, 1983, ISBN: 044-48-6479-2.

[11] Benson D. J. "Computational Methods in Lagrangian and Eulerian Hydrocodes". Computer Methods in Applied Mechanics and Engineering, Vol. 99, N° 2-3, pp. 235-394, 1992.

[12] Berger M. and Bokhari S. "A Partitioning Strategy for Nonuniform Problems on Multiprocessors". IEEE Transactons on Computers, Vol. 36, N° 5, pp. 570-580, 1987.

[13] Brewer J. C. "Effects of Angles and Offsets in Crash Simulations of Automobiles with Light Trucks". Proceedings of 17th International Technical Conference on the Enhanced Safety of Vehicles, paper N° 308, Amsterdam, 2001.

[14] Burnett D. S. Finite Element Analysis from concepts to applications. AT&T Bell Laboratories. Addison-Wesley, Publishing Compañy, New Jersey, 1987. ISBN 0-201-10806-2.

[15] Damianakis S. N., Bilas A. and Felten E. W. "Reducing Waiting Costs in User-Level Communication". Proceedings of 11th International Parallel Processing Symposium, Switzerland, pp. 381-387, 1997.

[16] Diekman R., Meyer D. and Monien B. "Parallel Decomposition of Unstructured FEM-Meshes". Proceedings of 2nd International Workshop on Parallel Algorithms for Irregular Structured Problems. Lecture Notes in Computer Science, Vol. 980, pp. 199-215, 1995.

[17] Dimemas, Internet, 2002. http://www.cepba.upc.es/dimemas/

[18] Donea J. Advanced Structural Dynamics. Pergamon, 1980, ISBN: 0-85334-859-6.

[19] Farhat C. "A Simple and Efficient Automatic FEM Domain Decomposer". Computers & Structures, Vol. 28, N° 5, pp. 579-602, 1988.

[20] Farhat C. "On the Mapping of Massively Parallel Processors onto Finite Element Graphs". Computers and Structures, Vol. 32, N° 2, pp. 347-353, 1989.

[21] Frisch N., Rose D., Sommer O. and Ertl Th. "Visualization and Pre-Processing of Independent Finite Element Meshes for Car Crash Simulations". The Visual Computer, Vol. 18, N° 4, pp. 236-249, 2002.

[22] Garey M. R. and Johnson D. S. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, New York, 1979, ISBN: 0-71671-045-5.

[23] Girona S., Labarta J. and Badía R. M. "Validation of Dimemas Communication Model for MPI Collective Operations". Lecture Notes in Computer Science, Vol. 1908, EuroPVM/MPI, Hungary, 2000.

[24] Goicolea J. M. "Análisis Termomecánico No Lineal Mediante Métodos Explícitos de Diferencias Finitas y Elementos Finitos". Revista Internacional de Métodos Numéricos para Cálculo y Diseño en Ingeniería, Vol. 8, N° 3, pp. 235-265, 1992.

[25] Goicolea J. M. "Estructuras Sometidas a Impacto". Escuela Técnica Superior de Ingenieros de Caminos, Universidad Politécnica de Madrid, EAD 2000, España.

[26] Hendrickson B. and Leland R. "A Multilevel Algorithm for Partitioning Graphs". Proceedings of Supercomputing'95, San Diego, California, ACM Press, New York, 1995.

[27] Hendrickson B. and Leland R. "The Chaco User's Guide, Version 2.0". Technical Report SAND95-2344, Sandia National Laboratories, Albuquerque, New Mexico, 1995.

[28] http://www.csgnetwork.com/bandwidth.html

[29] http://clik.to/adslvelocidad

[30] http://geant.net/server/show/nav.128

[31] http://www.intel.com/network/connectivity/emea/eng/resources/doc_library/data_sheets/np2038.pdf

[32] http://staffweb.cms.gre.ac.uk/~c.walshaw/jostle

[33] http://www.cs.sandia.gov/~bahendr/chaco.html

[34] http://www.force10networks.com/products/test-336p1.asp

[35] http://www.myri.com/myrinet/overview/

[36] http://www.npac.syr.edu/NPAC1/PUB/ranka/part/part.html

[37] http://www.rediris.es/red/

[38] http://wwwcs.uni-paderborn.de/fachbereich/AG/monien/RESEARCH/PART/party.html

[39] Huang S., Aubanel E. and Bhavsar V. C. "Mesh Partitioners for Computational Grids: A Comparison". Proceedings of the 2003 International Conference on Computational Science and Its Applications (ICCSA 2003), Montreal, Canada, Lecture Notes in Computer Science, Vol. 2269, Springer Verlag, pp. 60-68, 2003.

[40] Hughes T. J. "Analysis of Transient Algorithms with Particular Reference to Stability Behaviour". In Belytschko, Chapter 2. Reference [10].

[41] Hughes, T. J. R. The Finite Element Method: Linear Static and Dynamic Finite Element Analysis. Dover Publications, 2000, ISBN: 0-48641-181-8.

[42] Izaguirre Paz J. A. "Evaluation of Parallel Domain Decomposition Algorithms". Department of Computer Science and Beckman Institute, University of Illinois at Urbana-Champaign, Computer Science Department, USA, pp. 1-7, 1997.

[43] Karypis G. "Multi-Constraint Mesh Partitioning for Contact/Impact Computations". Proceedings of Supercomputing'03, ACM Press, Phoenix, Arizona, 2003.

[44] Karypis G. and Kumar V. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". SIAM Journal on Scientific Computing, Vol. 20, N° 1, pp. 359-392, 1998.

[45] Karypis G. and Kumar V. "METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices". Version 4.0, Technical Report, Department of Computer Science, University of Minnesota, Army HPC Research Center, Minneapolis, 1998.

[46] Karypis G. and Kumar V. "Parallel Multilevel K-way Partitioning Scheme for Irregular Graphs". SIAM Review, Vol. 41, N° 2, pp. 278-300, 1999.

[47] Kernighan B. and Lin L. "An Effective Heuristic Procedure for Partitioning Graphs". Bell Systems Technical Journal, pp. 291-308, 1970.

[48] Keyes D. E. "Domain Decomposition Methods in the Mainstream of Computational Science". Proceedings of 14th International Conference on Domain Decomposition Methods, Mexico City, pp. 79-93, 2003.

[49] Kirkpatrick S. W. and Mac Neill R. A. "Development of a Computer Model for Prediction of Collision Response of a Railroad Passenger Car". Proceedings of the 2002 ASME/IEEE Joint Rail Conference, Washington, DC, pp. 9-16, 2002.

[50] Kirkpatrick S. W., Simons J. W. and Antoun T. H. "Development and Validation of High Fidelity Vehicle Crash Simulation Models". International Journal of Crashworthiness, Vol. 4, N° 4, 1999.

[51] Kirkpatrick S., Gelatt C. and Vecchi M. "Optimization by Simulated Annealing". Science, pp. 220-671, 1983.

[52] Komatitsch D. and J. P. Vilotte. "The Spectral-element Method: An efficient Tool to Simulate the Seismic Response of 2D and 3D Geological Structures". Bulletin of the Seismological Society of America, Vol. 88, N° 2, pp. 368-392, 1998.

[53] Komatitsch D. and Tromp G. "Introduction to the Spectral-element Method for 3D Seismic Wave Propagation". Geophysical Journal International, Vol. 139, pp. 806-822, 1999.

[54] Komatitsch D. and Tromp G. "Spectral-element Simulations of Global Seismic Wave Propagation-I. Validation". Geophysical Journal International, Vol. 149, pp. 390-412, 2002.

[55] Komatitsch D., Tsuboi S., Chen J. and Tromp G. "A 14.6 Billion Degrees of Freedom, 5 teraflops, 2.5 Terabyte Earthquake Simulation on the Earth Simulator" . Conference on High Performance Networking and Computing. Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, Phoenix, Arizona, 2003.

[56] Kulak R. F. Critical Time Step Estimation for Three-Dimensional Explicit Analysis. Structure under Shock and Impact, (ed. P. S. Bulson), Elseiver, Amsterdam, 1989.

[57] Metis, Internet. http://www.cs.umn.edu/~metis

[58] Nour-Omid B., Raefsky A. and Lyzenga G. "Solving Finite Element Equations on Concurrent Computers". In A. K. Noor, editor, Parallel Computations and their Impact on Mechanics. ASME, New York, pp. 209-227, 1986.

[59] Popov E. P. Mecánica de Sólidos. Editorial Pearson Educación, México, 2000, ISBN: 970-17-0398-7.

[60] Pothen A., Simon H. and Liou K. "Partitioning Sparse Matrices with Eigenvectors of Graphs". SIAM Journal of Matrix Analysis and Applications, Vol. 11, N° 3, pp. 430-452, 1990.

[61] Sadayappan P. and Ercal F. "Nearest-Neighbor Mapping of Finite Element Graphs onto Processors Meshes". IEEE Transactions on Computers, Vol. 36, N° 12, pp. 1408-1424, 1987.

[62] Simó J. C. and Hughes T. J. Computational Inelasticity. Springer, New York, 2000.

[63] Simon H. D. "Partitioning of Unstructured Problems for Parallel Processing". Computing Systems in Engineering, Vol. 2, N° 2/3 pp. 135-148, 1991.

[64] Sosnowski W. "Flow Approach-Finite Element Model for Stamping Processes versus Experiment". Computer Assisted Mechanics and Engineering Sciences, Vol. 1, N° 1/2 pp. 49-75, 1994.

[65] Walshaw C. and Cross M. "Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm". SIAM Journal on Scientific Computing, Vol. 22, N° 1, pp. 63-80, 2000.

[66] Wolman A., Voelker G. M., Sharma N., Cardwell N., Karlin A. and Levy H. M. "On the Scale and Performance of Cooperative Web Proxy Caching". Proceedings of 17th ACM Symposium on Operating Systems Principles, pp. 16-31, ACM Press New York, USA 1999.

[67] Zhong Z. H. Finite Element Procedures for Contact-Impact Problems. Oxford University Press, Oxford, New York, Tokio, pp. 1-372, 1993.

[68] Zienkiewicz O. C. and Taylor R. L. "The Finite Element Method: Solid and Fluid Mechanics Dynamics and Non-linearity". McGraw-Hill Book Company, Vol. 2, 4th edition, 1989.

[69] Zienkiewicz O. C. y Taylor R. L. El Método de los Elementos Finitos: Formulación Básica y Problemas Lineales. Mc Graw Hill, Vol. 1, 4ta edición, 1994.

Capítulo 4

[1] Brown K., Attaway S., Plimpton S. and Hendrickson B. "Parallel Strategies for Crash and Impact Simulations". Computational Methods in Applied Mechanics & Engineering, Vol. 184, pp. 375-390, 2000.

[2] Bui T. and Jones C. "A Heuristic for Reducing Fill in Sparse Matrix Factorization". In Proceedings 6th SIAM Conference Parallel Processing for Scientific Computing, pp. 445-452, 1993.

[3] Chen J. and Taylor V. E. "Mesh partitioning for distributed systems: Exploring optimal number of partitions with local and remote communication". In Proceedings of 9th Conference on Parallel Processing and Scientific Computing, SIAM, Philadelphia, 1999.

[4] Chen J. and V. E. Taylor. "ParaPART: Parallel Mesh Partitioning Tool for Distributed Systems". Concurrency: Practice and Experience, Vol. 12, N° 2-3, pp. 111-123, 2000.

[5] Chien Y. P. et. al. "Load-balancing for Parallel Computation of Fluid Dynamics Problems". Computer Methods in Applied Mechanics and Engineering, Vol. 120, pp. 119-130, 1995.

[6] Cybenko, G. "Dynamic Load Balancing for Distributed Memory Multiprocessors". Journal of Parallel and Distributed Computing, Vol. 7, N° 2, pp. 279-301, 1989.

[7] Dasa S. K., Harvey D. J. and Biswas R. "MinEX: A Latency-tolerant Dynamic Partitioner for Grid Computing Applications". Future Generation Computer Systems, Vol. 18, N° 4, pp. 477-489, 2002.

[8] Devine K., Hendrickson B., Boman E., John M. S. and Vaughan C. "Design of Dynamic Load-balancing Tools for Parallel Applications". In Proceedings of the International Conference on Supercomputing, Santa Fe, pp. 110-118, 2000.

[9] Faik J., Flaherty J. E., Gervasio L. G., Teresco J. D., Devine K. D. and Boman E. G. "A Model for Resource-aware Load Balancing on Heterogeneous Clusters". In Proceedings of the Conference on Cluster'04, San Diego, California, 2004.

[10] Foster I. and Kesselman C. "The Grid: Blueprint for a New Computing Infrastructure". Morgan Kaufmann Publishers, San Francisco, California, 1999.

[11] Genaud S., Giersch A. and Vivien F. "Load-Balancing Scatter Operations for Grid Computing". Journal Parallel Computing, Vol. 30, N° 8, pp. 923-946, 2004.

[12] Gropp W. D., Kaushik D. K., Keyes D. E. and Smith B. F. "Latency, Bandwidth and Concurrent issue limitations in high-performance CFD". In Proceedings of the 1st M.I.T. Conference on Computational Fluid and Solid Mechanics, Cambridge, MA, 2000.

[13] Harvey D. J., Das S. K. and Biswas R. "Performance of a heterogeneous Grid

partitioner for N-body applications". In Proceedings of International Conference on Parallel Processing, Kaohsiung, Taiwan, pp. 399-406, 2003.

[14] Hendrickson B. and Devine K. "Dynamic Load Balancing in Computational Mechanics". Computational Methods in Applied Mechanics & Engineering, Vol. 184, pp. 485-500, 2000.

[15] Hendrickson B. and Kolda T. "Graph Partitioning Models for Parallel Computing". Journal of Parallel Computing, Vol. 26, N° 12, pp. 1519-1534, 2000.

[16] Hendrickson B. and Leland R. "A Multilevel Algorithm for Partitioning Graphs". Proceedings of Supercomputing'95, San Diego, California, ACM Press, New York, 1995.

[17] Hendrickson B. and Leland R. "The Chaco User's Guide, Version 2.0". Technical Report SAND95-2344, Sandia National Laboratories, Albuquerque, New Mexico, 1995.

[18] Hui C. and Chanson S. "Theoretical Analysis of the Heterogeneous Dynamic Load Balancing Problem Using a Hydro-Dynamic Approach". Technical Report HKUST-CS96-01, 1996.

[19] Johnston W., Gannon D. and Nitzberg B. "Grids as Production Computing Environments: The Engineering Aspects of NASA's Information Power Grid". IEEE Computer Society Press, 1999.

[20] Karypis G. and Kumar V. "A Coarse-Grain Parallel Multilevel k-way Partitioning Algorithm". In Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing, 1997.

[21] Karypis G. and Kumar V. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". SIAM Journal on Scientific Computing, Vol. 20, N° 1, pp. 359-392, 1998.

[22] Karypis G. and Kumar V. "A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering". Journal of Parallel and Distributed Computing, Vol. 48, N° 1, pp. 71-95, 1998.

[23] Karypis G. and Kumar V. "Parallel Multilevel K-way Partitioning for Irregular Graphs". SIAM Review, Vol. 41, N° 2, pp. 278-300, 1999.

[24] Karypis G. and Kumar V. "Parallel Multilevel K-way Partitioning Scheme for Irregular Graphs". SIAM Review, Vol. 41, N° 2, pp. 278-300, 1999.

[25] Ko S., Kim C., Rho O. and Lee S. "A Grid-based Flow Analysis and Investigation of Load Balance in Heterogeneous Computing Environment". In Proceedings of Parallel Computational Fluid Dynamics, Elsevier Science, Moscow, 2003.

[26] Laxmikant V. K. and Krishnan S. "CHARM++: A Portable Concurrent Object Oriented System Based on C++". Proceedings of the 8th Annual Conference on Object-oriented Programming Systems, Languages, and Applications,

Washington, D. C., pp. 91-108, 1993.

[27] Li Y. and Lan Z. "A Survey of Load Balancing in Grid Computing". Computational and Information Science (CIS). Proceedings of 1st International Symposium. Lecture Notes in Computer Science Vol. 3314, pp. 280-285, China, 2004.

[28] Monien J., Preis R. and Diekmann R. "Quality Matching and Local Improvement for Multilevel Graph-Partitioning". Technical Report, University of Paderborn, 1999.

[29] Otero B., Cela J. M., Badía R. M. and Labarta J. "Data Distribution Strategies for Domain Decomposition Applications in Grid Environments". Proceedings of the 6th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'05). Lecture Notes in Computer Science Vol. 3719, Melbourne, Australia, pp. 214-224, 2005.

[30] Otero B., Cela J. M., Badía R. M. and Labarta J. "Domain Decomposition Strategy for Grid Environments". Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting, Proceedings. Lecture Notes in Computer Science Vol. 3241/2004, Budapest, Hungary, pp. 353-361, 2004.

[31] Otero B., Cela J. M., Badía R. M. and Labarta J. "Performance Analysis of Domain Decomposition Applications Using Unbalanced Strategies in Grid Environments". Proceedings of the 4th International Conference on Grid an Cooperative Computing (GCC2005). Lecture Notes in Computer Science Vol. 3795, Beijing, China, pp. 1031-1042, 2005.

[32] Otero B., Cela J. M., Badía R. M. and Labarta J. "Strategies of Domain Decomposition for Mesh-Based Applications in Grid Environments", Journal Future Generation Computer Systems (to appear).

[33] Plimpton S., Attaway S., Hendrickson B., Swegle J., Vaughan C. and Gardner D. "Transient Dynamics Simulations: Parallel Algorithms for Contact Detection and Smoothed Particle Hydrodynamics". Journal of Parallel and Distributed Computing, Vol. 50, pp. 104-122, 1998.

[34] Pothen A. "Graph Partitioning Algorithms with Applications to Scientific Computing". In D. Keyes, A. Samed, and V. Venkatakrishnan, editors, Parallel Numerical Algorithms. Kluwer Academic Press, Vol. 4 of ICASE/LaRC Interdisciplinary Series in Science and Engineering, pp. 323-368, 1997.

[35] Rantakokko J. "Partitioning Strategies for Structured Multiblock Grids". Parallel Computing, Vol. 26, pp. 1661-1680, 2000.

[36] Schloegel K. and Karypis G. "Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes". Journal of Parallel and Distributed Computing, Vol. 47, N° 2, pp. 109-124, 1997.

[37] Schloegel K., Karypis G. and Kumar V. "Graph Partitioning for High-Performance Scientific Simulations". In Jack Dongara, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White, editors, Sourcebook on Parallel Computing, Chapter 18, pp. 491-541. Morgan Kaufmann, San Francisco, CA, 2002.

[38] Schloegel K., Karypis G. and Kumar V. "Multilevel Diffusion Algorithms for Repartitioning of Adaptive Meshes". Journal of Parallel and Distributed Computing. Vol. 47, N° 2, pp. 109-124, 1997.

[39] Simon H. D. and Farhat C. "TOP/DOMDEC: A Software Tool for Mesh Partitioning and Parallel Processing". Technical Report RNR-93-011, NASA, 1993.

[40] Simon H.D., Sohn A. and Biswas R. "HARP: A Fast Spectral Partitioner". In Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures, Newport, Rhode Island, pp. 43-52, 1997.

[41] Sinha S. and Parashar M. "Adaptive System-sensitive Partitioning of AMR Applications on Heterogeneous Clusters". Cluster Computing: The Journal of Networks, Software Tools and Applications, Kluwer Academic Publishers, Vol. 5, N° 4, pp. 343-352, 2002.

[42] Taylor V. E., Chen J., Canfield T. and Stevens R. "A Decomposition Method for Efficient Use of Distributed Supercomputers for Finite Element Applications". In Proceedings of the 10th International Conference on Application Specific Systems, Architectures and Processors, Chicago, Illinois, pp. 12-24, 1996.

[43] Taylor V. E., Schwabe E. J., Hribar M. R. and Holmer B. K. "Balancing Load versus Decreasing Communication: Parameterizing the Tradeoff". Journal of Parallel and Distributed Computing, Vol. 61, N° 5, San Diego, pp. 567-580, 2001.

[44] Walshaw C. and Cross M. "Parallel Optimisation Algorithms for Multilevel Mesh Partitioning". Technical Report 99/IM/44, University of Greenwich, London, UK, 1999.

[45] Walshaw C., Cross M. Johnson S. and Everett M. "JOSTLE: Partitioning of Unstructured Meshes for Massively Parallel Machines". Parallel Computational Fluid Dynamics: New Algorithms and Applications. Elsevier, Amsterdam, pp. 273-280, 1995.

[46] Willebeek-LeMair M. H. and Reeves A. P. "Strategies for Dynamic Load Balancing on Highly Parallel Computers". IEEE Transactions on Parallel and Distributed Systems, Vol. 4, pp.979-993, 1993.

[47] Yang L., Schopf J. M. and Foster I. "Conservative Scheduling: Using Predicted Variance to Improve Scheduling Decisions in Dynamic Environments". Conference on High Performance Networking and Computing. Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, Phoenix, Arizona, 2003.