UNIVERSIDADE DE SANTIAGO DE COMPOSTELA
DEPARTAMENTO DE ELECTRÓNICA E COMPUTACIÓN

Tesis Doctoral

Técnicas paralelas aplicadas a optimización no lineal en sistemas de memoria distribuida

Inmaculada Pardines Lence
Santiago de Compostela, Enero 2007

Dr. **Francisco Fernández Rivera**, Catedrático del Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela.

CERTIFICA:

Que la memoria titulada "**Técnicas paralelas aplicadas a optimización no lineal en sistemas de memoria distribuida**", ha sido realizada por Dña. **Inmaculada Pardines Lence** bajo mi dirección en el Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela y constituye la Tesis que presenta para optar al grado de Doctora en Ciencias Físicas.

Santiago de Compostela, Enero de 2007

Fdo. Francisco Fernández Rivera
Director de la tesis.

Fdo. Javier Díaz Bruguera
Director del Departamento de Electrónica y Computación.

Fdo. Inmaculada Pardines Lence,
Doctoranda.

A Jorge
y a Iago

Este trabajo ha sido posible gracias a la financiación de la Comisión Interministerial de Ciencia y Tecnología a través de los proyectos CICYT TIC 2002/750, TIC 2001-3694-C02, y de la Xunta de Galicia a través del proyecto PGID99PXI20602B.

Agradecimientos

Quiero dar mi más sincero agradecimiento a todas las personas que han contribuido a la realización de esta tesis tanto en el ámbito académico como en el personal.

En primer lugar, quiero dar las gracias al director de este trabajo, el catedrático D. Francisco Fernández Rivera, por todo el tiempo y esfuerzo que me ha dedicado, así como la confianza que ha depositado en mí. Él me dio la oportunidad de realizar mis estudios de doctorado, y durante todo este tiempo siempre he podido contar con su apoyo.

También quiero expresar mi agradecimiento a todos los miembros del Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela, en particular a aquellos con los que he compartido café y/o despacho. Sin nuestras charlas esa etapa de mi vida no hubiese sido tan agradable.

Gracias especialmente, a María, con la que colaboré estrechamente en el inicio de mi etapa investigadora. A Patricia y a Juan, por los muchos momentos que hemos compartido durante mis cuatro años de estancia en Santiago. A Antonio y a Dora, que cuando compartían despacho siempre estaban dispuestos a escuchar mis agobios y hoy, a pesar de la distancia, puedo seguir contando con ellos.

A David, por su preocupación constante porque todo el mundo sea feliz, y por su aportación a ello, con todas esas anécdotas raras y divertidas, que sólo él conoce. A Paula, porque me ha demostrado el valor de la amistad, con su continua ayuda y sus muchos ánimos. ¡No cambies, guapa! Te agradezco especialmente que hayas leído esta tesis y que me hayas aportado tantas sugerencias. Muchas gracias a los dos, por disfrutar de las anécdotas que os cuento de Iago casi tanto como yo.

También quiero dar las gracias a todos los miembros del Departamento de Arquitectura de Computadores y Automática de la Universidad Complutense, que me han tratado tan bien, donde desde el primer día me encuentro perfectamente integrada. En especial, a su director, el catedrático D. Francisco Tirado Fernández, que me acogió en su grupo de investigación, y ha tenido una gran paciencia en espera de la conclusión de este trabajo.

Gracias a todos los que me hacéis más amenas las comidas o los cafés: Dani, Silvia, Sara, Javi, Fredy, Sonia, ... Especialmente, a Juan Carlos y a Marcos por haberme ayudado a tomar la decisión más importante de mi vida. A Jose, por compartir conmigo su alegría y, por supuesto, sus recursos *software*. A Rubén, por sus múltiples sugerencias en la escritura de esta memoria. A Horten, Katza y Mari Carmen, por haberme dado tan buenos consejos.

Quiero agradecer al CESGA (Centro de Supercomputación de Galicia) por darme acceso a sus recursos de computación distribuida.

A mis amigos, en particular, a Ana, Marta e Isa, que me ofrecieron su compañía, y con los que siempre he podido contar cuando más los necesitaba.

A mi familia, por su apoyo incondicional durante todos estos años. En especial, a mis padres, a ellos les debo lo que soy.

A Jorge, por la paciencia que ha demostrado, sobre todo en los últimos meses que, al compaginar las tareas de madre con la escritura de esta tesis, se me hicieron muy duros.

Y a mi pequeño, Iago, por haber dado sentido a mi vida. Gracias, por tu alegría y continuas ganas de jugar. Gracias por esos besos que me das cuando menos me lo espero.

Índice general

Índice de Tablas

Índice de figuras

Introducción

Los problemas de optimización aparecen en áreas de las matemáticas, ingeniería, economía, medicina y estadística, de ahí el gran desarrollo que han experimentado en las últimas décadas. Son especialmente útiles en el estudio de las características de modelos matemáticos utilizados para explicar complejos fenómenos del mundo real.

Se entiende por solución de un problema de optimización un conjunto de valores permitidos de las variables para los cuales se alcanza el valor óptimo de una determinada función. Gran cantidad de algoritmos han sido propuestos e implementados con éxito, de ahí que el número de científicos en diversas disciplinas que usan técnicas de este tipo para resolver problemas haya aumentado considerablemente. Algunas de sus aplicaciones incluyen la economía en escala, finanzas, optimización estructural, problemas de redes y transporte, problemas de diseño de circuitos integrados, diseño nuclear y mecánico, diseño y control de ingeniería química, y biología molecular.

Los problemas de optimización que aparecen en la práctica suelen ser computacionalmente muy costosos. No sólo por su tamaño sino también por factores adicionales como la existencia de relaciones no lineales o la no convexidad. En los computadores secuenciales, los algoritmos para resolver estos problemas suelen necesitar tiempos de computación excesivos llegando a ser en muchas ocasiones imposibles de resolver. El desarrollo de algoritmos paralelos para la resolución de problemas de optimización sobre multiprocesadores o arquitecturas *cluster* permite encontrar la solución óptima a estos problemas con mayor rapidez, y resolver además aquellos problemas que requieren muchos recursos de memoria y cuya ejecución resultaba imposible en un computador secuencial convencional.

En este trabajo hemos tratado de disminuir los tiempos de computación, mediante la ejecución paralela en sistemas de memoria distribuida, de uno de los algoritmos más utilizados en optimización no lineal, el método cuasi-Newton. El mayor coste computacional de este algoritmo reside en las operaciones algebraicas y en la evaluación de la función a optimizar y de su gradiente. En esta memoria hemos abordado

dos estrategias para mejorar los tiempos de computación de este método: por un lado, se han propuesto algoritmos paralelos para realizar las computaciones algebraicas y las evaluaciones de la función objetivo y su gradiente en paralelo. De este modo, se reduce el tiempo de computación de cada iteración del método de resolución. Por otro lado, hemos propuesto un paralelismo de grano grueso en el que se trata de reducir el número de iteraciones, proponiendo distintas técnicas multipaso, cada una de las cuales avanza hacia la solución del problema siguiendo un camino distinto. Todas las modificaciones propuestas se han realizado sobre el algoritmo cuasi-Newton utilizado por la herramienta de optimización MINOS. Asimismo, este *software* es utilizado para evaluar el rendimiento de los algoritmos paralelos propuestos. MINOS[1] es una herramienta de optimización desarrollada en la universidad de Stanford que desde 1980 ha sido instalada y utilizada por un gran número de instituciones académicas y de investigación en todo el mundo. Su principal característica es que permite encontrar la solución de pequeños y grandes problemas en cuatro de las grandes áreas de la optimización continua: programación lineal, optimización sin restricciones, optimización con restricciones lineales y optimización con restricciones no lineales.

En el primer capítulo se define formalmente que se entiende por problema de optimización general. Se realiza una clasificación de los distintos problemas de optimización, analizando las características de cada uno y los distintos métodos que se utilizan en su resolución. Además, se introduce el *software* de optimización MINOS con el que se compara nuestro trabajo.

En el segundo capítulo se describen brevemente algunas de las características de la computación paralela, y se realiza una introducción a los sistemas multiprocesador y a las arquitecturas en *cluster*. También se describen los sistemas que se han empleado para la evaluación de los algoritmos que se mencionan en esta memoria.

En el tercer capítulo, el método cuasi-Newton se estudia con mayor detenimiento. Se muestra un estudio del coste computacional de las distintas subrutinas que lo forman, encontrándose entre las subrutinas más costosas las resoluciones triangulares, utilizadas en el cálculo de la dirección de búsqueda, y las rotaciones globales, que sirven para actualizar la aproximación de la Hesiana, matriz en la que se basa este método. También se muestra en este capítulo un análisis de las dependencias de datos de estas subrutinas, lo que permitirá posteriormente paralelizarlas de forma eficiente.

En el cuarto capítulo se describen los algoritmos paralelos propuestos para las subrutinas algebraicas más costosas. Estos algoritmos tratan de reducir el tiempo

[1] http://www.sbsi-sol-optimize.com/Minos.htm

de ejecución de cada iteración. Para comprobar si se han alcanzado los objetivos buscados se realiza un estudio de su rendimiento en sistemas multiprocesador de memoria distribuida como el AP3000 de Fujitsu y en arquitecturas *cluster* con red de interconexión Myrinet como el sistema *beowulf* del CESGA.

En el quinto capítulo se presenta un estudio del balanceo de la carga para la distribución de datos inicialmente propuesta. A medida que el algoritmo evoluciona en la búsqueda de la solución óptima se puede llegar a producir un cierto desbalanceo. Se proponen y describen distintas heurísticas de redistribución para corregir este desequilibrio, y se comparan con otras ya existentes, como la heurística Mejor ajuste y la heurística Peor ajuste. Asimismo, se evalúa el coste computacional debido al desbalanceo y a la redistribución, para justificar en qué casos compensa redistribuir la carga.

En el sexto capítulo se aborda una paralelización de grano grueso de un método cuasi-Newton. En este caso, se busca reducir el tiempo de computación del algoritmo mediante la disminución del número total de iteraciones requerido para alcanzar la solución del problema. Distintas estrategias multipaso son propuestas, resaltando su eficiencia en términos de reducción en el número de iteraciones y de evaluaciones de la función objetivo.

Por último, se indican las principales conclusiones de este trabajo, y se mencionan las líneas de investigación abiertas por el mismo.

Capítulo 1

Introducción a los métodos de optimización no lineal

En las últimas décadas la aplicación de métodos de optimización ha ido adquiriendo cada vez más importancia en diversas áreas de la ciencia y la ingeniería, debido al gran número de problemas que pueden resolver. Básicamente, el objetivo de todo problema de optimización es encontrar el valor máximo o mínimo que alcanza una función en un determinado espacio de búsqueda. Un problema puede poseer más de un mínimo, de ahí que distingamos entre optimización global y optimización local. Entendemos por óptimo global la mejor solución a un determinado problema, y por óptimo local la mejor solución a un problema dentro de un entorno determinado.

Existen muchos algoritmos capaces de resolver problemas de optimización local. Estos métodos suelen recibir el nombre de métodos determinísticos. Su principal característica es que garantizan la convergencia a una solución, siempre y cuando se cumplan los criterios de convergencia del algoritmo. Son métodos iterativos que utilizan tanto técnicas de programación lineal como no lineal, dependiendo del tipo de problema. Computacionalmente se caracterizan por el uso de rutinas de álgebra numérica y evaluaciones de la función objetivo, del gradiente, y en algunos casos de la Hesiana de la función. Por lo tanto, requieren el conocimiento de la estructura matemática del problema a resolver. Según el tipo de problema, será más eficiente usar un determinado método de resolución. Así, si el problema no presenta restricciones, puede resolverse empleando un método de Newton, cuasi-Newton, un Newton discreto o métodos del gradiente conjugado, etc [34]. Si el problema presenta restricciones lineales y la función objetivo es lineal, el método *simplex* [35] es uno de los más utilizados por su baja complejidad computacional. Por el contrario, si la función objetivo

es no lineal, resultan eficientes los métodos basados en un conjunto activo de restricciones o métodos como el gradiente reducido, computando la dirección de búsqueda mediante algoritmos similares a los empleados en problemas sin restricciones. Cuando las no linealidades también aparecen en las restricciones [82], métodos como el gradiente reducido generalizado, el gradiente proyectado, la Lagrangiana proyectada o la Lagrangiana aumentada, abordan el problema aproximando las restricciones a un modelo lineal.

Un amplio espectro de métodos también puede ser utilizado en problemas de optimización global. Sin embargo, puesto que presentan el inconveniente de que su coste computacional crece exponencialmente con la dimensión de la parte no convexa del problema a optimizar, y que una gran parte de los problemas reales no están bien estructurados matemáticamente, para este tipo de optimización se suelen utilizar los llamados métodos estocásticos. La mayoría de los métodos estocásticos se basan en la evaluación de la función objetivo en una serie de puntos elegidos de forma aleatoria entre el conjunto de puntos factible, y posteriores manipulaciones de dicho conjunto. Estos métodos no pueden garantizar absolutamente la convergencia al óptimo global, pero pueden alcanzar una aproximación suficientemente buena. Entre los métodos estocásticos destacamos los métodos de búsqueda aleatoria, como los algoritmos de Enfriamiento Simulado [21, 49], Búsqueda Tabú [37, 38] y los algoritmos genéticos [39], que se basan en distintas técnicas de emulación de los procesos naturales de evolución.

Todas las técnicas descritas en este trabajo corresponden a problemas de optimización local, y por lo tanto, se engloban dentro de los llamados métodos determinísticos. En este capítulo de introducción definiremos, en primer lugar, el problema de la optimización local, y las condiciones que se deben verificar para que un determinado punto pueda ser considerado mínimo local. Diferenciaremos entre distintas clases de problemas de optimización, centrando posteriormente nuestro estudio en los problemas no lineales con restricciones lineales. Por último, se describen algunos métodos eficientes para resolver esta clase de problemas, en concreto, los utilizados en el código de optimización MINOS [66].

1.1. Definición de un problema de optimización

Un problema de optimización se puede definir de forma general partiendo de un conjunto de variables independientes, una serie de restricciones que definen valores aceptables de las variables y una función de estas variables que se denomina función objetivo. Se entiende por solución de un problema de optimización cualquier

conjunto aceptable de valores de las variables para el cual la función objetivo alcanza un valor óptimo. Formalmente, el proceso de optimización implica maximizar o minimizar la función objetivo dentro de un conjunto de valores factibles de las variables.

La mayor parte de los problemas de optimización se pueden expresar matemáticamente mediante la forma general:

$$\begin{array}{lll} \text{minimizar} & F(x), & x \in IR^n \\ \text{sujeto a} & c_i(x) = 0, & i = 1, 2, ..., m' \\ & c_i(x) \geq 0, & i = m' + 1, ..., m \end{array} \tag{1.1}$$

donde F es la función objetivo y las funciones c_i las restricciones sobre las variables. En la definición 1.1, la optimización de una determinada función objetivo se entiende como minimización. Sin embargo, si el problema de optimización consistiese en maximizar la función F, su definición podría ajustarse a la dada en 1.1 sin más que sustituir $F(x)$ por $-F(x)$. Asimismo, la función $c(x)$ engloba la existencia de cualquier tipo de restricciones: desigualdades e igualdades, linealidades y no linealidades, y límites superior e inferior.

Sin embargo, la existencia de una formulación estándar no implica que las diferencias entre los problemas deban ser ignoradas. Para resolver un problema de forma eficiente será importante determinar sus características, de ahí que resulte interesante clasificar los problemas para poder proponer los métodos de resolución más adecuados.

1.2. Clasificación de los problemas de optimización

La clasificación de los problemas de optimización se suele basar en un compromiso entre las mejoras en la eficiencia obtenidas al considerar ciertas propiedades de los problemas frente a la complejidad de tener que elegir un método de resolución dentro de una gran librería de métodos. La forma más obvia de caracterizar los problemas se basa en las propiedades matemáticas de la función objetivo y de las funciones de restricciones. Así, por ejemplo, podremos hablar de problemas sin restricciones, problemas con una función objetivo lineal con las variables ligadas a una frontera superior y otra inferior, etc.

En la tabla 1.1 se muestra un esquema de las propiedades que típicamente caracterizan la naturaleza de las funciones que definen el problema.

Se pueden utilizar otros criterios para clasificar los problemas de optimización, por ejemplo, el tamaño del problema. La dimensión del problema va a afectar a la

capacidad de almacenamiento y al esfuerzo computacional requerido para obtener una solución, de modo que técnicas muy efectivas para problemas con pocas variables pueden no ser adecuadas si el problema tiene cientos de miles de variables. Sin embargo, el concepto de "tamaño" es relativo, y dependerá mucho del entorno de trabajo.

Otra característica que distingue a los problemas de optimización es la cantidad de información disponible durante el proceso de resolución. Algunos problemas permiten computar de forma analítica la primera y segunda derivada de la función objetivo, mientras que otros sólo proporcionan los valores de la función.

Finalmente, algunas aplicaciones de optimización pueden incluir requisitos especiales que reflejan la fuente del problema y el marco dentro del cual se resuelve. Por ejemplo, algunos problemas pueden necesitar que ciertas restricciones se satisfagan exactamente en cada iteración.

1.3. Definición de mínimo local

Un problema de optimización implica la minimización de la función objetivo verificándose las restricciones impuestas a las variables. Cualquier punto x^* que verifique las restricciones del problema se dirá que es un punto factible. De la definición de problema de optimización se deduce que únicamente los puntos factibles pueden ser solución del problema.

Sea x^* un punto factible para el caso general de un problema de optimización como el enunciado en la ecuación 1.1, y $N(x^*, \delta)$ un conjunto de puntos factibles que distan de x^* a lo sumo una distancia δ. Diremos que el punto x^* es un mínimo local fuerte si existe $\delta > 0$ tal que:

1. $F(x)$ está definida en $N(x^*, \delta)$; y
2. $F(x^*) < F(y)$, $\forall y \in N(x^*, \delta)$, $y \neq x^*$.

Diremos que el punto x^* es un mínimo local débil si existe $\delta > 0$ tal que:

1. $F(x)$ está definida en $N(x^*, \delta)$;
2. $F(x^*) \leq F(y)$, $\forall y \in N(x^*, \delta)$; y
3. x^* no es un mínimo local fuerte.

La definición de mínimo local sirve para verificar si un punto dado es o no óptimo, y para tratar de seleccionar algoritmos de resolución del problema de optimización

Propiedades de F(x)	Propiedades de $\{c_i(x)\}$
Función de una sola variable	Sin restricciones
Función lineal	Fronteras simples
Suma de cuadrados de funciones lineales	Funciones lineales
Función cuadrática	Funciones lineales dispersas
Suma de cuadrados de funciones no lineales	Funciones no lineales suaves
Función no lineal suave	Funciones no lineales dispersas
Función no lineal dispersa	Funciones no lineales no suaves
Función no lineal no suave	

Tabla 1.1: Propiedades para la clasificación de los problemas de optimización.

basados en la definición de mínimo. Si basamos estos algoritmos en las definiciones anteriores sería necesaria la evaluación de la función objetivo F en la δ-vecindad de cualquier punto x^* que se esté evaluando, lo que implicará generalmente un gran coste computacional. Pero afortunadamente, las propiedades de la función objetivo y de las funciones de restricciones, suelen permitir definir los mínimos locales de una forma más práctica. De modo que en función del tipo de problema habrá distintos métodos de resolución que se adecúan más a las condiciones que debe cumplir un punto óptimo para esa clase de problemas.

1.4. Problemas con función objetivo no lineal y restricciones lineales

Este tipo de problemas se pueden expresar de forma general como:

$$\begin{array}{lll} \text{minimizar} & F(x), & x \in I\!R^n \\ \text{sujeto a} & Ax = b, & \\ & l \leq x \leq u & \end{array} \tag{1.2}$$

donde F es dos veces continuamente diferenciable, l y u son límites inferior y superior de x, y A es una matriz $m \times n$ $(m \leq n)$, donde la fila i-ésima de A contiene los coeficientes correspondientes a la i-ésima restricción. Nótese que la definición 1.2 también incluye el caso de problemas con restricciones de desigualdad como $Ax \geq b$, ya que esta expresión se puede sustituir por una restricción de igualdad añadiendo variables residuales, con el fin de obtener la restricción equivalente $Ax - y = b$.

El algoritmo que va a ser utilizado para resolver este tipo de problemas es una extensión del método *simplex* combinado con un método cuasi-Newton para tratar las no linealidades. El método cuasi-Newton es también ampliamente utilizado en

la resolución de problemas de optimización sin restricciones, de ahí que las mejoras propuestas en esta memoria se puedan asimismo aplicar a este tipo de problemas. Sin embargo, antes de describir los algoritmos empleados para resolver problemas de optimización no lineal, resulta interesante enunciar, para esta clase de problemas, las condiciones que debe verificar un punto $x^* \in I\!R^n$ para ser un mínimo local; es decir, qué condiciones debe chequear el método de optimización utilizado para decidir si se ha alcanzado o no la solución óptima.

1.4.1. Condiciones de óptimo

Como se ha visto en la sección 1.3, un punto x^* es un mínimo local sólo si $F(x^*) \leq F(x)$ para todo x perteneciente a la vecindad de x^*. En los problemas de la forma 1.2, las restricciones constituyen un sistema lineal, y utilizando las propiedades de los subespacios lineales, podemos caracterizar la dirección p que permite el movimiento de un punto factible a otro. El vector p que define esta dirección debe ser perpendicular al espacio definido por las filas de la matriz A, es decir,

$$Ap = 0. \tag{1.3}$$

Supongamos una matriz Z cuyas columnas forman una base del subespacio de vectores que verifica la ecuación 1.3, por lo tanto, $AZ = 0$, entonces toda dirección factible p se puede definir como una combinación lineal de las columnas de Z, de la forma $p = Zp_z$. Podemos aproximar el valor de F en un punto factible de la vecindad de x^* como:

$$F(x^* + \epsilon Z p_z) = F(x^*) + \epsilon p_z^T Z^T g(x^*) + \frac{1}{2}\epsilon^2 p_z^T Z^T G(x^* + \epsilon\theta p) Z p_z, \tag{1.4}$$

donde $0 \leq \theta \leq 1$, y ϵ es un número positivo. Cualquier vector $p = Zp_z$ tal que $p_z^T Z^T g(x^*)$ sea negativo se dice que es una dirección descendente. Dada una dirección descendente Zp_z, siempre existirá un escalar positivo $\bar{\epsilon}$ tal que para todo entero positivo ϵ, con $\epsilon \leq \bar{\epsilon}$, se verifica que $\epsilon p_z^T Z^T g(x^*) + \frac{1}{2}\epsilon^2 p_z^T Z^T G(x^* + \epsilon\theta p) Z p_z < 0$. De la ecuación 1.4 se deduce que $F(x^* + \epsilon Z p_z) < F(x^*)$ para estos ϵ, y por tanto x^* no sería un mínimo local. Con lo que, para que x^* sea un mínimo se tendrá que verificar que:

$$Z^T g(x^*) = 0. \tag{1.5}$$

O lo que es lo mismo, el gradiente $g(x^*)$ debe ser una combinación lineal de las filas de la matriz de restricciones A,

$$g(x^*) = \sum_{i=1}^{m} a_i \sigma_i^* = A\sigma^*, \tag{1.6}$$

donde σ^* se denomina vector de los multiplicadores de Lagrange.

De la ecuación 1.4, y siguiendo el mismo razonamiento que para el término de primer orden, se deduce una condición de segundo orden necesaria para que x^* sea un mínimo local: la matriz $Z^T G(x^*)Z$ debe ser semidefinida positiva. Resumiendo, las condiciones suficientes para que un punto x^* sea un mínimo local para un problema no lineal con restricciones de igualdad lineales son:

1. $Ax^* = b$;
2. $Z^T g(x^*) = 0$ ó $g(x^*) = A^T\sigma^*$; y
3. $Z^T G(x^*)Z$ sea definida positiva.

Si el problema tiene restricciones lineales de desigualdad, las condiciones suficientes para que el punto x^* sea un mínimo local son ligeramente diferentes. En este caso, se distinguen dos tipos de restricciones en un punto factible x^*: las restricciones activas si $a_i^T x^* = b_i$, y las restricciones inactivas si $a_i^T x^* > b_i$.

Las restricciones activas limitan las posibles perturbaciones válidas en torno a un punto factible, de ahí la necesidad de identificarlas para determinar si un punto x^* es óptimo. Si la restricción i es activa en x^* existen dos posibles direcciones factibles verificando $a_i^T p = 0$ (perturbación vinculante) ó $a_i^T p > 0$ (perturbación no vinculante).

Supongamos la matriz $\hat{A}$ cuyas t filas contienen los coeficientes de las restricciones activas; adoptando una nomenclatura similar para el vector $\hat{b}$, tendremos $\hat{A}x^* = \hat{b}$. Si definimos Z como una matriz cuyas columnas forman una base del conjunto de vectores ortogonales al espacio definido por las filas de $\hat{A}$, todo vector que verifique $\hat{A}p = 0$ se puede expresar como una combinación lineal de vectores de Z.

Para una dirección p vinculante, siguiendo los mismos razonamientos aplicados para el caso de restricciones de igualdad llegamos a unas condiciones necesarias equivalentes que se deben verificar en un punto factible x^* para que sea mínimo. Sin embargo, en el caso de restricciones activas, las perturbaciones no vinculantes también son válidas, y para que x^* sea un mínimo local, una dirección p no vinculante deberá verificar que $g(x^*)^T p \geq 0$. Si x^* es óptimo entonces $g(x^*)$ es una combinación lineal de las filas de $\hat{A}$, con lo que será necesario que:

$$g(x^*)^T p = \sigma_1^* \hat{a}_1^T p + \cdots + \sigma_t^* \hat{a}_t^T p \geq 0, \tag{1.7}$$

donde $\hat{a}_i^T p \geq 0$, $i = 1, \cdots, t$.

Para que el punto x^* sea un mínimo local se tendrá que verificar la condición 1.7, y esta condición será cierta si y sólo si $\sigma_i^* \geq 0$, $i = 1, \cdots, t$.

Resumiendo, el que los multiplicadores de Lagrange sean mayores que cero, será condición suficiente para que el punto x^* sea un mínimo local. Con lo que las condiciones suficientes son:

1. $Ax^* \geq b$, con $\hat{A}x^* = \hat{b}$;
2. $Z^T g(x^*) = 0$; o, equivalentemente, $g(x^*) = \hat{A}^T \sigma^*$;
3. $\sigma_i^* > 0$, $i = 1, \cdots, t$; y
4. $Z^T G(x^*) Z$ es definida positiva.

1.4.2. Métodos basados en un conjunto activo de restricciones

Si el problema de optimización posee restricciones de desigualdad, sólo las restricciones activas en el punto x^* son significativas para evaluar si este punto verifica las condiciones de óptimo expuestas en la sección 1.4.1. Si este conjunto de restricciones activas fuese conocido y sus coeficientes fuesen los elementos de las filas de $\hat{A}$, la solución del problema de optimización

$$\begin{array}{lll} \text{minimizar} & F(x), & x \in IR^n \\ \text{sujeto} \quad \text{a} & Ax \geq b, & \end{array} \tag{1.8}$$

también sería solución del problema con restricciones de igualdad

$$\begin{array}{lll} \text{minimizar} & F(x), & x \in IR^n \\ \text{sujeto} \quad \text{a} & \hat{A}x = \hat{b}, & \end{array} \tag{1.9}$$

Por lo tanto, si el conjunto de restricciones activas en un punto x^* fuese conocido, bastaría con resolver el problema con restricciones lineales de igualdad 1.9. Sin embargo, este conjunto es desconocido a priori. Los métodos basados en un conjunto de restricciones activas predicen una serie de restricciones que serán activas en el punto óptimo. El conjunto formado por estas restricciones se denomina conjunto de trabajo. Asimismo, este tipo de métodos necesitan incluir una serie de procedimientos para testear si el conjunto de trabajo actual es correcto, y de no ser así, permitir modificarlo, añadiendo o eliminando alguna restricción.

En cada iteración k de este tipo de algoritmos, se determina una dirección de búsqueda p_k y un tamaño de paso λ_k para calcular la nueva estimación de la solución $x_{k+1} = x_k + \lambda_k p_k$. La dirección de búsqueda deberá pertenecer al subespacio definido por el conjunto de trabajo, verificando $\hat{A}_k p_k = 0$. Además, el algoritmo tendrá que

garantizar que la longitud de paso seleccionada no viola ninguna de las restricciones que no pertenecen al conjunto de trabajo. La longitud de paso a la restricción más próxima a lo largo de la dirección p_k se convierte en un límite superior para λ_k. Si denotamos por λ_{max} al mayor paso no negativo factible que se puede tomar desde p_k, en la iteración k, la longitud λ_k deberá verificar que $\lambda_k \leq \lambda_{max}$. Si existe $\lambda_k < \lambda_{max}$, el conjunto de trabajo permanecerá inalterado, pero si $\lambda_k = \lambda_{max}$, entonces una nueva variable se ha hecho activa en el punto solución, y deberá ser añadida al conjunto de trabajo actual.

En cada iteración, se testea si el conjunto de trabajo es el adecuado. Este conjunto no estará bien definido, y el problema 1.9 estará mal planteado si alguno de los multiplicadores de Lagrange asociados a las restricciones de la matriz $\hat{A}_k$ es negativo. Esta situación significaría que existe otro punto diferente de x_k para el cual la función F tiene un valor menor. La restricción asociada al multiplicador de Lagrange negativo deberá ser eliminada del conjunto de trabajo.

Los cambios en este conjunto se traducen en modificaciones de la matriz $\hat{A}_k$. Añadir una restricción implica añadir una nueva fila a la matriz, y eliminar una restricción es equivalente a borrar una fila. Si la matriz $\hat{A}_k$ varía también lo hará la matriz Z_k, y el resto de matrices utilizadas, como la proyección de la Hesiana $Z^T G Z$. Recalcular estas matrices resulta computacionalmente muy costoso, así que se suelen utilizar técnicas que permitan realizar las actualizaciones de un modo más eficiente.

1.4.3. Problemas de optimización de gran tamaño con restricciones lineales

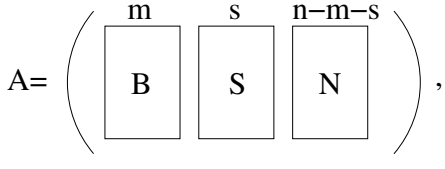
La formulación estándar para este tipo de problemas es la especificada en la ecuación 1.2, asumiendo un gran número de variables y que la matriz de restricciones A es una matriz dispersa. Las técnicas empleadas para resolver problemas pequeños suelen resultar ineficientes, en cuanto a tiempos de computación y a capacidad de almacenamiento, para problemas de gran tamaño. En este tipo de problemas el trabajo asociado a las operaciones algebraicas o a las operaciones de manejo de datos adquiere incluso más importancia que el coste de las evaluaciones de la función F. Será, por lo tanto, recomendable utilizar algoritmos que, aunque aumenten el número de evaluaciones necesarias de la función objetivo, consigan disminuir el coste de las operaciones algebraicas.

Si utilizamos un método basado en un conjunto activo de restricciones, en una determinada iteración, la matriz $\hat{A}$ contiene todas las filas de A más un conjunto

adicional de filas de la matriz identidad que se corresponden con variables fijas al valor de una de sus fronteras.

Aplicando un método similar al método *simplex*, podemos suponer una división de las variables en básicas y no básicas. De modo que para encontrar la solución óptima a un problema no será necesario estudiar todas las soluciones factibles sino que bastará con analizar únicamente las soluciones asociadas a las variables básicas, reduciendo de este modo la dimensionalidad del problema.

Si la matriz A tiene m columnas linealmente independientes, podemos subdividirla en una matriz que denominamos básica B formada por estas m columnas, y una matriz no básica N formada por las $n-m$ columnas restantes. Una posible solución al problema, que se denomina solución básica, será aquella cuyas m primeras componentes son iguales a $x_B = B^{-1}b$ y las demás tendrán valor cero. Por otro lado, si la función objetivo presenta no linealidades, la solución en el punto óptimo probablemente no será básica; sin embargo, si hay pocas variables no lineales cabe esperar que la solución sea "casi" básica. Entonces, generalizando, introducimos una nueva clase de variables que denominamos superbásicas, de modo que la matriz de restricciones A se divide en:



donde B es una matriz cuadrada y no singular $m \times m$, y sus columnas se corresponden con las variables básicas. S es una matriz $m \times s$, con $0 \leq s \leq n-m$, y la matriz N está formada por las restantes columnas de A, asociadas a las variables no básicas (variables fijas a sus fronteras). Las variables básicas y superbásicas se caracterizan porque pueden variar libremente entre sus fronteras. Las superbásicas lo podrán hacer en cualquier dirección, pero las variaciones de las variables básicas siempre tienen que garantizar la verificación de las restricciones $Ax = b$.

En una determinada iteración del método, las restricciones que forman el conjunto de trabajo vienen dadas por:

$$\hat{A}x = \begin{pmatrix} B & S & N \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} x_B \\ x_S \\ x_N \end{pmatrix} = \begin{pmatrix} b \\ b_N \end{pmatrix}, \tag{1.10}$$

donde los componentes de b_N se toman de los límites l ó u dependiendo de si la que se verifica es una frontera inferior o superior.

La dirección de búsqueda p, en una iteración dada, será igual a Zp_z, siendo Z una matriz cuyas columnas son ortogonales a las filas de $\hat{A}$. Una matriz que cumple estos requisitos será,

$$Z = \begin{pmatrix} -B^{-1}S \\ I \\ 0 \end{pmatrix}. \tag{1.11}$$

Si además consideramos que con $p_z = p_S$ la dirección de búsqueda p es una dirección descendente factible tendremos que,

$$p = \begin{pmatrix} p_B \\ p_S \\ p_N \end{pmatrix} = Zp_S = \begin{pmatrix} -B^{-1}Sp_S \\ p_S \\ 0 \end{pmatrix}. \tag{1.12}$$

De la ecuación 1.12, deducimos que $p_N = 0$, y

$$Bp_B = -Sp_S. \tag{1.13}$$

Por lo tanto, bastará con calcular la dirección de búsqueda para las variables superbásicas p_S mediante, por ejemplo, un método cuasi-Newton, y p_B se obtendrá a partir de la ecuación 1.13.

A lo largo de las distintas iteraciones del método se producirán cambios en el conjunto de trabajo, bien porque una variable básica o superbásica ha alcanzado alguna de sus fronteras, y por tanto, se convierte en no básica, o bien porque una variable no básica se convierte en superbásica con el objetivo de mejorar el conjunto de trabajo actual y aproximarse más a la solución óptima. Todos los cambios en el conjunto de trabajo implican costosas actualizaciones de las matrices. Normalmente, en la práctica, la matriz Z nunca se computa, y se suele trabajar únicamente con la matriz S y con una factorización de la matriz B.

1.5. Métodos cuasi-Newton

Estos métodos se basan, al igual que los métodos de Newton, en el desarrollo de un modelo local cuadrático de la función objetivo F a partir de la información sobre la curvatura de dicha función. Sin embargo, frente a los métodos de Newton, que extraen esta información de la Hesiana de la función, los métodos cuasi-Newton la elaboran a partir del comportamiento de la función F y su gradiente g a lo largo de distintas iteraciones del método descendente.

Sea s_k el paso aplicado al punto solución de la iteración k-*ésima*, x_k, el gradiente en el siguiente punto usando series de Taylor será:

$$g(x_k + s_k) = g_k + G_k s_k + \cdots \tag{1.14}$$

La curvatura de la función F viene dada por $s_k^T G_k s_k$ la cual se puede aproximar usando información de primer orden:

$$s_k^T G_k s_k \approx (g(x_k + s_k) - g_k)^T s_k. \tag{1.15}$$

Al comienzo de la iteración k-*ésima*, el método cuasi-Newton proporciona una aproximación de la Hesiana, B_k, que refleja la información sobre la curvatura acumulada. Si B_k representa la matriz Hesiana, la dirección de búsqueda p_k es la solución del sistema lineal:

$$B_k p_k = -g_k. \tag{1.16}$$

Como valor inicial de la aproximación de la Hesiana B_0 se puede considerar la matriz identidad. Después de que el punto x_{k+1} haya sido calculado, se debe obtener la nueva aproximación de la Hesiana B_{k+1} actualizando B_k para recoger la nueva información adquirida sobre la curvatura. De modo que:

$$B_{k+1} = B_k + U_k, \tag{1.17}$$

donde U_k es una matriz de actualización. La aproximación B_{k+1} debe verificar la condición cuasi-Newton, que deriva de la ecuación 1.15,

$$B_{k+1} s_k = y_k, \tag{1.18}$$

donde $s_k = x_{k+1} - x_k$ e $y_k = g_{k+1} - g_k$. Nótese que $s_k = \lambda_k p_k$.

Durante una iteración, se obtiene información nueva sobre el comportamiento de segundo orden de F a lo largo de una dirección, por lo tanto, B_{k+1} diferirá de B_k en una matriz de bajo rango. Estas modificaciones pueden ser de rango uno o de rango dos, pero deben elegirse de forma que la matriz B_{k+1} sea simétrica y definida positiva. Dependiendo de las actualizaciones empleadas obtendremos distintas expresiones para B_{k+1}, como son, por ejemplo:

- Actualización simétrica de rango uno

$$B_{k+1} = B_k + \frac{1}{(y_k - B_k s_k)^T s_k}(y_k - B_k s_k)(y_k - B_k s_k)^T. \tag{1.19}$$

- Actualización Powell-Symmetric-Broyden (PSB)

$$B_{k+1} = B_k + \frac{1}{s_k^T s_k}((y_k - B_k s_k)s_k^T + s_k(y_k - B_k s_k)^T) - \frac{(y_k - B_k s_k)^T s_k}{(s_k^T s_k)^2} s_k s_k^T. \tag{1.20}$$

- Actualización Davidon-Fletcher-Powell (DFP)

$$B_{k+1} = B_k - \frac{1}{s_k^T B_k s_k} B_k s_k s_k^T B_k + \frac{1}{y_k^T s_k} y_k y_k^T + (s_k^T B_k s_k) w_k w_k^T, \tag{1.21}$$

donde

$$w_k = \frac{1}{y_k^T s_k} y_k - \frac{1}{s_k^T B_k s_k} B_k s_k. \tag{1.22}$$

- Actualización Broyden-Fletcher-Goldfarb-Shanno (BFGS)

$$B_{k+1} = B_k + \frac{1}{s_k^T B_k s_k} B_k s_k s_k^T B_k + \frac{1}{y_k^T s_k} y_k y_k^T. \tag{1.23}$$

Partiendo de la ecuación 1.16, $B_k s_k = -\lambda_k g_k$, por lo tanto, la actualización BFGS se reduce a:

$$B_{k+1} = B_k + \frac{1}{g_k^T p_k} g_k g_k^T + \frac{1}{\lambda_k y_k^T p_k} y_k y_k^T. \tag{1.24}$$

Los métodos cuasi-Newton son muy utilizados en la resolución de problemas sin restricciones. Sin embargo, también se utilizan en combinación con otros métodos en problemas con restricciones lineales para hallar la dirección de búsqueda en cada iteración. Dentro de las actualizaciones descritas, la más utilizada y la más potente es la BFGS.

Para computar la dirección de búsqueda en los problemas no lineales con restricciones lineales se han utilizado dos actualizaciones cuasi-Newton: la actualización BFGS y una actualización de rango uno, en la que:

$$B_{k+1} = B_k + \frac{1}{\lambda_k w_k^T p_k} w_k^T, \tag{1.25}$$

donde $w_k = y_k + \lambda_k g_k$.

La fórmula BFGS se utiliza tanto si la longitud de paso $\lambda = \lambda_{max}$ como si $\lambda > \lambda_{max}$, pudiéndose usar la alternativa de la actualización de rango uno cuando $\lambda = \lambda_{max}$, siempre y cuando el resultado sea una matriz definida positiva.

Por último, el algoritmo 1 muestra una descripción de alto nivel del método BFGS para la iteración k-*ésima*. Esta descripción resalta las principales etapas del algoritmo, lo que nos será muy útil en próximas secciones cuando estudiemos el coste de cada una de ellas y abordemos su paralelización.

Algoritmo 1 Algoritmo BFGS

1. Cálculo de la dirección de búsqueda
   Resolver $B_k p_k = -g_k$
2. Búsqueda lineal: Cálculo del tamaño de paso
   Repetir
   Seleccionar el tamaño de paso λ_k
   Evaluar $f(x_k + \lambda_k p_k)$ (y posiblemente $g(x_k + \lambda_k p_k)$)
   Hasta encontrar un λ_k satisfactorio
   $x_{k+1} = x_k + \lambda_k p_k$
   Calcular $g_{k+1} = \nabla f(x_{k+1})$ si no se ha hecho durante la búsqueda lineal
3. Chequear criterio de convergencia. Si se verifica **STOP** sino ir al paso 4.
4. Actualizar la aproximación de la Hesiana
   $s_k = x_{k+1} - x_k$, $y_k = g_{k+1} - g_k$
   $B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}$

1.6. El código de optimización MINOS

MINOS es un programa diseñado por el Laboratorio de Optimización de Sistemas (SOL) de la Universidad de Stanford para resolver problemas de optimización. En este trabajo hemos usado la versión 5.4 [66], revisada en 1995. MINOS está especialmente diseñado para trabajar con problemas de gran tamaño, pudiendo procesar un elevado número de restricciones no lineales. Está implementado en Fortran77 y se puede instalar en cualquier sistema computacional que disponga de una cantidad de memoria razonable y un compilador Fortran.

MINOS se caracteriza por su aplicación en los campos de la ingeniería, economía y finanzas. Resolviendo problemas de control óptimo, redes no lineales, modelos de comercio, análisis del portafolio y equilibrio espacial.

Comprende una serie de algoritmos que permiten resolver problemas de las cuatro principales áreas de la optimización suave:

- El método *simplex* para resolver problemas de programación lineal, es decir, función objetivo y restricciones lineales.
- Un método cuasi-Newton para abordar problemas sin restricciones.
- El método de gradiente reducido junto con un algoritmo cuasi-Newton para resolver problemas de optimización con restricciones lineales (función objetivo no lineal y restricciones lineales) [64].

- El método de la Lagrangiana proyectada para resolver problemas de optimización con restricciones no lineales. Se trata de un método iterativo que considera subproblemas con las restricciones linealizadas y una Lagrangiana aumentada como función objetivo [65].

MINOS es especialmente eficiente resolviendo problemas lineales, y problemas con una función objetivo no lineal y restricciones dispersas y lineales, como por ejemplo, los problemas cuadráticos. Entre las principales características de MINOS destaca el uso de derivadas primeras exclusivamente, y la estabilidad numérica de sus algoritmos. Otra característica reseñable es que utiliza como entrada de datos el formato MPS [63]. Este formato, introducido por IBM, se ha convertido en un estándar de la industria, adoptado por la mayoría de los códigos comerciales para definir los problemas de optimización. MINOS también es accesible a través de otros tipos de lenguajes o entornos, como son: AMPL [25], GAMS [10], MATLAB y la librería de problemas CUTE [8].

En este trabajo nos hemos centrado en problemas no lineales de gran tamaño sin restricciones o con restricciones lineales, para los que MINOS es especialmente eficiente. Los algoritmos utilizados por MINOS en la resolución de este tipo de problemas han sido descritos en las secciones anteriores. Hemos utilizado este código para analizar estos algoritmos y determinar cuáles son las subrutinas computacionalmente más costosas para posteriormente paralelizarlas. El rendimiento obtenido en nuestras implementaciones paralelas se compara con los algoritmos secuenciales utilizados en MINOS.

1.6.1. Método del gradiente reducido

El método del gradiente reducido es una de las principales técnicas propuestas para la optimización de problemas no lineales sujetos a restricciones lineales. Inicialmente, fue propuesto por Wolfe [90], y posteriormente fue generalizado para solucionar también problemas con restricciones no lineales [1].

Este tipo de métodos posee la característica de reducir el problema original a un problema con restricciones de frontera ($l \leq x \leq u$) en un espacio de menor dimensión. Suponiendo que las m filas de la matriz A del problema 1.2 son linealmente independientes, el método del gradiente reducido divide el espacio n-dimensional en dos subespacios: un subespacio m-dimensional definido por las filas de A, y el subespacio complementario, formado por vectores ortogonales a las filas de A. De este modo, distinguimos dos clases de variables, las llamadas variables básicas asociadas al espacio m-dimensional, y las variables no básicas asociadas al subespacio com-

plementario. Si además el problema presenta no linealidades en su función objetivo, las variables no básicas se dividen en dos categorías: las variables que permanecen fijas al valor de sus fronteras, que conservan el nombre de no básicas, y las variables superbásicas, que son libres de moverse entre sus fronteras en cada iteración.

Siguiendo con la nomenclatura de las secciones anteriores pero desprendiéndonos de los subíndices que indican la iteración, definimos s como el paso que aplicamos al punto x para obtener una nueva aproximación de la solución. Si la función $F(x)$ es cuadrática para que el punto $x+s$ sea un punto estacionario, el paso s deberá de verificar dos propiedades:

- Deberá permanecer en la superficie determinada por la intersección de las restricciones activas. Lo que equivale a decir que,

$$\begin{pmatrix} B & S & N \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} s_B \\ s_S \\ s_N \end{pmatrix} = 0, \tag{1.26}$$

  donde s_B, s_S y s_N representan el paso seleccionado para las variables básicas, superbásicas y no básicas respectivamente.

- Deberá ser tal que el gradiente en el nuevo punto $x+s$ sea ortogonal a la superficie determinada por las restricciones activas,

$$\begin{pmatrix} g_B \\ g_S \\ g_N \end{pmatrix} + G \begin{pmatrix} s_B \\ s_S \\ s_N \end{pmatrix} = \begin{pmatrix} B^T & 0 \\ S^T & 0 \\ N^T & I \end{pmatrix} \begin{pmatrix} \mu \\ \sigma \end{pmatrix}. \tag{1.27}$$

Si la función F no es cuadrática, el paso s así definido no conducirá directamente a un punto estacionario, pero si garantizará una dirección factible descendente. De la ecuación 1.26, deducimos que: $s_N = 0$ y $s_B = B^{-1}Ss_S$. Por lo tanto,

$$s = \begin{pmatrix} -B^T S \\ I \\ 0 \end{pmatrix} s_S, \tag{1.28}$$

o lo que es lo mismo, $s = Zs_S$.

Si denotamos por W el producto $B^{-1}S$, multiplicando la ecuación 1.27 por la matriz

$$\begin{pmatrix} I & 0 & 0 \\ -W^T & I & 0 \\ 0 & 0 & I \end{pmatrix},$$

obtenemos las expresiones para las estimaciones de los multiplicadores de Lagrange

$$B^T\mu = g_B + \begin{pmatrix} I & 0 & 0 \end{pmatrix} G \begin{pmatrix} -W \\ I \\ 0 \end{pmatrix} s_S, \tag{1.29}$$

y

$$\sigma = g_N - N^T\mu + \begin{pmatrix} 0 & 0 & I \end{pmatrix} G \begin{pmatrix} -W \\ I \\ 0 \end{pmatrix} s_S, \tag{1.30}$$

Si el punto x es un punto estacionario, entonces $||s_S|| = 0$, y las ecuaciones 1.29 y 1.30 se reducen a

$$B^T\mu = g_B, \tag{1.31}$$

y

$$\sigma = g_N - N^T\mu, \tag{1.32}$$

respectivamente. De estas expresiones, deducimos que μ es análogo al vector de precios y σ al vector de costes reducidos del método *simplex* [15]. Finalmente, de las modificaciones efectuadas sobre la ecuación 1.27, también se deduce la expresión que debe verificar el paso seleccionado para que sea apropiado:

$$\begin{pmatrix} -W^T & I & 0 \end{pmatrix} \begin{pmatrix} -W \\ I \\ 0 \end{pmatrix} s_S = -h, \tag{1.33}$$

siendo $h = (-W^T \quad I \quad 0)g = g_S - W^T g_B = g_S - S^T\mu$.

La expresión 1.33 define el paso de Newton para las variables independientes (s_S). Por lo tanto, $h = Z^T g$ equivale al gradiente de la función, y se denomina gradiente reducido. Asimismo, la función de la Hesiana la realiza el producto $Z^T GZ$, que recibe el nombre de Hesiana reducida.

Si distinguimos en el cálculo del paso s el cómputo de la dirección de paso p y del tamaño de paso λ tendremos que, en cada iteración del método del gradiente reducido, los principales pasos son:

1. Computar el gradiente reducido $h = Z^T g$.

2. Obtener una aproximación de la Hesiana reducida $Z^T GZ$.

3. Obtener la dirección de búsqueda para las variables superbásicas resolviendo el sistema de ecuaciones $Z^T GZp_S = -h$.

4. Computar la dirección de búsqueda para las variables básicas $p_B = -Wp_S$, y por lo tanto, determinar la dirección de búsqueda:

$$p = \begin{pmatrix} p_B \\ p_S \\ 0 \end{pmatrix}. \tag{1.34}$$

5. Hacer una búsqueda lineal para obtener una aproximación del tamaño de paso λ^* tal que,

$$F(x + \lambda^* p) = min \quad F(x + \theta p), \quad 0 < \theta \leq \lambda_{max}. \tag{1.35}$$

En cada iteración se comprobará si el método converge en el subespacio de trabajo actual, y de no ser así se proporcionarán los mecanismos necesarios para modificarlo. A lo largo del método se producen cambios en los conjuntos de variables básicas, superbásicas y no básicas. Esos cambios se tendrán que reflejar en la matriz básica B y en la aproximación de la Hesiana. Para que estas modificaciones se realicen de forma eficiente, se suelen utilizar factorizaciones y métodos numéricos estables basados en transformaciones ortogonales.

Capítulo 2

Sistemas paralelos y su programación

La evolución de los computadores ha estado siempre fuertemente ligada a la evolución de la tecnología y a las necesidades de las aplicaciones computacionales. El área de las ciencias de la computación y la ingeniería demandan cada vez más recursos de memoria y mayores velocidades de operación. Estas necesidades de mejora han originado el desarrollo de arquitecturas con mejores prestaciones. En esta evolución de los computadores, las arquitecturas paralelas son un paso inevitable en la consecución de mayores velocidades de procesamiento.

El procesamiento paralelo [52] se basa en la idea de dividir un problema en un conjunto de partes resolubles de forma concurrente, de manera que el tiempo total de resolución del problema queda dividido por el número de procesadores utilizados. El grado de consecución de este objetivo depende básicamente de dos factores: de la sobrecarga computacional generada por la paralelización del problema y del equilibrio de carga conseguido entre el conjunto de procesadores.

Una de las principales ventajas de los computadores masivamente paralelos es que la potencia global del sistema no se fundamenta en la potencia individual de cada procesador sino en la del conjunto. De este modo, este tipo de sistemas se pueden configurar a partir de elementos modestos, obteniendo un alto rendimiento a bajo coste.

Sin embargo, existe un obstáculo fundamental para el avance de la computación paralela, y éste es el problema de su programación, ya que los compiladores que detectan paralelismo automáticamente presentan todavía límites a su aplicabilidad. La programación paralela de una arquitectura multiprocesador se realiza mediante lenguajes que permitan expresar el paralelismo e intercambiar información entre los procesadores. En este sentido la meta principal ha sido encontrar un mecanismo para la programación paralela independiente de la máquina.

En este capítulo se describen brevemente las características más importantes de los sistemas paralelos. Así como las arquitecturas del multiprocesador AP3000 de Fujitsu y de un *cluster beowulf*, que se han utilizado para evaluar el rendimiento de los códigos paralelos que han sido desarrollados en este trabajo.

2.1. Arquitecturas paralelas

La definición de procesamiento paralelo dada en la sección anterior implica que las distintas partes en las que se subdivide un problema deberán cooperar y comunicarse entre sí. De esta afirmación se deduce que las arquitecturas paralelas añaden al concepto de arquitectura del computador el de arquitectura de las comunicaciones. En función de como se realicen estas comunicaciones se distinguen varios modelos de programación paralela. Desde una perspectiva tradicional, cada modelo de programación está asociado a una determinada máquina paralela. Así, se puede diferenciar entre multiprocesadores de memoria compartida y multiprocesadores de memoria distribuida [19]. En la actualidad, esta distinción no es muy precisa debido a las muchas similitudes existentes entre diferentes máquinas paralelas y a que muchas máquinas soportan diversos modelos de programación.

Los multiprocesadores de memoria compartida se caracterizan principalmente porque la comunicación entre procesos consiste en lecturas y escrituras de variables compartidas, y ocurren implícitamente como resultado de instrucciones convencionales de acceso a memoria. La arquitectura de estas máquinas consiste en un espacio físico de memoria que puede ser accedido por todos los procesadores y un dispositivo de interconexión que permite el acceso a memoria a los distintos procesadores y dispositivos de entrada/salida. Este dispositivo de interconexión ha evolucionado desde un *crossbar switch* hasta un bus, pasando por una red de interconexión de múltiples etapas. Con el mecanismo de acceso a través de bus todas las direcciones de memoria son equidistantes para todos los procesadores, de ahí que se denominen máquinas de acceso uniforme a memoria.

La desventaja de este tipo de multiprocesadores está en que el bus no permite una alta escalabilidad, debido a que posee un ancho de banda fijo. Una solución a este problema son las máquinas NUMA (*Non Uniform Memory Access*) donde cada procesador posee su propia memoria local, y se permite el acceso de un procesador a los datos situados en la memoria de otro procesador a través de una red de interconexión escalable. Entre los ejemplos de este estilo de diseño nos encontramos con el Cray T3E [84] que presenta coherencia cache parcial; y con el SGI Origin 2000 [53] con coherencia cache.

Otra clase de máquinas paralelas son los multiprocesadores de memoria distribuida. Estos sistemas tienen una arquitectura similar a la de una máquina NUMA, un conjunto de procesadores, con su memoria y dispositivos de entrada/salida, unidos mediante una red de interconexión escalable. La diferencia estriba en que en este tipo de máquinas las comunicaciones consisten en operaciones de entrada/salida mientras que en una máquina NUMA consisten en simples instrucciones de acceso a memoria. El estilo de diseño de los multiprocesadores de memoria distribuida es similar al de una red de estaciones de trabajo o un *cluster*, pero con una red de mayor capacidad. Puesto que la programación en este tipo de máquinas se basa en comunicaciones punto a punto, los procesadores y la red deberán estar fuertemente interconectados. En los primeros multiprocesadores de memoria distribuida la topología de la red era un factor importante porque sólo se podían realizar comunicaciones con los procesadores vecinos. Ejemplos, de distintos tipos de topología son los hipercubos y las mallas de dos y tres dimensiones. En la actualidad, con la introducción de redes de propósito general se ha conseguido reducir la importancia de la topología de la red. Los costes de comunicación entre los distintos procesadores se han uniformado, y el coste de un mensaje está dominado por la latencia de la red. Todas estas mejoras en la red de interconexión han simplificado considerablemente la programación de este tipo de máquinas. Algunos multiprocesadores de memoria distribuida son el IBM SP Power3, el Intel Paragon XP MP, y el Fujitsu AP3000. En [81] podemos encontrar información sobre estos sistemas así como una relación de los supercomputadores actuales más potentes.

La evolución a nivel *hardware* y *software* ha difuminado los límites entre los multiprocesadores de memoria compartida y los multiprocesadores de memoria distribuida. A nivel de programación, las máquinas de memoria compartida soportan comunicaciones punto a punto a través de *buffers* compartidos; y las máquinas de memoria distribuida pueden realizar la comunicación entre dos procesos mediante el acceso a variables compartidas a través de código creado por el sistema operativo. A nivel de arquitectura de la máquina la convergencia se ve claramente en los sistemas NUMA. En los últimos años, además de esta convergencia, y debido a la introducción de redes locales basadas en *switches*, como la Fast Ethernet y la Myrinet, han surgido los *clusters*, que son conjuntos de máquinas que pueden funcionar como sistemas paralelos para tratar de resolver un problema concreto, o como máquinas individuales soportando multiprogramación. Ejemplos de esta clase de sistemas son el Compaq HPC320 [22] y el HP AlphaServer GS1280 [88].

2.2. Arquitecturas *Beowulf*

El precio, mantenimiento y desarrollo de los supercomputadores fuertemente acoplados suele estar muy por encima del coste requerido para justificar la inversión de las diversas áreas industriales que potencialmente se beneficiarían de la computación de alto rendimiento. Esta descompensación entre la necesidad de computación de alto rendimiento y la disponibilidad de los recursos requeridos para desarrollarla, ha sido subsanada por una nueva tendencia en arquitectura paralela, *cluster computing*, que de una forma innovadora conjuga conceptos clásicos en computación paralela, como el modelo de paso de mensajes, con nuevas tecnologías de bajo coste provenientes del mercado de la informática de consumo [3].

La experiencia acumulada durante décadas en el área del procesamiento paralelo y la arquitectura de computadores paralela establece una serie de factores que caracterizan el comportamiento de los sistemas multiprocesador como son [24]:

- La dependencia crítica del rendimiento con la latencia y ancho de banda de la red de interconexión.

- La necesidad de *software* de control ligero.

En general, las arquitecturas en *cluster* no presentan un buen comportamiento en ninguno de estos puntos. La latencia y ancho de banda típico de las redes de interconexión empleadas en los *clusters* pueden llegar a diferir en dos o más ordenes de magnitud respecto a las empleadas en los supercomputadores fuertemente acoplados. Además, habitualmente, en los nodos computacionales de los *clusters* se instalan sistemas operativos completos que ocupan más memoria y responden de forma más lenta que el *software* específico que usan los nodos de los sistemas multiprocesador convencionales.

Los inconvenientes citados anteriormente impiden alcanzar un rendimiento satisfactorio para un determinado tipo de cargas de trabajo, sin embargo, en un amplio rango de aplicaciones es posible emplear eficientemente las arquitecturas en *cluster*.

La principal ventaja de los *clusters* es la relación coste-rendimiento que presentan en comparación con los sistemas multiprocesador convencionales. Además, en relación con estos supercomputadores, los *clusters* ofrecen una mayor flexibilidad ya que permiten especificar todos los componentes del sistema (el número de nodos, la cantidad memoria y número de procesadores por nodo, la red de interconexión y su topología, ...) en función de las necesidades particulares del usuario final, sin añadir ningún coste adicional derivado de esta configuración a medida. Esta gran

flexibilidad de las arquitecturas en *cluster* permite una rápida respuesta de estos sistemas a los avances tecnológicos, ya que resulta fácil instalar nuevos dispositivos.

En los últimos años, dentro de las arquitecturas en *cluster* se ha extendido el empleo de configuraciones duales de los nodos computacionales hasta convertirse gradualmente en un estándar. El ahorro en el coste y en el consumo de potencia, la reducción del espacio y complejidad de cableado, y la posibilidad de emplear el paradigma de memoria compartida, entre otras, han sido las principales razones que han llevado a este hecho. A modo de ejemplo[1], el coste de un *cluster* de 16 procesadores equipado con nodos Compaq Proliant DL320 G5 (Dual Core Intel Xeon 3070 a 2.66 GHz) empleando un *switch* Myrinet de 16 puertos resulta un 23 % más caro que el correspondiente sistema de 8 nodos Compaq Proliant DL360 G5 duales (Dual Core Intel Xeon 5150 a 2.66 GHz), interconectados con un *switch* Myrinet de 8 puertos. Además, esta diferencia en precio crece significativamente al aumentar el número de nodos, ya que el coste de la red de interconexión no crece linealmente con el tamaño del sistema; por ejemplo, el coste de un sistema de 128 nodos es un 27 % mayor que el de su sistema homólogo de 64 nodos.

Una de las principales ventajas del uso de nodos duales es la potencial reducción en el tiempo de comunicaciones intranodo (entre dos procesos ejecutándose en el mismo nodo), ya que pueden realizarse mediante memoria compartida. Sin embargo, para determinadas aplicaciones las configuraciones duales pueden resultar altamente ineficientes debido a la competición de los procesadores de un mismo nodo por los recursos compartidos, memoria principal e interfaz de red.

2.2.1. Redes de Interconexión

En el diseño de un *cluster* es posible elegir entre una amplio abanico de redes de interconexión que van, desde redes LAN estándar de bajo coste, hasta redes especialmente diseñadas para la computación en *cluster* de precio elevado. La elección de la tecnología de la red de interconexión caracteriza en gran medida tanto el precio, como el rendimiento final del sistema y ha de realizarse en base a múltiples factores que incluyen el coste, el rendimiento de la red en términos de su latencia y ancho de banda, la compatibilidad con el hardware y software de otros componentes del *cluster* y el patrón específico de comunicaciones de las aplicaciones que se ejecutarán en el sistema.

Dentro de las distintas redes de interconexión, podemos distinguir; la red TCP/IP sobre Fast Ethernet que emplea en las transmisiones realizadas el protocolo de co-

[1] Estimación basada en los precios oficiales de Compaq y Myricom, Enero de 2007.

municaciones TCP/IP, y la red Myrinet basada en *switches*. Típicamente, el envío de mensajes usando el protocolo TCP/IP implica la realización de copias entre *buffers* de sistema y el espacio de memoria de usuario del proceso. Estas copias, y las interrupciones que llevan asociadas, representan una parte significativa del tiempo total de envío del mensaje, lo que supone una gran sobrecarga en las comunicaciones que se hace más ostensible según aumenta el ancho de banda de la red.

En los últimos años se ha trabajado mucho para mejorar el rendimiento de las comunicaciones en este tipo de sistemas. En este contexto, se han propuesto dos técnicas: el protocolo de red a nivel de usuario y el protocolo de copia cero [32].

El protocolo de red a nivel de usuario elimina la sobrecarga producida por el procesamiento del sistema operativo habilitando el acceso directo del proceso a la red, de forma que una vez reservada la memoria de los *buffers* de envío y recepción, no es necesario realizar ninguna llamada al sistema para enviar o recibir un mensaje. El diseño y desarrollo de este protocolo ha conducido a la consolidación del estándar VIA (*Virtual Architecture Interface*) [18].

La otra técnica utilizada, el protocolo de copia cero, se basa en la idea de que los datos se pueden enviar/recibir directamente desde/en las aplicaciones sin necesidad de hacer copias de los mismos en *buffers* intermedios. Un ejemplo de implementación experimental de esta técnica se puede encontrar en Trapeze [2] basado en Myrinet.

Una red que combina ambas técnicas es la red Myrinet. Desarrollada por la compañía Myricom, surge de dos proyectos de investigación, el California Institute of Technology Mosaic y el University of Southern California Information Sciences Institute Atomic LAN. La red Myrinet [7] deriva de la tecnología de comunicación de los procesadores masivamente paralelos, tratándose, por lo tanto, de una red de baja latencia. Consiste en una serie de conexiones *full-duplex* punto a punto entre los procesadores y los *switches*. Los *switches* Myrinet pueden proporcionar hasta 16 puertos, pudiéndose conectar formando topologías regulares o irregulares dando lugar a una gran red. Otra característica destacada de la tecnología Myrinet es la utilización de adaptadores de red programables, consistentes en: un procesador RISC de 32 bits denominado LANai9, una memoria SRAM desde 2 Mbytes a 8 Mbytes, un puente PCI y un controlador de DMA. Esta arquitectura hace de la red Myrinet una red de alto rendimiento y alta disponibilidad que proporciona una interconexión idónea en *clusters*.

2.3. Modelos de programación

Las aplicaciones paralelas deben ser escritas siguiendo un modelo de programación concreto. El caso más simple de ejecución paralela consiste en el modelo de multiprogramación, en el cual varios programas secuenciales son ejecutados simultáneamente sobre diferentes procesadores sin ninguna interacción entre ellos. Pero el caso más interesante lo constituyen los programas paralelos propiamente dichos. Básicamente se puede decir que existen cuatro alternativas para construir un programa paralelo: la paralelización manual usando pase de mensajes, la paralelización manual de sistemas de memoria compartida, la paralelización semi-automática basada en el paradigma de paralelismo de datos, y la paralelización automática. A continuación vamos a realizar una breve introducción a cada una de ellas.

- **Modelo de programación de pase de mensajes**: en este modelo se define un conjunto de procesos con su propio espacio de memoria, pero que pueden comunicarse con otros procesos mediante el envío y la recepción de mensajes a través de la red de interconexión. La implementación de esta metodología se suele realizar añadiendo librerías específicas a los lenguajes de programación estándar, fundamentalmente C y Fortran. Actualmente, existen también librerías estándar ampliamente difundidas como PVM (*Parallel Virtual Machine*) [30, 31] y MPI (*Message Passing Interface*) [41, 59]. Se utiliza principalmente en multiprocesadores de memoria distribuida y en redes de estaciones de trabajo, lo cual permite usar este modelo en redes heterogéneas compuestas por diferentes sistemas con distintas arquitecturas.

- **Modelo de programación de sistemas de memoria compartida**: los programas paralelos ejecutados en sistemas de memoria compartida se descomponen en varios procesos que comparten los datos asociados a una porción de su espacio de direcciones. La coordinación y cooperación entre los procesos se realizan a través de la lectura y escritura de variables compartidas y a través de variables de sincronización. Cada proceso puede llevar a cabo la ejecución de un subconjunto de iteraciones de un lazo común, o bien, de forma más general, cada proceso puede obtener sus tareas de una cola compartida. La programación más eficiente, aunque difícil, se establece a través de construcciones de bajo nivel tales como barreras de sincronización, regiones críticas, *locks*, semáforos, etc.

- **Modelo de programación de paralelismo de datos**: se utiliza principalmente para simplificar la programación de los sistemas de memoria distribuida.

En esta aproximación, a un programa secuencial se le añaden directivas o se le insertan anotaciones para guiar al compilador en su tarea de distribuir los datos y las computaciones. Los lenguajes de paralelismo de datos más extendidos son CM-Fortran (*Connection Machine Fortran*) [86], Fortran D [26], Craft [77], Vienna Fortran [91] y, más recientemente, el *High Performance Fortran* (HPF) [45].

- **Paralelización automática**: el compilador asume todas las estrategias y decisiones, y genera automáticamente la versión paralela equivalente al código secuencial escrito en un lenguaje de programación convencional. En general los paralelizadores automáticos actuales ofrecen buenos resultados cuando los códigos a paralelizar tienen patrones de acceso regulares a los datos. Sin embargo, la paralelización automática de códigos irregulares, y entre ellos los códigos dispersos, es una tarea mucho más complicada y que no ha encontrado todavía una solución eficiente.

2.4. Grado de paralelismo

Otra característica de los sistemas paralelos es lo que se denomina granularidad. La granularidad es una forma de medir el grado de paralelismo que explota el sistema [27], indicando la cantidad de computaciones que pueden realizar los procesadores sin interaccionar entre ellos. Así, en un modelo de granularidad gruesa, el programa se divide en varias partes que precisan poca comunicación entre sí. Los sistemas paralelos formados por procesadores potentes y débilmente interconectados, parecen más adecuados para aplicaciones de granularidad gruesa. Por el contrario, en los modelos de granularidad fina, la comunicación entre los procesadores es más intensa y se ejecutan relativamente pocas instrucciones sin necesidad de comunicaciones. Para las aplicaciones de granularidad fina, los sistemas con procesadores sencillos y fuertemente interconectados entre sí resultan más eficientes.

Para la realización de esta memoria se han diseñado programas paralelos que se ejecutan sobre sistemas multiprocesador de memoria distribuida y sobre sistemas *cluster*. Por un lado, hemos explotado el paralelismo a nivel de lazo (granularidad media) y se ha utilizado un modelo SPMD (Simple Programa, Múltiple flujo de Datos) en el que todos los procesadores ejecutan el mismo programa de forma independiente sobre distintos datos. Por otro lado, hemos propuesto también un paralelismo de grano grueso, de modo que las mismas iteraciones del algoritmo se computan a la vez en todos los procesadores pero con distintos parámetros iniciales.

Cada cierto tiempo, los distintos procesadores se comunican entre sí para ver qué procesador converge mejor hacia la solución; y todos continúan con las iteraciones partiendo de los datos recogidos, aportando cada uno ligeras modificaciones.

2.5. Sistemas paralelos sobre los que se ha desarrollado la programación

La programación paralela de un sistema multiprocesador se realiza mediante lenguajes que permitan expresar el paralelismo e intercambiar información de los procesadores entre sí [89].

A lo largo de esta memoria utilizaremos el sistema paralelo AP3000 de Fujitsu y un *cluster beowulf* para validar nuestras propuestas paralelas. Los códigos han sido paralelizados siguiendo el modelo de programación de pase de mensajes utilizando MPI (*Message Passing Interface*). Para diseñar nuestros programas hemos utilizado los lenguajes de programación C y Fortran junto con las rutinas de comunicación de pase de mensajes. El modelo básico que usaremos será el SPMD, de tal forma que todos los procesadores ejecutan el mismo código sobre diferentes conjuntos de datos. Los procesadores trabajan de forma independiente, sincronizándose, si es necesario, en las etapas de comunicación.

2.5.1. Multiprocesador de memoria distribuida AP3000 de Fujitsu

El AP3000 de Fujitsu [46] es un sistema que puede considerarse como un computador paralelo de memoria distribuida o como un grupo (*cluster*) de estaciones de trabajo unidas por una red de alta velocidad. Este sistema utiliza procesadores UltraSPARC de 64–bits. En estos procesadores la cache de primer nivel se encuentra particionada para instrucciones y datos, con 16 Kbytes de capacidad para cada una de las partes. La cache de segundo nivel se comparte entre instrucciones y datos y tiene una capacidad de 2 Mbytes.

Entre las características más importantes del AP3000 se encuentran las siguientes:

- Implementación de alto rendimiento utilizando procesamiento paralelo multinodo: habitualmente en el procesamiento paralelo, las comunicaciones entre los nodos afectan en gran modo al rendimiento general del sistema. Para poder alcanzar un alto rendimiento en procesamiento paralelo, es necesario un sistema

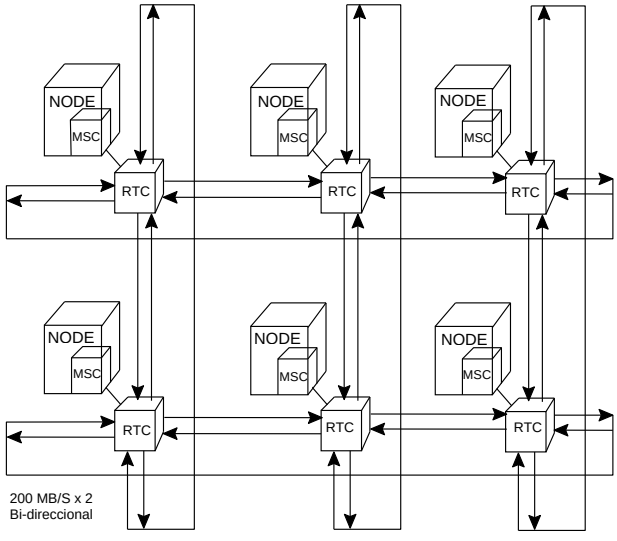


Figura 2.1: AP–Net del Fujitsu AP3000.

de comunicaciones entre nodos que presente una baja latencia y un alto ancho de banda. Para conseguir aumentar la velocidad de transmisión, el AP3000 utiliza una red de comunicación de alta velocidad denominada AP–Net. Además, el AP3000 consigue un nivel de latencia reducido y un elevado ancho de banda en las comunicaciones entre nodos. Para que el nivel de latencia sea bajo, es importante no sólo aumentar la velocidad de transmisión de datos en la red, sino también reducir de forma significativa el tiempo necesario para establecer y configurar el envío de los mensajes. Por ello, el AP3000 soporta un sistema de comunicaciones a nivel de usuario de forma que la comunicación de mensajes puede ser activada directamente sin ningún tipo de ayuda por parte del sistema operativo.

- Soporte para el aumento del número de nodos y de canales de entrada/salida: el AP3000 tiene la capacidad de aumentar de un modo sencillo, el grupo de estaciones de trabajo que lo forman, así como la ampliación de su red de comunicación de alta velocidad. Para la red AP–Net se utiliza una red toroidal bidimensional con alto nivel de expansión, y su escalabilidad le permite soportar desde 4 hasta 1024 nodos, pudiendo llegar a obtener un pico de rendimiento de la máquina en su configuración máxima de 614 Gflops/s.

La red de interconexión AP–Net está basada en una topología toroidal bidimensional y consiste en un conjunto de controladores de encaminamiento (RTCs)

Número de PEs	4 a 1024
Ciclo de reloj	3.3 ns
Rendimiento teórico por procesador	600 Mflops
Rendimiento teórico máximo	614 Gflops
Procesador	SPARC
Memoria máxima por nodo	2 Gbytes
Memoria máxima	2 Tbytes
Velocidad de la red de interconexión	200 Mbytes/s
Estructura de la red	toro 2–D

Tabla 2.1: Características del AP3000.

encargados de dirigir los mensajes. En la figura 2.1 se representa un esquema de la AP–Net. La tarjeta MSC (*Message controller*) se encuentra conectada a cada nodo a través de un SBus (bus de entrada/salida) para su conexión con la AP–Net. La tarjeta MSC está formada por un chip controlador de mensajes (MSC) y una memoria *buffer*. El MSC incluye controladores DMA para la transferencia de datos con la AP–Net. La velocidad máxima de transferencia es de 200 Mbytes/s. Otras características de este sistema se muestran en la tabla 2.1.

2.5.2. *Cluster beowulf* del CESGA

Existen varios sistemas que pueden encuadrarse dentro de las arquitecturas en *cluster*, en particular únicamente consideraremos sistemas tipo *beowulf*. Un *cluster beowulf* es un sistema de cálculo local compuesto por un conjunto independiente de computadores o nodos y una red que los conecta. El término local hace referencia al hecho de que todos los componentes y subsistemas del *beowulf* se supervisan y administran atendiendo a criterios comunes, de forma que el conjunto de nodos se trata como un único sistema. Los nodos computacionales que constituyen el *beowulf* son PCs o multiprocesadores simétricos (*Symmetric Multiprocessors*, SMP) de PCs integrados mediante una red de área local (*Local Area Network*, LAN) o una red de área de sistema (*System Area Network*, SAN). El sistema operativo que se instala en los nodos es alguna variedad de Unix, típicamente Linux o Solaris, aunque también es posible encontrar plataformas basadas en Windows NT.

El sistema *beowulf* evaluado en este trabajo, instalado en el Centro de Supercomputación de Galicia [14] es una plataforma homogénea que consta de 16 servidores Compaq DL320 que forman un total de 16 CPUs Pentium III a 1GHz, con un rendi-

Componente	**Tipo**
CPU	Intel Pentium III 1GHz
Memoria	512 Mbytes
Disco	ATA 40 Gbytes
Sistema Operativo	Rocks 4.0
Compilador	Portland Group version 3.3-2
Librería MPI	MPI Software Technology MPICH-GM 1.2.1..7b

Tabla 2.2: Configuración *hardware* y *software* de los nodos del *cluster beowulf* del CESGA.

miento pico de 16 Gflops. Cada nodo tiene 512 Mbytes de memoria y un disco local ATA de 40 Gbytes (ver la tabla 2.2). Los nodos están interconectados mediante una red Myrinet 2000 [67] de baja latencia (para mensajes MPI $<8\mu s$) y alto ancho de banda (200 Mbytes/s). Además, todos los nodos están conectados mediante un *switch* Fast Ethernet a un servidor Compaq DL380 (ver figura 2.2) que actúa como servidor de almacenamiento y como *front-end* para la conexión de los usuarios y las tareas iterativas (compilación, edición de archivos, etc, ...). Este servidor cuenta con 4 discos Ultra3 SCSI de 36 Gbytes en configuración RAID 5.

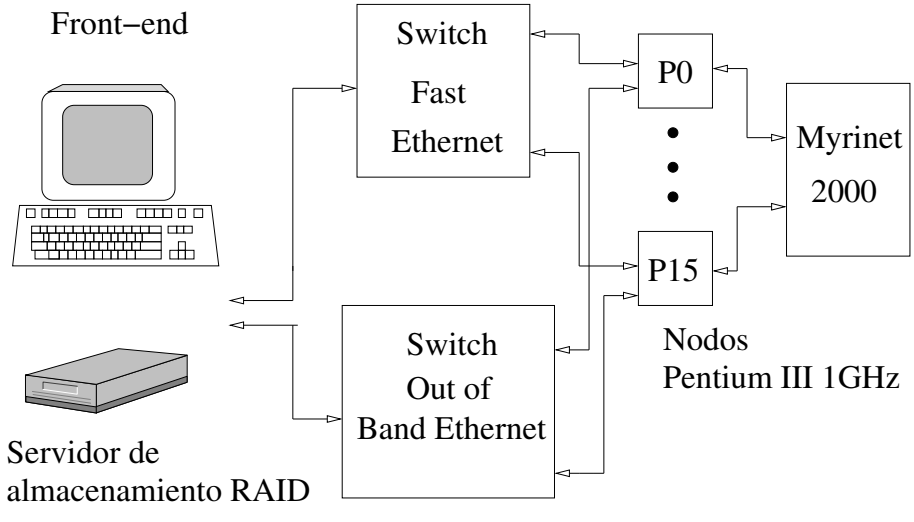


Figura 2.2: Esquema del sistema *beowulf* instalado en el CESGA.

Capítulo 3

Perfil computacional del código de optimización MINOS

En este trabajo se aborda la paralelización de un algoritmo de optimización cuasi-Newton sobre multiprocesadores de memoria distribuida y *clusters* de PCs. El código del que partimos como base es el propuesto en el *software* de optimización MINOS [66].

Abordaremos, en primer lugar, una paralelización de las rutinas algebraicas computacionalmente más costosas. En este campo contamos con una gran experiencia en la paralelización de factorizaciones de matrices dispersas tanto en sistemas de memoria distribuida como en sistemas de memoria compartida [57, 58, 71].

Como fases previas a la paralelización, se deben realizar dos pasos de gran importancia. El primero consiste en analizar el perfil computacional del algoritmo que se va a implementar. Este paso es imprescindible para determinar las rutinas del código que necesitan un mayor tiempo de computación, y establecer qué rutinas son lo suficientemente costosas para que la reducción en el coste computacional compense los gastos inherentes a la implementación paralela de acuerdo con la ley de Amdahl. Este análisis simplificará nuestro trabajo, de tal modo que realizaremos únicamente una implementación paralela de estas últimas rutinas, descartando la paralelización de aquéllas poco costosas que apenas influyen en el rendimiento del algoritmo.

En segundo lugar, habrá que realizar un análisis de las dependencias para cada una de las subrutinas candidatas a ser paralelizadas. De modo que nuestras propuestas paralelas traten de adaptarse a estas dependencias para obtener el máximo rendimiento posible.

En este capítulo, se aborda el estudio del perfil computacional del algoritmo empleado por el código MINOS para resolver problemas de optimización no lineal

en los términos indicados anteriormente. Además, se presenta una descripción de la función que realizan las subrutinas más costosas, y de los algoritmos que las implementan. Por último, se analizan las dependencias de cada una de las rutinas que hemos decidido paralelizar, con el fin de distribuir los datos y las computaciones paralelas adaptándonos a estas dependencias, minimizando el número de mensajes entre procesadores, y explotando al máximo el grado de paralelismo.

3.1. Estudio computacional de un código de optimización basado en un algoritmo cuasi-Newton

En este trabajo tratamos de resolver problemas de optimización con función objetivo no lineal y restricciones, si existen, lineales. Nuestro objetivo es que el método se pueda aplicar eficazmente a problemas de gran tamaño. Por lo tanto, vamos a utilizar el algoritmo del gradiente reducido, basado en un conjunto activo de restricciones, que utiliza un método cuasi-Newton para tratar las no linealidades de la función objetivo.

Para decidir qué rutinas se deben paralelizar se ha realizado un análisis computacional del código de optimización sobre tres de los problemas que se han utilizado como banco de pruebas. Los resultados, especificando los porcentajes de tiempo requeridos por los tres estados de un método cuasi-Newton, se muestran en la tabla 3.1. Se puede observar que el método cuasi-Newton consume la mayor parte del tiempo del código de optimización, de ahí que resulte tan interesante su paralelización. Dentro del algoritmo cuasi-Newton, es la actualización de la Hesiana la que requiere un mayor coste computacional. Por lo tanto, un algoritmo paralelo eficiente tratará de obtener el máximo rendimiento de este estado, aunque ello implique un pequeño coste en la paralelización de los otros dos. Para ello resulta fundamental adaptarse a las dependencias de datos existentes en la aplicación de la actualización de la Hesiana, y realizar la distribución de la matriz aproximación de la Hesiana de forma que cada procesador pueda realizar la mayor cantidad de computaciones posibles antes de que sea necesaria una comunicación con otros procesadores del sistema. Con este objetivo, en las secciones que vienen a continuación se describen las subrutinas que se pretenden paralelizar y se analizan sus dependencias de datos.

En el método de optimización existen otras subrutinas en las que se modifica la aproximación de la Hesiana. Estas subrutinas no tienen prácticamente coste computacional pero se deben tener en cuenta a la hora de distribuir esta matriz si no se desean realizar envíos de datos entre procesadores cada vez que se deba ejecutar

Nombre del problema		HUES-MOD	AUG2D	DTOC1L
Número de variables		1430	1600	1998
Porcentaje de tiempo consumido en los distintos estados	Actualización cuasi-Newton	82.6 %	89.6 %	81.4 %
	Resolución triangular	9.5 %	4.1 %	6.9 %
	Búsqueda del tamaño de paso	1.7 %	3.4 %	5.4 %
	Otros	6.2 %	2.9 %	6.5 %

Tabla 3.1: Porcentaje del tiempo computacional consumido en cada estado de un método cuasi-Newton.

una nueva subrutina. Básicamente lo que hacen estas subrutinas es añadir o eliminar variables del conjunto de trabajo, lo que implica añadir o borrar columnas de la aproximación de la Hesiana. Cada vez que se realiza alguna de estas modificaciones esta matriz pierde su forma triangular, lo que implica una nueva actualización. Este es uno de los motivos por los que la actualización de la Hesiana consume la mayor parte del tiempo de ejecución de nuestro código de optimización. La descripción de la paralelización de estas subrutinas se va a obviar dada su trivialidad y su escasa influencia en el coste computacional del algoritmo.

3.2. Actualización de la Hesiana

En un método cuasi-Newton, como ya se ha indicado previamente, no se utiliza la matriz Hesiana sino una aproximación de la misma. Si la matriz Hesiana es simétrica y definida positiva se puede utilizar el método BFGS, que descompone la matriz según la factorización de Cholesky $B_k = U_k^T U_k$, siendo U_k una matriz triangular superior [64]. Además, para encontrar la dirección de búsqueda se debe resolver el sistema $B_k p_k = -g_k$. Dado que $B_k = U_k^T U_k$, la resolución de este sistema implica la resolución de dos sistemas triangulares. Sin embargo, a lo largo del proceso de optimización, en cada iteración, la matriz U_k sufre modificaciones de rango uno [34],

$$\bar{U}_k = U_k + uv^T,$$

donde u y v son vectores de tamaño n.

Para poder seguir resolviendo estos sistemas modificados como si fuesen triangulares, lo que implica un menor número de operaciones, tendremos que realizar de nuevo la factorización de Cholesky de la matriz modificada. Con el fin de evitar el cálculo de una nueva factorización de Cholesky en cada iteración, que tendría

un coste $O(n^3)$, se trata de obtener dicha descomposición a partir de la actualización de los factores de la matriz U_k [33] en $O(n^2)$. Esta actualización consiste en la aplicación de un conjunto de rotaciones planas que le devuelven a la matriz U_k su forma triangular. Dichas rotaciones se describen a continuación. En primer lugar, definimos formalmente el concepto de matriz de rotación y cuál es el efecto de su aplicación sobre un vector o una matriz. Se generaliza la idea de rotación plana, consiguiendo lo que se denomina rotación global plana, encadenando la aplicación de varias matrices de rotación. Por último, explicamos como se realiza, utilizando las rotaciones globales planas, la factorización QR de una matriz tras haber sufrido una modificación de rango uno. En nuestro algoritmo cuasi-Newton se produce, en cada iteración, una situación similar, siendo necesaria la aplicación de las dos rotaciones globales planas que se describen en esta sección.

3.2.1. Rotaciones planas

En este trabajo se utilizan rotaciones planas para actualizar la factorización de una determinada matriz. La importancia de las factorizaciones es la generación de ceros en posiciones estratégicas de la matriz. Si estos ceros se pierden tras sufrir la factorización una serie de modificaciones, habrá que volver a obtener el patrón triangular original, y las rotaciones planas se aplican con esta finalidad.

Dado un vector v, podemos anular el elemento de v localizado en la posición j, sin más que aplicar a dicho vector una rotación plana. Esta rotación lleva asociada una matriz ortogonal Ψ_{ij} que se define, asumiendo que $v_i \neq 0$ y $v_j \neq 0$, del siguiente modo [40]:

$$r = \sqrt{v_i^2 + v_j^2}, \quad c = \frac{v_i}{r} \quad y \quad s = \frac{v_j}{r}, \tag{3.1}$$

donde c y s se corresponden respectivamente con el coseno y el seno del ángulo de rotación. De modo que la matriz Ψ_{ij} será una matriz que sólo difiere de la identidad en las filas i y j donde los elementos toman los valores:

$$\psi_{ii} = c, \quad \psi_{ij} = s, \quad \psi_{ji} = -s \quad y \quad \psi_{jj} = c, \tag{3.2}$$

Cuando se aplica esta matriz de rotación sobre el vector v ($\Psi_{ij}v$) sólo se modifican las componentes i y j de v permaneciendo las demás componentes inalteradas.

Nuestro algoritmo cuasi-Newton no usa exactamente rotaciones planas sino rotaciones seguidas de una reflexión sobre un eje [33], denominadas transformaciones de Givens. Las reflexiones tienen las mismas propiedades numéricas que las matrices asociadas a rotaciones planas, pero además tienen la característica de ser simétricas.

Para la matriz Ψ_{ij} usada, los elementos de las filas i y j tomarán los siguientes valores:

$$\psi_{ii} = c, \quad \psi_{ij} = s, \quad \psi_{ji} = s \quad y \quad \psi_{jj} = -c. \tag{3.3}$$

3.2.2. Rotaciones globales planas

Una rotación global plana es un conjunto de rotaciones planas aplicadas siguiendo un determinado orden. Formalmente, se define como la matriz ortogonal que resulta de multiplicar las matrices asociadas a las distintas rotaciones que forman la secuencia [35].

Estas rotaciones permiten obtener un patrón específico de ceros, anulando elementos en posiciones y en un orden preestablecidos. Por ejemplo, si se quiere transformar un vector v de dimensión n a un múltiplo de e_0, tendremos que anular las componentes de v_1 a v_{n-1}. Una forma de conseguirlo, aunque no la única, es realizar una rotación global plana en el orden $(n-2, n-1), (n-3, n-2), \cdots, (0, 1)$. Si denominamos a esta rotación S_1, su representación formal sería la matriz,

$$S_1 = \Psi_{0,1} \cdots \Psi_{n-3,n-2} \Psi_{n-2,n-1}.$$

Este proceso se ilustra en la figura 3.1 para un vector de dimensión 4. Cada flecha indica la aplicación de una rotación y los elementos primados son los que han sufrido algún cambio en su valor una vez hecha la rotación.

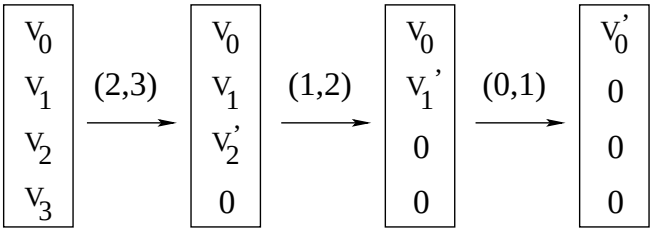


Figura 3.1: Efecto de la rotación global plana $(2, 3), (1, 2), (0, 1)$ sobre un vector v.

3.2.3. Factorización QR

Supongamos el caso general de una factorización QR [40] de una matriz A:

$$A = QR \quad o \quad Q^T A = R,$$

donde Q es una matriz ortogonal y R una matriz triangular superior. Sea $\bar{A}$ una modificación de rango uno de la matriz A, es decir,

$$\bar{A} = A + uv^T,$$

donde $u \neq 0$ y $v \neq 0$. Existe un método, denominado actualización QR, para obtener los factores QR de $\bar{A}$ directamente modificando los de A, sin tener que volver a calcular la factorización [35].

El primer paso del proceso de actualización implica aplicar una parte de la factorización antigua a la nueva matriz. Multiplicando $\bar{A}$ por Q^T, obtenemos:

$$Q^T\bar{A} = Q^T(A + uv^T) = R + wv^T, \quad \textit{donde} \quad w = Q^T u. \tag{3.4}$$

El producto $Q^T\bar{A}$ es una modificación de rango uno de R provocada por los vectores w y v.

Para actualizar los factores, tendremos que considerar una matriz ortogonal $\widetilde{Q}^T$ que transforme la matriz $R + wv^T$ en una matriz triangular superior $\bar{R}$:

$$\widetilde{Q}^T(R + wv^T) = \bar{R}. \tag{3.5}$$

Por lo tanto, $\widetilde{Q}^T$ y $\bar{R}$ son los factores de $R + wv^T$. Dada una matriz $\widetilde{Q}^T$ satisfaciendo 3.5, la ecuación 3.4 implica que:

$$\bar{R} = \widetilde{Q}^T(R + wv^T) = \widetilde{Q}^T Q^T \bar{A}.$$

Como las matrices Q y $\widetilde{Q}$ son ortogonales, su producto $\bar{Q} = Q\widetilde{Q}$ también lo será. El proceso de actualización se reduce a encontrar una matriz ortogonal $\widetilde{Q}^T$ que transforme una modificación de rango uno de una matriz triangular superior R en una matriz triangular superior $\bar{R}$. Esta matriz ortogonal se puede obtener aplicando dos rotaciones globales planas. A continuación, describimos una de las posibilidades para determinar estas rotaciones.

La primera rotación, que denotamos por S_1, transforma la matriz de rango uno wv^T a una estructura que cause la menor influencia posible sobre la estructura de la matriz triangular R. Por lo tanto, se elige S_1 para reducir el vector w a un múltiplo de e_{n-1}, es decir, un vector que tiene todas sus componentes, excepto la última, iguales a cero. De este modo,

$$S_1 w = \gamma e_{n-1}, \quad \textit{donde} \quad |\gamma| = \|w\|_2, \tag{3.6}$$

donde $|\cdot|$ es el módulo de un escalar y $||\cdot||_2$ la norma 2 de una matriz, que se define como el mayor valor singular de la matriz.

Se sigue que:

$$\gamma e_{n-1} v^T = \gamma \begin{pmatrix} 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \\ v_0 & v_1 \cdots & v_{n-2} \quad v_{n-1} \end{pmatrix}, \tag{3.7}$$

y, por tanto, todas las filas de $S_1 w v^T$ son cero excepto la última.

La rotación S_1, llamada rotación global hacia atrás, se define a partir de $n-1$ rotaciones planas siguiendo el orden $(n-1, n-2)$, $(n-1, n-3)$, $\cdots$, $(n-1, 0)$. Formalmente, S_1 se representa como el producto de matrices:

$$S_1 = \Psi_{n-1,0} \cdots \Psi_{n-1,n-3} \Psi_{n-1,n-2}. \tag{3.8}$$

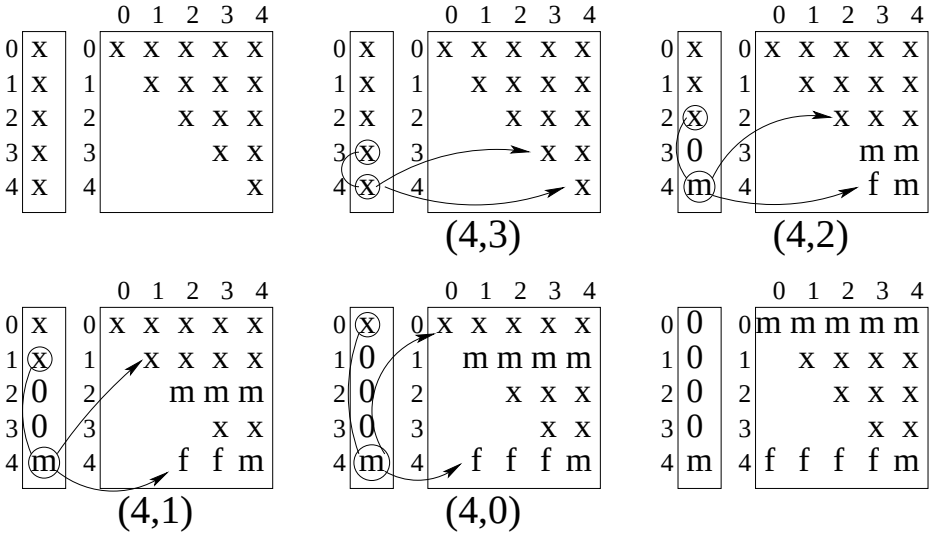


Figura 3.2: Evolución del algoritmo de la rotación global hacia atrás.

El orden utilizado en las rotaciones planas produce el siguiente efecto sobre la matriz triangular superior R: cuando se aplica S_1 a R, la estructura de las primeras $n-1$ filas permanece inalterada, pero la última fila de $S_1 R$ se hace densa. Sin embargo, a pesar de que la forma triangular superior de las $n-1$ primeras filas permanece inalterada, los valores numéricos de estas filas se ven modificados.

El efecto de la rotación S_1 sobre la matriz R puede apreciarse en la figura 3.2 para el caso de una matriz de orden cinco. Cada paso representa una rotación plana. Los elementos de R se representan simbólicamente después de cada rotación del siguiente modo: "x" denota un elemento no nulo de la matriz que no ha sido alterado; "m" representa un elemento modificado; "f" denota un previo cero que se ha convertido en un elemento no nulo.

Después de la primera rotación global, S_1R es una matriz triangular superior con la última fila densa. Dada la estructura que tienen S_1R y S_1wv^T, la suma de estas matrices resulta:

$$R' = S_1R + S_1wv^T = S_1(R + wv^T), \tag{3.9}$$

que es también una matriz triangular superior con la última fila densa.

Para que la matriz R' recupere la forma triangular es necesario eliminar los $n-1$ primeros elementos de su última fila, para ello se aplica una segunda rotación S_2 (denominada rotación global hacia adelante) a través de rotaciones planas en el orden $(0, n-1)$, $(1, n-1)$, $\cdots$, $(n-2, n-1)$.

Para determinar S_2 se utiliza la primera columna de R' para definir la rotación $\Psi'_{0,n-1}$ que crea un elemento nulo en la entrada $(n-1, 0)$ de R', y altera todas las entradas de la primera fila de R'. A continuación, a partir de la segunda columna de $\Psi'_{0,n-1}R'$ definimos la rotación $\Psi'_{1,n-1}$ que produce un cero en la posición $(n-1, 1)$. En este punto, el orden de las rotaciones es crucial, ya que como los elementos $(1, 0)$ y $(n-1, 0)$ son cero, la aplicación de $\Psi'_{1,n-1}$ no modifica el cero introducido anteriormente en $(n-1, 0)$.

Las siguientes rotaciones aplicadas tienen la propiedad de anular los demás elementos de la última fila conservando los ceros creados por las rotaciones previas, como se ilustra en el ejemplo de la figura 3.3.

Finalmente, la rotación global hacia adelante será:

$$S_2 = \Psi'_{n-2,n-1} \cdots \Psi'_{1,n-1}\Psi'_{0,n-1}. \tag{3.10}$$

Después de aplicar completamente esta rotación, los primeros $n-1$ elementos de la última fila de R' vuelven a ser cero por lo que la matriz resultante $\bar{R} = S_2R'$ será una matriz triangular superior. Por otro lado, la matriz ortogonal $\widetilde{Q}$ buscada, se define como:

$$\widetilde{Q}^T = S_2S_1 \quad y \quad \bar{Q} = Q\widetilde{Q} = QS_2S_1, \tag{3.11}$$

de donde se sigue que,

$$\bar{A} = QS_1^TS_2^T\bar{R} = \bar{Q}\bar{R}, \tag{3.12}$$

y $\bar{Q}$ y $\bar{R}$ son los factores de QR de $\bar{A}$. En el caso concreto de un algoritmo cuasi-Newton, la matriz $A = U_k$ sería una matriz triangular superior, de modo que sus

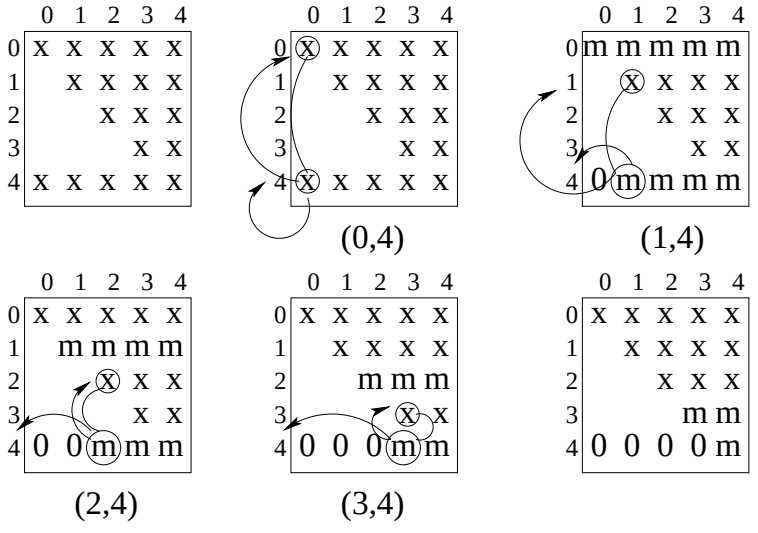


Figura 3.3: Evolución del algoritmo de la rotación global hacia adelante.

factores serían $Q = I$ y $R = U_k$. Sustituyendo en la ecuación 3.4, la ecuación 3.5 queda:

$$\widetilde{Q}^T \bar{U}_k = \bar{R}. \tag{3.13}$$

Por lo tanto, aplicando las dos rotaciones globales planas descritas se obtiene de nuevo una matriz triangular.

3.3. Cálculo de la dirección de búsqueda

En cada iteración de un método cuasi-Newton es necesario calcular la dirección de búsqueda p_k hacia la cual se debe mover la solución desde el punto actual x_k. Para ello es necesario resolver el sistema lineal,

$$B_k p_k = -g_k \tag{3.14}$$

Si sustituimos la aproximación de la Hesiana B_k por su factorización $U_k^T U_k$, la ecuación 3.14 se convierte en:

$$U_k^T U_k p_k = -g_k \tag{3.15}$$

La resolución del sistema lineal de la ecuación 3.15 implica la resolución de dos sistemas triangulares: primero, un sistema triangular inferior $U_k^T q = -g_k$, y posteriormente un sistema triangular superior $U_k p_k = q$.

3.3.1. Resolución de sistemas triangulares

Supongamos el sistema triangular inferior,

$$Lx = b,$$

donde L es una matriz triangular inferior no singular. En forma matricial, el sistema sería:

$$\begin{pmatrix} l_{0,0} & & & \\ l_{1,0} & l_{1,1} & & \\ \vdots & & \ddots & \\ l_{n-1,0} & l_{n-1,1} & \cdots & l_{n-1,n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{pmatrix}. \tag{3.16}$$

En la primera ecuación,

$$l_{0,0}x_0 = b_0,$$

existe sólo una incógnita, la variable x_0, que puede obtenerse inmediatamente de la división,

$$x_0 = \frac{b_0}{l_{0,0}}.$$

La segunda ecuación,

$$l_{1,0}x_0 + l_{1,1}x_1 = b_1,$$

implica dos variables, x_0 y x_1. Como x_0 ya la hemos calculado, x_1 viene dada por,

$$x_1 = \frac{b_1 - l_{1,0}x_1}{l_{1,1}}.$$

Siguiendo con este proceso, para $k > 1$, x_k se puede definir en términos de x_0, $\cdots$, x_{k-1}, como:

$$x_k = \frac{(b_k - l_{k,0}x_0 - l_{k,1}x_1 - \cdots - l_{k,k-1}x_{k-1})}{l_{k,k}}. \tag{3.17}$$

Para $k = 0, \cdots, n-1$, el valor de x_k está bien definido porque $l_{k,k}$ es distinto de cero.

Este proceso eminentemente secuencial también se puede aplicar a la resolución de un sistema triangular superior pero comenzando por la obtención de x_{n-1}. El

proceso de resolución de un sistema triangular inferior se denomina sustitución hacia adelante, mientras que el de la resolución triangular superior recibe el nombre de sustitución hacia atrás.

3.4. Dependencias de datos: actualización de la Hesiana y dirección de búsqueda

En esta sección se analizan las dependencias existentes en dos de las subrutinas más costosas de un método cuasi-Newton [70, 73]. El objetivo de este estudio es proponer un algoritmo paralelo que se adapte a estas dependencias de datos, y que a su vez, sea lo más eficiente posible.

El proceso de actualización de la matriz aproximación de la Hesiana consiste en la aplicación de dos rotaciones globales planas que introducen en esta matriz una modificación de rango uno, pero sin que esto suponga la pérdida de su estructura triangular. Para ilustrar el proceso, supongamos que se dispone de una matriz triangular R de orden 3. Tras la aplicación de una rotación global hacia atrás (S_1) la matriz R se convierte en R', una matriz triangular con la última fila densa. Esta última fila de la matriz se almacena en un vector v. Para recuperar la forma triangular original, la matriz R' sufre una nueva rotación, S_2, convirtiéndose en la matriz triangular $\overline{R}$. Este proceso, así como los grafos de dependencias asociados a la aplicación de estas dos rotaciones, se ilustra en la figura 3.4.

Analizando los grafos se observan muchas similitudes en las dependencias de datos existentes en las dos rotaciones. De hecho, en las operaciones del lazo interno, que abarcan el cómputo de los elementos de la nueva matriz resultante y las actualizaciones de la última fila, observamos el mismo tipo de dependencias. Sin embargo, dentro del lazo externo, éstas son más restrictivas en la rotación global hacia adelante (S_2). Mientras que en la rotación global hacia atrás, el cálculo de los ángulos de rotación se hace de forma independiente y se podría implementar de forma totalmente paralela, en la rotación global hacia adelante, el cálculo de cada ángulo depende de datos que han sido modificados en la iteración anterior. Por lo tanto, debido a que las dependencias más fuertes aparecen en la rotación global hacia adelante, nos centramos en su estudio.

En la figura 3.4 se puede observar que cada rotación $(j, n-1)$ de S_2 modifica los elementos no nulos de la última fila de la matriz así como los elementos de la fila j. A su vez para calcular el ángulo asociado a la rotación $(j+1, n-1)$ se necesita conocer el valor del elemento v_{j+1} que ha sido modificado en la anterior iteración

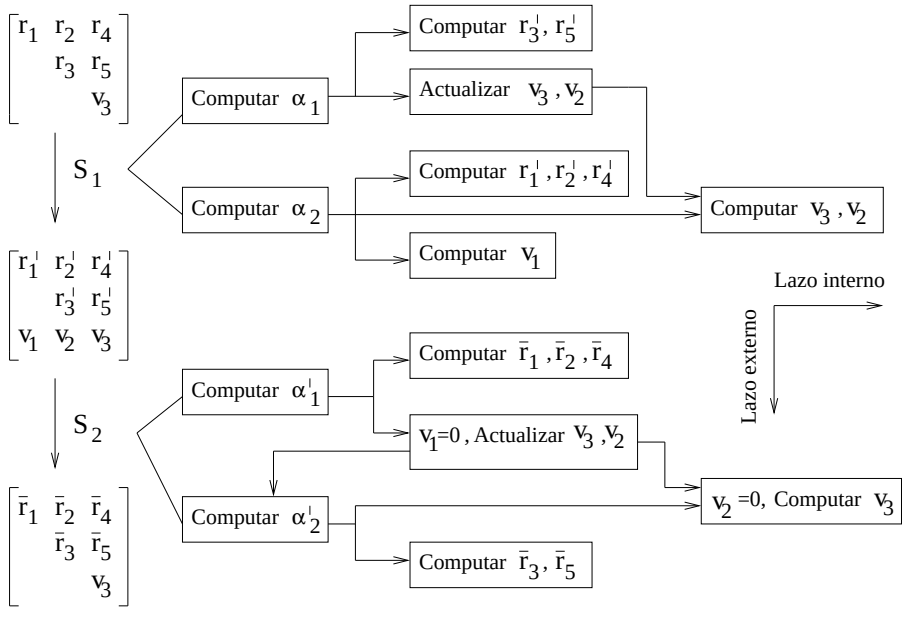


Figura 3.4: Grafos de dependencias asociados a la rotaciones globales planas.

del lazo externo.

El único paralelismo que podemos extraer está asociado a las operaciones de lazo interno, que son independientes entre sí. Pero, en cada iteración del lazo externo es necesario conocer las modificaciones realizadas en la última fila de la matriz durante la iteración anterior, lo que implica que el procesador que actualice los elementos de la última fila tendrá que transmitir esta información a los procesadores que la necesiten para computar los siguientes ángulos de rotación. El proceso es esencialmente secuencial, y su aplicación implica un gran número de mensajes.

De lo dicho anteriormente se deduce que el flujo de datos de la rotación global hacia adelante sigue una estructura triangular, de ahí que se puedan extraer de manera inmediata similitudes entre este algoritmo y el de la resolución triangular inferior.

El grafo de dependencias de la resolución triangular inferior (figura 3.5) es más relajado que el grafo asociado a la rotación global hacia adelante. Las dependencias en sentido horizontal son las mismas, pero en sentido vertical son más débiles. Mientras en la rotación hacia adelante cada modificación de un elemento de la última fila depende de las modificaciones anteriores, el cálculo de los productos $r_{ij}^T q_j$ son independientes entre sí, y sólo van a influir en el cómputo del elemento del vector solución inmediatamente posterior.

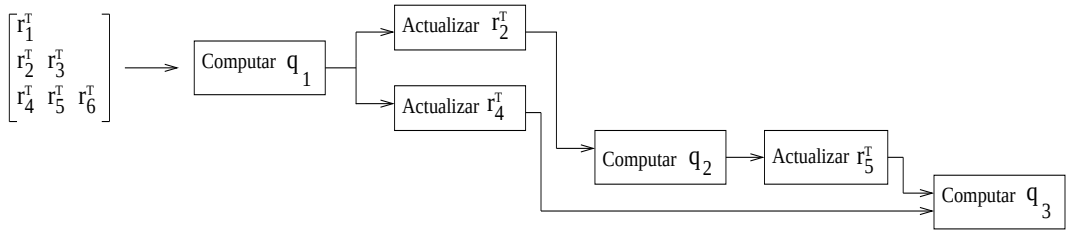


Figura 3.5: Grafo de dependencias asociado a la resolución triangular inferior.

En el siguiente capítulo describiremos nuestra propuesta paralela que se basa en un particionamiento de la matriz triangular R en ciertos bloques, a cada uno de los cuales se le asigna un patrón determinado de computaciones. Esta propuesta minimiza el número de comunicaciones necesarias. La asignación del trabajo dentro de cada bloque así como las comunicaciones necesarias entre bloques, se planifican tratando de evitar las dependencias de datos descritas anteriormente.

3.5. Cálculo del tamaño de paso

El objetivo de esta fase del método de optimización es determinar la longitud de paso λ_k a lo largo de la dirección de búsqueda p_k desde el punto solución actual x_k. Se distinguen dos etapas:

1. Obtención de λ_k mediante la resolución del siguiente problema unidimensional de optimización:

$$f(x_k + \lambda_k p_k) = min \quad f(x_k + \theta p_k), \quad con \quad 0 < \theta \leq \lambda_{max}. \tag{3.18}$$

2. Sustitución de x_k por el nuevo punto solución $x_{k+1} = x_k + \lambda_k p_k$, lo que implica el cálculo del valor de la función y del gradiente en el nuevo punto solución.

Existen distintos métodos para resolver el problema de optimización del paso 1, como son el método de la bisección, el método de Newton, el método de Fibonacci o los métodos de interpolación polinomial, entre otros [34]. Estos últimos son los más eficientes cuando la función a minimizar es continua. Los métodos polinomiales aproximan la función a un polinomio, usando información sobre el valor de la función o del gradiente en algunos puntos. Si además, consideramos interpolación polinomial, lo que estamos afirmando es que el mínimo del polinomio aproximación

de la función está dentro del área que delimitan los puntos usados para crear dicho polinomio. MINOS utiliza interpolación cúbica o cuadrática, según se disponga o no del gradiente de la función en forma analítica. La interpolación cúbica es más precisa que la cuadrática, sin embargo cuando no se dispone del gradiente de la función también será mucho más costosa.

Cuando no existe un procedimiento para calcular el gradiente de forma analítica, se puede usar una aproximación del gradiente mediante diferencias finitas:

$$\bigtriangledown f(x_k)_i \cong (g_k)_i = \frac{f(x_k + he_i) - f(x_k)}{h}, \tag{3.19}$$

donde h es una cantidad infinitesimal. Si m representa el número de variables no lineales, la aproximación para calcular el gradiente requiere m evaluaciones de f además de $f(x_k)$.

El coste computacional del algoritmo de búsqueda de paso se debe fundamentalmente a las evaluaciones de la función objetivo. Si tenemos que aplicar diferencias finitas para computar el gradiente necesitaremos, como se ha dicho anteriormente, $m + 1$ evaluaciones de la función f, lo que hará que el algoritmo sea computacionalmente muy costoso. Sin embargo, no existen dependencias entre las distintas evaluaciones de f, por lo tanto, se pueden realizar completamente en paralelo. El rendimiento del algoritmo paralelo será óptimo, ya que sólo es necesaria una comunicación entre los distintos procesadores del sistema una vez hayan sido evaluados el gradiente y la función objetivo en el nuevo punto solución. Si el primer tamaño de paso elegido verifica las condiciones de Wolfe habrá una única comunicación por iteración. Si por el contrario, no se verifican las condiciones de Wolfe será necesario repetir todo el proceso para buscar un nuevo tamaño de paso. Normalmente, esta búsqueda no suele repetirse más de una vez, y de hacerlo, el proceso se repite pocas veces hasta obtener el tamaño de paso adecuado, lo que implica la necesidad de muy pocas comunicaciones, incluso en el peor de los casos. Por lo tanto, se puede afirmar que el rendimiento de este algoritmo paralelo estará limitado prácticamente por el número de variables no lineales.

Por lo tanto, si utilizamos un algoritmo paralelo para realizar todas las evaluaciones de la función necesarias para computar el gradiente, podremos aplicar un método de interpolación cúbica para determinar el tamaño de paso sin que el coste sea excesivo. Además, una vez que se haya determinado el valor de λ_k, este algoritmo paralelo también se utilizará para computar el valor de la función y del gradiente en el nuevo punto solución.

El algoritmo de paralelización anteriormente descrito es analizado en el capítulo 4. El posible paralelismo que se puede extraer dentro del propio método de interpo-

lación cúbica o cuadrática para acelerar el proceso de búsqueda de tamaño de paso es objeto de estudio en el capítulo 6.

Capítulo 4

Paralelización sobre sistemas de memoria distribuida

En este capítulo se propone una implementación paralela de un código de optimización en el que la dirección de búsqueda se computa mediante un algoritmo cuasi-Newton. En la aplicación de este método existen dos partes principales en cuanto a coste computacional, ambas susceptibles de paralelización. Por un lado, los cálculos algebraicos asociados a los distintos algoritmos que conforman el método; y por otro, las evaluaciones de la función objetivo y su gradiente para cada posible punto solución. Las implementaciones paralelas de métodos cuasi-Newton que se han propuesto en los últimos años se han centrado en alguno de estos aspectos.

Para problemas en los que la evaluación de la función objetivo es computacionalmente costosa, Schnabel [83] introduce una técnica denominada evaluación especulativa del gradiente. La idea consiste en que, mientras parte de los procesadores del sistema computan las correspondientes componentes de la función objetivo en un posible punto solución, el resto calculan el gradiente de la función en ese punto mediante la aproximación de diferencias finitas. De modo que, si finalmente el punto evaluado no es elegido como solución para la siguiente iteración, no se ha perdido tiempo, pues en otro caso, esos procesadores estarían inactivos. Sin embargo, si el punto es seleccionado se ha conseguido adelantar trabajo. Basándose en esta idea, Byrd, Schnabel y Shultz [11, 12] proponen un algoritmo paralelo de un método de Newton [62] que realiza además evaluaciones especulativas de la Hesiana. Describen distintas formas de utilizar la información parcial de la Hesiana, y analizan las propiedades computacionales y de convergencia de cada propuesta.

Si nos centramos en las operaciones algebraicas de un método cuasi-Newton, la más costosa es la actualización de la aproximación de la Hesiana. Trabajos propues-

tos en este campo son el de Kontoghiorghes [50], donde se proponen estrategias de Givens para resolver la actualización de rango k de una factorización QR para un modelo computacional EREW PRAM (*Exclusive-read Exclusive-write Parallel Random Access Machine*); y el de Pinilla [78] donde el algoritmo paralelo propuesto ha sido validado sobre el multiprocesador de memoria compartida PowerChallenge de Silicon y sobre un *cluster* de PCs. En el primer trabajo, se analiza el rendimiento del algoritmo propuesto en unidades de tiempo. Si el número de procesadores disponible es ilimitado, el código paralelo requiere 3/8 unidades de tiempo menos que el algoritmo secuencial; sin embargo, si el número de procesadores está limitado, el algoritmo secuencial es más eficiente en el caso de que todos los procesadores se usen para ejecutar simultáneamente las rotaciones de Givens. En [78], los datos se distribuyen bidimensionalmente de forma cíclica, de modo que cuando haya que aplicar una rotación utilizando dos filas, y éstas hayan sido asignadas a procesadores diferentes, será necesaria una comunicación entre ellos para intercambiar dichas filas. Por lo tanto, el número de mensajes necesario para computar todas las rotaciones será elevado.

Todas las implementaciones mencionadas hasta el momento se centran en la paralelización de cada iteración del método cuasi-Newton. Su objetivo es disminuir el coste computacional del algoritmo en cada iteración. Otros autores, como Van Laarhoven [87] y Phua [79, 80] abordan el problema de forma diferente, en vez de disminuir el coste por iteración, su objetivo es reducir el número de iteraciones. Este tipo de algoritmos será estudiado con más detenimiento en el capítulo 6.

En este capítulo se introducen propuestas eficientes de implementación paralela de las distintas etapas de un método cuasi-Newton. Nos centraremos en la paralelización de las computaciones algebraicas y de las evaluaciones tanto de la función objetivo como del gradiente en cada iteración. Compararemos los algoritmos paralelos propuestos con otros algoritmos existentes. Presentaremos resultados experimentales de las implementaciones propuestas sobre el multiprocesador de memoria distribuida AP3000 de Fujitsu y sobre un *cluster beowulf* de Pentium III.

4.1. Algoritmos paralelos para la resolución triangular

Las eficiencias de los distintos algoritmos paralelos que han sido propuestos en los últimos años para la resolución de sistemas triangulares poseen una cota superior determinada por las características de latencia y ancho de banda del sistema

multiprocesador utilizado y por las dependencias del propio algoritmo. La resolución triangular es un proceso con fuertes dependencias de datos y cuya paralelización requiere un esquema de grano fino, lo que va a implicar un gran número de mensajes de pequeño tamaño entre los distintos procesadores. Además, las pocas computaciones necesarias para resolver un sistema triangular difícilmente compensan el gasto debido a las comunicaciones. Ejemplos clásicos de este tipo de algoritmos son: los algoritmos de *fan-out*, *fan-in* y de frente de ondas [44]. A continuación, comentaremos brevemente algunas de sus características, haciendo hincapié en las dependencias de datos y el número de comunicaciones. Este estudio se introduce con la intención de resaltar las ventajas de nuestra propuesta paralela frente a estos bien conocidos códigos, y así poder utilizarlos como referencia para las validaciones de eficiencia. Nos centraremos en la descripción de la resolución triangular inferior, ya que la resolución triangular superior es análoga.

4.1.1. Algoritmos *fan-out* y *fan-in*

Estos algoritmos constituyen la implementación paralela más inmediata de la resolución triangular. Deben su nombre al tipo de comunicación utilizado en cada uno de ellos, la información es difundida desde un procesador a todos los demás (*fan-out*) o recopilada en un procesador procedente de los demás (*fan-in*). A continuación, mostramos sus respectivos pseudocódigos (algoritmos 2 y 3), donde se emplea una nomenclatura según la cual el procesador $P(j)$ tiene asignada la columna o fila j, según corresponda (identificadas con los conjuntos mis_filas y $mis\_columnas$, respectivamente). Asumimos que, previamente, en el algoritmo *fan-out* se ha realizado una distribución cíclica de la matriz por filas, y en el *fan-in* por columnas.

Algoritmo 2 Algoritmo *fan-out*

```
for j = 0 to n − 1 do
  if j ∈ mis_filas then
    x_j = b_j / l_{j,j}
  end if
  broadcast(x_j, P(j))
  for i ∈ mis_filas, i > j do
    b_i = b_i − x_j l_{i,j}
  end for
end for
```

La principal desventaja de estos algoritmos son los tiempos de espera y el gran número de comunicaciones que se requieren entre procesadores. Hemos realizado las comparaciones con la versión más simple de los dos algoritmos, la cual tiene asociados mayores tiempos de espera. Así por ejemplo, en el algoritmo *fan-out* [51], la iteración del lazo externo $j+1$ no comienza hasta que se han realizado las correspondientes operaciones sobre los distintos elementos de b en la iteración anterior j. Cuando estas operaciones concluyen, todos los procesadores chequean cuál es el que tiene que calcular el elemento x_{j+1}, operación que se podría haber realizado una vez que el procesador $P(j+1)$ ha actualizado b_{j+1}. Una situación similar sucede en el algoritmo *fan-in*, el procesador $P(i)$ será el último en computar su contribución al componente de la solución x_{i+1}. Para evitar que este procesador sea el cuello de botella en la iteración $i+1$, Cleary [16] propone que $P(i)$ adelante su aportación a x_{i+1} computando todos los productos $x_j l_{i,j}$, con $j < i$ y $j \in mis\_columnas$, aprovechando así los tiempos de espera por las contribuciones a x_i del resto de procesadores.

Algoritmo 3 Algoritmo *fan-in*

```
for i = 0 to n − 1 do
  t = 0
  for j ∈ mis_columnas, j < i do
    t = t + x_j l_{i,j}
  end for
  s = reduce (suma t, P(i))
  if i ∈ mis_columnas then
    x_i =(b_i − s)/l_{i,i}
  end if
end for
```

4.1.2. Algoritmo de frente de ondas

Otra aproximación clásica a la paralelización de la resolución triangular la constituyen los algoritmos de frente de ondas. La fuente de paralelismo de este tipo de algoritmos es el solapamiento de las computaciones [23, 54]. Como se ha comentado, el algoritmo es inherentemente secuencial ya que después de que $P(i)$ computa x_i, las actualizaciones del vector independiente b sólo las podrá realizar este procesador.

Algoritmo 4 Algoritmo frente de ondas

```
for j ∈ mis_columnas do
  for k = 1 to #segmentos do
    recibir segmento
    if k = 1 then
      x_j = (b_j − z_j)/l_{j,j}
      segmento = segmento − {z_j}
    end if
    for z_i ∈ segmento, i > j do
      z_i = z_i + x_j l_{i,j}
    end for
    if |segmento| > 0 then
      enviar segmento al procesador P(j + 1)
    end if
  end for
end for
```

La idea del algoritmo de frente de ondas consiste en dividir el vector de actualizaciones z en varios segmentos de tamaño fijo κ y solapar las computaciones de estos segmentos sobre los P procesadores del sistema.

En el algoritmo 4 se muestra el pseudocódigo de esta estrategia de paralelización de forma abreviada, omitiendo la fase de inicialización. El procedimiento es el siguiente: el procesador $P(0)$ computa el primer elemento de la solución, x_0. A continuación, comienza a calcular los productos $z_i = x_0 l_{i,0}$ del vector de actualizaciones z. Después de computar los primeros κ elementos de este vector, $P(0)$ envía un mensaje al procesador $P(1)$, de modo que éste puede computar x_1 y continuar con las actualizaciones de z. Mientras $P(1)$ realiza estas computaciones, el procesador $P(0)$ prosigue con las siguientes κ actualizaciones de z que tiene asignadas. El proceso continúa iterativamente de modo que tan pronto como un procesador completa su actualización sobre un bloque de κ elementos del vector z, envía ese segmento al siguiente procesador. El procedimiento finaliza cuando el último elemento de la solución ha sido computado.

El tamaño del segmento, κ, es un parámetro ajustable que controla la granularidad del algoritmo. Cuanto más pequeño es κ, mayor será el número de computaciones concurrentes pero también implicará un mayor número de comunicaciones. Por lo tanto, el parámetro κ debe garantizar un buen equilibrio entre computación paralela y coste de las comunicaciones. En la figura 4.1 se muestran los tiempos de ejecución

del algoritmo de frente de ondas utilizando distintos tamaños de segmento para una matriz de orden $n = 1998$ en el multiprocesador AP3000. Podemos observar que para tamaños de segmento pequeños, el algoritmo presenta una mayor escalabilidad, pero los tiempos de computación son elevados. A medida que el tamaño de κ aumenta, los tiempos de computación disminuyen pero se degrada la escalabilidad.

Una importante desventaja de este algoritmo es que mientras no se hayan computado los P primeros elementos del vector solución, no estarán ocupados todos los procesadores. Además, como el tamaño y el número de segmentos en que se ha dividido el vector z disminuye a medida que avanza el procedimiento, habrá procesadores que terminen sus computaciones antes que otros. Si el coste de las comunicaciones es muy elevado, apenas se producirá solapamiento entre computaciones, y el algoritmo será altamente ineficiente.

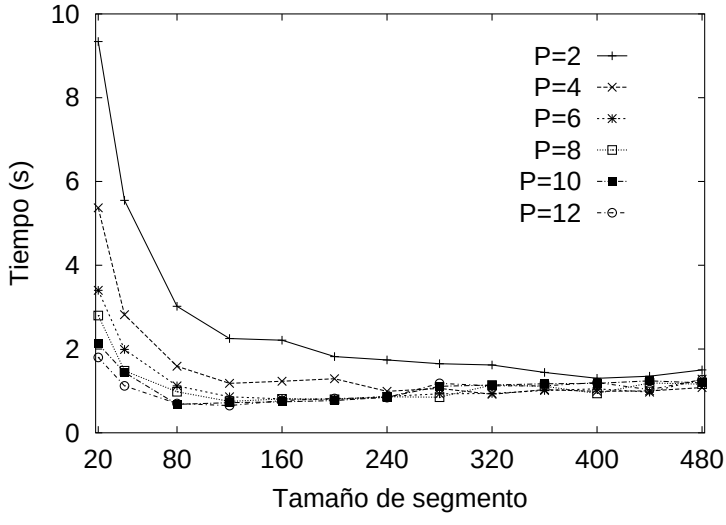


Figura 4.1: Tiempos de ejecución del algoritmo de frente de ondas para la resolución de una matriz triangular de orden $n = 1998$ utilizando distintos tamaños de segmento.

4.1.3. Algoritmo paralelo por bloques (PB)

Esta propuesta divide la matriz triangular en bloques triangulares $L_{i,i}$ de tamaño $h_i \times h_i$, y bloques cuadrados $L_{i,j}$ $(i > j)$ de dimensiones $h_i \times h_j$ [85]. Asimismo, el vector independiente y el vector solución se dividen en bloques de tamaño h_i. De modo que el sistema triangular a resolver quedaría del siguiente modo:

$$\begin{pmatrix} L_{0,0} & & & \\ L_{1,0} & L_{1,1} & & \\ \vdots & \vdots & \ddots & \\ L_{t,0} & L_{t,1} & \cdots & L_{t,t} \end{pmatrix} \begin{pmatrix} X_0 \\ X_1 \\ \vdots \\ X_t \end{pmatrix} = \begin{pmatrix} B_0 \\ B_1 \\ \vdots \\ B_t \end{pmatrix} \tag{4.1}$$

La partición siempre es tal que $n = \sum_{i=0}^{t} h_i$, siendo n el orden de la matriz triangular L. La matriz L está particionada en $t+1$ bloques de columnas, $L_{*,i} = (0 \quad \cdots \quad 0 \quad L_{i,i} \quad \cdots \quad L_{t,i})^T$, de la matriz original. Para distribuir la matriz L entre los P procesadores de nuestro sistema optamos por una distribución cíclica por bloques de columnas. Un ejemplo de como quedaría distribuida la matriz en este algoritmo para 3 procesadores se muestra en la figura 4.2(a).

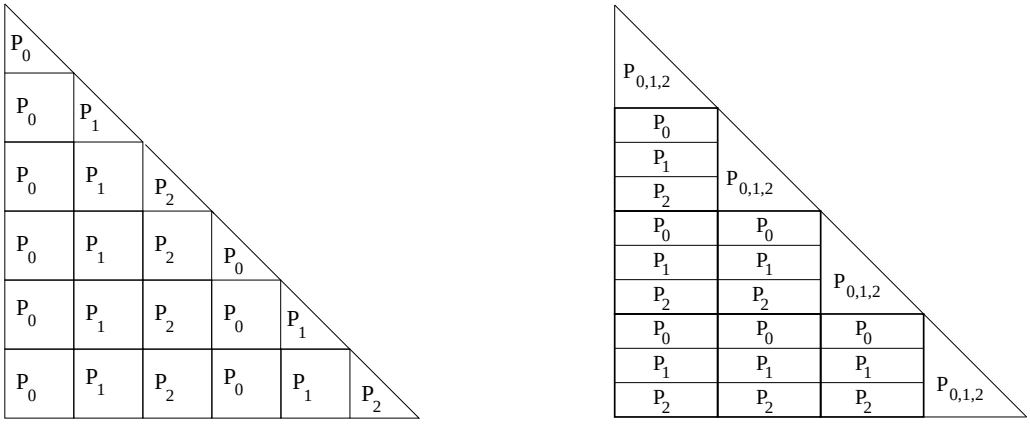


Figura 4.2: Distribución de una matriz triangular en un sistema de 3 procesadores, P_0, P_1 y P_2, para: (a) Algoritmo PB. (b) Algoritmo SPB.

Tanto el bloque de columnas $L_{*,i}$ como la porción B_i del vector b son asignados al procesador P_k, siendo $k = i\,\%P$. El algoritmo 5 muestra el pseudocódigo de esta estrategia de resolución triangular, donde $mis\_bloques$ representa el conjunto de bloques asignados a un determinado procesador. Los productos parciales asociados al bloque $L_{i,j}$ se suman en el vector V. Una vez que todos los productos $L_{i,j}$ $(j < i)$ han sido computados se realiza una operación de reducción que almacena en el vector W del procesador P_k la suma de los productos parciales acumulada. Finalmente, el procesador P_k computa los elementos del vector solución ligados al bloque triangular $L_{i,i}$.

Algoritmo 5 Algoritmo paralelo por bloques

```
for i = 0 to t do
  V = 0
  for j ∈ mis_bloques, j < i do
    V = V + X_j L_{i,j}
  end for
  W = reduce (suma V, P_k), siendo k = i %P
  if i ∈ mis_bloques then
    X_i =(B_i − W)/L_{i,i}
  end if
end for
```

Una importante decisión es la selección del tamaño de bloque h_i. El criterio de selección es puramente empírico, seleccionando aquel valor con el que se obtienen los menores tiempos de ejecución. Para ilustrar el algoritmo, consideramos una matriz de orden $n = 1998$ sobre un sistema de 12 procesadores del multiprocesador AP3000 de Fujitsu, los resultados se muestran en la figura 4.3. Inicialmente, puede observarse que para cualquier valor de P, el tiempo de ejecución decrece rápidamente a medida que se incrementa el tamaño de bloque, hasta llegar a un cierto valor a partir del cual el tiempo permanece prácticamente constante. Una de las características más destacables de este algoritmo es la escalabilidad que se observa a medida que se aumenta el número de procesadores y el tamaño de bloque. Por ejemplo, para un sistema de 12 procesadores y un tamaño de bloque de 52 el algoritmo se ejecuta 9.5 veces más rápido que con 2 procesadores y un tamaño de bloque de 4.

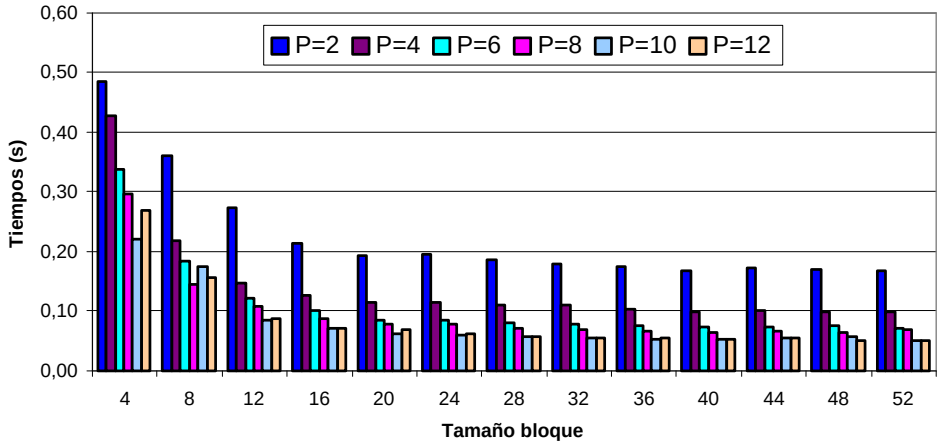


Figura 4.3: Tiempos de ejecución del algoritmo PB para una matriz de orden $n = 1998$ utilizando distintos tamaños de bloque.

4.1.4. Algoritmo secuencial-paralelo por bloques (SPB)

En los sistemas multiprocesadores de memoria distribuida actuales, la relación T_C/T_E es muy pequeña, siendo T_C el tiempo necesario para ejecutar una operación en punto flotante, y T_E el tiempo que transcurre desde que un mensaje de 1 byte es enviado hasta que llega a su destino. Por lo tanto, un programa paralelo con gran número de comunicaciones entre procesadores será posiblemente poco eficiente. Tanto el algoritmo de sustitución hacia adelante, para la resolución triangular inferior, como el de sustitución hacia atrás, para la resolución triangular superior, poseen fuertes dependencias de datos como se ha visto en la sección 3.4. Los algoritmos paralelos de *fan-out* y *fan-in* tratan de explotar todo el paralelismo existente en cada iteración del lazo interno del algoritmo secuencial. Sin embargo, debido a las dependencias existentes entre las iteraciones del lazo externo, será necesario el envío de un mensaje para computar cada elemento del vector solución. En el algoritmo de frente de ondas, estas dependencias tampoco se evitan, pero se trata de minimizar su efecto aprovechando el solapamiento de las computaciones. Por lo tanto, para obtener buenas eficiencias es necesario reducir el número de comunicaciones y los tiempos de espera.

La estrategia paralela que proponemos divide el flujo de datos del algoritmo en bloques triangulares $bloqtg_i(L_{i,i})$ de ancho $s = \alpha P$ ($\alpha \in IN$), y bloques rectangulares $bloqrt_{i,j}(L_{i,j})$, de modo similar al algoritmo PB, como se muestra en la figura 4.2(b).

Algoritmo 6 Algoritmo secuencial-paralelo por bloques

for $i = 0$ to $n_\alpha - 1$ **do**
  Computación secuencial redundante
    for $j \in filas\_bloqtg_i$ **do**
      $x_j = (b_j - w_j)/l_{j,j}$
      for $k = is$ to j **do**
        $b_{j+1} = b_{j+1} - x_k l_{j+1,k}$
      end for
    end for
  Computación paralela
    for $j \in filas\_bloqrt_i$ **do**
      if $j \in mis\_filas$ **then**
        for $k = 0$ to $(i+1)s - 1$ **do**
          $z_j = z_j + x_k l_{j,k}$
        end for
      end if
    end for
  $[w((i+1)s), w((i+2)s-1)]$ = allreduce (suma $[z((i+1)s, z((i+2)s-1)])$
end for

Dentro de los bloques triangulares se computan elementos del vector solución x y productos $x_j l_{i,j}$. Todas estas operaciones serán ejecutadas por todos los procesadores del sistema de forma secuencial y redundante. Una vez realizadas las computaciones asociadas a un bloque triangular se pueden calcular los productos de los elementos de la matriz que están por debajo de ese bloque. Estos elementos están distribuidos de forma cíclica por filas entre los P procesadores del sistema. Más formalmente, tras la computación del bloque triangular i, en la que se han obtenido los elementos del vector solución $[x_{is}, x_{(i+1)s-1}]$, los P procesadores calculan de forma concurrente los productos $x_k l_{j,k}$ (con $k \in [0, (i+1)s-1]$, y $(i+1)s < j < (i+2)s-1)$ correspondientes a las filas que tienen asignadas dentro de los bloques rectangulares que están situados inmediatamente debajo del triángulo i. Antes de ejecutar el siguiente bloque triangular, el $i+1$, es necesario conocer los elementos $[(i+1)s, (i+2)s-1]$

del vector de actualizaciones z para obtener los elementos del vector solución correspondientes a ese bloque. Como los cálculos asociados al bloque triangular van a ser realizados por todos los procesadores, será necesaria una comunicación colectiva para que los P procesadores tengan el valor actualizado de z antes de empezar a computar el siguiente bloque triangular. El proceso continúa hasta que se recorren todos los bloques triangulares.

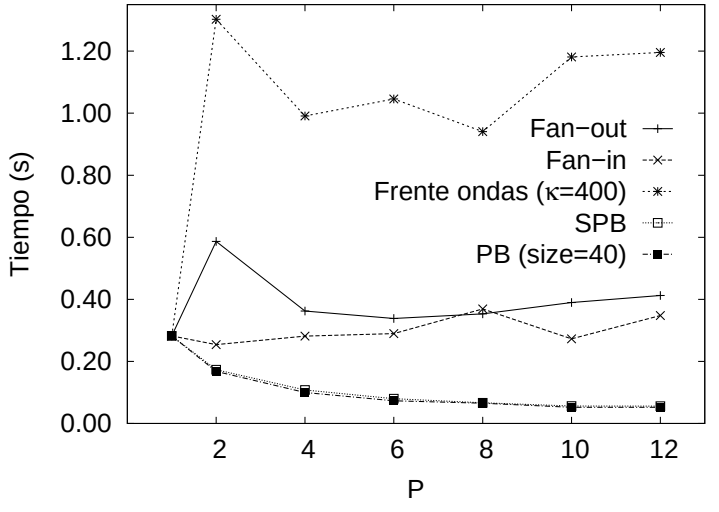


Figura 4.4: Comparación del tiempo de ejecución de los algoritmos de resolución triangular.

La primera implicación de nuestra propuesta es un fuerte decremento en el número de comunicaciones. Al considerar bloques triangulares de ancho αP, se tienen $n/\alpha P$ bloques triangulares de ancho $s = \alpha P$, y un bloque adicional de ancho $s_u = n\,\%(\alpha P)$. En este algoritmo sólo es necesario realizar comunicaciones antes de comenzar los cálculos de cada bloque triangular para recoger los elementos de z que han sido modificados previamente en cada procesador de forma paralela. De este modo, el número de comunicaciones se reduce a $n_\alpha - 1$, siendo $n_\alpha = \lceil n/\alpha P \rceil$ el número de bloques triangulares en que ha sido dividida la estructura original. El algoritmo 6 muestra el pseudocódigo de esta estrategia paralela.

4.2. Estudio comparativo

Para validar la eficiencia del algoritmo paralelo propuesto para la resolución triangular, en esta sección se realiza una comparativa de todos los algoritmos descritos en la sección anterior. En la figura 4.4 se muestran los tiempos de ejecución en el multiprocesador de memoria distribuida AP3000 de Fujitsu para una matriz triangular de tamaño $n = 1998$. Para ilustrar la eficiencia del algoritmo propuesto, hemos utilizado esta matriz porque su tamaño es el adecuado para que la correcta explotación del paralelismo no se vea afectada por problemas de falta de memoria en el AP3000. El coste del algoritmo secuencial de sustitución hacia adelante basado en el producto escalar se ha utilizado como tiempo de ejecución para $P = 1$ en todos los casos.

Algoritmo	Número de mensajes	Tipo de comunicación
Fan-out	$n-1$	*broadcast*
Fan-in	$n-1$	*reduce*
Frente de ondas	$\kappa\frac{\lceil\frac{n-1}{\kappa}\rceil(\lceil\frac{n-1}{\kappa}\rceil+1)}{2}-(\kappa-\kappa_u)$	*send-receive*
PB	$\lceil\frac{n}{size}\rceil-1$	*reduce*
SPB	$\lceil\frac{n}{\alpha P}\rceil-1$	*allreduce*

Tabla 4.1: Número de comunicaciones necesarias en los distintos algoritmos de resolución triangular.

En general, el algoritmo SPB que hemos propuesto consigue menores tiempos de ejecución y una mejor escalabilidad que los algoritmos clásicos de *fan-in*, *fan-out* y frente de ondas. La explicación de esta disminución de los tiempos de ejecución se encuentra en la reducción del número de mensajes totales. En la tabla 4.1 presentamos el número de comunicaciones requeridas por cada algoritmo para una matriz de orden n y un sistema de P procesadores. κ es el tamaño de los segmentos en que se divide cada columna del algoritmo de frente de ondas, κ_u es el tamaño del último segmento y $size$ es el tamaño de bloque del algoritmo PB. Además de una reducción en el número de comunicaciones, en esta tabla se observa que únicamente para el algoritmo SPB este número es inversamente proporcional al número de procesadores del sistema, lo que da lugar a una mejor escalabilidad. Esta mejora se observa a pesar del mayor coste que implica una comunicación tipo *allreduce* frente a una

comunicación *broadcast* o *reduce* porque el incremento en el coste se compensa con un número total de mensajes inferior. En la tabla 4.2 se compara el número total de comunicaciones *allreduce* del algoritmo SPB para $\alpha = 4$ con el máximo número de este tipo de mensajes que se tendrían que realizar para igualar el coste de las comunicaciones de este algoritmo con el coste asociado al algoritmo *fan-in* y *fan-out*. Estos datos resaltan lo beneficioso que es el algoritmo SPB en cuanto a ahorro en el coste de las comunicaciones.

P	SPB	*Fan-out*	*Fan-in*
2	249	813	941
4	124	271	398
6	83	284	288
8	62	242	345
10	49	256	222
12	41	236	249

Tabla 4.2: Número de *allreduces* del algoritmo SPB frente al máximo número de *allreduces* necesario para igualar el coste de las comunicaciones de los algoritmos *fan-in* y *fan-out*.

Los tiempos de ejecución del algoritmo PB con $size = 40$ (tamaño de bloque que presenta el mejor comportamiento) y del algoritmo SPB son muy similares, aunque ligeramente mejores para el primero (ver figura 4.5). La principal diferencia entre ambos métodos está en la forma de distribuir los elementos de la matriz entre los distintos procesadores (figura 4.2). Mientras en el algoritmo PB los bloques triangulares son asignados a un único procesador, en el SPB los elementos de estos bloques están replicados en todos los procesadores. Del mismo modo, los bloques rectangulares en el algoritmo SPB tienen su carga repartida por igual entre todos los procesadores, y en el PB cada bloque se asigna a un sólo procesador. Dado que el paralelismo en el algoritmo SPB se encuentra a nivel de bloque, el valor de la carga paralela depende del ancho del mismo y del número de bloques existentes. Cada procesador tiene asignados en cada bloque $\alpha^2 P$ elementos de la matriz. A medida que avanza la ejecución del método, y tras i bloques triangulares computados, el sistema ejecutará en paralelo los i bloques rectangulares, de modo que cada procesador trabaja sobre $i\alpha^2 P$ elementos de la matriz.

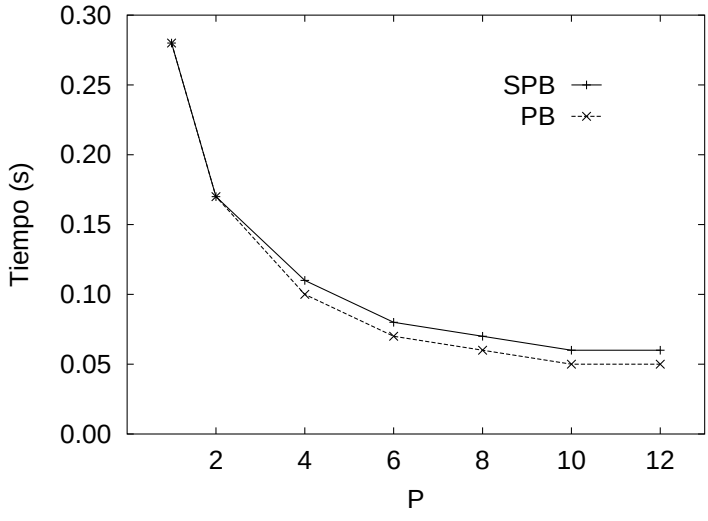


Figura 4.5: Comparación del tiempo de ejecución de los algoritmos PB y SPB.

Por otro lado, en el algoritmo PB el paralelismo se introduce mediante un cierto grado de segmentación. Cada bloque rectangular $L_{*,i}$ está asignado a un procesador, de modo que cuantos más bloques triangulares hayan sido computados, más bloques cuadrados será posible computar en la siguiente etapa. Los procesadores computan los bloques que tienen asignados, y cuando terminan con todos los bloques asociados a una etapa realizan una operación de reducción. Al comienzo del algoritmo habrá procesadores ociosos hasta que hayan sido computados $P-1$ triángulos. Entonces, hasta ese momento resulta más eficiente la propuesta SPB, en la que no hay ningún procesador desocupado. En la etapa de computación del triángulo *(P-1)*-ésimo, y para triángulos posteriores, en el algoritmo PB ya no habría ningún procesador ocioso, resultando ambos algoritmos de similar eficiencia.

Se puede determinar que la reducción en el número de comunicaciones es la principal causa de la mejor eficiencia del algoritmo SPB frente a los algoritmos de *fan-in*, *fan-out* y frente de ondas. Para precisar este hecho, se ha realizado una estimación analítica del rendimiento de estos algoritmos paralelos sobre el AP3000 de Fujitsu. El estudio se ha dividido en dos partes: el análisis del coste de las computaciones (operaciones en punto flotante) y el análisis del coste de las comunicaciones (operaciones de comunicación colectiva). La estimación del coste de aplicación de estos métodos se ha realizado sobre la matriz ejemplo utilizada para obtener las medidas empíricas de la figura 4.4.

La estimación del coste de las computaciones consiste en determinar el tiempo

de una operación en punto flotante en el AP3000, así como el número de operaciones en punto flotante que realiza el procesador que tiene más carga asignada.

Operación	Tamaño mensaje (bytes)	μ (μs)	τ (μs/byte)	γ (μs/byte)
broadcast	<96	[17,33]	[0.060,0.140]	0
	96-32K	[35,67]	[0.011,0.027]	0
	>32K	[0,500]	[0.010,0.014]	0
reduce	<96	[17,37]	[0.043,0.266]	0.19
	96-32K	[34,75]	[0.017,0.036]	0.19
	>32K	[-6,112]	[0.017,0.031]	0.029
allreduce	<96	[22,36]	0.034	0.19
	96-32K	[54,86]	0.0052	0.19
	>32K	[-75,225]	0.0094	0.029

Tabla 4.3: Valores de los parámetros μ, τ y γ para las operaciones de *broadcast*, *reduce* y *allreduce* en el AP3000.

La estimación del coste de las comunicaciones es algo más compleja. En primer lugar, se tiene que conocer cómo están implementadas las funciones de comunicación colectiva en el AP3000. Un estudio previo [6] ha demostrado que estas funciones en la arquitectura de esta máquina siguen una aproximación típica de algoritmos de potencia de dos (árboles binarios e hipercubos). Mitra et al [61] realizan la caracterización de las operaciones colectivas del siguiente modo:

- **Difusión** (*broadcast*). La implementación típica se basa en un modelo de árbol (MST: *Minimum Spanning Tree*). Para P procesadores el coste de esta operación se puede expresar como:

$$T_{comu} = \lceil log_2 P \rceil (\mu + \tau l), \tag{4.2}$$

  donde l es el tamaño del mensaje en bytes, y μ y τ son parámetros que caracterizan este tipo de operación colectiva en el AP3000.

- **Reducción** (*Combine-to-One*). Utiliza una implementación equivalente a la operación de difusión, pero en orden inverso, realizando en cada paso la operación de combinación que en nuestro caso es un sumatorio. El coste viene dado por:

$$T_{comu} = \lceil log_2 P \rceil (\mu + \tau l + \gamma l), \tag{4.3}$$

donde μ y τ son parámetros que caracterizan las comunicaciones, y γ caracteriza el coste de computación de la operación de combinación.

- **Allreduce**. El modelo de coste de esta operación se basa en una implementación de una operación de reducción, que en nuestro caso es un sumatorio, seguida de una operación de difusión. Es decir:

$$T_{comu} = 2\lceil log_2 P \rceil \mu + (\frac{P-1}{P} + \lceil log_2 P \rceil)\tau l + \lceil log_2 P \rceil \gamma l \qquad (4.4)$$

Estas expresiones han sido estimadas suponiendo ausencia de conflictos en la red. Esta hipótesis es realista en nuestro caso, ya que el sistema dispone de una red dedicada de alto rendimiento. Los parámetros μ, τ y γ utilizados han sido determinados para cada operación colectiva en el AP3000 [6]. El valor del parámetro γ es constante para todos los procesadores, sin embargo para los parámetros μ y τ se obtienen distintos valores para diferentes valores de P. En la tabla 4.3 se muestran los valores empíricos obtenidos en el AP3000 para estos parámetros distinguiendo tres tamaños de mensaje. En el caso de que los parámetros no sean constantes, tomarán distintos valores en función de P, pero siempre dentro de los intervalos indicados en la tabla.

Los resultados de la estimación teórica del coste de los algoritmos se muestran en la figura 4.6. De esta figura se han eliminado los resultados del algoritmo de frente de ondas ya que su implementación en el AP3000 es poco eficiente. En este método las comunicaciones son punto a punto. La estimación de su coste es lineal: $T_{comu} = \mu + \tau l$. Para una matriz dada, el número de comunicaciones sólo depende del tamaño de segmento elegido, y por tanto, será independiente del número de procesadores. Para el ejemplo que estamos estudiando, la estimación del coste total debido a las comunicaciones es $T_{comu} = 0.8$ s, un valor muy elevado en comparación con los resultados obtenidos por los otros algoritmos. A esta predicción se le deben añadir las estimaciones del coste de las computaciones para obtener el coste teórico total del algoritmo. Este coste es similar al de los algoritmos *fan-in* y *fan-out*, ya que este método aplica una distribución cíclica por columnas. Si existiese un cierto grado de solapamiento entre comunicaciones y computaciones, el coste experimental del algoritmo debería ser menor que el estimado, ya que, en este caso, el coste de algunos mensajes sería enmascarado por el coste de algunas operaciones en punto flotante. Sin embargo, si se comparan los resultados estimados con los obtenidos experimentalmente, se observa que estos últimos son ligeramente más costosos (figuras 4.4 y 4.6). Por lo tanto, deducimos que el solapamiento entre comunicaciones y computaciones que busca el algoritmo presenta un efecto mínimo. La causa se

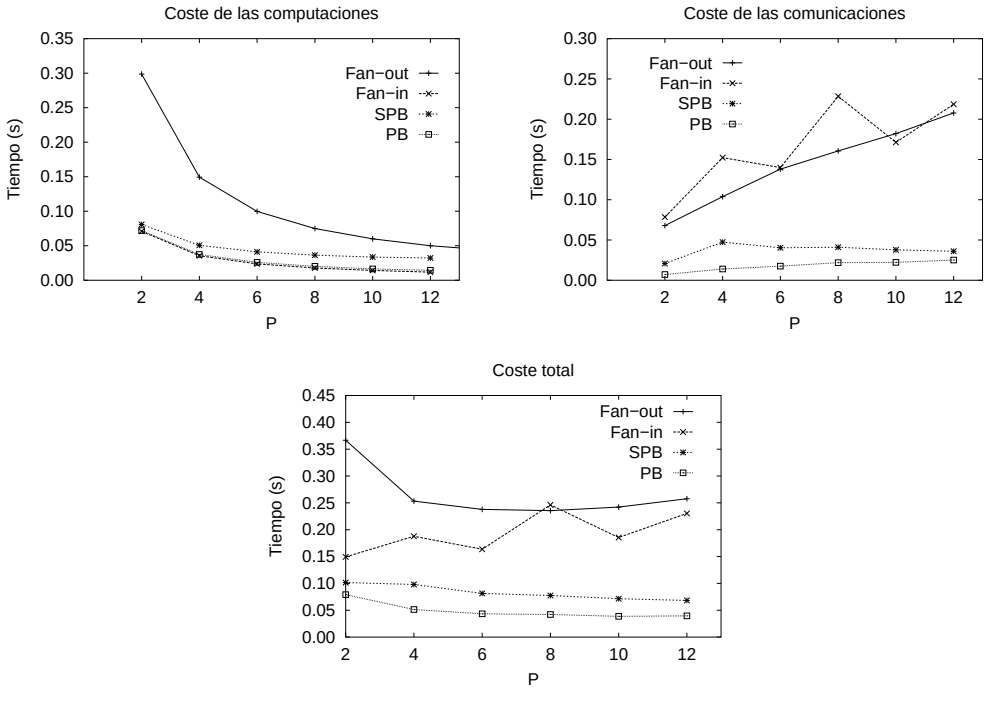


Figura 4.6: Estimación teórica del coste de los algoritmos paralelos para la resolución triangular.

encuentra de nuevo en el mayor coste de una operación de comunicación frente a una operación de punto flotante.

En la figura 4.6 se puede observar que a medida que P aumenta, las operaciones colectivas se hacen más costosas. En ese caso, sólo el tiempo de comunicación del algoritmo SPB disminuye, ya que el número de mensajes en este método es inversamente proporcional al número de procesadores. En esta figura también podemos observar que en el algoritmo PB las comunicaciones son menos costosas que en el SPB, debido por un lado a que en nuestro caso hasta $P = 10$ el número de mensajes generado por el algoritmo PB es menor, y por otro, a que el coste de una operación de reducción es siempre menor que el coste de una reducción a todos (*allreduce*).

El tiempo total estimado y el medido experimentalmente sobre el AP3000 para cada uno de los algoritmos paralelos de resolución triangular se muestra en la figura 4.7. Aunque en algún caso sus valores difieren significativamente, el comportamiento global que predicen nuestras estimaciones coincide con el medido. Este hecho es claramente constatable para el algoritmo *fan-in* en el que los picos existen-

tes en el tiempo de computación son debidos a las variaciones que el parámetro μ (el cual caracteriza una operación de reducción de pequeño tamaño) sufre al variar el número de procesadores.

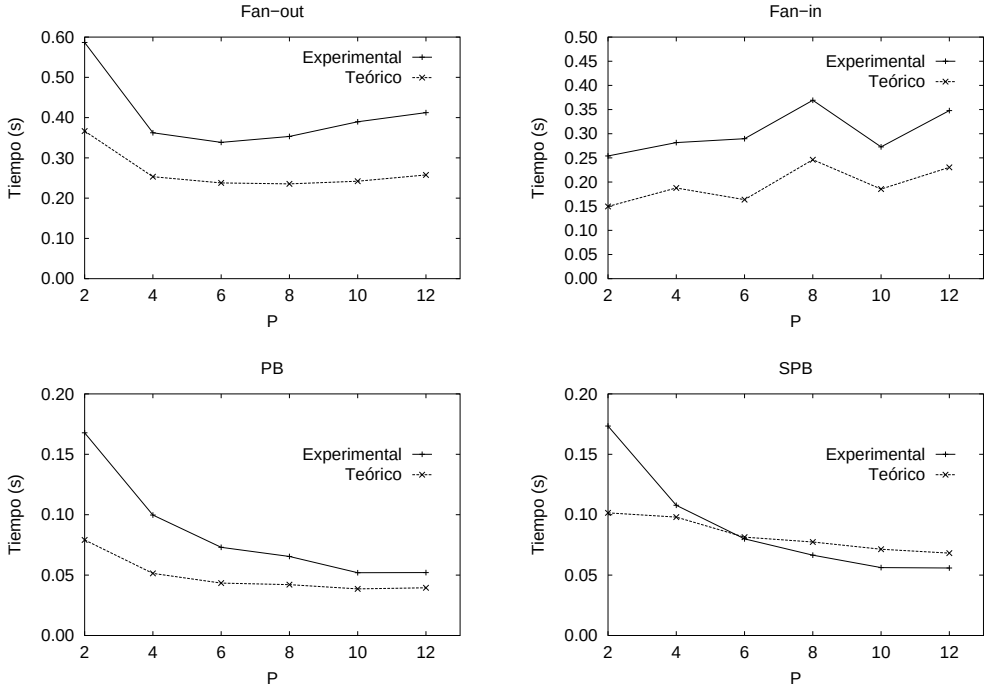


Figura 4.7: Medidas teóricas y experimentales del coste de los algoritmos paralelos para la resolución triangular.

Por último, hemos realizado un estudio de la escalabilidad. Utilizamos un modelo de escalabilidad crítico en tiempo [55]. La escalabilidad del algoritmo se establece en función de métricas como la eficiencia, la aceleración o el tiempo de ejecución en función del número de procesadores [24].

Si definimos la eficiencia relativa como:

$$E_{rel} = \frac{T_{comp}}{PT_{pal}}, \tag{4.5}$$

donde T_{comp} es el tiempo de ejecución del algoritmo en un solo procesador y T_{pal} es el tiempo de ejecución paralelo con P procesadores. Si consideramos que el tiempo de ejecución con P procesadores es la suma del tiempo de computación (T_{compal})

más el tiempo de comunicaciones (T_{comu}) y que $T_{comp} = PT_{compal}$, la ecuación 4.5 puede expresarse como,

$$E_{rel} = \frac{1}{1 + \frac{T_{comu}}{T_{comp}}}. \tag{4.6}$$

El rendimiento de un algoritmo paralelo se incrementa conforme el valor de la eficiencia se aproxime a la unidad, o lo que es lo mismo, cuanto menor sea el cociente T_{comu}/T_{comp}. En la figura 4.8 se muestra un estudio de la escalabilidad del coste de las computaciones y las comunicaciones para los distintos algoritmos paralelos. Se utiliza como métrica el tiempo de ejecución en función del número de procesadores.

En las gráficas se puede observar que el algoritmo SPB es el que presenta peores resultados de escalabilidad en cuanto a coste de computaciones. Esto se debe a la dependencia del tamaño de los bloques triangulares con el número de procesadores. A medida que aumenta éste, aumenta el número de computaciones secuenciales y disminuye la carga paralela del algoritmo, aproximándose cada vez más a un algoritmo secuencial. Sin embargo, este algoritmo es el que presenta mejor escalabilidad en cuanto a coste de comunicaciones, ya que su número disminuye al aumentar P, decreciendo así el tiempo total de comunicación. Para los otros algoritmos, el número de comunicaciones se mantiene constante con P, pero se incrementa su coste. El comportamiento global de los algoritmos, teniendo en cuenta coste de computaciones y comunicaciones, se muestra en la tercera gráfica de la figura 4.8, donde se representa el valor del cociente T_{comu}/T_{comp} en función del número de procesadores. Se observa que a pesar de que el algoritmo PB es el más eficiente para valores de P pequeños, a medida que el número de procesadores aumenta el algoritmo SPB se convierte en el más eficiente. Concluimos, por tanto, que los algoritmos PB y SPB ofrecen un buen rendimiento para la resolución triangular y que ambos muestran una buena escalabilidad, mejorando considerablemente los resultados ofrecidos por los algoritmos clásicos de *fan-in*, *fan-out* y frente de ondas.

4.2.1. Ventajas del algoritmo SPB en el método de optimización

De la comparativa realizada en la sección anterior podemos deducir que para calcular la dirección de búsqueda en cada iteración del método de optimización, los algoritmos PB y SPB se pueden utilizar indistintamente en la resolución de los dos sistemas triangulares implicados. Sin embargo, se debe tener en cuenta que la distribución de datos que lleva asociada cada una de las soluciones sea adecuada para la computación del resto de rutinas del método de optimización, y no sea

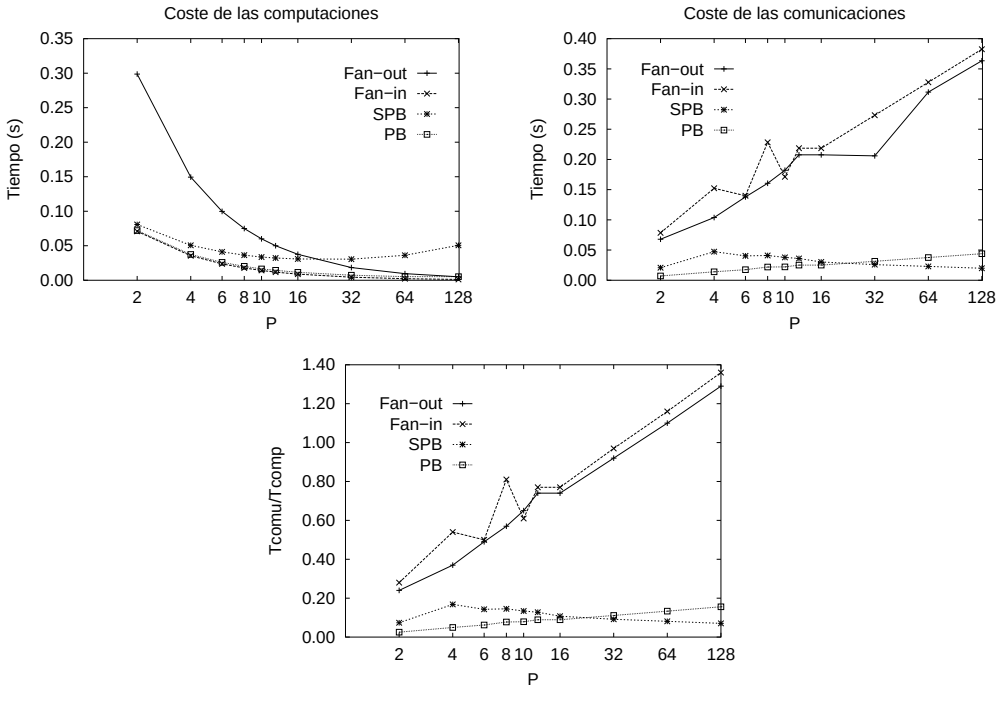
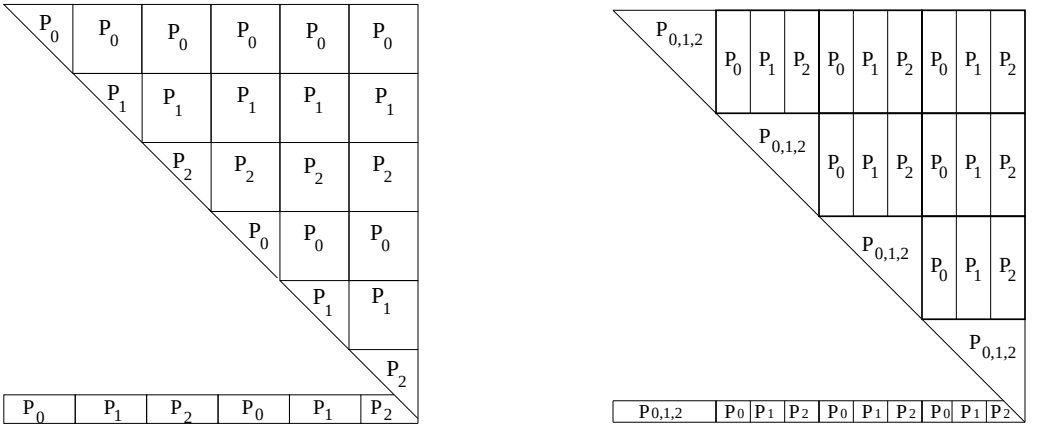


Figura 4.8: Estudio de la escalabilidad de los distintos algoritmos paralelos para la resolución triangular.

necesario realizar comunicaciones intermedias de redistribución. De este modo, el algoritmo que se elija para la resolución triangular tendrá que adaptarse al algoritmo de las rotaciones planas, que es la rutina del código que requiere el mayor coste computacional.

En la sección 3.4 se han descrito las dependencias de datos de la rotación global hacia adelante. El algoritmo SPB ha sido desarrollado para explotar el máximo paralelismo teniendo en cuenta estas dependencias. Ahora, analizamos cómo se comporta el método PB bajo las mismas. Al comienzo de la rotación global hacia adelante, y de acuerdo con lo visto en la sección 4.1.3, la matriz R', que es una matriz triangular superior con la última fila densa, se distribuye entre los P procesadores del sistema como se muestra en la figura 4.9(a).

Si aplicamos una nomenclatura análoga a la utilizada en la sección 4.1.3, una matriz triangular superior U se divide en bloques triangulares superiores $U_{i,i}$ de tamaño $h_i \times h_i$ y bloques cuadrados $U_{i,j}$ $(i < j)$ de tamaño $h_i \times h_j$. El flujo del algoritmo PB aplicado a la rotación global hacia adelante sería como sigue: el procesador P_0



(a) (b)

Figura 4.9: Distribución de la matriz R' antes de la aplicación de la rotación global hacia adelante para un sistema de 3 procesadores: (a) Algoritmo PB. (b) Algoritmo SPB.

a partir de los elementos diagonales del bloque triangular $U_{0,0}$ y de los elementos de la última fila de R', situados debajo de ese triángulo ($U_{t,0}$), calcula los s primeros ángulos de rotación. Estos ángulos son enviados mediante una difusión al resto de procesadores para que puedan calcular las modificaciones que estas rotaciones producen en sus elementos de la última fila. Sin embargo, cada procesador necesita el valor no actualizado de los bloques $U_{0,j}$ para poder calcular las primeras actualizaciones de sus bloques $U_{t,j}$. Por lo tanto, los procesadores P_1, P_2, $\cdots$, P_{P-1}, necesitan los datos de los rectángulos $U_{0,j}$ para poder calcular las actualizaciones sobre sus segmentos de la última fila $U_{t,j}$. Adicionalmente, el procesador P_0 necesita las sucesivas modificaciones de los elementos de $U_{t,j}$ para actualizar los rectángulos $U_{0,j}$. Con la distribución de datos de la figura 4.9(a) sería necesaria una comunicación de dispersión para distribuir los bloques $U_{0,j}$ entre los distintos procesadores con el objetivo de poder realizar las correspondientes actualizaciones de los segmentos $U_{t,j}$. Una vez modificados estos segmentos todos los procesadores envían sus modificaciones a P_0 con una operación de recolección, y este procesador puede computar las actualizaciones de los bloques $U_{0,j}$. Por lo tanto, son necesarias $(t-1)$ operaciones de dispersión y $(t-1)$ operaciones de recolección, lo que implica un mínimo de $2(t-1)$ comunicaciones más que las realizadas en el algoritmo SPB. El algoritmo PB se basa

en una estructura segmentada eficiente para la resolución triangular; sin embargo, mantener la misma distribución de R' para el caso de los algoritmos de rotaciones planas implica un gran número de mensajes e importantes tiempos de espera, y por lo tanto, una descompensación y un funcionamiento ineficiente de la segmentación.

Otra posible solución que mantiene más equilibrada la segmentación consiste en que el procesador que computa los ángulos de rotación asociados a un bloque $U_{i,i}$ realice las actualizaciones de la última fila de la matriz correspondientes a esos ángulos. Posteriormente, radiará a los demás procesadores los ángulos de rotación computados en esa iteración del algoritmo así como la parte de la última fila de la matriz necesaria en el cálculo de los ángulos de rotación que todavía no han sido computados. En este caso, el número total de mensajes se reduce a $(t-1)$ radiaciones, pero los mensajes serían de mayor tamaño que en el caso del algoritmo SPB, y por tanto, más costosos.

En el algoritmo SPB, si un procesador tiene asignada la columna $u_{*,i}$ también tiene asignado el elemento $u_{t,i}$ (ver figura 4.9(b)), de modo que es este mismo procesador el que realiza todas las modificaciones de $u_{t,i}$, y por lo tanto, tiene todos los valores de este elemento necesarios para actualizar la columna $u_{*,i}$ sin necesidad de realizar ninguna comunicación.

Por lo tanto, concluimos que el método SPB es la mejor alternativa para implementar nuestro algoritmo cuasi-Newton paralelo.

4.3. Algoritmo paralelo para la resolución de rotaciones globales planas

En cada iteración del método cuasi-Newton la matriz aproximación de la Hesiana sufre dos rotaciones globales planas. El algoritmo paralelo para computarlas debe ser lo más eficiente posible porque se va a ejecutar tantas veces como iteraciones necesita el algoritmo cuasi-Newton para encontrar la solución al problema de optimización. Un algoritmo paralelo eficiente en un multiprocesador de memoria distribuida implica la minimización de las comunicaciones entre procesadores y de los tiempos de espera. Como hemos visto en la sección 3.4, dentro del algoritmo secuencial de rotaciones globales planas, es la rotación global hacia adelante la que presenta mayores dependencias. Por lo tanto, el algoritmo paralelo propuesto se trata de adaptar al flujo de datos de esta rotación en detrimento de la rotación global hacia atrás.

En la sección anterior, se ha comparado el rendimiento de nuestro algoritmo

paralelo SPB frente a otros algoritmos paralelos para la resolución triangular. Se ha visto que nuestro algoritmo presenta un rendimiento adecuado, dadas las fuertes dependencias de datos de la resolución triangular. Asimismo, se ha comprobado que otro de los algoritmos paralelos analizados (el algoritmo PB) alcanza un rendimiento similar. El motivo por el que nos decantamos por el algoritmo SPB para aplicarlo a un método cuasi-Newton, es su mejor adaptación a las dependencias de datos de la rotación global hacia adelante como se describe en la sección 4.2.1.

La clave para que un programa paralelo sobre un sistema multiprocesador de memoria distribuida sea eficiente es conseguir que todos los procesadores estén trabajando a la vez la mayor cantidad de tiempo posible. Nuestra estrategia SPB consigue exactamente eso cuando su distribución de datos se aplica a la rotación global hacia adelante [72].

Dada una rotación $(j, n-1)$, para que P procesadores puedan calcular en paralelo las modificaciones de la fila j $(j > P)$ y la fila $n-1$ de la matriz R' (ver sección 3.2.2), es necesario que se hayan calculado previamente al menos P ángulos de rotación. Si la matriz R' se distribuye de forma cíclica por columnas, el procesador que tiene asociada la columna j no puede empezar a computar el ángulo asociado a la rotación $(j, n-1)$ hasta que no ha recibido los ángulos asociados a todas las rotaciones previas. De igual modo, si optamos por una distribución cíclica por bloques de columnas, disminuye el número de mensajes, pero se incrementa el tiempo de espera de los procesadores, de modo que éstos no pueden empezar hasta que reciban el primer mensaje. Por lo tanto, se deduce que resulta más eficiente que P ángulos de rotación sean computados por todos los procesadores de forma redundante, con el fin de disminuir el número de mensajes. El algoritmo SPB que divide a la matriz R' en bloques triangulares $bloqtg_i$ $(U_{i,i})$ de ancho $s = \alpha P$ $(\alpha \in IN)$ y bloques rectangulares $bloqrt_{i,j}$ $(U_{i,j})$ se adapta a esta estructura de dependencias de la rotación global hacia adelante. Por lo tanto, proponemos que los elementos de la matriz asociados a los bloques triangulares se repliquen en todos los procesadores. Mientras que dentro de cada bloque rectangular, las columnas se distribuyen de forma cíclica entre todos los procesadores del sistema, la última fila de R', almacenada en un vector v, inicialmente se copia en todos los procesadores. De esta manera, las computaciones asociadas a los bloques triangulares, es decir, la computación de los ángulos de rotación y las actualizaciones de los elementos de la matriz pertenecientes al bloque triangular, son ejecutadas por todos los procesadores. Las actualizaciones de los elementos de la última fila se computan a veces de forma redundante por todos los procesadores del sistema, y a veces de forma concurrente.

Una vez realizadas las operaciones asociadas al bloque triangular i, se dispone

Algoritmo 7 Algoritmo SPB para la rotación global hacia adelante

```
for i = 0 to n_α − 1 do
  Computación secuencial redundante
    for j ∈ columnas_bloqtg_i do
      Cálculo del ángulo asociado a la rotación ψ_{j,n−1}
      Actualizaciones de los elementos [is, (i+1)s − 1] de v.
      Actualización de los elementos de R' asociados a bloqtg_i
    end for
  Computación paralela
    for j ∈ columnas_bloqrt_{k,i+1}, k ≤ i do
      if j ∈ mis_columnas then
        Actualización de la columna j de R'.
        Actualización del elemento v_j
      end if
    end for
  allgather [v((i+1)s), v((i+2)s − 1)]
end for
```

de s nuevos ángulos de rotación ($[\psi_{is,n-1}, \psi_{(i+1)s-1,n-1}]$). A continuación, se pueden computar las actualizaciones, asociadas a las rotaciones $\psi_{0,n-1}$, ... , $\psi_{(i+1)s-1,n-1}$, de los elementos de los bloques rectangulares situados encima del bloque triangular $i+1$, y del segmento de última fila $v[(i+1)s, (i+2)s-1]$ situado bajo el triángulo $i+1$.

Las actualizaciones de los elementos de v debidas a las últimas s rotaciones se computan, primero, de forma secuencial en todos los procesadores después de haberse calculado los últimos s ángulos de rotación. Sin embargo, en los demás elementos de v ($\forall v_j, j \geq (i+1)s$) las $(i+1)s$ primeras rotaciones se aplican en paralelo, con los elementos distribuidos de forma cíclica entre los P procesadores del sistema. Para computar los ángulos de rotación asociados al bloque triangular $i+1$ se necesita el valor actualizado de los elementos $[(i+1)s, (i+2)s-1]$ de v después de haber sufrido la aplicación de las primeras $(i+1)s$ rotaciones planas. Como el valor más reciente de estos elementos está distribuido entre los distintos procesadores, es necesaria una operación de recolección de datos para que los P procesadores del sistema tengan el valor correcto de este segmento del vector v y puedan calcular los siguientes s ángulos de rotación. El proceso continúa de este modo hasta que se han recorrido todos los bloques triangulares y se han aplicado las n rotaciones planas.

El pseudocódigo asociado al algoritmo SPB aplicado a las rotación global hacia

adelante se muestra en el algoritmo 7. Podemos observar que, de forma similar a lo que ocurre con la resolución triangular, el número de comunicaciones se ha reducido a $n_\alpha - 1$ (siendo $n_\alpha = \lceil n/\alpha P \rceil$).

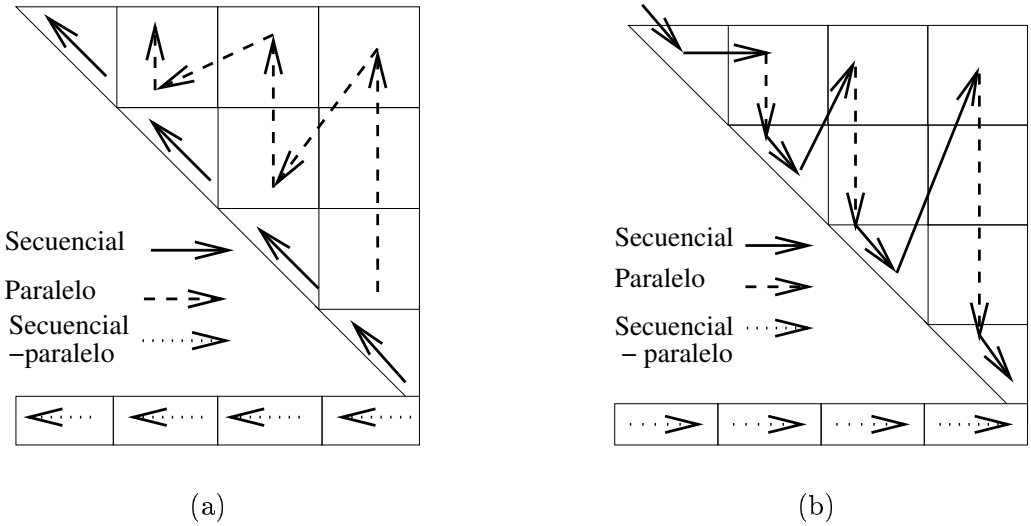
Algoritmo 8 Algoritmo SPB para la rotación global hacia atrás

Computación secuencial redundante
  for $i = n_\alpha - 1$ to 0 **do**
    for $j \in columnas\_bloqtg_i$ **do**
      Cálculo del ángulo asociado a la rotación $\psi_{n-1,j}$
      Actualizaciones de los elementos de R asociados a $bloqtg_i$.
      Primeras s actualizaciones de $v[is, (i+1)s - 1]$.
    end for
  end for
Computación paralela
  for $i = n_\alpha - 1$ to 1 **do**
    for $j \in columnas\_bloqrt_{k,i}, k < i$ **do**
      if $j \in mis\_columnas$ **then**
        Actualización de la columna j de R.
        Actualización del elemento v_j
      end if
    end for
  end for
allgather $v[s, n-1]$

En la rotación global hacia atrás no hay dependencias de datos en el cálculo de los ángulos de rotación. Sin embargo, proponemos mantener la distribución de los datos en bloques triangulares y rectangulares para evitar comunicaciones entre la ejecución de las dos rotaciones. Por lo tanto, aplicamos la misma distribución de datos del algoritmo SPB a la rotación global hacia atrás. Primero, se calculan todos los ángulos de rotación de forma redundante en todos los procesadores. A continuación, todos los procesadores actualizan los elementos de los bloques triangulares y de la parte de la última fila (v) situada debajo de cada bloque tras la aplicación de las rotaciones planas asociadas a dicho bloque. Posteriormente, y de forma concurrente, cada procesador actualiza los elementos de las columnas de R, pertenecientes a bloques rectangulares, y los elementos de v que tiene asignados. Finalmente, y para poder aplicar la rotación global hacia adelante, el valor de esta última fila, ahora

densa, se debe copiar en todos los procesadores, siendo necesaria una comunicación para recoger los datos. El algoritmo 8 muestra el pseudocódigo del código paralelo de la rotación global hacia atrás.

Si comparamos los algoritmos 7 y 8 podemos observar como este último no presenta las dependencias de datos entre computaciones secuenciales y paralelas del primero, reduciéndose el número de mensajes de $n_\alpha - 1$ a uno. Otra diferencia existente entre los dos algoritmos reside en el flujo de datos, mientras el algoritmo 7 empieza a aplicar las rotaciones por las primeras filas de la matriz, el algoritmo 8 comienza por las últimas. En la figura 4.10 se muestra mediante flechas el sentido que sigue el flujo de datos de las operaciones redundantes y de las concurrentes en ambos algoritmos. La flecha denominada secuencial-paralelo hace referencia a computaciones que son realizadas de forma secuencial por todos los procesadores cuando estos actualizan los bloques triangulares, y que además, se ejecutan de forma concurrente cuando cada procesador actualiza las columnas que tiene asignadas dentro de cada bloque rectangular.



(a) (b)

Figura 4.10: (a) Flujo de datos del algoritmo SPB para la rotación global hacia atrás. (b) Flujo de datos del algoritmo SPB para la rotación global hacia adelante.

4.4. Resultados experimentales del algoritmo paralelo SPB

Los métodos de optimización basados en un conjunto activo de restricciones parten de un conjunto de trabajo que aumenta en cada iteración, aumentando también el tamaño de la aproximación de la Hesiana. Por lo tanto, se debe decidir a partir de qué tamaño de matriz resulta eficiente aplicar nuestro algoritmo paralelo a las etapas de cálculo de la dirección de búsqueda y actualización de la aproximación cuasi-Newton. Las figuras 4.11 y 4.12 muestran los tiempos de ejecución del algoritmo paralelo SPB aplicado a estas dos etapas sobre el AP3000 de Fujitsu para distintos tamaños de matriz. En estas figuras se observa que en el caso del algoritmo de las rotaciones planas, empleado en la actualización cuasi-Newton, la aplicación del algoritmo paralelo SPB compensa a partir de un tamaño de matriz de aproximadamente $n = 600$ para todos los valores de α utilizados ($1 \leq \alpha \leq 10$). En el caso de los métodos de resolución triangular, y debido a que poseen un menor número de operaciones en punto flotante, el algoritmo paralelo será eficiente para tamaños de matriz mayores, con los valores de α utilizados. Sin embargo, observamos que si el orden de la matriz es aproximadamente $n = 600$, y el parámetro $\alpha \geq 4$, entonces el algoritmo paralelo es escalable. Por lo tanto, se puede concluir que es eficiente ejecutar los dos algoritmos de forma paralela para matrices de orden mayor o igual a 600, siempre que se elija un parámetro $\alpha \geq 4$.

Similares conclusiones se pueden extraer de la aplicación de este algoritmo sobre el *cluster beowulf* del CESGA. Las figuras 4.13 y 4.14 muestran los tiempos de ejecución en este sistema del algoritmo paralelo SPB aplicado al cálculo de la dirección de búsqueda y a la actualización de la Hesiana respectivamente, para matrices de prueba del mismo tamaño que las utilizadas en el AP3000 de Fujitsu. Se puede observar que en el caso de la actualización de la Hesiana, el tiempo del algoritmo paralelo empieza a ser inferior al del secuencial para una matriz de orden $n = 600$ aproximadamente. En la resolución triangular, debido al menor número de operaciones en punto flotante y a que, ahora, la relación T_{comu}/T_{comp} es mayor, el algoritmo paralelo no compensa hasta que el algoritmo iterativo trabaje con una matriz de orden mayor o igual a 600, siempre que se elija un parámetro $\alpha \geq 6$. Al incrementar este parámetro, en nuestro algoritmo paralelo aumenta el número de computaciones que se realizan en forma secuencial, y disminuye el número de comunicaciones entre los distintos procesadores del sistema.

Este límite de $n = 600$ se tiene que tener en cuenta a la hora de seleccionar distintos problemas de prueba para validar la eficiencia del algoritmo paralelo SPB.

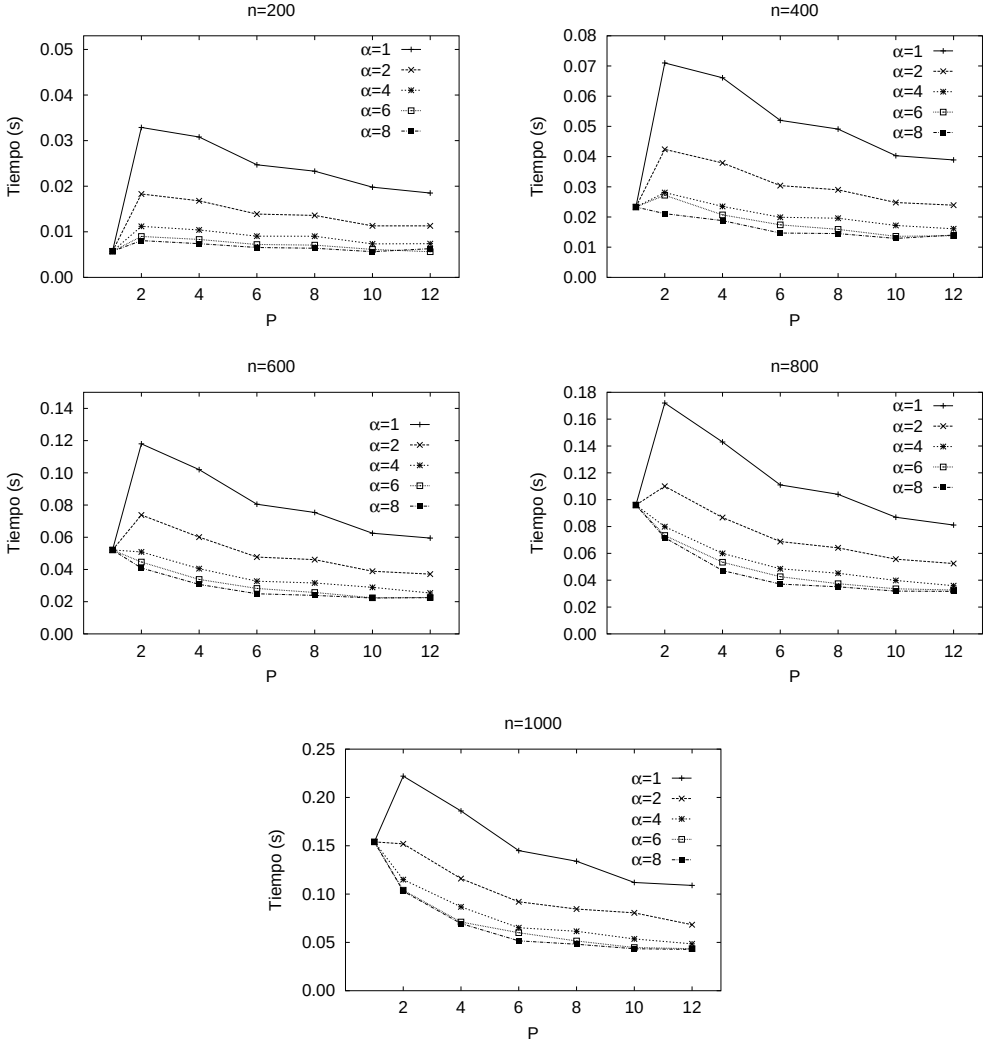


Figura 4.11: Tiempos de ejecución del algoritmo de resolución triangular SPB en el AP3000 de Fujitsu.

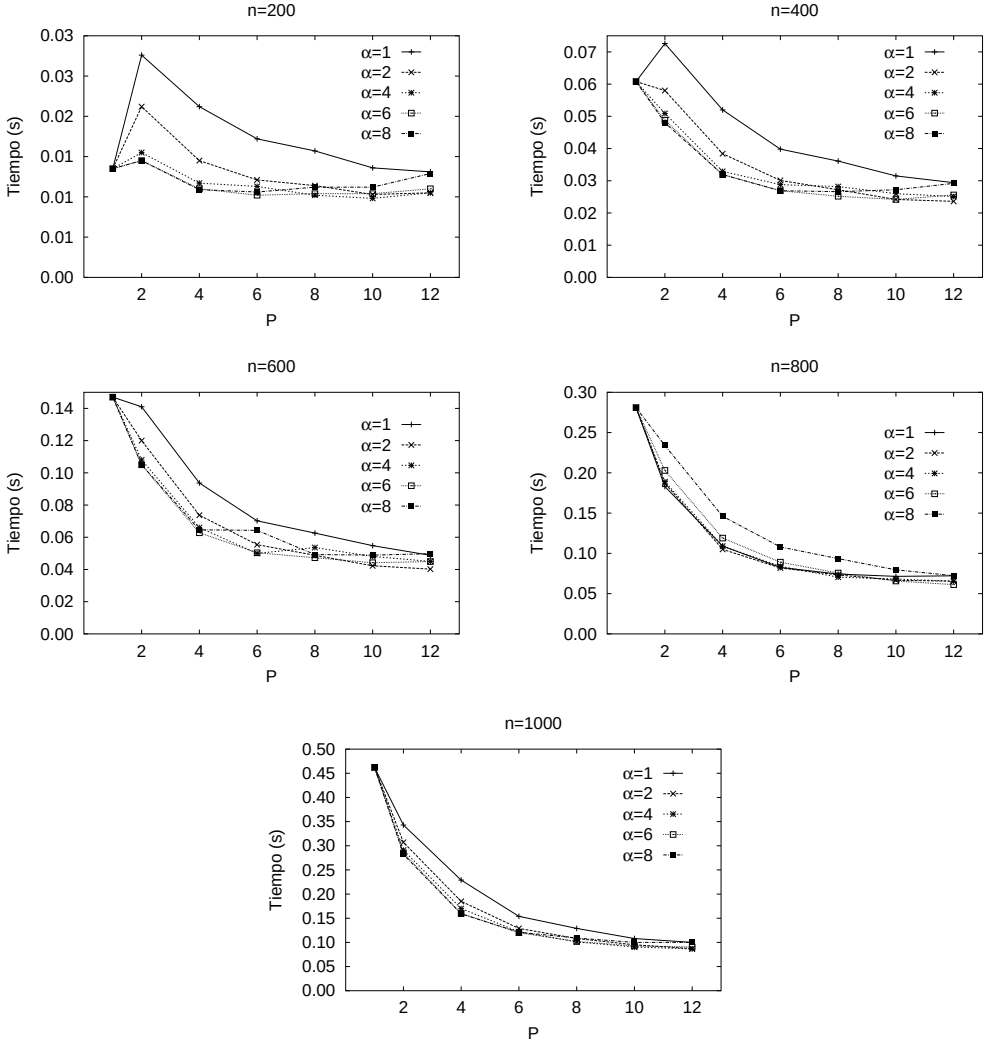


Figura 4.12: Tiempos de ejecución del algoritmo paralelo de rotaciones planas en el AP3000 de Fujitsu.

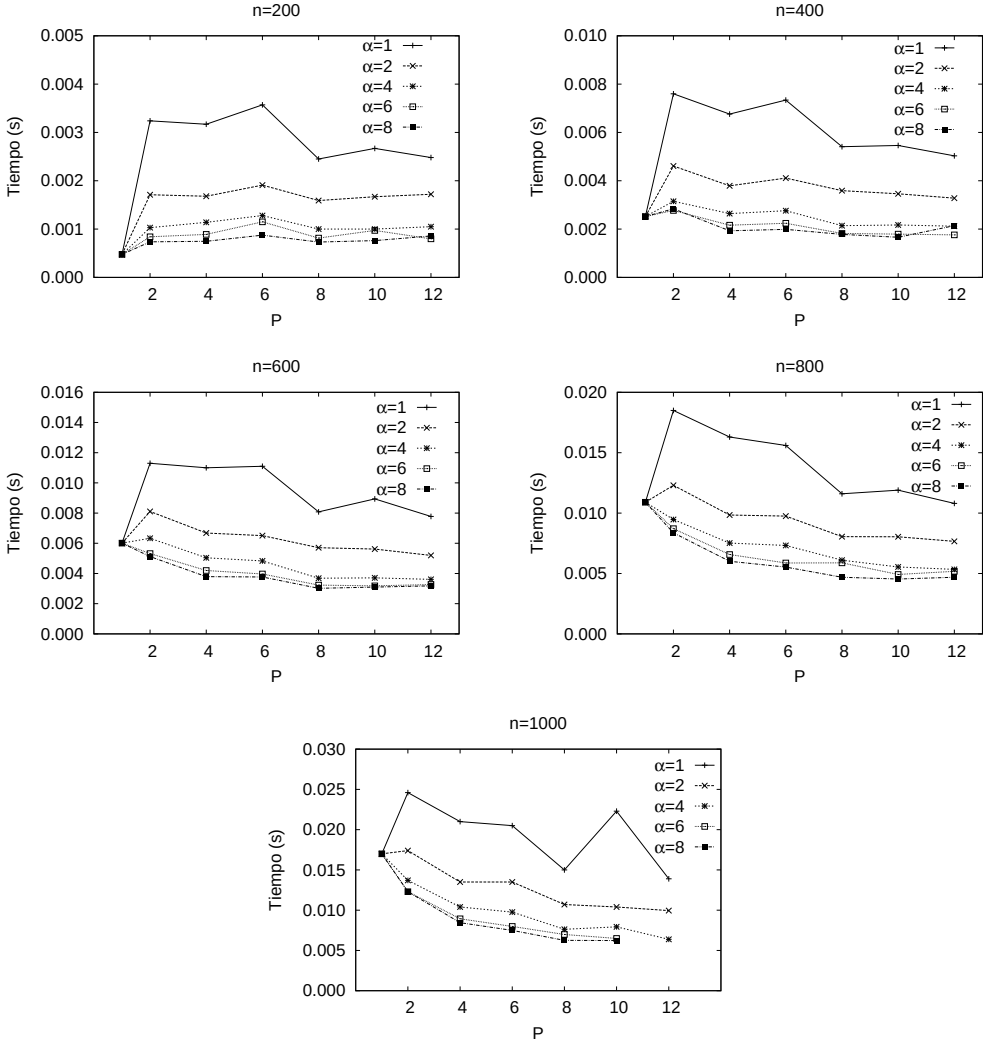


Figura 4.13: Tiempos de ejecución del algoritmo de resolución triangular SPB en el *cluster beowulf* del CESGA.

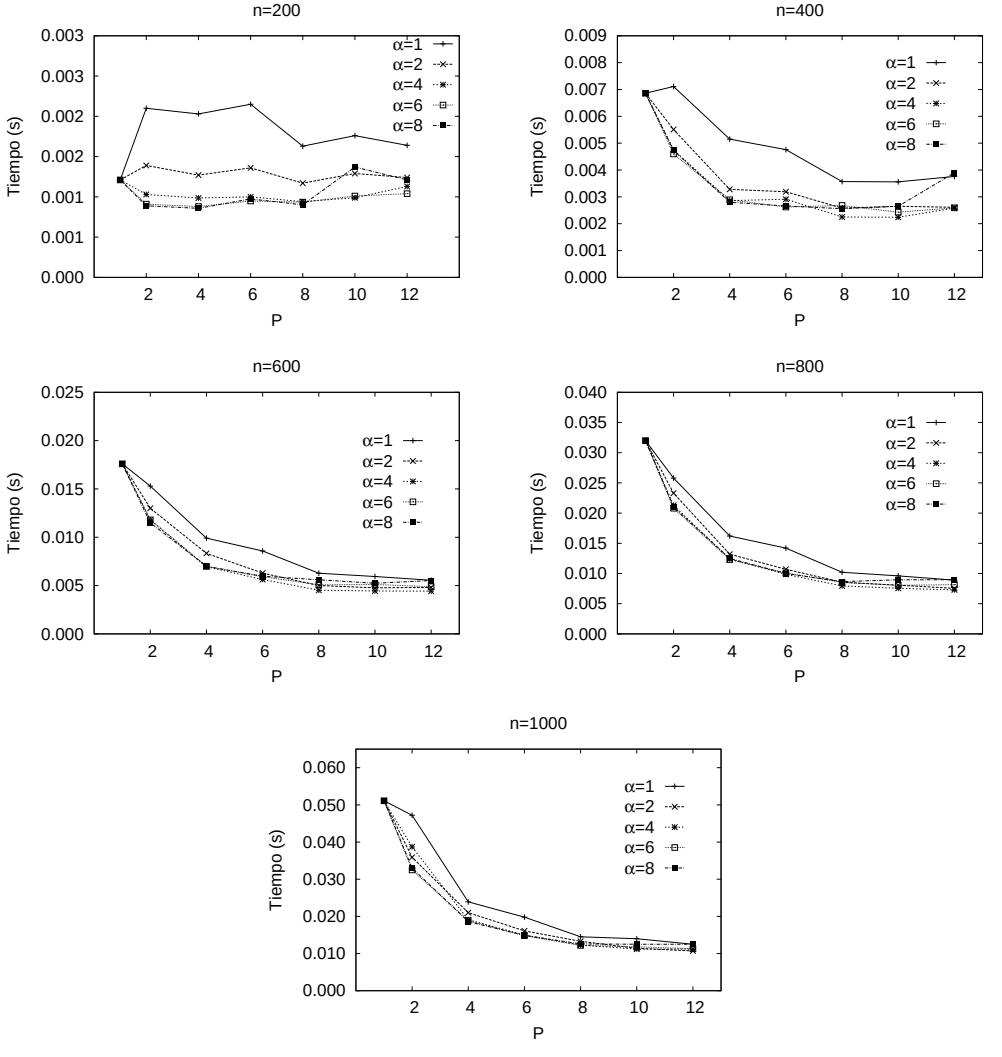


Figura 4.14: Tiempos de ejecución del algoritmo paralelo de rotaciones planas en el *cluster beowulf* del CESGA.

A continuación, se muestran los resultados de aplicación del algoritmo SPB sobre las dos subrutinas analizadas en el multiprocesador de memoria distribuida AP3000 de Fujitsu y en el *cluster beowulf* del CESGA. Los códigos paralelos han sido implementados utilizando la librería de paso de mensajes MPI [41]. Hemos utilizado matrices de prueba de distintos tamaños y asociadas a distintos problemas de optimización de la librería CUTE [9]. La tabla 4.4 resume sus características. En ella se describe el tipo de restricciones y el tipo de función objetivo de los problemas de optimización utilizados, y el tamaño de la matriz aproximación de la Hesiana.

Problema	Matriz	Restricciones	Función objetivo
AUG2D	1600	Lineales	Cuadrática
DTOC1L	1998	Lineales	No lineal, no cuadrática
HUES-MOD	1430	Lineales	Cuadrática
OBSTACLE	2200	Bordes	Cuadrática
WOODS	3500	Sin restricciones	Suma de cuadrados

Tabla 4.4: Problemas de prueba.

En los resultados experimentales que se muestran en todas las figuras de este capítulo, la medida de los tiempos de ejecución para $P = 1$ se corresponde con el tiempo que tarda la subrutina secuencial empleada por MINOS en realizar la actualización de la aproximación cuasi-Newton y en resolver los sistemas triangulares para obtener la dirección de búsqueda.

4.4.1. Implementación sobre el AP3000 del algoritmo SPB para la aplicación de rotaciones planas

A modo de ejemplo ilustrativo, los resultados obtenidos al aplicar el algoritmo paralelo SPB a las rotaciones planas sobre el AP3000 de Fujitsu se muestran en las figuras 4.15, 4.17, 4.19 para dos de los problemas de la librería CUTE. La matriz triangular generada por el problema AUG2D es de orden 1600, y la generada por el problema OBSTACLE es de orden 2200. Se muestran resultados de la aceleración obtenida para la aplicación de las rotaciones por separado y también para su aplicación conjunta.

En particular, la figura 4.15 muestra la aceleración del algoritmo paralelo SPB aplicado a la rotación global hacia atrás. Se observa que la aceleración empeora

a medida que aumenta el tamaño de los bloques triangulares, es decir, a medida que aumenta el número de computaciones que se realizan de forma secuencial. En la rotación global hacia atrás no existen dependencias de datos en el cálculo de los ángulos de rotación, por lo tanto, el algoritmo paralelo que hemos propuesto no necesita ningún mensaje para computar estos ángulos. Debido a esta ausencia de dependencias, la realización de computaciones de forma secuencial tampoco sería necesaria en este caso. Sin embargo, se incluyen debido a que como ya hemos visto, es la que mejor se adapta a la estructura de la rotación global hacia adelante. Así pues, el comportamiento observado en la figura 4.15 se debe a que cuanto mayor es el valor de α mayor es el tamaño de los bloques secuenciales, disminuyendo la cantidad de tareas ejecutadas de forma concurrente. Esta disminución en el paralelismo supone un deterioro de la eficiencia del algoritmo paralelo que no se ve compensada con una disminución en el coste de las comunicaciones, ya que esta rotación implica un único mensaje.

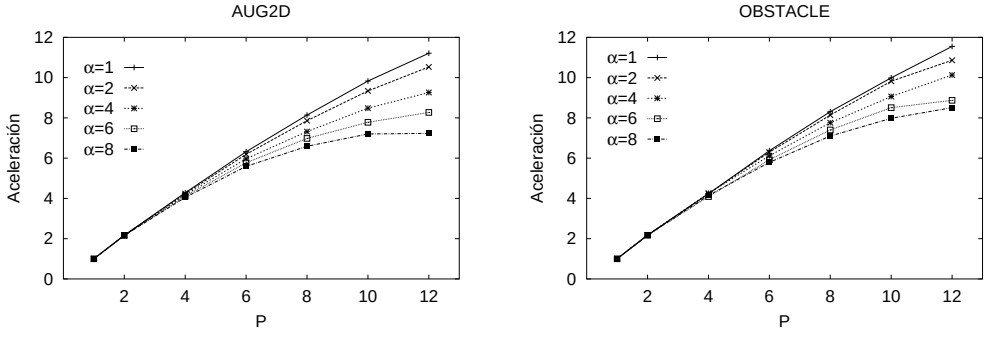


Figura 4.15: Aceleración de la rotación global hacia atrás en el AP3000 de Fujitsu.

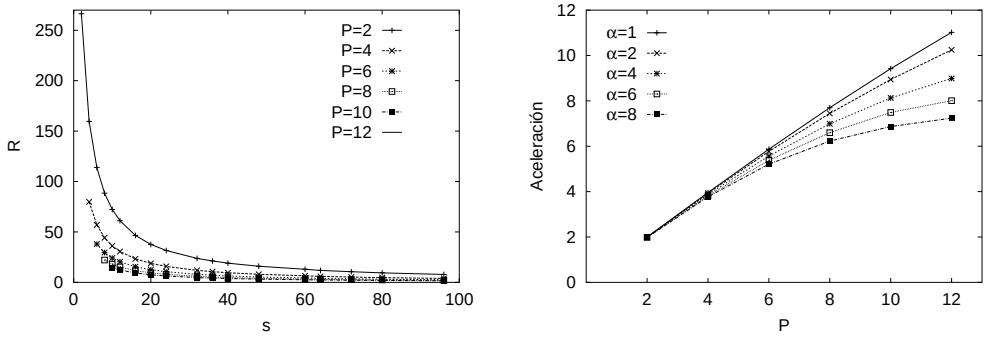
Se ha realizado un estudio teórico de la relación entre operaciones en punto flotante realizadas en paralelo por cada procesador ($FLOPS_{prect}$) y la relación de operaciones en punto flotante a realizar de forma secuencial dentro de los bloques triangulares por todos los procesadores ($FLOPS_{stg}$) en el algoritmo paralelo SPB. En la figura 4.16(a) se muestran los resultados de $R = FLOPS_{prect}/FLOPS_{stg}$ para el problema AUG2D. Se observa que a medida que aumenta el número de procesadores esta relación es más pequeña para un mismo valor de tamaño de los bloques. Esta disminución se debe a que como el ancho de bloque s es múltiplo del número de procesadores, aunque el número de operaciones en punto flotante a realizar en secuencial sea el mismo, el número de tareas a realizar por cada procesador de for-

ma paralela (dentro de los bloques rectangulares) es inversamente proporcional al número de procesadores.

En la figura 4.16(b) se ilustra la relación teórica de la aceleración en función de R basada en la ley de Amdahl. No hemos considerado para este estudio el tiempo de comunicación, por ser, en este caso, su coste muy inferior al coste de las computaciones. De esta manera, se verifica que:

$$Aceleracion = \frac{T_{sec}}{T_{pal}} \approx \frac{FLOPS_{stg} + FLOPS_{prect}P}{FLOPS_{stg} + FLOPS_{prect}} = \frac{1 + RP}{1 + R}. \qquad (4.7)$$

Los resultados de la figura 4.16(b) coinciden prácticamente con los medidos de forma experimental en el AP3000 de Fujitsu.



(a) (b)

Figura 4.16: (a) Relación entre los FLOPS ejecutados en paralelo y los FLOPS computados secuencialmente en el algoritmo SPB frente al tamaño de bloque. (b) Aceleración teórica basada en la ley de Amdahl para el algoritmo de las rotación global hacia atrás.

Sin embargo, para el caso de la rotación global hacia adelante, en la figura 4.17 se observa un comportamiento diferente, la escalabilidad se incrementa con el tamaño de los bloques triangulares hasta llegar a bloques de un determinado ancho $s = \alpha P$ en los que el comportamiento empeora. La rotación global hacia adelante implica fuertes dependencias de datos en el cálculo de los ángulos de rotación. En este caso, antes de calcular estos ángulos es necesario realizar una comunicación colectiva para

que todos los procesadores dispongan de los valores de los elementos de la última fila, que han sido modificados previamente de forma paralela. Cuando el tamaño de los bloques triangulares aumenta, disminuye el tiempo de cálculo paralelo, pero también el número de mensajes, compensándose la pérdida de paralelismo con la disminución en el coste de las comunicaciones hasta un determinado valor de αP. La mayor eficiencia se obtiene para bloques triangulares de ancho $s = 4P$ o $s = 6P$, siendo los resultados de aceleración para los dos casos prácticamente iguales.

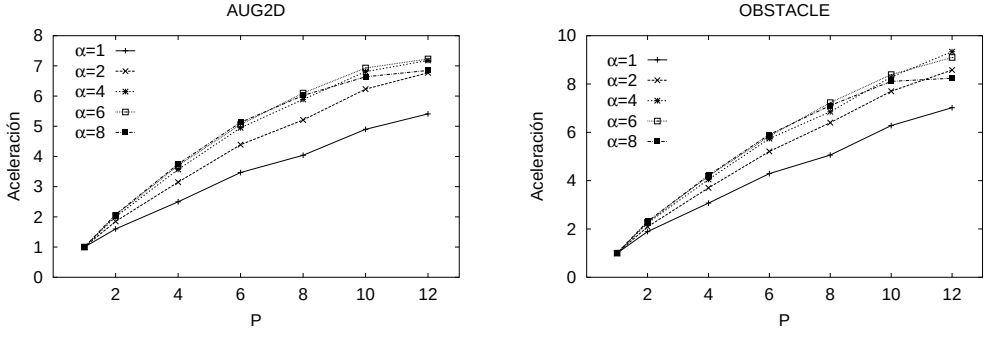


Figura 4.17: Aceleración de la rotación global hacia adelante en el AP3000 de Fujitsu.

En la figura 4.18 se muestra el número de mensajes del algoritmo SPB aplicado a la rotación global hacia adelante frente al número de operaciones en punto flotante asociado a los bloques triangulares para el problema AUG2D y para diferentes valores de s. Se observa que a medida que aumenta el ancho de estos bloques, el número de mensajes decrece exponencialmente mientras que el número de FLOPS secuenciales crece de forma cuadrática. Para este caso a partir de un tamaño de bloque $s = 64$ se observa que el coste de las comunicaciones no disminuye apreciablemente. Sin embargo, el coste de los FLOPS secuenciales se incrementa en detrimento de las computaciones paralelas, produciéndose un deterioro de la aceleración como muestra la figura 4.17.

Finalmente, en la figura 4.19 se muestra la aceleración obtenida por el algoritmo SPB tras la aplicación de las dos rotaciones planas. La aceleración está determinada fundamentalmente por el algoritmo de rotación global hacia adelante, que es el que implica mayor número de dependencias. Globalmente la escalabilidad de la propuesta decrece, y los mejores resultados se obtienen para bloques de ancho $s = 4P$.

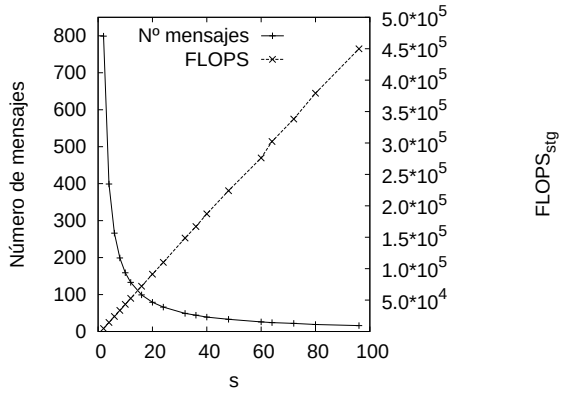


Figura 4.18: Número de mensajes frente al número de operaciones en punto flotante ejecutadas de forma secuencial en la rotación global hacia adelante.

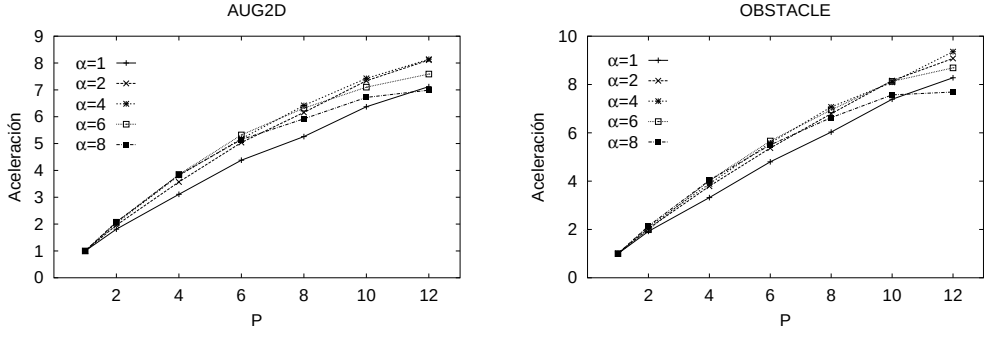


Figura 4.19: Aceleración de las dos rotaciones globales planas en el AP3000 de Fujitsu.

4.4.2. Implementación sobre el AP3000 del algoritmo SPB para el cálculo de la dirección de búsqueda

A modo ilustrativo, los resultados obtenidos al aplicar el algoritmo paralelo SPB al cálculo de la dirección de búsqueda sobre el AP3000 de Fujitsu se muestran en la figura 4.20 para dos nuevos problemas de la librería CUTE. El algoritmo de cálculo de la dirección de búsqueda implica la resolución de dos sistemas triangulares: una resolución triangular inferior, y posteriormente una resolución triangular superior. La matriz triangular generada por el problema HUESMOD es de orden 1430 y la generada por el problema DTOC1L es de orden 1998.

En estos resultados se observa que aunque el algoritmo paralelo SPB es escalable, cuando se aplica a la resolución triangular, su eficiencia es menor que cuando se aplica al algoritmo de las rotaciones planas. El motivo radica en el mayor número de operaciones en punto flotante que se deben computar al realizar una rotación. Por ejemplo, mientras que en la resolución triangular inferior sólo se realizan n^2 operaciones en punto flotante (siendo n el orden de la matriz triangular), en la rotación global hacia adelante se tienen que ejecutar $6n + 3n(n-1)$ operaciones en punto flotante.

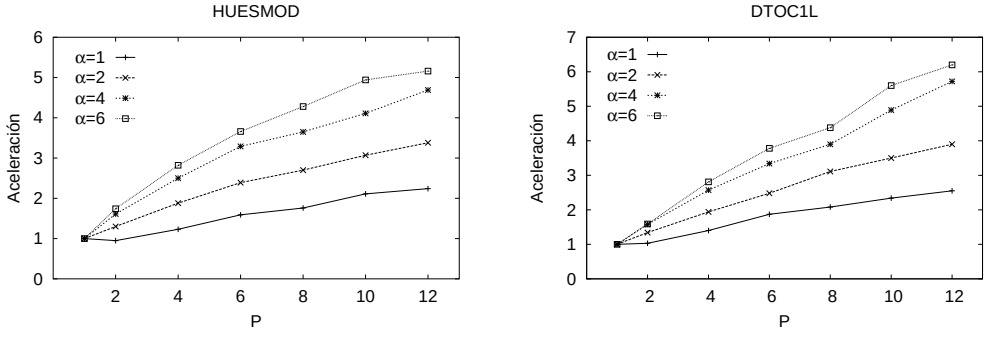


Figura 4.20: Aceleración del algoritmo paralelo para el cálculo de la dirección de búsqueda en el AP3000 de Fujitsu.

Debido al reducido número de operaciones en punto flotante que implica una resolución triangular, observamos, por ejemplo, que cuando $\alpha = 1$ (bloques de ancho $s = P$) en el problema HUESMOD, se produce un ligero aumento del tiempo de computación cuando se pasa de uno a dos procesadores (figura 4.20). Esto se debe a que el ahorro computacional derivado de la ejecución del algoritmo paralelo no es suficiente para superar la sobrecarga introducida por las comunicaciones. Para el problema DTOC1L esto no sucede de manera tan apreciable porque en este caso la matriz triangular es de mayor tamaño, lo que implica un mayor número de operaciones en punto flotante.

El efecto del reducido número de operaciones en punto flotante de la resolución triangular también se aprecia en la diferencia de las curvas de aceleración para distintos tamaños de bloque. En las rotaciones planas los resultados de aceleración son buenos para pequeños tamaños de bloque, produciéndose un pequeño decremento de la aceleración al aumentar el ancho del bloque (figura 4.19). Sin embargo, en

las resoluciones triangulares para un ancho $s = P$, los resultados son inferiores; no obstante, a medida que aumenta ese ancho, las aceleraciones mejoran notablemente hasta llegar a un tamaño de los bloques secuenciales que reduce tanto la cantidad de carga concurrente que se pierde el beneficio del paralelismo.

4.4.3. Implementación sobre el *cluster beowulf* del CESGA del algoritmo SPB

Para ilustrar el rendimiento de nuestro código paralelo sobre el *cluster beowulf* del CESGA se han utilizado dos problemas de la librería CUTE, el problema DTOC1L y el problema WOODS con matrices triangulares del orden de 1998 y 3500, respectivamente. Los resultados de aceleración obtenidos al aplicar el algoritmo SPB a la actualización cuasi-Newton se muestran en la figura 4.21. Lo primero que se puede destacar de estas gráficas es que debido al incremento de la relación T_{comu}/T_{comp} con relación al AP3000 de Fujitsu, la aceleración de nuestro algoritmo en este sistema paralelo es menor. Así, se observa que en el *cluster beowulf* del CESGA el problema WOODS con una matriz de orden 3500 alcanza la misma aceleración que el problema OBSTACLE con una matriz de orden 2200 en el AP3000. Este efecto se ve todavía más acentuado en la figura 4.22 donde se muestran los resultados de aplicar el algoritmo SPB al cálculo de la dirección de búsqueda.

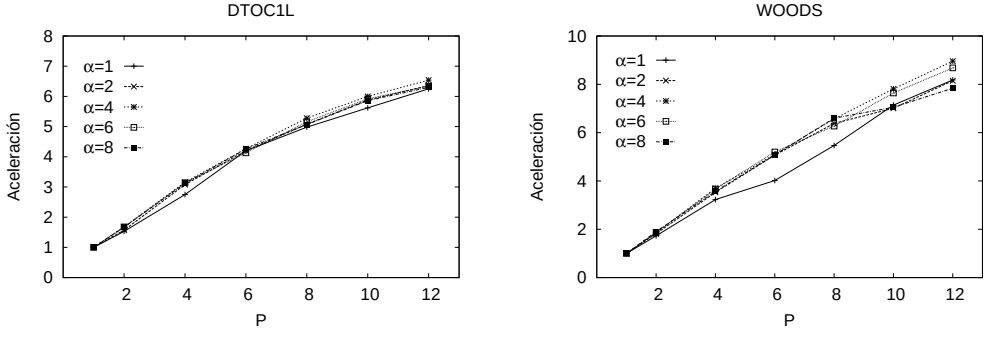


Figura 4.21: Aceleración de las dos rotaciones globales planas en el *cluster beowulf* del CESGA.

En estas dos figuras también se puede observar que si el número de operaciones en punto flotante es pequeño (algoritmo de cálculo de la dirección de búsqueda),

al aumentar el tamaño del parámetro α aumenta la aceleración para un mismo número de procesadores. Esto es debido a que las comunicaciones son más costosas y compensa reducir su número a pesar de disminuir el número de computaciones realizadas concurrentemente. A medida que el algoritmo implica un mayor número de computaciones en punto flotante la situación cambia (figura 4.21). Para el problema DTOC1L se observa que la aceleración prácticamente no depende del valor de α. Aumentar el tamaño de bloque implica una reducción en el número de mensajes y un incremento de las operaciones realizadas de forma secuencial. Ambos costes se compensan anulando el efecto de aumentar los tamaños de bloque. Sin embargo, si el número de operaciones en punto flotante es mayor (problema WOODS), llega un momento que aumentar el tamaño de bloque supone que el coste de las operaciones secuenciales sea superior al coste de las comunicaciones, produciéndose un decremento de la aceleración.

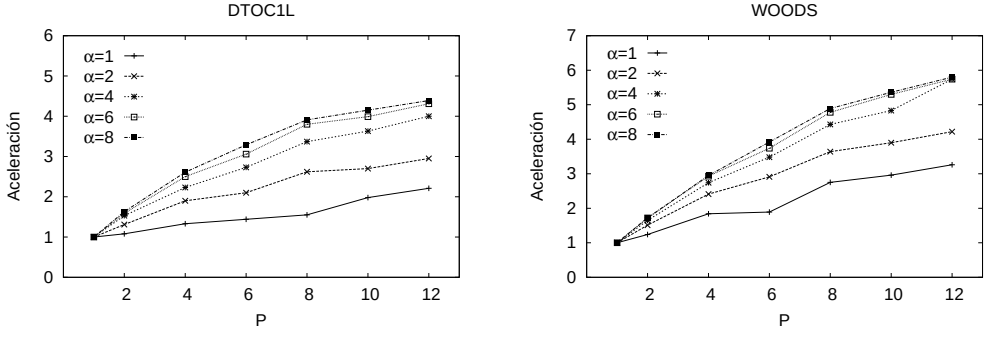


Figura 4.22: Aceleración del algoritmo paralelo para el cálculo de la dirección de búsqueda en el *cluster beowulf* del CESGA.

Finalmente, en ambas figuras, pero sobre todo en la figura 4.22, se aprecia como existe un mayor incremento en la aceleración al pasar de 6 a 8 procesadores que al pasar de 4 a 6. Esto es debido a que en la red de interconexión del *beowulf*, las comunicaciones colectivas de *allreduce* y *allgather* son más costosas para 6 que para 8 procesadores.

4.5. Algoritmo paralelo de búsqueda de tamaño de paso

Este algoritmo está compuesto por la computación secuencial del tamaño de paso (todos los procesadores realizan esta operación), la evaluación de la función objetivo y de su gradiente en cada nuevo punto solución, y la posterior verificación de las condiciones de Wolfe para determinar si el nuevo punto solución es válido o si por el contrario será necesario calcular un nuevo tamaño de paso. La secuencia de realización de estos pasos en la iteración k-ésima se muestra en el algoritmo 9. Hemos usado como nomenclatura el subíndice k para hacer referencia al valor del gradiente y de la dirección en el nuevo punto solución $x_k + \lambda_k p_k$.

Algoritmo 9 Algoritmo paralelo para el cálculo del tamaño de paso

Cálculo del tamaño de paso λ_k
Evaluación de la función objetivo en el nuevo punto solución $f(x_{k+1})$
Cálculo del gradiente mediante diferencias finitas
V=0
for $j = 1$ to m **do**
  if $j \in mis\_elementos$ **then**
    $(g_{k+1})_j = \frac{f(x_{k+1}+he_j)-f(x_{k+1})}{h}$
    $V = V + (g_{k+1})_j^T (p_k)_j$
  end if
end for
$g_{k+1}^T p_k$= **Allreduce** (SUMA (V))
if Si se verifican las condiciones de Wolfe **then**
  STOP
else
  Volver al principio
end if

En este algoritmo se supone que el gradiente de la función no está disponible analíticamente, y por lo tanto, es necesario calcularlo en cada nuevo punto solución. Este cálculo, necesario como mínimo una vez en cada iteración, es lo que encarece el coste computacional de esta rutina. Para el cálculo del gradiente se utiliza el método de diferencias finitas, lo que supone la realización de $m + 1$ evaluaciones de

la función, siendo m el número de variables no lineales. Estas evaluaciones no tienen dependencias de datos y se pueden ejecutar completamente en paralelo. El algoritmo propuesto, distribuye estas computaciones de forma cíclica entre los P procesadores del sistema, que las ejecutan sin necesidad de comunicaciones. Una vez que cada procesador ha computado los elementos del gradiente que tiene asignados, tiene que calcular la parte del producto escalar $V = g_{k+1}p_k$ que le corresponde. Finalmente, es necesaria una operación colectiva (*allreduce*) para que todos los procesadores conozcan el valor de V, y puedan comprobar si se cumplen las condiciones de Wolfe.

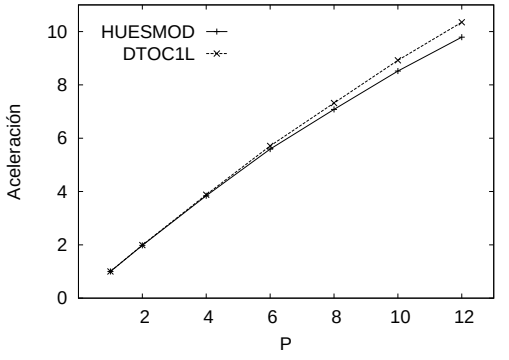


Figura 4.23: Aceleración del algoritmo paralelo de cálculo del tamaño de paso en el AP3000 de Fujitsu.

En la figura 4.23 se muestran los resultados de aceleración del algoritmo paralelo de cálculo del tamaño de paso en el AP3000 de Fujitsu. La práctica ausencia de comunicaciones y la ejecución totalmente concurrente de las evaluaciones, implica que la eficiencia del algoritmo propuesto es prácticamente óptima. En este caso, la única limitación en el rendimiento del algoritmo paralelo está impuesta por el número de variables no lineales y por mínimos desbalanceos de la carga. Para un número de procesadores superior a 10, la eficiencia del algoritmo paralelo disminuye debido al peso que adquiere la operación colectiva de *allreduce* (ver ecuación 4.4). Sin embargo, un incremento del número de variables no lineales, compensa el coste de la operación colectiva y mejora el rendimiento del algoritmo.

Los resultados para el *cluster beowulf* del CESGA se muestran en la figura 4.24. Hay que destacar que en este caso los resultados son muy inferiores a los obtenidos en el AP3000. La razón está en el elevado coste de la comunicación colectiva *allreduce* en

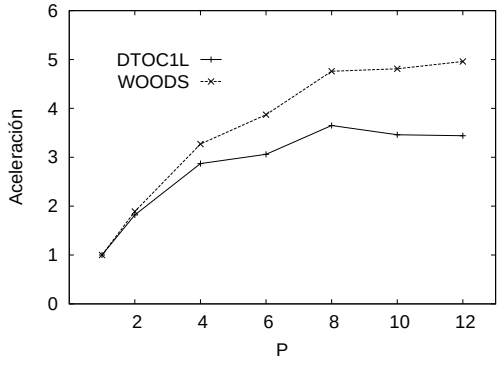


Figura 4.24: Aceleración del algoritmo paralelo de cálculo del tamaño de paso en el *cluster beowulf* del CESGA.

el *beowulf*, siendo ésta más costosa a medida que aumenta el número de procesadores. En la gráfica se observa que con dos procesadores la eficiencia del algoritmo es prácticamente óptima, y se degrada al aumentar el número de procesadores.

Capítulo 5

Balanceo de la carga y estrategias de redistribución dinámica

El balanceo de la carga es uno de los aspectos que más se debe cuidar para alcanzar un alto rendimiento en cualquier implementación paralela. Su objetivo es distribuir el trabajo equitativamente entre los distintos procesadores del sistema evitando los tiempos de espera y minimizando el coste de las comunicaciones. Una solución adaptativa del problema consiste en que, tras una distribución inicial estática, si la carga computacional de los procesadores cambia durante la ejecución del código, se realicen redistribuciones dinámicas de la misma.

El estudio del balanceo de la carga ha sido uno de los temas relevantes de la computación paralela. En la literatura sobre el tema se han propuesto distintos algoritmos [47] que se pueden clasificar de diferentes formas. En primer lugar, se puede distinguir entre algoritmos centralizados o descentralizados, según tengan o no un proceso especial que funcione como coordinador único o maestro y que se encargue de recoger la información sobre la carga y de distribuir la misma procesando dicha información. También podemos hablar de algoritmos trasladables, si tienen la capacidad de monitorizar la carga de los distintos nodos y hacer migrar computaciones desde los nodos más sobrecargados a los menos, siempre que el desbalanceo supere un cierto umbral predefinido. Finalmente, los algoritmos se pueden clasificar atendiendo a su carácter estático o dinámico. Los algoritmos estáticos usan solamente información inicial para distribuir la carga en el sistema, mientras que los dinámicos utilizan el estado de los nodos durante la ejecución del algoritmo.

En este capítulo, se aborda el problema del balanceo de la carga en la distribución inicial propuesta en nuestro algoritmo paralelo. Se estudian posibles situaciones de desbalanceo debido a la eliminación de variables dentro del conjunto de trabajo,

lo que implica que algunos procesadores pueden ver reducida su carga computacional a medida que se ejecuta el programa. Eventualmente, para recuperar el buen rendimiento de la implementación paralela, será necesario aplicar estrategias de redistribución de la carga eficientes. Proponemos tres posibles heurísticas de redistribución, que atendiendo a los modos de clasificación dados anteriormente, se pueden considerar como descentralizadas, trasladables y dinámicas. Estas heurísticas son comparadas, tanto entre ellas como con otras propuestas existentes en la literatura del tema, en cuanto a coste y eficacia para determinar cuál se adapta mejor a nuestro algoritmo paralelo. Además, se establece cuál es el umbral de desbalanceo que implica una redistribución de la carga eficiente. Este umbral se determina en función del coste de las computaciones y de las comunicaciones del sistema paralelo utilizado.

5.1. Balanceo de la carga

Hemos utilizado para caracterizar cuantitativamente el balanceo de la carga computacional (en FLOPs) la siguiente métrica:

$$B = \frac{FLOP_{pal}}{P \cdot FLOP_{max}} = \frac{FLOP_{total} - FLOP_{sec}}{P \cdot FLOP_{max}}, \tag{5.1}$$

donde $FLOP_{total}$ es el número de operaciones en punto flotante necesarias para ejecutar el algoritmo secuencial. Dentro del código paralelo, $FLOP_{pal}$ es el número de operaciones en punto flotante que se ejecutan de modo concurrente, y $FLOP_{sec}$ es el número de operaciones en punto flotante que realizarán todos los procesadores de forma secuencial. $FLOP_{max} = \text{máx}\{FLOP_i\}$ $(1 \leq i \leq P)$, es el valor de la carga computacional máxima asignada a los procesadores en el código paralelo. Por último, P es el número de procesadores del sistema.

El valor de B es tal que $0 < B \leq 1$. En el caso de carga completamente balanceada, el resultado es la unidad, ya que $FLOP_{max} = FLOP_{pal}/P$. Cuanto peor balanceada se encuentre la carga, B se aproximará más a $1/P$. El balanceo de la carga computacional constituye un límite superior en la eficiencia E del programa paralelo. Teniendo en cuenta que en nuestro algoritmo paralelo $FLOP_{sec}$ es mucho menor que $FLOP_{pal}$, y considerando el coste de las comunicaciones T_{comu}, la eficiencia del código paralelo será:

$$E = \frac{T_{sec}}{P \cdot T_{pal}} \quad \alpha \quad \frac{FLOP_{sec} + FLOP_{pal}}{P \cdot (FLOP_{max} + T_{comu})} \approx \frac{FLOP_{pal}}{P \cdot (FLOP_{max} + T_{comu}}) \leq B. \tag{5.2}$$

En nuestro algoritmo paralelo, se concretiza el estudio del balanceo al caso de una matriz triangular superior R de orden n. Supongamos, además, una distribución de la matriz en bloques triangulares y rectangulares de ancho $s = \alpha P$, y que dentro de cada bloque rectangular se realiza una distribución cíclica por columnas entre los P procesadores del sistema (ver sección 4.3).

Por lo tanto, asumiendo que $n > \alpha P$, el número de bloques triangulares en que queda dividida la matriz será $\lceil n/\alpha P \rceil$, donde $\lfloor n/\alpha P \rfloor$ bloques tendrán ancho s, mientras que el último será de ancho $r = n \,\%s$. Cada bloque triangular i ($i \geq 0$) irá precedido de i bloques rectangulares. De modo que si hay $\lfloor n/\alpha P \rfloor$ bloques triangulares de ancho s, tendremos $\frac{\lfloor n/\alpha P \rfloor (\lfloor n/\alpha P \rfloor - 1)}{2}$ bloques cuadrados de s^2 elementos, y $\lfloor n/\alpha P \rfloor$ bloques rectangulares de $r \cdot s$ elementos.

Como tanto para el algoritmo de las rotaciones planas como para la resolución triangular, el número de operaciones en punto flotante es proporcional al número de elementos de la matriz, el balanceo de la carga se puede caracterizar en términos del número de elementos de la matriz. Teniendo en cuenta esto, la carga paralela total $FLOP_{pal}$ será proporcional a:

$$FLOP_{pal} \propto \frac{\lfloor \frac{n}{\alpha P} \rfloor (\lfloor \frac{n}{\alpha P} \rfloor - 1)}{2} \cdot s^2 + \lfloor \frac{n}{\alpha P} \rfloor \cdot r \cdot s. \tag{5.3}$$

Simplificando, la ecuación 5.3 es equivalente a:

$$FLOP_{pal} \propto (\frac{n^2}{\alpha P} - n) \cdot \frac{\alpha P}{2} + n \cdot r. \tag{5.4}$$

En los bloques cuadrados de ancho s, la carga está perfectamente balanceada entre los P procesadores del sistema. El desbalanceo será debido a la carga del último bloque de ancho $n \,\%\alpha P$, en caso de que exista. El procesador con carga máxima tendrá siempre asignada una columna más que el resto debido a que el algoritmo usa una distribución cíclica por columnas. Así,

$$FLOP_{max} \propto \frac{\lfloor \frac{n}{\alpha P} \rfloor (\lfloor \frac{n}{\alpha P} \rfloor - 1)}{2P} \cdot s^2 + \lfloor \frac{n}{\alpha P} \rfloor \cdot \lceil \frac{r}{P} \rceil \cdot s. \tag{5.5}$$

De nuevo simplificando, y sustituyendo s por su valor,

$$FLOP_{max} \propto (\frac{n^2}{\alpha P} - n) \cdot \frac{\alpha}{2} + n \cdot \lceil \frac{r}{P} \rceil. \tag{5.6}$$

Sustituyendo el valor de $FLOP_{pal}$ y $FLOP_{max}$ en la ecuación 5.1,

$$B = \frac{(\frac{n^2}{\alpha P} - n) \cdot \frac{\alpha P}{2} + n \cdot r}{(\frac{n^2}{\alpha P} - n)\frac{\alpha P}{2} + n \cdot P \cdot \lceil \frac{r}{P} \rceil}. \tag{5.7}$$

De esta fórmula del balanceo se deduce que si $r = 0$ ó múltiplo de P ($\lceil r/P \rceil = r/P$), entonces $B = 1$, y la carga computacional de nuestro algoritmo está perfectamente balanceada. En cualquier otro caso, se producirá un cierto desbalanceo ($B < 1$). En este caso, el valor de B dependerá del orden de la matriz n y del número de procesadores del sistema.

Si el orden de la matriz es grande ($n \to \infty$), los términos de segundo orden en n se hacen dominantes en la ecuación 5.7, de modo que,

$$\lim_{n\to\infty} B = \frac{\frac{n^2}{\alpha P} \cdot \frac{\alpha P}{2} + O(n)}{\frac{n^2}{\alpha P} \cdot \frac{\alpha P}{2} + O(n)} = 1. \tag{5.8}$$

Por lo tanto, el balanceo del algoritmo mejora al aumentar el tamaño del problema, tendiendo al óptimo para problemas de gran tamaño, como muestra la figura 5.1. Para valores de n pequeños, $n \to 0$, nos encontraremos con el peor caso. Concretamente,

$$\lim_{n\to 0} B = \frac{(\frac{n}{\alpha P} - 1)\frac{\alpha P}{2} + r}{(\frac{n}{\alpha P} - 1)\frac{\alpha P}{2} + \lceil \frac{r}{P} \rceil} = \frac{\alpha P - 2r}{\alpha P - 2\lceil \frac{r}{P} \rceil}. \tag{5.9}$$

Por lo tanto, cuando $n \to 0$, el valor del balanceo dependerá del número de procesadores del sistema. Al aumentar P, disminuye $\lceil r/P \rceil$, incrementándose el valor del divisor en la ecuación 5.9, y por tanto, reduciéndose el valor del balanceo. Este efecto se observa en la figura 5.1. Cuando P aumenta se aprecia como para valores de n pequeños el balanceo es peor, y además el valor de n necesario para alcanzar el balanceo óptimo es mayor.

5.2. Pérdida de la situación de equilibrio inicial

El algoritmo del gradiente reducido que se utiliza junto con un método cuasi-Newton para la resolución de problemas no lineales con restricciones lineales (sección 1.6.1) clasifica las variables en tres grupos: básicas, superbásicas y no básicas [60]. Las variables no básicas se fijan al valor de una de sus fronteras, mientras las básicas y las superbásicas pueden variar dentro de ciertos límites. Esto da lugar a un continuo cambio de las variables que forman cada uno de estos grupos. Específicamente, si partiendo de una determinada definición de variables básicas, superbásicas y no básicas no se consigue ninguna mejora en la obtención de la solución óptima, alguna de las variables no básicas pasa a ser superbásica, incrementándose el orden de la matriz Hesiana y, por tanto, también de su aproximación, la matriz R. Esta variación del número de columnas de R no afecta al balanceo de la carga de nuestro algoritmo ya que las columnas que se añaden se asignan a los distintos procesadores siguiendo

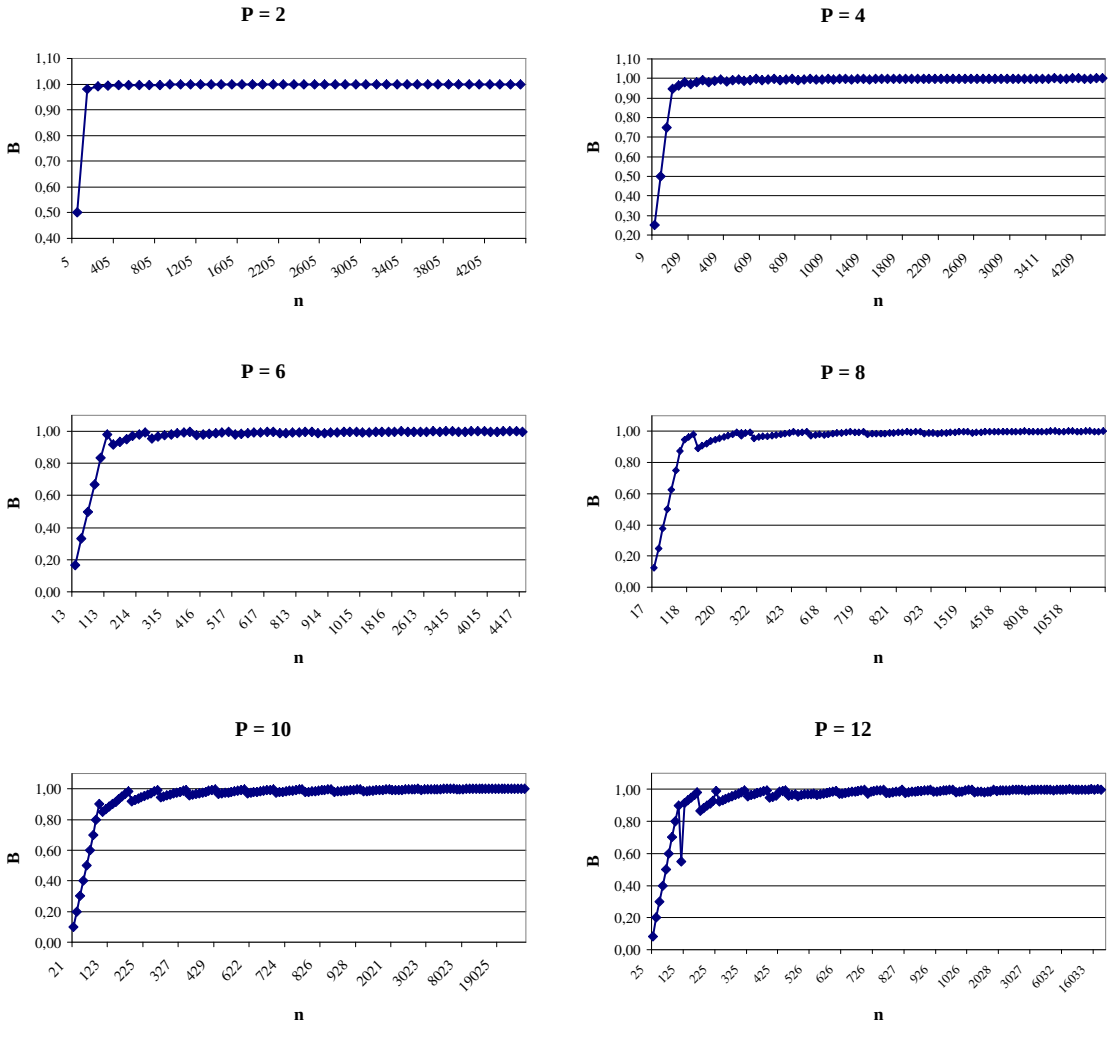


Figura 5.1: Balanceo de la carga en función del orden de la matriz.

una distribución cíclica. Sin embargo, cuando una variable superbásica alcanza una de sus fronteras debe ser eliminada, reduciéndose en ese caso el número de columnas de R. Esta situación lleva a la aparición de desbalanceos de la carga, siendo mayor cuantas más columnas hayan sido eliminadas de la matriz R.

De esta forma, cuando el desbalanceo en uno o más bloques rectangulares sea tan importante que provoque un efecto negativo suficientemente significativo en los tiempos de computación del algoritmo, será necesario realizar una redistribución de las columnas asociadas a dicho bloque rectangular entre los procesadores del sistema. La tarea, ahora, consistirá en determinar el coste, en tiempo, de este desbalanceo, y decidir cuando compensa la sobrecarga de comunicaciones necesarias para volver a una situación de equilibrio. Por otro lado, el uso de una heurística de redistribución óptima será fundamental. A continuación, se describen y evalúan las tres heurísticas de redistribución propuestas.

5.3. Estrategias de redistribución de la carga

El problema de la redistribución óptima de la carga computacional se puede definir formalmente del siguiente modo: sea un conjunto $D = \{1, \cdots, n_d\}$ de procesadores a los que denominaremos donadores, cada procesador i con un exceso de carga w_i ($i \in D$), medido en número de columnas, deben distribuir ese exceso de carga entre un conjunto $R = \{1, \cdots, n_r\}$ de procesadores receptores que poseen defecto de carga, y que tienen una capacidad de recepción de c_j columnas, $j \in R$. El valor de la sobrecarga de un procesador donador puede ser repartido entre distintos procesadores receptores, al contrario de lo que sucede en los típicos problemas de empaquetamiento de objetos (*Multiple Knapsack problems* [56]) donde puede quedar carga sin asignar debido a que su peso es fijo, y no puede ser asignado a ninguno de los receptores existentes. El problema puede formularse del siguiente modo:

$$\begin{array}{lllll} \text{minimizar} & \max\{\max \sum_{i\in D} x_{ij} & \forall j \in R, & \max \sum_{j\in R} x_{ij} & \forall i \in D\} \\ \text{sujeto a} & \sum_{i\in D} w_i(j) x_{ij} = c_j, & \forall j \in R, & & \\ & \sum_{j\in R} w_i(j) x_{ij} = w_i, & \forall i \in D, & & \\ & x_{ij} \in \{0,1\}, & i \in D, & j \in R, & \end{array} \tag{5.10}$$

donde x_{ij} es 1 si existe un envío entre el donador i y el receptor j, y por lo tanto, $\sum_{i\in D} x_{ij}$ es el número de mensajes recibidos por el procesador j, $\sum_{j\in R} x_{ij}$ es el número de mensajes enviados por el procesador i y $w_i(j)$ es el número de columnas del procesador i que ha sido asignado al procesador j.

El objetivo del problema es equilibrar el reparto de la carga entre los P procesadores del sistema. En la literatura, aparecen problemas de redistribución similares, las soluciones propuestas tratan de minimizar el número total de mensajes existentes en el sistema [42]. Sin embargo, nuestras heurísticas pretenden balancear el sistema minimizando el máximo número de mensajes recibidos y el máximo número de envíos por procesador, de este modo se minimizan los tiempos de espera de los procesadores y el tiempo de ejecución total del algoritmo, ya que el procesador que envíe o reciba el mayor número de mensajes marcará el tiempo crítico del algoritmo. Este valor está directamente relacionado con el tiempo necesario para restablecer el balanceo óptimo de la carga en el caso de que la red de interconexión permite el envío de mensajes simultáneos. Adicionalmente, reducir el número total de mensajes que circulan por la red evita la contención de la misma, lo que desembocaría en una degradación del rendimiento del algoritmo paralelo. Nótese que se prima la minimización del número de mensajes, no su tamaño, ya que la influencia del tamaño de los mensajes es menos relevante en los sistemas multiprocesador con suficiente ancho de banda en su red de interconexión.

Se trata de un problema NP-completo [13, 28, 36], por lo tanto, es imposible encontrar un algoritmo que nos garantice la obtención de la solución óptima en todos los casos y en un tiempo razonable. De ahí que distintas heurísticas como las de Primer ajuste, Último ajuste, Peor ajuste, Mejor ajuste, ..., hayan sido propuestas para resolver este tipo de problemas. En este trabajo se proponen tres estrategias que se adaptan mejor a las características del problema de optimización planteado en nuestro caso. A continuación, describiremos las estrategias propuestas y, para validarlas, las compararemos con las heurísticas clásicas de Peor ajuste y Mejor ajuste. También compararemos nuestros resultados con los de una heurística propuesta recientemente para resolver un problema de redistribución de la carga genérico en un sistema de computación paralelo. La heurística se conoce con el nombre de Pares dentro de un lazo (PDL) [42]. Puesto que el problema que aborda esta heurística es similar al que nosotros tratamos de resolver, la comparación de sus resultados con los obtenidos por nuestras propuestas resalta de manera precisa las características de las mismas.

5.3.1. Estrategia basada en donadores

Utilizando una terminología similar a la de los problemas de empaquetamiento de objetos, supongamos que las columnas en exceso de los procesadores donadores representan objetos a empaquetar. Consideremos que este número de columnas

representa el peso del objeto. Inicialmente estos pesos se encuentran ordenados de mayor a menor y almacenados en un vector W (vector de pesos). Los procesadores receptores representan contenedores que hay que llenar con los objetos, cuya capacidad, ordenada también en orden decreciente, se almacena en un vector C (vector de capacidades). Nótese que al ordenar los vectores W y C, el peso w_i no tiene por que corresponder al procesador i.

Para resolver este problema de optimización proponemos tres estrategias. La primera de ellas, denominada Estrategia basada en donadores (EBD), trata de minimizar el máximo número de mensajes enviados por procesador, intentando al mismo tiempo que no se incremente significativamente el máximo número de mensajes recibidos. En una primera fase, el algoritmo recorre los dos vectores identificando pesos y capacidades iguales. Un peso w_i del mismo valor que una capacidad c_j, se traduce en un único mensaje del donador k (con peso w_i) al receptor q (con capacidad c_j).

El siguiente paso consiste en realizar los emparejamientos entre los procesadores que aún tengan carga desbalanceada. La heurística recorre, para cada elemento del vector de pesos, el vector de capacidades buscando receptores a los que enviar la carga sobrante. Para ilustrar la heurística, consideremos el ejemplo de aplicación de este método de la figura 5.2 para un vector de pesos $W = \{34, 9, 7, 7\}$ y un vector de capacidades $C = \{26, 22, 8, 1\}$. La flecha que se encuentra en la parte superior del vector W indica que el algoritmo realiza los emparejamientos recorriendo este vector en primer lugar. Las otras flechas representan el establecimiento de mensajes entre dos procesadores.

Para cada elemento del vector de pesos, se distinguen dos casos:

1. El peso w_1 del procesador más cargado (i) es mayor que la capacidad de cualquier procesador receptor ($w_1 > c_j$, $\forall j \in R$). En este caso (paso (a) de la figura 5.2), el donador i envía c_1 de sus columnas al receptor j siendo este procesador el de mayor capacidad receptora. La carga sobrante, $w_1 - c_1$ columnas, se almacena de nuevo en el vector de pesos, siendo necesaria la correcta ubicación de la misma para mantener el orden decreciente del vector. Con esta estrategia, se trata de minimizar el número de envíos por procesador, ya que al eliminar la mayor carga sobrante posible en un sólo envío, previsiblemente el procesador donador necesitará enviar, a continuación, menos mensajes para librarse de la sobrecarga que todavía le queda.

2. El peso w_1 del procesador más cargado es menor que la mayor capacidad receptora ($w_1 < c_1$). En este caso, la estrategia consiste en buscar algún procesador q con capacidad receptora $c_j = w_1$. Si esta capacidad existe, se realiza

el emparejamiento, y será a este procesador al que se envíe la carga en exceso (paso (c) de la figura 5.2). De no existir ningún procesador que cumpla este requisito (pasos (b), (d) y (e) de la figura 5.2), la selección se hará entre todos aquellos procesadores cuya capacidad sobrante, después de realizarse la comunicación, verifique $c_j - w_1 > min_{k \in D}\{w_k\}$. De este modo, se trata de evitar que un remanente de capacidad receptora tenga un tamaño tan pequeño que sea insuficiente para que el procesador donador que posteriormente anule esa capacidad pueda eliminar toda su sobrecarga con un único envío. Tratando así de minimizar el número de envíos por procesador.

Dentro del grupo de procesadores que cumplen esta condición, se elegirá como receptor del mensaje a aquel que hasta el momento haya recibido el menor número de mensajes, con el fin de que el máximo número de recepciones por procesador no se incremente. En la figura 5.2, el número de recepciones y envíos en cada procesador se indica entre paréntesis. En caso de que la condición $c_j - w_1 > min_{k \in D}\{w_k\}$ no se cumpla, el mensaje se establecerá con el procesador receptor de mayor capacidad como se muestra en la figura 5.2 (d).

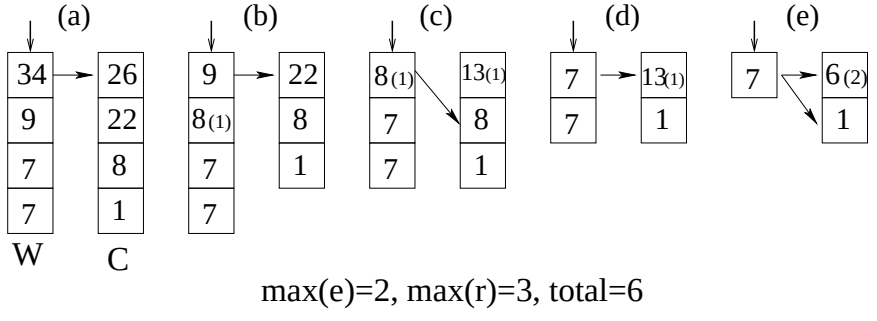


Figura 5.2: Ejemplo de aplicación de la heurística EBD para balancear un sistema de 8 procesadores.

5.3.2. Estrategia basada en receptores

Esta estrategia, denominada EBR, es similar a la descrita en el apartado anterior, la única diferencia es que ahora el objetivo es minimizar el máximo número de

recepciones por procesador, y tratar de mantener controlado, evitando que se incremente excesivamente, el máximo número de envíos. Para conseguirlo, proponemos aplicar la misma técnica pero, ahora, recorriendo primero el vector de capacidades en lugar del vector de pesos, tratando de encontrar para cada procesador receptor la carga adecuada que lo lleve a la situación de equilibrio. La descripción de este método, del que podemos ver un ejemplo en la figura 5.3, es la siguiente:

En una primera fase, el algoritmo recorre los dos vectores en busca de pesos y capacidades iguales. De encontrarse un peso w_j del mismo valor que una capacidad c_i, el procesador k (con peso w_i) enviará un mensaje con sus w_j columnas sobrantes al procesador q (con capacidad c_j).

De nuevo, el siguiente paso consiste en realizar los emparejamientos entre los procesadores del sistema que todavía no han sido balanceados, recorriendo, en primer lugar, el vector de capacidades y buscando para cada procesador receptor, un procesador donador que le proporcione la carga necesaria. En cada iteración del método, tendremos que distinguir dos casos:

1. La capacidad receptora c_1 del procesador más desbalanceado (i) es mayor que el peso asociado a cualquier procesador donador ($c_1 > w_j$, $\forall j \in D$). En este caso (pasos (b), (c) y (d) de la figura 5.3), el procesador j, con mayor peso, envía sus w_1 columnas al receptor i. La capacidad sobrante, $c_1 - w_1$ columnas, se almacena de nuevo en el vector de capacidades, siendo necesaria la reordenación del mismo. Con esta estrategia, se pretende minimizar el número de recepciones por procesador al igual que el número de envíos en la estrategia EBD.

2. La capacidad receptora c_1 es menor que el número de columnas sobrantes del procesador con mayor peso ($c_1 < w_1$). Primero, el algoritmo busca si existe algún procesador i con peso $w_j = c_1$ para que sea éste el que realice el envío. De no existir ningún procesador que cumpla este requisito (pasos (a) y (e) de la figura 5.3), la selección se hará entre todos aquellos procesadores que, después de establecerse la comunicación, verifiquen la condición $w_j - c_1 > min_{k \in R}\{c_k\}$.

   Dentro del grupo de procesadores que cumplen la condición 2, se elegirá como receptor del mensaje a aquel que hasta el momento haya realizado el menor número de envíos. Análogamente a lo que sucedía en la estrategia EBD, si no existe ningún procesador donador que verifique la condición 2, será el de mayor peso el que realice el envío.

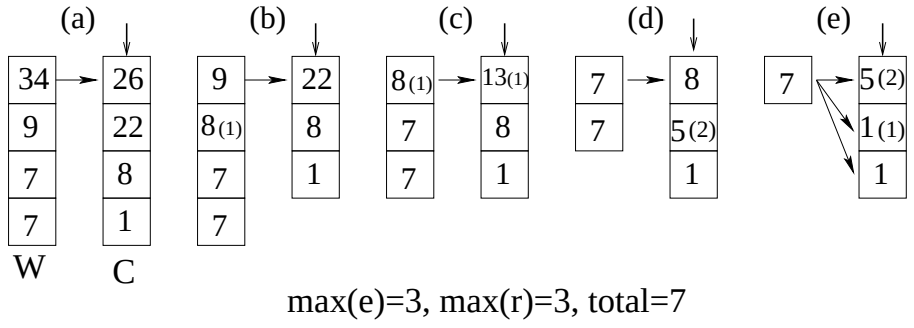


Figura 5.3: Ejemplo de aplicación de la heurística EBR para balancear un sistema de 8 procesadores.

5.3.3. Estrategia híbrida

La idea básica de la tercera estrategia propuesta (EH) consiste en combinar las características de las heurísticas EBD y EBR, sacando partido de las ventajas de cada una de ellas. Por lo tanto, esta heurística trata de minimizar por igual el máximo número de mensajes enviados y recibidos por procesador.

Partimos, como en los dos métodos anteriores, de un vector de pesos y un vector de capacidades ordenados decrecientemente. La primera etapa del algoritmo, consistirá de nuevo en la búsqueda de pesos y capacidades iguales, estableciéndose un único mensaje entre cada par de procesadores donador y receptor que verifiquen esta condición.

En la segunda etapa, se recorren los dos vectores (pesos y capacidades) simultáneamente, distinguiéndose para cada elemento dos casos:

1. Si el mayor peso w_1 es menor o igual que la máxima capacidad receptora c_1, entonces se aplica el método EBD (pasos (b), (c) y (d) de la figura 5.4). Se busca alguna capacidad c_k igual al peso w_1 (paso (c) de la figura 5.4). Si esta capacidad existe, el donador i (con peso w_1) enviará un mensaje al receptor j (con capacidad c_k), consiguiendo ambos procesadores balancear su carga. De no encontrarse esta capacidad, se elige, de entre todos los procesadores receptores que verifiquen que $c_k - w_1 > min_{v \in D}\{w_v\}$, el que haya recibido hasta el momento el menor número de mensajes. En este caso, se prima la minimización del máximo número de envíos por procesador.

2. Si el mayor peso w_1 es mayor que la máxima capacidad receptora c_1, entonces

se aplica el método EBR (pasos (a) y (e) de la figura 5.4). Se intenta encontrar algún donador k con peso w_i igual a la capacidad c_1. De existir ese procesador será necesario un único mensaje entre esos dos procesadores para balancearlos. Si no existe, se elige al procesador donador j (con peso w_i), que verificando $w_i - c_1 > min_{v \in R}\{c_v\}$, haya realizado el mínimo número de envíos. En esta situación, se da mayor importancia a la minimización del máximo número de recepciones por procesador.

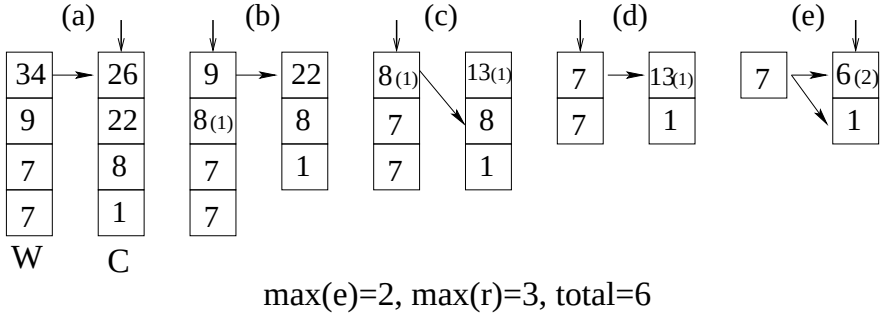


Figura 5.4: Ejemplo de aplicación de la heurística EH para balancear un sistema de 8 procesadores.

5.3.4. Estrategia Peor ajuste

La heurística Peor ajuste WF (*Worst Fit*) [17] es uno de los métodos más intuitivos de los utilizados en la optimización de problemas de empaquetamiento de objetos. En el planteamiento de este tipo de problemas se dispone de una lista de N objetos de diferentes pesos w_i que se deben empaquetar en M contenedores de igual capacidad c. La heurística consiste simplemente en tomar el primer elemento no asignado de la lista de objetos y colocarlo en el recipiente que disponga de la mayor cantidad de espacio vacío una vez que el objeto haya sido depositado en él.

Nuestro problema es ligeramente diferente, ya que en algunos casos, el peso asociado a un procesador donador es mayor que la capacidad de cualquier procesador receptor, y además los contenedores no tienen necesariamente la misma capacidad. Ante estas situaciones no podemos aplicar la técnica del método Peor ajuste pura, y

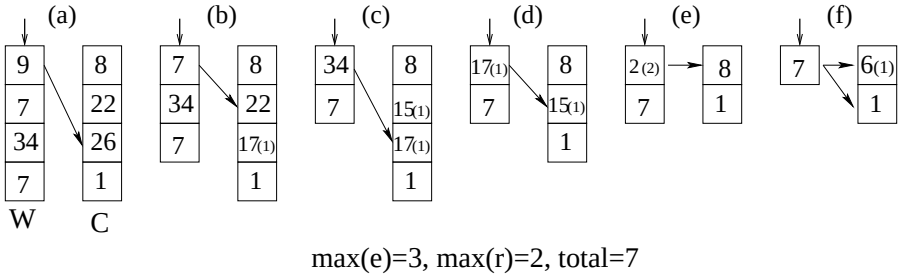


Figura 5.5: Ejemplo de aplicación de la heurística Peor ajuste para balancear un sistema de 8 procesadores.

optamos por adaptarla. Podemos ver un ejemplo de este método en la figura 5.5, en la que se observa que en este caso no es necesario el reordenamiento de los vectores de pesos y capacidades. Por lo tanto, es válida la nomenclatura en la que al procesador i le corresponde una carga en exceso w_i. Para los procesadores receptores se puede usar una notación análoga.

Por lo tanto, para el problema de optimización que nos ocupa, la implementación del algoritmo Peor ajuste se adaptaría de la siguiente forma:

- Si el peso w_i del procesador donador i es mayor que cualquier capacidad receptora ($w_i > c_j$, $\forall j \in R$), entonces el procesador i envía c_j de sus columnas sobrantes al receptor j de mayor capacidad (pasos (c), (d) y (f) de la figura 5.5).

- En caso contrario, se aplica la heurística Peor ajuste, y el donador i enviará sus w_i columnas al receptor j en el que quede más hueco una vez se haya recibido el mensaje ($c_j - w_i = max\{c_k - w_i\}$, $\forall k \in R$) (pasos (a), (b) y (e) de la figura 5.5).

Claramente esta heurística no garantiza la minimización del máximo número de recepciones por procesador.

5.3.5. Estrategia Mejor ajuste

Esta heurística, al igual que ocurría con la de Peor ajuste, sólo podrá ser aplicada a nuestro problema de optimización cuando el peso a asignar w_i sea menor que alguna capacidad receptora. Por lo tanto, el algoritmo Mejor ajuste BF (*Best Fit*) [48] adaptado quedaría formulado del siguiente modo:

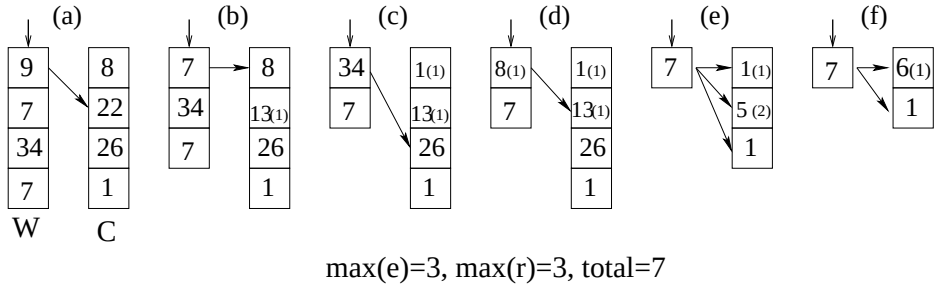


Figura 5.6: Ejemplo de aplicación de la heurística Mejor ajuste para balancear un sistema de 8 procesadores.

- Si el peso w_i del procesador donador i es mayor que cualquier capacidad receptora ($w_i > c_j$, $\forall j \in R$), entonces el procesador i envía c_j de sus columnas sobrantes al receptor j de mayor capacidad (pasos (c), (e) y (f) de la figura 5.6).

- En caso contrario, se aplica la heurística Mejor ajuste, y el donador i enviará sus w_i columnas al procesador j en el que la capacidad sobrante sea menor después de establecerse la comunicación ($c_j - w_i = min\{c_k - w_i\}$, $\forall k \in R$) (pasos (a), (b) y (e) de la figura 5.6).

Nótese que al realizar los envíos primando remanentes de capacidades pequeñas, en la mayor parte de los casos se garantiza un mínimo número de recepciones. Sin embargo, con frecuencia algún procesador donador para liberarse de su exceso de carga, y debido a las pequeñas capacidades receptoras existentes en un determinado momento, se ve forzado a realizar muchos envíos.

5.3.6. Estrategia Pares dentro de un Lazo

Esta estrategia, que denominamos PDL [42], tiene como objetivo minimizar el número total de mensajes necesarios para redistribuir la carga computacional. Se trata de un algoritmo avaro que recorre los vectores de pesos (W) y capacidades (C), ordenados en orden decreciente. La primera etapa del algoritmo es análoga a nuestras propuestas, se recorren ambos vectores buscando pesos y capacidades iguales. En la segunda etapa se establece un lazo que se ejecuta mientras existan procesadores con exceso de carga. En cada iteración de este lazo, se distinguen dos casos:

1. El peso w_i del procesador más cargado k es mayor que la mayor capacidad receptora, c_j, entonces el procesador k envía $w_i - c_j$ columnas al procesador q (con capacidad c_j).

2. El desbalanceo w_i del procesador k con mayor carga en exceso es menor que el que presenta el procesador q con menor carga asignada, de capacidad c_j. En este caso, el procesador k enviará w_i columnas al procesador q.

Al final de cada iteración, el algoritmo busca si el remanente de carga o capacidad resultante de este envío se puede emparejar con la ya existente en algún procesador, estableciéndose el equilibrio entre ambos procesadores como consecuencia de un único mensaje entre ellos.

Como se puede observar, el algoritmo es muy similar a nuestra propuesta EBD cuando $w_i > c_j$. En otro caso, se establece una comunicación entre los dos procesadores más desbalanceados sin considerar el número de mensajes que ya ha recibido el receptor. A pesar de que esta heurística no trata de minimizar el máximo número de recepciones, al haber sido propuesta para resolver un problema ligeramente diferente al nuestro, consigue resultados más próximos a nuestras propuestas que las heurísticas de Mejor ajuste y Peor ajuste como concluiremos en la siguiente sección.

5.4. Análisis del rendimiento de las estrategias de balanceo de la carga

Las heurísticas EBD, EBR y EH han sido diseñadas con carácter general para poder adaptarlas a problemas de desbalanceo en la ejecución de cualquier aplicación sobre un sistema paralelo homogéneo donde el tiempo crítico esté determinado por el coste de las comunicaciones. Sin embargo, probaremos su eficiencia para redistribuir la carga computacional cuando se produzca una situación de desbalanceo en nuestro algoritmo paralelo. Se ha realizado un estudio comparativo entre todas las heurísticas descritas en la sección anterior. El objetivo es resaltar la eficiencia de las estrategias propuestas así como decidir qué heurística se va a utilizar finalmente en nuestra implementación paralela.

El estudio se ha realizado sobre un sistema genérico de P procesadores, y considerando la matriz R dividida en bloques rectangulares y triangulares de distintos tamaños. Debido a que en el algoritmo paralelo existe una sincronización después de la computación de cada bloque rectangular, el tiempo total de ejecución del algoritmo está fuertemente ligado a las computaciones de los bloques rectangulares.

Además, como la computación paralela se reduce a estos bloques, el desbalanceo de la carga computacional de un bloque rectangular determina el tiempo de computación asociado a ese bloque. Por lo tanto, centraremos el estudio sobre un único bloque rectangular de entre todos en los que quedaría dividida la matriz, pero las conclusiones que se extraigan se podrán aplicar a todos los demás bloques de manera inmediata.

Consideremos sistemas con distinto número de procesadores $P = \{4, 8, 16, 32, 64, 128\}$, y diferentes tamaños de bloque, desde 4 hasta 3000 columnas por procesador. Seleccionamos el número de columnas que pueden ser eliminadas en cada procesador, de modo que controlemos el desbalanceo de la carga computacional. Cuanto mayor sea la cantidad de columnas eliminadas, mayor podrá ser el desbalanceo existente. Este control sobre el desbalanceo se ejerce permitiendo la eliminación como máximo de un porcentaje del número total de columnas asignadas a un procesador (en un rango del 10 % al 90 %). Así, por ejemplo, en un sistema con $P = 64$ procesadores y $n = 2000$ columnas por procesador, si establecemos un desbalanceo del 20 %, para cada procesador el número de columnas eliminadas está comprendido dentro del intervalo $[0, 400]$.

Definido este intervalo, el número de columnas eliminadas se genera aleatoriamente para cada procesador. Una vez establecida la carga por procesador en la situación de balanceo y la que realmente tiene cada procesador, se conoce el número de columnas que tiene asignadas en exceso o las que le faltan para estar balanceado. De este modo, hemos generado 1740 ejemplos sobre los que se han aplicado las seis heurísticas de redistribución descritas en la sección 5.3.

Para visualizar los resultados hemos representado para cada valor de P y del desbalanceo permitido, gráficas que representan el número máximo de envíos por procesador, el número máximo de recepciones y el número total de mensajes frente al número de columnas asignadas inicialmente a cada procesador. Estos resultados pueden verse en las figuras 5.7 a 5.12 para todos los posibles valores de P y para desbalanceos del 10 % y 90 %. Los resultados de la heurística PDL no se muestran en ellas debido a que su similitud con los obtenidos mediante nuestras propuestas dificulta la visualización de estas gráficas. Posteriormente, mostraremos como, siendo los resultados entre nuestras heurísticas y la heurística PDL muy similares, las primeras alcanzan mejores valores de la función objetivo.

En los resultados se observa que para un sistema de 4 procesadores, las tres medidas analizadas toman valores idénticos para todas las heurísticas. Esto es debido a que con un número de procesadores tan pequeño, se pueden dar sólo tres situaciones de desbalanceo, y para cada una de ellas existe una única alternativa para redistribuir

la carga computacional y llevar de nuevo el sistema al equilibrio. De ahí que todas las heurísticas presenten soluciones idénticas. Las tres situaciones posibles son:

- Máximo número de recepciones o envíos por procesador igual a 1. Esto ocurre si en el sistema existe un único procesador con exceso de carga y un único procesador con capacidad receptora; o bien, dos procesadores donadores y dos receptores, siendo el peso de cada donador igual a una de las capacidades receptoras.

- Máximo número de recepciones o envíos por procesador igual a 2. Si existen dos procesadores donadores y un procesador receptor, o un donador y dos receptores.

- Máximo número de recepciones o envíos por procesador igual a 3. Se distingue un procesador donador y tres procesadores receptores, o la situación inversa, tres donadores y un receptor.

Como cabía esperar, a medida que aumenta el número de procesadores del sistema, se observan comportamientos diferentes entre las distintas heurísticas, sobre todo, en las gráficas del número máximo de envíos por procesador (figuras 5.7 y 5.10) y del número total de mensajes(figuras 5.9 y 5.12). Estas diferencias son especialmente relevantes cuando el sistema tiene el mayor número de procesadores, $P = 64$ ó $P = 128$. En estos casos, se observan claramente los aspectos comentados en la sección 5.3. Globalmente, los peores resultados en el número máximo de envíos por procesador se obtienen con la heurística Mejor ajuste, ya que se caracteriza por realizar los envíos a aquellos procesadores de menor capacidad de recepción, lo que conduce a que el procesador que se libere de su carga en exceso en último lugar, debe repartir esa carga entre un número alto de procesadores receptores, lo que implica un gran número de envíos.

En las figuras 5.7 y 5.10 se observa la eficiencia de las heurísticas propuestas que reducen el número máximo de envíos respecto a las estrategias Peor ajuste y Mejor ajuste de forma significativa a medida que aumenta el número de procesadores. En la figuras 5.8 y 5.11 puede verse que las cinco heurísticas consiguen mantener valores reducidos en el número máximo de recepciones por procesador. En este caso, la heurística Mejor ajuste presenta los mejores resultados, aunque difiere muy poco de los obtenidos por nuestras propuestas EBR y EH. Nótese que EBR se comporta correctamente en número de recepciones, y EBD en envíos, mientras que EH presenta buenos comportamientos en ambos casos.

Las resultados del número total de mensajes se muestran en las figuras 5.9 y 5.12. En ellas puede observarse un comportamiento similar. Todas las heurísticas obtienen

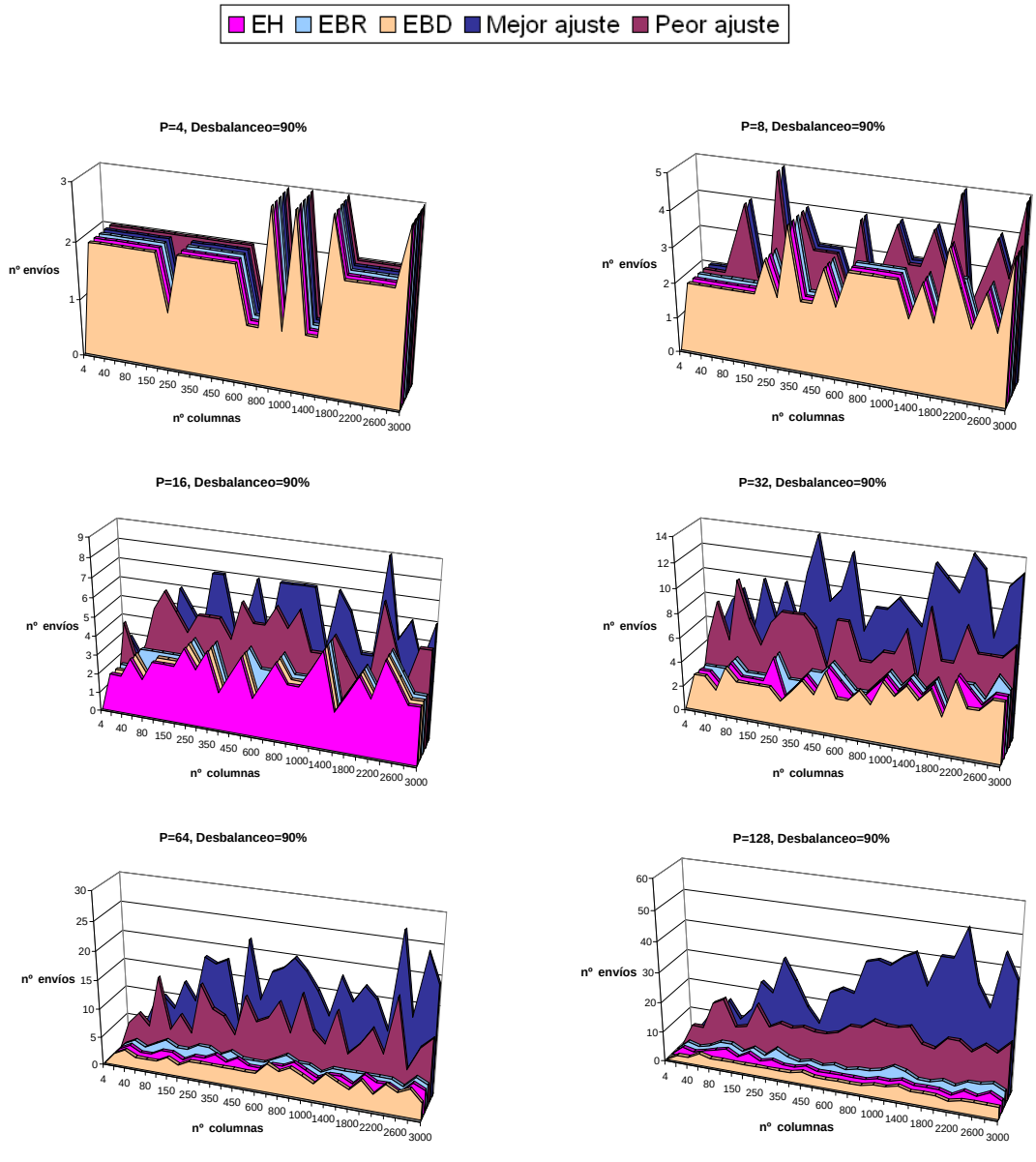


Figura 5.7: Número máximo de envíos por procesador necesarios tras la aplicación de los distintos algoritmos de redistribución para un desbalanceo del 90 %.

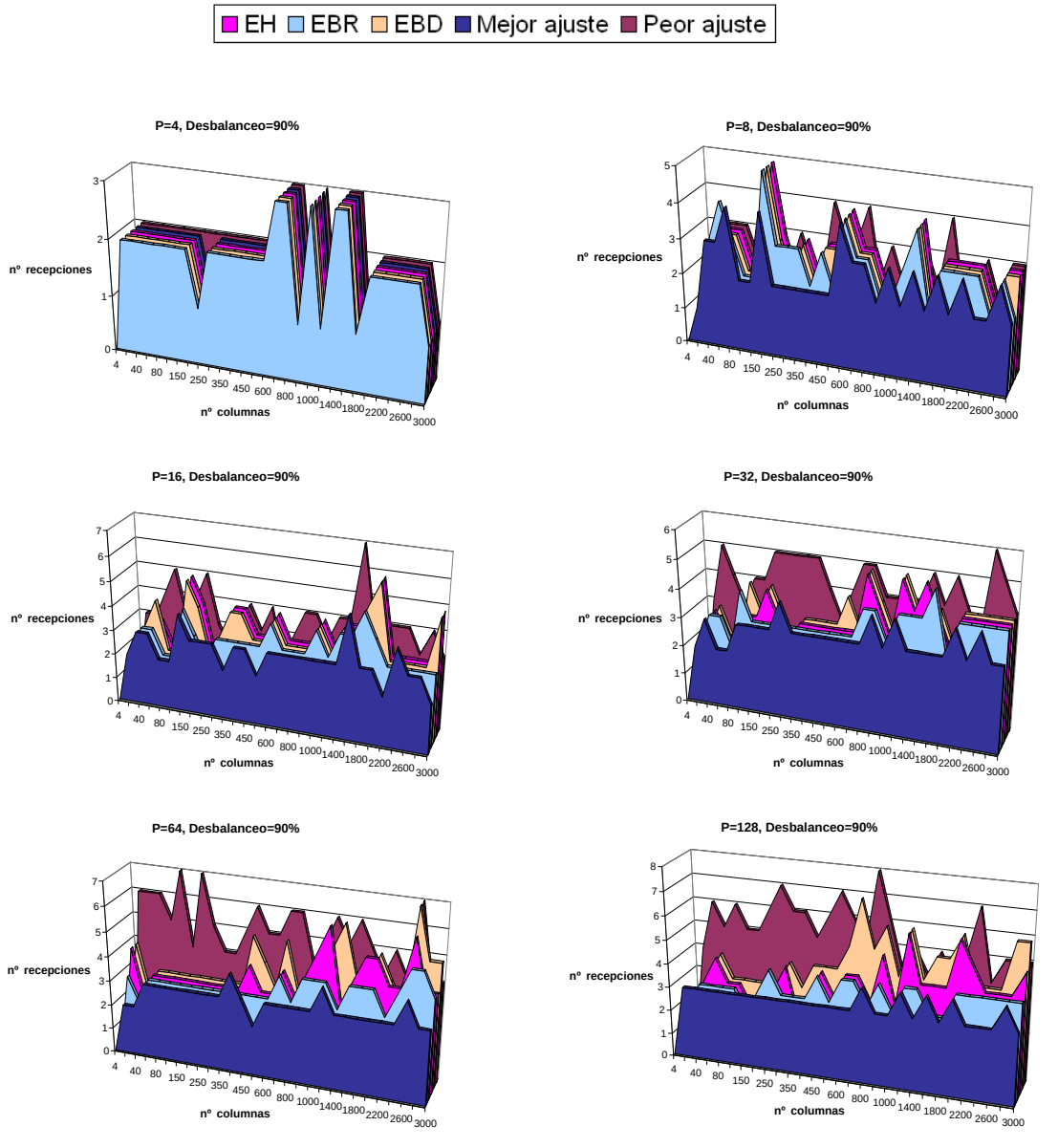


Figura 5.8: Número máximo de recepciones por procesador que genera la aplicación de los distintos algoritmos de redistribución para un desbalanceo del 90 %.

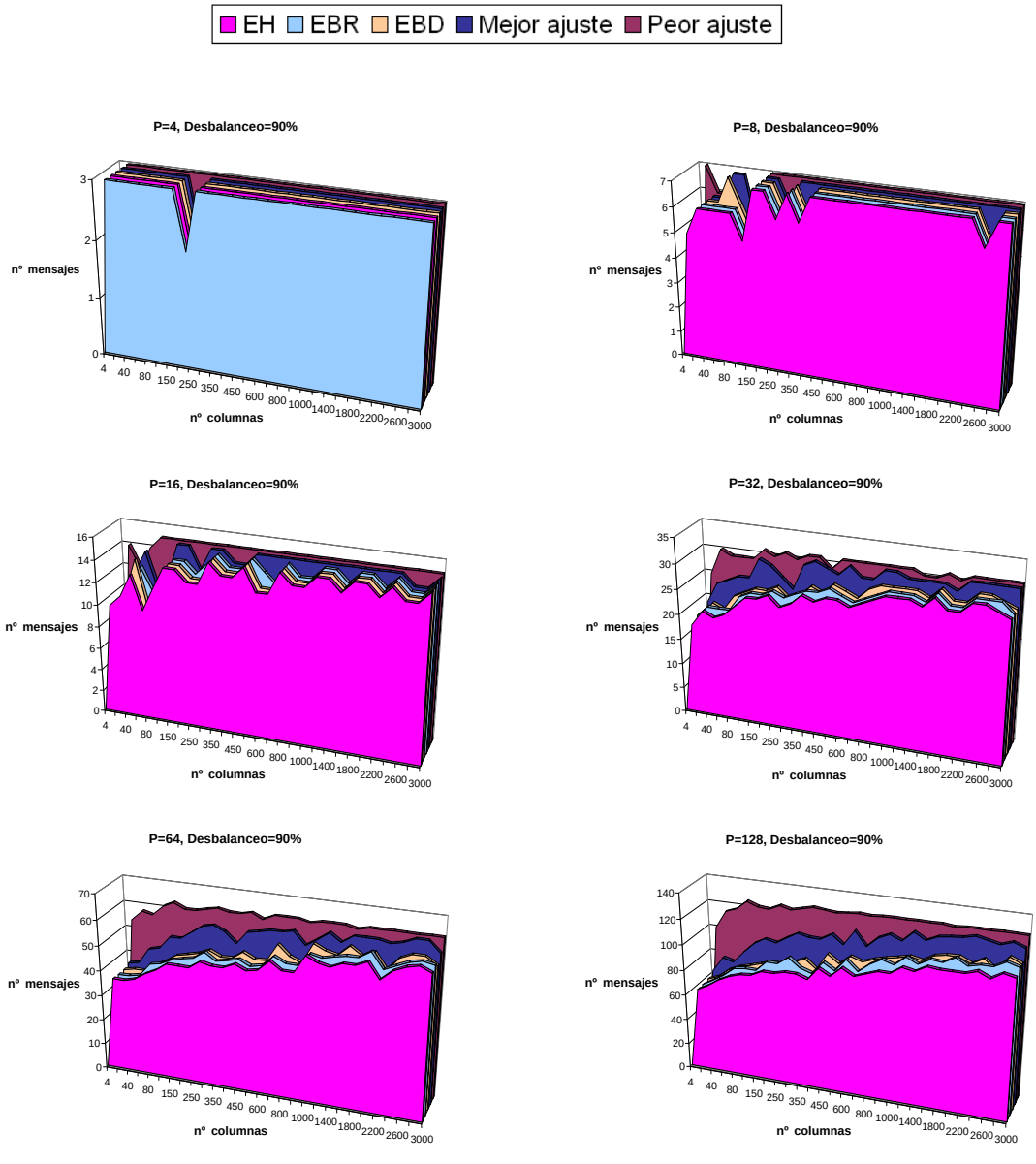


Figura 5.9: Número total de mensajes generados en el sistema tras la aplicación de los distintos algoritmos de redistribución para un desbalanceo del 90 %.

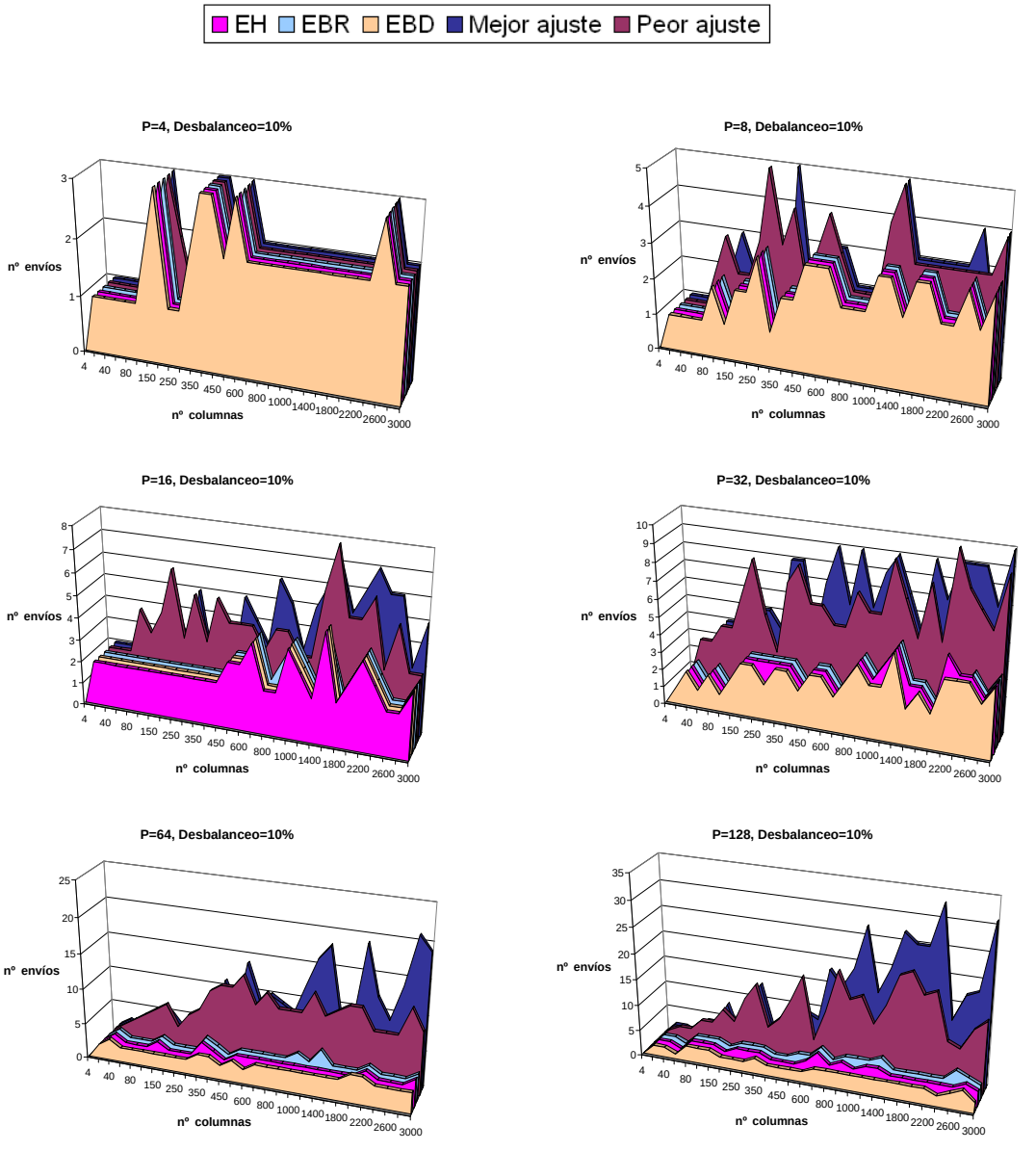


Figura 5.10: Número máximo de envíos por procesador necesarios tras la aplicación de los distintos algoritmos de redistribución para un desbalanceo del 10 %.

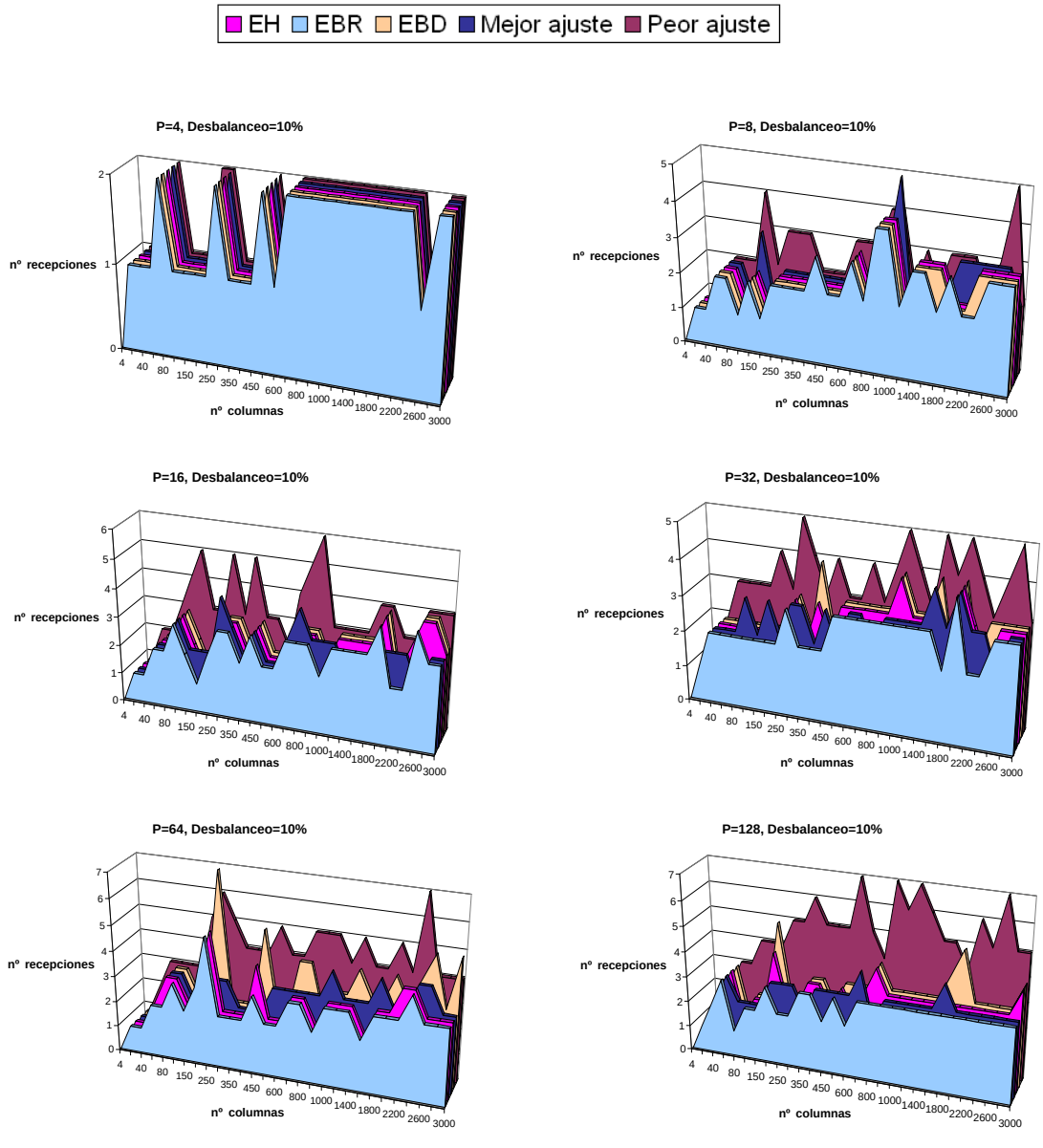


Figura 5.11: Número máximo de recepciones por procesador generado tras la aplicación de los distintos algoritmos de redistribución para un desbalanceo del 10 %.

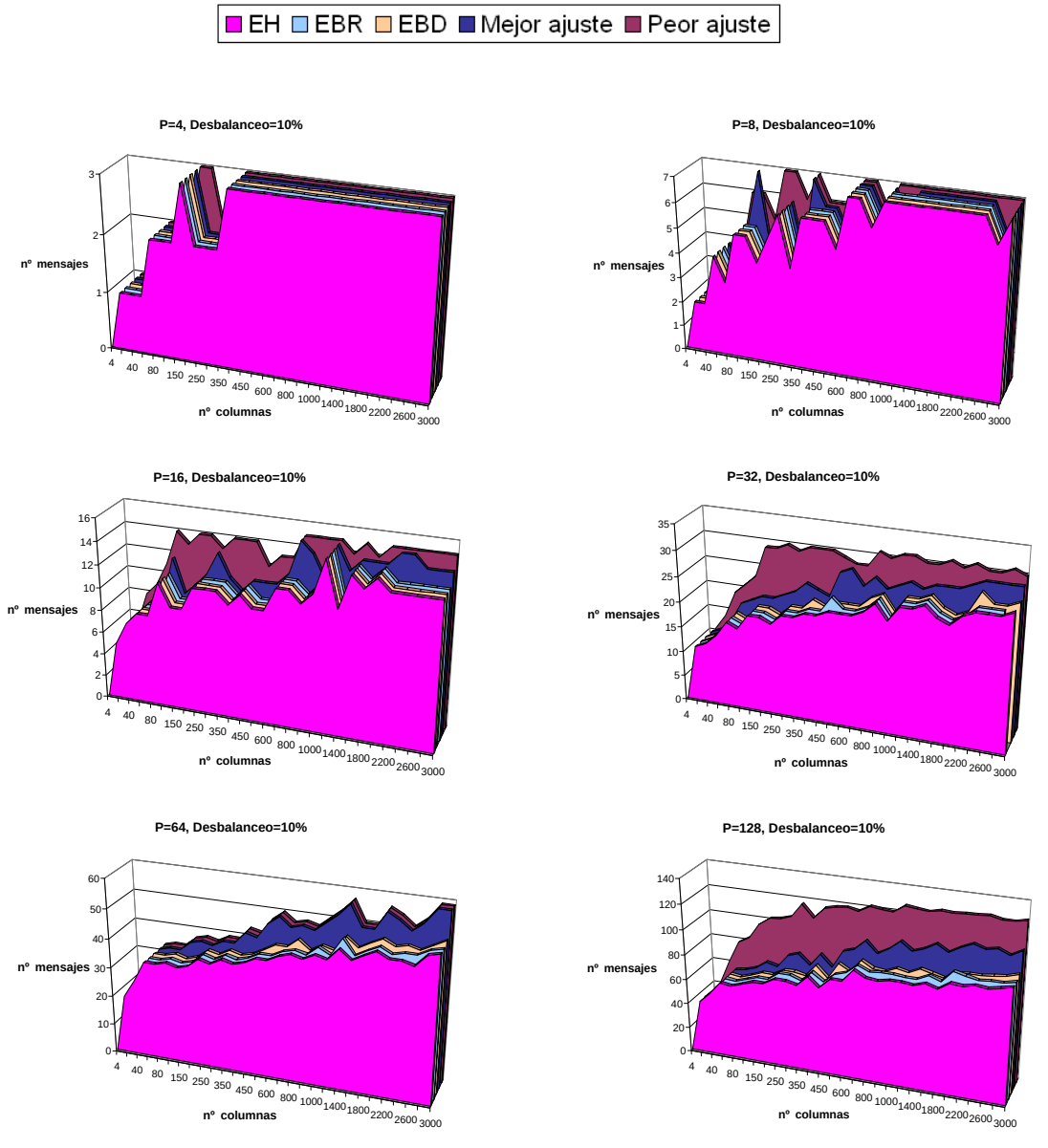


Figura 5.12: Número total de mensajes generados en el sistema tras la aplicación de los distintos algoritmos de redistribución para un desbalanceo del 10 %.

resultados muy próximos para sistemas con un número de procesadores pequeño, diferenciándose más conforme el número de procesadores aumenta. En este caso, aunque los mejores resultados se obtienen con la heurística EH, las diferencias con los otros métodos propuestos son mínimas. Los peores resultados se obtienen ahora con la heurística Peor ajuste, sobre todo, para $P = 128$ procesadores.

BF					WF				
P	EBD	EBR	EH	PDL	P	EBD	EBR	EH	PDL
4	0.00	0.00	0.00	0.00	4	1.66	1.66	1.66	1.66
8	12.61	12.61	12.85	12.73	8	11.00	11.00	11.25	11.12
16	35.92	35.12	35.77	35.41	16	29.71	28.83	29.55	29.15
32	62.23	60.38	61.08	61.12	32	50.94	48.53	49.44	49.49
64	77.30	74.23	76.15	75.71	64	66.36	61.81	64.66	64.01
128	86.61	83.20	85.28	84.96	128	76.33	70.30	73.98	73.41

Tabla 5.1: Tasa de mejora (en %) del número máximo de envíos de las heurísticas descritas frente a los casos clásicos de Mejor ajuste y Peor ajuste.

En estas gráficas se observa que es en el máximo número de envíos y en el número total de mensajes donde se producen las mayores diferencias entre nuestras propuestas y las heurísticas clásicas. La comparación con la heurística PDL no presenta conclusiones reseñables, sin embargo, se observan resultados ligeramente superiores para nuestras propuestas. En la tabla 5.1 se muestra la tasa de decrecimiento del número máximo de envíos en nuestros métodos y en la heurística PDL frente al generado por los métodos Mejor ajuste y Peor ajuste. Para calcular este porcentaje de mejora se ha trabajado con el máximo número de envíos medio que resulta de aplicar cada estrategia de redistribución a todos lo tamaños de bloque y todos los desbalanceos de nuestra batería de pruebas sobre un sistema de P procesadores.

Para completar el estudio del comportamiento de nuestras propuestas frente a las otras alternativas descritas, en la tabla 5.2 se muestra el número de veces que cada heurística obtiene la mejor solución para el problema de optimización 5.10 sobre el conjunto de pruebas utilizado. En muchos casos, la solución óptima es alcanzada por más de una heurística al mismo tiempo. Sin embargo, en algunas ocasiones, sólo una de las estrategias obtiene el mejor resultado. El número de veces que esto ocurre se muestra entre paréntesis. A medida que el número de procesadores aumenta, nuestras estrategias presentan las mejores eficiencias en un mayor número de ocasiones. La heurística que más veces alcanza el valor mínimo de la función objetivo aparece

P	EBD	EBR	EH	PDL	BF	WF
4	290	290	290	290	290	281
8	230	226(6)	**232**	231	193(10)	181(15)
16	239(5)	**247(14)**	242	238	84(6)	81(9)
32	230(19)	**234(31)**	227	226	27(2)	26(3)
64	202(27)	**214(32)**	211(4)	190(1)	28(4)	16
128	179(40)	179(32)	**202(10)**	173(5)	24(3)	15

Tabla 5.2: Número de veces, sobre un conjunto de 1740 ejemplos, que cada heurística de redistribución obtiene el mejor resultado.

resaltada en negrita.

Todo este análisis de la eficiencia de las distintas heurísticas se ha realizado a partir de los resultados obtenidos para desbalanceos del 10 % al 90 %. Hemos comprobado que para todos los desbalanceos las conclusiones son idénticas, y sólo mostramos los resultados de dos casos extremos. Sin embargo, cabe destacar el comportamiento observado en las gráficas del número máximo de envíos por procesador. Mientras que para nuestras propuestas se observa un número de envíos muy pequeño y casi constante, independientemente del desbalanceo, a medida que aumenta el número de procesadores, para el Mejor ajuste y el Peor ajuste, el número máximo de envíos crece considerablemente a medida que lo hace el desbalanceo y el número de procesadores del sistema. Este comportamiento puede apreciarse en la figura 5.13 para un sistema de $P = 128$ procesadores.

Una vez constatada la eficiencia de nuestras heurísticas, tendremos que seleccionar una de ellas como la más adecuada para redistribuir la carga computacional en nuestro código de optimización paralelo.

En un sistema formado por P procesadores, el desbalanceo es función de la carga computacional por procesador y del número de procesadores, caracterizados como donadores D o receptores R. La diferencia $D - R$ toma un amplio rango de valores a medida que se incrementa P. Si $D - R \gg 0$, habrá un gran número de procesadores donadores que tienen que redistribuir su exceso de carga entre los procesadores receptores. En este caso, el máximo número de recepciones por procesador es el factor dominante. Por lo tanto, la estrategia EBR, en este caso, proporciona la mejor solución. Esta situación se puede observar en el lado derecho de la figura 5.14 para un sistema de 128 procesadores. En el lado izquierdo de esta misma figura se

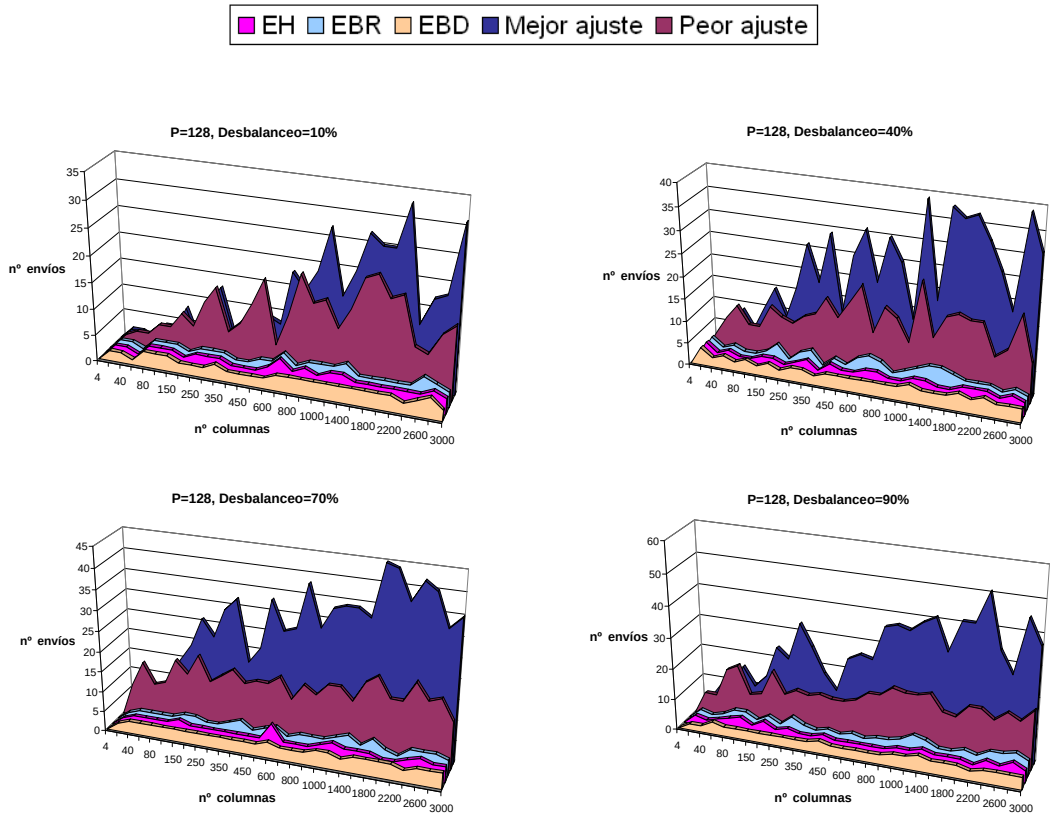


Figura 5.13: Número máximo de envíos por procesador tras la aplicación de las distintas heurísticas para distintos grados de desbalanceo y P=128.

observa la situación opuesta, el número de donadores es muy pequeño, y el máximo número de envíos por procesador se hace crítico. En este caso, la solución óptima es la aplicación de la heurística EBD. Sin embargo, para valores $|D - R|$ próximos a cero, las tres estrategias presentan un comportamiento similar, como se ha visto en las figuras 5.7 a 5.12. En esta región, la heurística EH, que combina las características de las estrategias EBD y EBR, parece la mejor opción. En muchos problemas reales, el desbalanceo típico de un código de optimización basado en un conjunto activo de restricciones no es alto, y la diferencia entre el número de donadores y receptores no puede ser excesiva, por lo que el máximo número de envíos y el máximo número de recepciones tiende a ser similar. Como consecuencia, la heurística EH es la que hemos seleccionado como el método a utilizar para redistribuir la carga computacional del sistema, y en ella centraremos el estudio del resto del capítulo. En cualquier caso,

la aplicación de cualquiera de las otras dos heurísticas nos llevaría a resultados y conclusiones análogas.

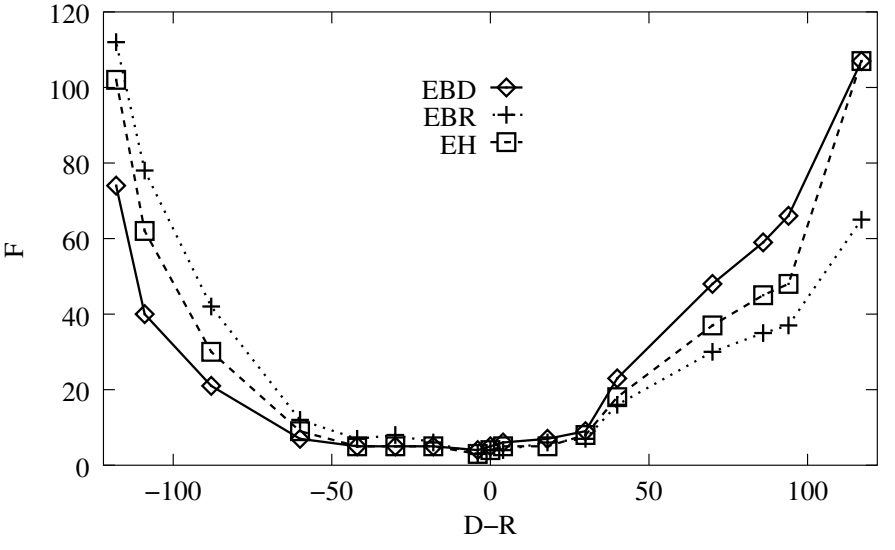


Figura 5.14: Función objetivo del problema de redistribución de la carga aplicando las tres heurísticas propuestas (EBD, EBR y EH) para desbalanceos con diferente relación entre el número de donadores y receptores.

5.5. Estimación del coste de la heurística EH

Un sistema que no se encuentra completamente balanceado implica un mayor coste computacional. Es necesario decidir si este incremento en el tiempo de computación es lo suficientemente importante como para que compense realizar una redistribución de la carga.

En primer lugar, habrá que determinar cuánto se aparta el sistema de la situación de equilibrio [74]. El procesador con mayor carga, en nuestro caso con el mayor número de columnas excediendo el equilibrio, es el que marca el tiempo de computación. El coste del desbalanceo se puede definir como el tiempo que este procesador necesita para realizar todas las computaciones existentes sobre los elementos de esas columnas en exceso. En cada iteración del método cuasi-Newton se ejecuta una serie

de procesos sobre la matriz R. Algunos se aplican bajo determinadas condiciones pero hay cuatro que se ejecutan siempre y que además suponen la mayor parte del coste computacional: la resolución triangular inferior, la resolución triangular superior, la rotación global hacia adelante y la rotación global hacia atrás. Por lo tanto, para evaluar el posible coste del desbalanceo existente vamos a calcular cuál es el tiempo computacional necesario para ejecutar las cuatro computaciones mencionadas sobre una porción de columna de tamaño αP perteneciente a uno de los bloques rectangulares, U_{ij}, en que se divide la matriz. Estos bloques se muestran en la figura 5.15.

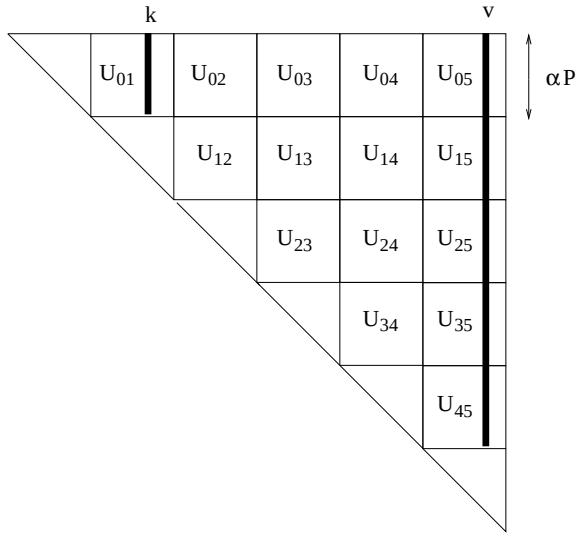


Figura 5.15: Particionamiento en bloques de la matriz R.

Hemos estudiado el coste computacional para $P = 1$ y distintos valores de α de las cuatro computaciones. A continuación, realizamos un ajuste lineal de los valores de tiempo de ejecución obtenidos para cada computación. Este proceso hay que realizarlo sobre cada sistema en el que se quiera estimar el coste de la heurística de redistribución EH. A modo ilustrativo, en la siguiente sección se presentan resultados experimentales obtenidos sobre el sistema multiprocesador AP3000 de Fujitsu y el *cluster beowulf* del CESGA.

Formalmente, sustituyendo el coste computacional de cada una de las cuatro rutinas mencionadas por la correspondiente ecuación de ajuste, el coste computacional

asociado al desbalanceo para una única columna en exceso de tamaño αP se puede ajustar linealmente de un modo suficientemente preciso,

$$\text{coste}(\alpha P) = a_{desba} + b_{desba} \cdot \alpha P. \tag{5.11}$$

Cada columna puede pertenecer a uno o más bloques, de modo que cuanto mayor sea el índice de la columna, mayor es el número de bloques a los que pertenece. Por lo tanto, el desbalanceo del sistema será mayor si se eliminan columnas pertenecientes a los últimos bloques (por ejemplo, la columna v de la figura 5.15) que si desaparecen columnas del principio, como la columna k de la misma figura. Podemos hablar, por tanto, de varios niveles dentro de la matriz, así el nivel $\Omega = 1$ estará compuesto por las columnas que pertenecen a un sólo bloque rectangular, el nivel $\Omega = 2$ por las que pertenecen a dos bloques y así sucesivamente.

Una vez que se conoce el coste computacional de las cuatro rutinas mencionadas sobre una porción de columna de tamaño αP, para medir el desbalanceo dentro de un conjunto de bloques U_{ij}, $i < j$, perteneciente a un determinado nivel Ω, basta con multiplicar el coste dado por la ecuación 5.11 por el número de columnas en exceso que posee el procesador con más carga asignada dentro de ese nivel. Ese resultado, además, habrá que multiplicarlo por el nivel considerado.

Si denotamos por C_{max} el máximo número de columnas que exceden el equilibrio dentro de un bloque rectangular de altura αP, podemos definir el coste computacional debido al desbalanceo de ese bloque como:

$$\text{coste(bloque)} = C_{max} \cdot \text{coste}(\alpha P) = C_{max} \cdot (a_{desba} + b_{desba} \cdot \alpha P). \tag{5.12}$$

El desbalanceo existente no sólo afectará a ese bloque sino a todos los que están alineados con él, por lo tanto, el coste total asociado a este desbalanceo será,

$$\text{coste(desba)} = \Omega \cdot C_{max} \cdot \text{coste}(\alpha P) = w \cdot \Omega \cdot C_{max} \cdot (a_{desba} + b_{desba} \cdot \alpha P). \tag{5.13}$$

El siguiente paso consiste en calcular el coste de la estrategia de redistribución, para decidir si compensa recomponer la carga computacional. Para caracterizar el coste computacional de la redistribución debemos distinguir dos componentes:

- $T1$, es el tiempo de computación de la heurística en sí. Nótese que todos los procesadores aplican el método de redistribución para determinar de quién van a recibir o a quién van a enviar su carga.

- $T2$, es el tiempo de comunicación necesario para redistribuir toda la carga computacional. Este tiempo será el de las comunicaciones del procesador que más envíos realice o más mensajes reciba.

Obtendremos estimaciones de $T1$ y $T2$ para decidir si es rentable la redistribución de la carga.

El coste computacional de la heurística depende del número de veces que se deba recorrer el vector de pesos y el vector de capacidades para establecer mensajes entre los distintos procesadores. El número total de mensajes no se puede conocer hasta que se haya aplicado la heurística de redistribución, pero en el peor caso su valor será $P-1$.

Este máximo se alcanzará cuando se dé alguna de las dos situaciones extremas:

- Un procesador ha perdido toda su carga, y los $P-1$ procesadores restantes no han perdido ninguna columna. Para volver a balancear el sistema, estos procesadores tendrán que enviar un mensaje al que ha perdido su carga, recibiendo este último $P-1$ mensajes.

- Todas las columnas de un bloque asignadas a $P-1$ procesadores son eliminadas. El procesador cuya carga no ha sido alterada debe enviar un mensaje a cada uno de estos $P-1$ procesadores. El número máximo de envíos será, por lo tanto, $P-1$ y el de recepciones será 1.

De este modo, disponemos de un límite por exceso del coste de la heurística de redistribución si conocemos el coste computacional de cada iteración:

$$\text{coste(heuristica)} \leq \text{coste(iter)} \cdot (P-1). \tag{5.14}$$

Por otro lado, y para poder evaluar el coste total de la redistribución, es necesario estimar el tiempo requerido para el envío y la recepción de todos los mensajes. El procesador que más mensajes envíe o reciba es el que más tardará en finalizar la redistribución, y será el que determine el tiempo de comunicación de esta etapa. Hemos utilizado los 1740 ejemplos de la batería de pruebas sobre la que se han validado las heurísticas propuestas. Se han realizado dos histogramas, uno del número máximo de envíos y otro del número máximo de recepciones. Ambos histogramas se muestran en la figura 5.16, y en ellos se observa un comportamiento gaussiano con su valor máximo centrado en 3. Por lo tanto, consideraremos que el tiempo de comunicación viene dado por el coste de 3 mensajes, que supone un valor medio con suficiente fiabilidad.

Así, el coste del tiempo de comunicación del algoritmo de redistribución será,

$$\text{coste(comunica)} \cong \chi \cdot \text{coste(mensaje)}, \tag{5.15}$$

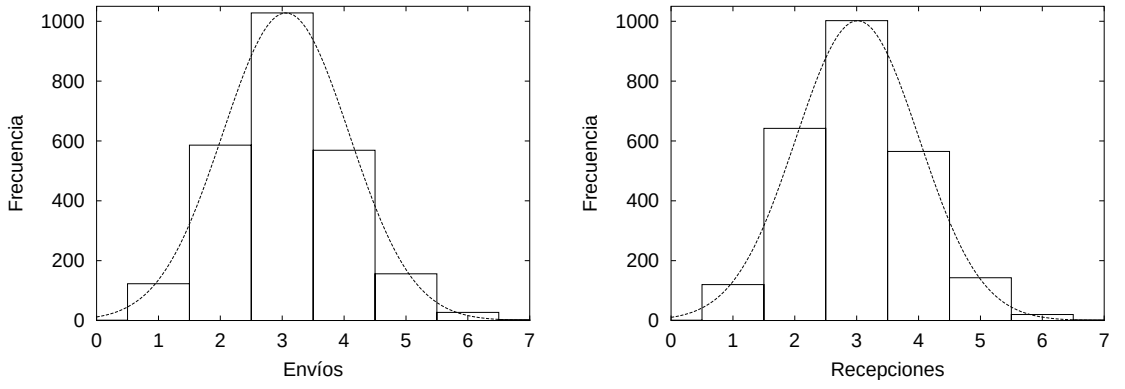


Figura 5.16: Histogramas del número máximo de envíos y recepciones que genera la heurística de redistribución EH.

siendo χ un parámetro que representa la relación entre el coste de un único y de tres mensajes punto a punto consecutivos, y cuyo valor depende del comportamiento de los mecanismos de gestión de los mensajes del sistema considerado.

Ahora, establecemos el coste de un mensaje punto a punto, que depende del tamaño del mismo y de las propiedades del sistema multiprocesador, en especial, de la red de interconexión. Así que, por un lado se estimará un tamaño por defecto para cada mensaje, y por otro, el coste computacional del mensaje en el sistema.

Para medir el coste de un mensaje punto a punto de tamaño l en un determinado sistema, se han ejecutado programas de prueba tipo *ping-pong* de distintos tamaños, medimos su coste, y ajustamos los valores obtenidos por tramos lineales. Los resultados experimentales obtenidos se muestran en la siguiente sección.

Si suponemos que el procesador con carga C_{max} es el que va a realizar más envíos, y de acuerdo con los histogramas realizados estimamos este número en 3, cada mensaje contendrá en media $\lceil C_{max}/3 \rceil$ segmentos de columnas. Como el tamaño de estos segmentos es αP y cada elemento está representado en doble precisión (8 bytes), el tamaño del mensaje será:

$$l = (\text{columnas/bloque}) \cdot (\text{elementos/columna}) \cdot (\text{bytes/elemento}) = \lceil \frac{C_{max}}{3} \rceil \cdot \alpha P \cdot 8. \tag{5.16}$$

Por lo tanto, la estimación del coste de la estrategia de redistribución EH será,

$$\text{coste}(\text{EH}_{\text{bloque}}) = \text{coste}(\text{iter}) \cdot (P-1) + \chi \cdot (a_{men} + b_{men} \cdot \lceil \frac{C_{max}}{3} \rceil \cdot \alpha P \cdot 8). \tag{5.17}$$

De nuevo, el coste computacional dado por la ecuación 5.17 se corresponde con

la redistribución de las porciones de columnas que pertenecen a un único bloque de tamaño αP. Sin embargo, la redistribución se hará de columnas enteras, por lo que tendremos que multiplicar el tamaño del mensaje por el nivel al que pertenece el bloque estudiado, Ω, de modo que:

$$\text{coste(EH)} = \text{coste(iter)} \cdot (P-1) + \chi \cdot (a_{men} + b_{men} \cdot \Omega \cdot \lceil \frac{C_{max}}{3} \rceil \cdot \alpha P \cdot 8). \quad (5.18)$$

Una vez estimados el coste computacional del desbalanceo y de la redistribución (ecuaciones 5.13 y 5.18), para determinar cuándo compensa realizar la redistribución bastará con detectar el punto de corte de estas dos ecuaciones, es decir, cuando

$$\Omega \cdot C_{max} \cdot (a_{desba} + b_{desba} \cdot \alpha P) = \text{coste(iter)} \cdot (P-1) + \chi \cdot (a_{men} + b_{men} \cdot \Omega \cdot \lceil \frac{C_{max}}{3} \rceil \cdot \alpha P \cdot 8). \quad (5.19)$$

En todos los problemas, los parámetros α y P determinan el tamaño del mensaje, y por tanto, los coeficientes a_{men} y b_{men} de la ecuación 5.18. Conocidos estos parámetros, y dependiendo de cuántas columnas han sido eliminadas, la variable C_{max} tomará distintos valores. C_{max} mide, en cierto modo, el desbalanceo dentro de un bloque. De la ecuación 5.19 deducimos que la redistribución de la carga computacional compensará cuando el número de columnas que excedan del equilibrio sea,

$$C_{max} \approx \frac{\text{coste(iter)} \cdot (P-1) + \chi \cdot a_{men}}{(a_{desba} + \alpha P \cdot b_{desba} - b_{men} \cdot 8/3 \cdot \alpha P) \cdot \Omega}. \quad (5.20)$$

Este valor de C_{max} es válido si consideramos que tras la redistribución sólo se va a realizar una iteración del algoritmo, pero si suponemos, como realmente sucederá en la práctica, que la nueva distribución que lleva al sistema a un balanceo de la carga computacional se va a mantener durante un número de iteraciones i, podemos considerar que el coste computacional de la redistribución es,

$$\text{coste(redis)} = \frac{\text{coste(iter)} \cdot (P-1) + \chi \cdot (a_{men} + b_{men} \cdot \Omega \cdot \lceil \frac{C_{max}}{3} \rceil \cdot \alpha P \cdot 8)}{i}. \quad (5.21)$$

De modo que compensará realizar la redistribución a partir de un valor de C_{max} igual a:

$$C_{max} = \frac{(\text{coste(iter)} \cdot (P-1) + \chi \cdot a_{men})/i}{(a_{desba} + \alpha P \cdot b_{desba} - (b_{men} \cdot 8/3 \cdot \alpha P)/i) \cdot \Omega}. \quad (5.22)$$

5.6. Análisis de las estimaciones

Como se ha descrito en la sección anterior, nuestro algoritmo paralelo se basa en estimaciones de los posibles costes computacionales para decidir si se realiza

Tamaño	bytes	a_{men} (μs)	b_{men} (μs/bytes)
pequeño	0-96	30.0347	0.1286
mediano	97-16K	62.2112	0.0270
grande	>16K	213.9943	0.0146

Tabla 5.3: Coeficientes de las rectas de ajuste del coste de una comunicación punto a punto para diferentes tamaños del mensaje en el AP3000 de Fujitsu.

o no una redistribución de la carga. El valor y la precisión de estas estimaciones dependerá del sistema paralelo elegido. A continuación, presentamos una evaluación precisa de las mismas sobre cada uno de los sistemas paralelos utilizados, así como medidas experimentales de la precisión lograda. Para realizar un análisis lo más completo posible hemos forzado situaciones de gran desbalanceo que serán bastante improbables en la resolución de un determinado problema de optimización real. Además, será importante demostrar que la redistribución compensa para pequeños valores de desequilibrio computacional. De este modo, la heurística EH se podrá aplicar de manera efectiva en la resolución, mediante el método cuasi-Newton, de un gran número de problemas de optimización.

5.6.1. Estimaciones en el multiprocesador de memoria distribuida AP3000 de Fujitsu

Las primeras estimaciones tratan de predecir el coste computacional del desbalanceo para las 4 rutinas de álgebra lineal, mencionadas en la sección anterior, en el AP3000 de Fujitsu. Los resultados del coste de estas subrutinas pueden verse en la figura 5.17, donde se muestran además los coeficientes de las rectas de ajuste y la desviación cuadrática media para cada caso. Nótese que en todos los casos $R^2 \simeq 1$ lo que demuestra la calidad del ajuste lineal.

El coste total de las 4 rutinas de la figura 5.17 viene dado por la ecuación 5.11, donde $a_{desba} = 3.0820 \cdot 10^{-6}$ s y $b_{desba} = 1.2957 \cdot 10^{-6}$ s/bytes para el AP3000 de Fujitsu.

El siguiente paso es determinar el coste de ejecución de la heurística EH, y por lo tanto, el coste debido a las comunicaciones que viene dado por la ecuación 5.15. Para determinar el coste de un mensaje punto a punto sobre el AP3000, a partir de los datos obtenidos de la ejecución de los programas de prueba *ping-pong*, realizaremos una serie de ajustes lineales [6]. Distinguiremos 3 tramos, que se corresponden con

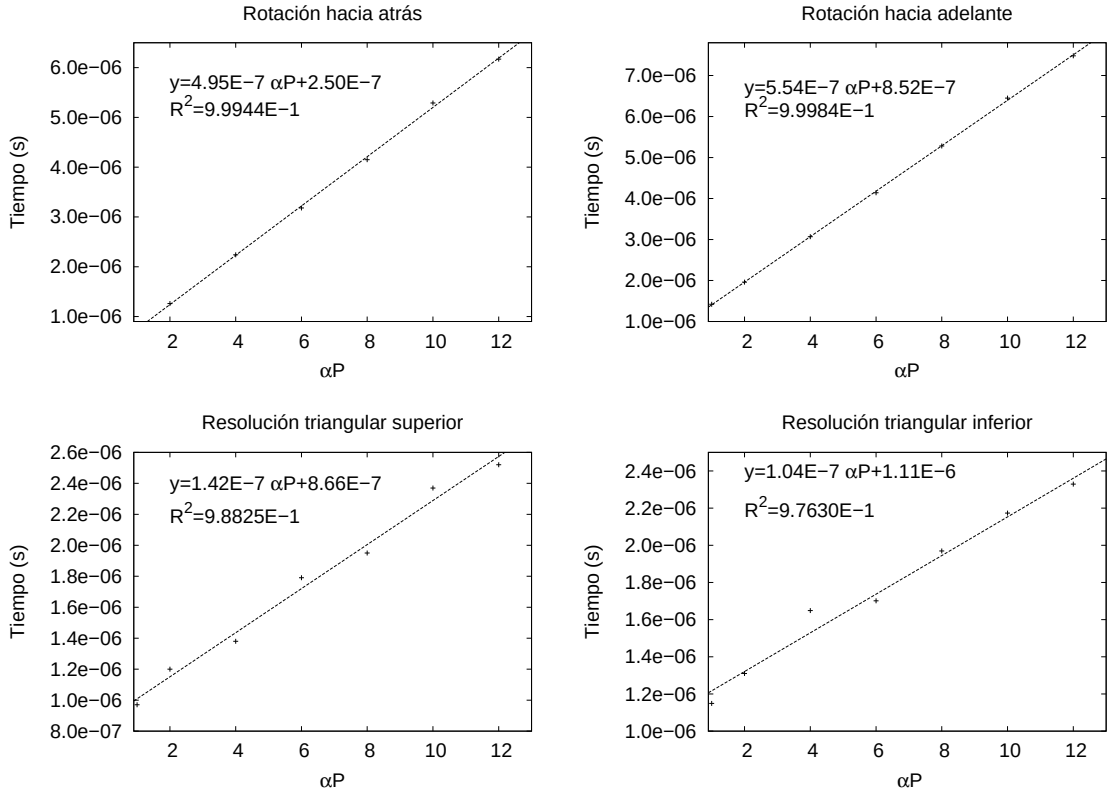


Figura 5.17: Ajuste lineal de los tiempos de computación de las cuatro rutinas más costosas del método cuasi-Newton en el AP3000.

mensajes de tamaño pequeño, mediano o grande. La elección de estos tramos se basa en la coincidencia de eventos significativos en la jerarquía de memoria que influyen en el manejo de los dispositivos de almacenamiento de información presentes en una comunicación. Para cada uno de estos tamaños, el coste del mensaje punto a punto se puede ajustar linealmente de la forma $a_{men} + b_{men} \cdot l$. Los valores de los coeficientes a_{men} y b_{men} varían dependiendo del tamaño del mensaje, y sus valores para el AP3000 de Fujitsu se muestran en la tabla 5.3. Además, en la figura 5.18 se muestra como varían los tiempos de comunicación en el AP3000 para mensajes punto a punto dependiendo de su tamaño. Podemos observar que para los tres tamaños del mensaje los ajustes son muy precisos (desviación cuadrática media próxima a la unidad), y que todos los valores medidos experimentalmente están en un entorno de la recta de ajuste con un margen de error inferior al $\pm 5\,\%$.

Finalmente, para determinar el coste asociado a las comunicaciones se necesita

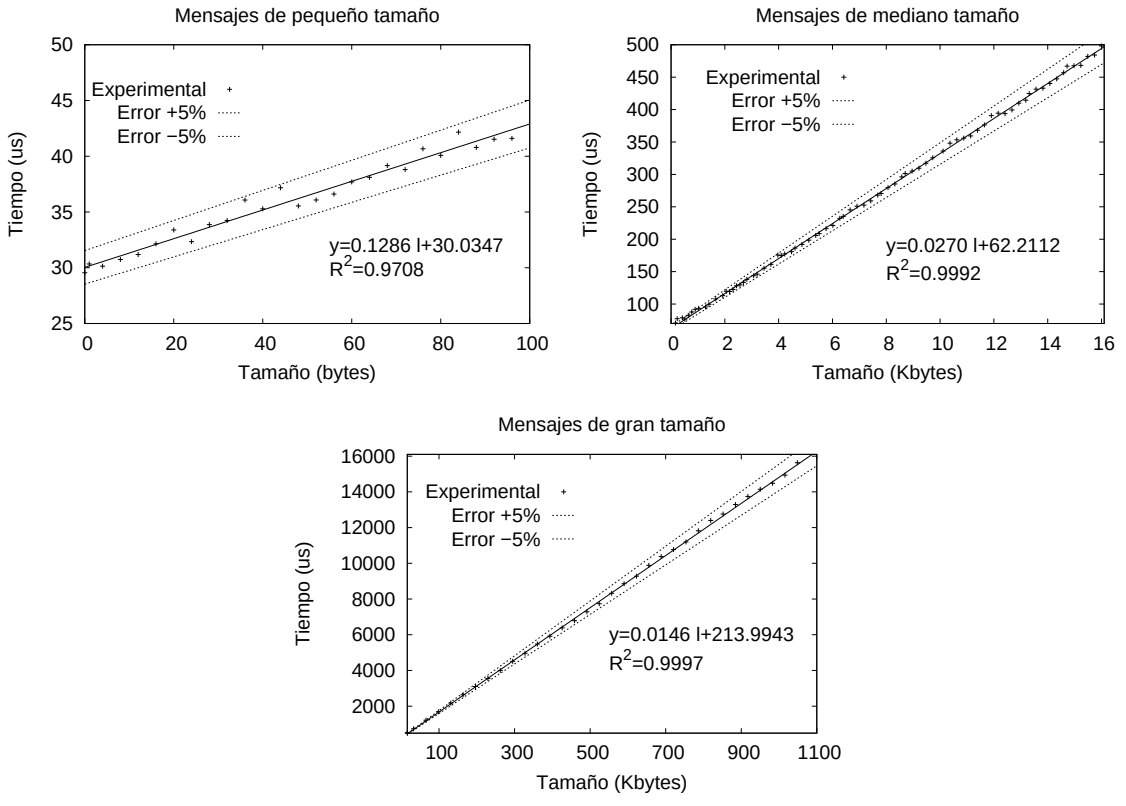
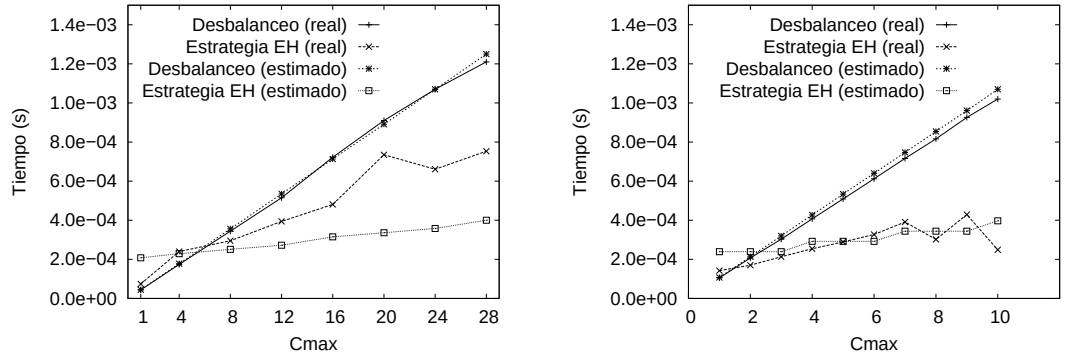


Figura 5.18: Coste de un mensaje punto a punto en el AP3000.

conocer el valor del parámetro χ. Experimentalmente en el AP3000 se ha demostrado que el coste de tres mensajes del tamaño de los generados es equivalente a tres mensajes punto a punto. Dentro del coste total de aplicación de la heurística EH, el coste de ejecución de la heurística propiamente dicho queda encubierto por el coste de las comunicaciones. Las estimaciones del coste del desbalanceo y de aplicación de la heurística EH se comparan en la figura 5.19 con los valores experimentales reales obtenidos en el AP3000. La calidad de estas estimaciones ha sido validada sobre 8 procesadores y para diferentes tamaños de los bloques rectangulares. En la figura se muestran solamente los resultados para $\alpha = 4$ y $\alpha = 10$, considerando desbalanceos en un único bloque rectangular ($\Omega = 1$).

Analizando los resultados, se observa que para pequeños valores de α (figura 5.19(a)), la estimación del coste de la heurística EH difiere significativamente de su coste real para grandes valores del desbalanceo. Sin embargo, a medida que α aumenta, la predicción realizada se aproxima con más precisión al comportamiento real de la heurística. La razón de este comportamiento radica en que, en una si-

tuación de equilibrio, en nuestro algoritmo de optimización cada procesador tiene asignadas α columnas. Al ser el valor de α pequeño un gran desbalanceo supondrá la pérdida total de carga por parte de algunos procesadores del sistema. De esta forma, el procesador con mayor carga en exceso tendrá que enviar, con cierta frecuencia, más de 3 mensajes para restablecer el balanceo computacional. Para grandes valores de α (figura 5.19(b)) la estimación es se aproxima más al comportamiento real ya que es más frecuente que el máximo número de recepciones o envíos por procesador dentro del sistema sea 3. Además, nuestra estimación es altamente presisa para pequeños desbalanceos, independientemente del valor de α, como se observa en la proximidad del punto de intersección de las estimaciones del coste del desbalanceo y la heurística EH con el punto de intersección real en las gráficas de la figura 5.19. La intersección se produce siempre para pequeños valores de desbalanceo. Decreciendo, incluso, a medida que α o P aumentan, ya que, en este caso, la carga computacional y el coste del desbalanceo son más significativos. Por lo tanto, en el caso del AP3000 de Fujitsu es adecuada la aplicación de la estrategia de redistribución EH prácticamente en todos los casos, incluso en aquellos donde la redistribución puede ser más costosa que el desbalanceo, ya que transcurridas varias iteraciones del método de optimización, la redistribución ya resulta beneficiosa.



(a) (b)

Figura 5.19: Comparación entre el coste real y el estimado de la estrategia EH para diferentes tamaños de bloque en el AP3000. (a) $\alpha = 4$, (b) $\alpha = 10$.

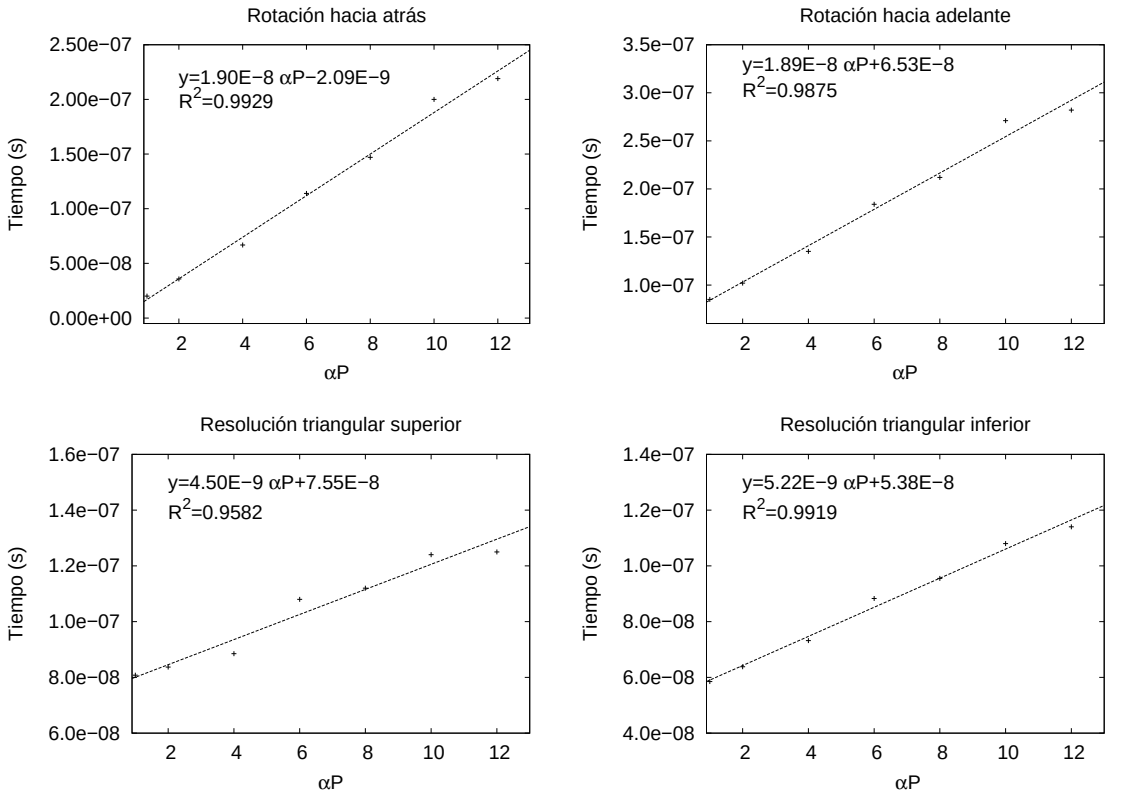


Figura 5.20: Ajuste lineal de los tiempos de computación de las cuatro rutinas más costosas del método cuasi-Newton en el *cluster beowulf* del CESGA.

5.6.2. Estimaciones en el *cluster beowulf* del CESGA

En esta sección se presentan los resultados obtenidos para las estimaciones realizadas sobre el *cluster beowulf* del CESGA. Las diferencias fundamentales con respecto al sistema multiprocesador AP3000 estriban en el incremento del cociente T_{comu}/T_{comp}, lo que implica una mayor necesidad de minimizar el número de mensajes para que la redistribución de la carga computacional resulte eficiente.

El análisis realizado es análogo al hecho para el AP3000 de Fujitsu. El coste del desbalanceo se modela según la ecuación 5.11, donde $a_{desba} = 1.9252 \cdot 10^{-7}$ s y $b_{desba} = 4.7649 \cdot 10^{-8}$ s/bytes. El coste computacional de las subrutinas más costosas del método cuasi-Newton se muestran en la figura 5.20, donde se indican además los coeficientes de las rectas de ajuste y la desviación cuadrática media para cada caso.

El coste de una comunicación punto a punto depende, de nuevo, del tamaño del

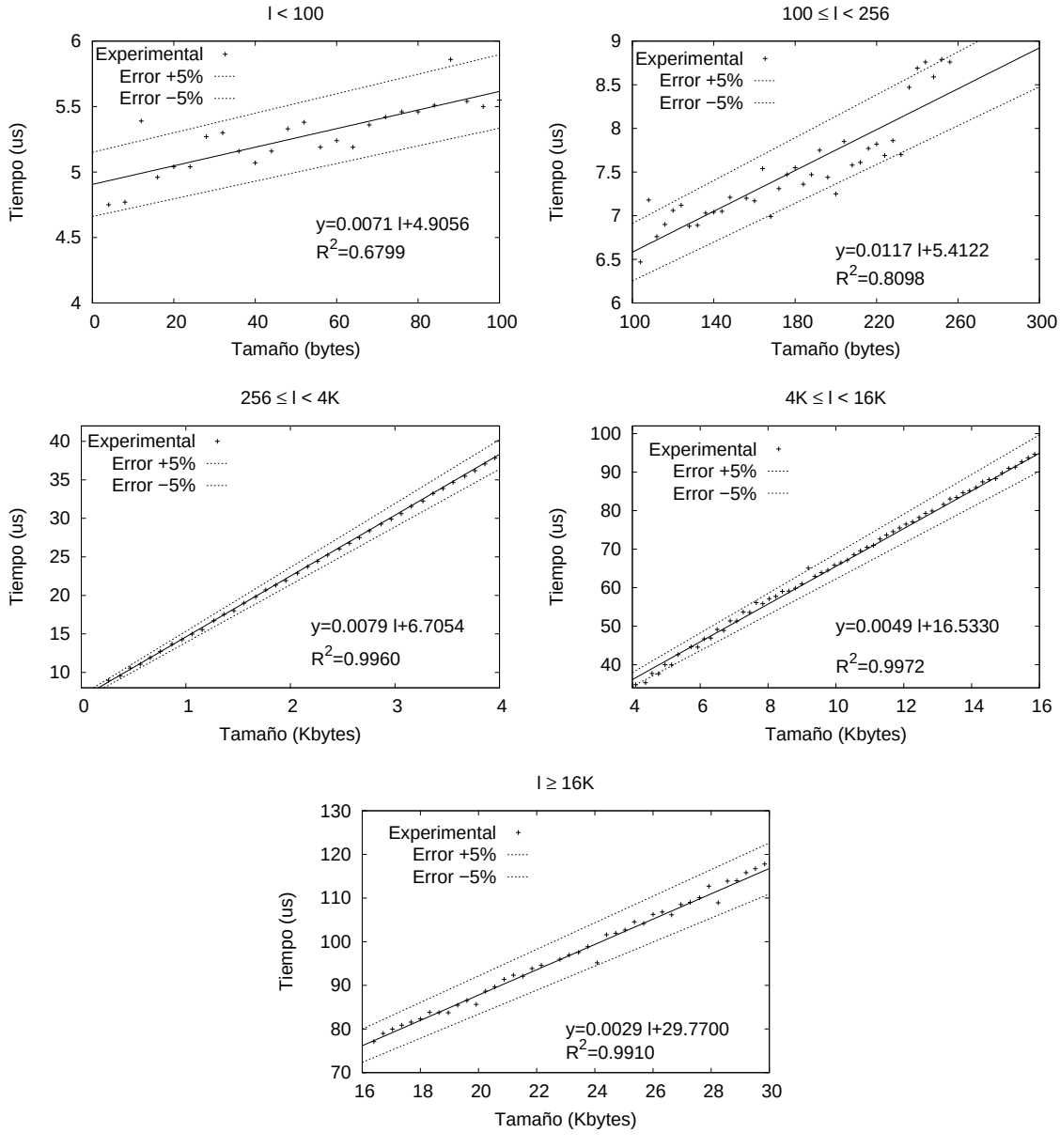


Figura 5.21: Coste de un mensaje punto a punto en el *cluster beowulf* del CESGA.

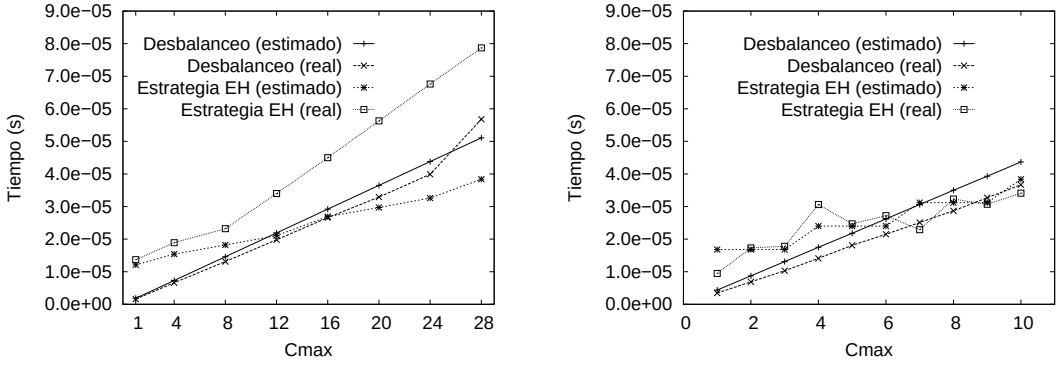
Tamaño en bytes	a_{mes} (μs)	b_{mes} (μs/bytes)
0-100	4.885	0.007
100-256	5.207	0.012
256-4K	6.684	0.008
4K-16K	16.963	0.005
$>$ 16K	139.095	0.002

Tabla 5.4: Coeficientes de las rectas de ajuste del coste de una comunicación punto a punto para diferentes tamaños de mensaje en el *cluster beowulf* del CESGA.

mensaje según la ecuación $a_{mes} + b_{mes} \cdot l$. Los resultados obtenidos a partir de la aplicación del programa *ping-pong* del banco de pruebas PMB [68], se muestran en la tabla 5.4. Los parámetros mostrados en la tabla han sido obtenidos a partir de los ajustes lineales que se muestran en la figura 5.21. El estudio realizado, análogo al llevado a cabo sobre el AP3000 de Fujitsu, nos conduce en este caso a la distinción de 5 tramos en el tamaño del mensaje para poder realizar un modelado óptimo del coste de una comunicación punto a punto en el *cluster beowulf*. Los ajustes son suficientemente precisos, aunque ligeramente inferiores a los del AP3000 de Fujitsu, ya que ahora, para mensajes de pequeño tamaño algunos puntos difieren de la recta estimada en un margen de error ligeramente superior al $\pm 5\,\%$. También hemos realizado un estudio empírico para determinar el valor del parámetro χ. El valor de este parámetro en el *beowulf* varía dependiendo del tamaño del mensaje como se puede observar en la tabla 5.5. El valor de χ depende de la latencia de la red y de su ancho de banda, de cómo se ha implementado el MPI en el *cluster*, y del modo en que el protocolo Myrinet intercambia información entre los distintos procesadores y la red. Para mensajes de pequeño tamaño, como los usados en nuestras pruebas, la latencia adquiere una mayor relevancia en el coste del mensaje que el ancho de banda. Los resultados obtenidos al aplicar al *beowulf* los programas de prueba PMB GmbH, muestran, por ejemplo, para un mensaje de tamaño 2 Kbytes una latencia de 40 μs y un ancho de banda de 40 Mbytes/s [4]. En cuanto al mecanismo para el envío de un mensaje, será necesaria la copia del mismo en dos *buffers* (protocolo Myrinet 1 copia) o un único *buffer* (protocolo Myrinet 0 copias). En cualquier caso, dependiendo del tamaño de los mensajes, puede que los tres quepan en el *buffer*, reduciendo de este modo la latencia total asociada a los tres mensajes, y siendo menor que tres

Tamaño en bytes	χ
< 1K	1.53
1K-4K	1.34
4K-8K	1.47
8K-16K	1.56
> 16K	2.96

Tabla 5.5: Valor de χ en función del tamaño del mensaje en el *cluster beowulf* del CESGA.



(a) (b)

Figura 5.22: Comparación entre el coste real y el estimado de la estrategia EH para diferentes tamaños de bloque en el *cluster beowulf* del CESGA. (a) $\alpha = 4$, (b) $\alpha = 10$.

veces la latencia de un mensaje. Sin embargo, para mensajes de gran tamaño, χ es prácticamente igual a 3 porque un sólo mensaje ocupa todo el *buffer* y hasta que la recepción de dicho mensaje es confirmada (nótese que estamos trabajando con envíos bloqueantes) dicho *buffer* no se puede utilizar para copiar el siguiente.

El coste del desbalanceo y la eficiencia de la estrategia EH han sido estudiados sobre el *cluster beowulf* para un máximo de 8 procesadores [75]. El análisis se ha realizado para distintos tamaños de bloque. A modo ilustrativo, la figura 5.22 muestra los resultados obtenidos para $\alpha = 4$ y $\alpha = 10$, y un único nivel de la matriz, $\Omega = 1$.

De nuevo, observamos que para pequeños valores de α (figura 5.22(a)) la estimación del coste de la estrategia LH difiere bastante de su coste real en el caso de desbalanceos grandes. El motivo es el mismo que el del caso del AP3000. Para valores mayores de α (figura 5.22(b)), el coste del desbalanceo adquiere mayor relevancia que el coste adicional que implica el establecimiento de mensajes. Ahora, el punto de intersección (ecuación 5.22) de las rectas de coste estimado de la estrategia LH y de coste estimado del desbalanceo se produce en torno a un valor de $C_{max} = 7.3$. En la figura 5.22(b) podemos observar la proximidad del punto de intersección real (en torno a $C_{max} = 8.8$). Por lo tanto, para valores de α elevados, se ha observado que en determinados problemas de optimización, si se decide realizar la redistribución para el valor teórico de C_{max}, el coste de la misma se verá compensado tras varias iteraciones del algoritmo paralelo.

Capítulo 6

Estrategias multipaso aplicadas a optimización no lineal

En capítulos anteriores hemos abordado el problema de paralelizar eficientemente un método cuasi-Newton tratando de disminuir el coste computacional de cada iteración del método. En este capítulo abordaremos el problema desde una perspectiva diferente. Del análisis computacional del método hemos deducido que lo más costoso en cada iteración es la resolución de algunas rutinas de álgebra matricial y la evaluación de la función objetivo y de su gradiente. Se han presentado algunas propuestas para acelerar cada iteración incidiendo en la reducción del tiempo de ejecución de las rutinas más costosas y en la evaluación de la función objetivo y su gradiente en paralelo. Sin embargo, existe otro modo de enfocar la paralelización de este algoritmo, que consiste en la disminución del tiempo de ejecución mediante la reducción del número de iteraciones y de las evaluaciones de la función objetivo necesarias para obtener la solución del problema. La alternativa propuesta en este capítulo es un conjunto de estrategias que reciben el nombre de métodos multipaso. En cada iteración de un método cuasi-Newton es necesario calcular un nuevo punto solución modificando el punto de solución actual un determinado tamaño de paso en la dirección computada. La correcta selección de este tamaño de paso es necesaria para la convergencia del método. El número total de evaluaciones de la función objetivo y de su gradiente dependen también de la elección adecuada del tamaño de paso. Una reducción en el número de evaluaciones implica una mejora significativa en el rendimiento del algoritmo.

En esta línea, existen algunos trabajos propuestos en la literatura del tema. En concreto, son destacables los métodos multipaso y multidirección basados en la métrica variable de Phua [80]. Estos algoritmos generan varias direcciones de

búsqueda en cada iteración, y para cada dirección generan distintos tamaños de paso. Cada dirección y tamaño de paso conducen a un punto solución diferente. Cada nueva solución se usa como punto de partida para la iteración siguiente, generándose distintos hilos de ejecución que se computan en paralelo. Una dirección de búsqueda y un tamaño de paso dados implican la ejecución de un método cuasi-Newton distinto en cada procesador. Nótese que se asigna un único proceso por procesador, de ahí que en el resto del capítulo ambos conceptos se usen indistintamente.

En este capítulo se proponen una serie de heurísticas basadas en un estructura en árbol. Cada rama del árbol se identifica con un hilo de ejecución del método cuasi-Newton que usa un tamaño de paso diferente al empleado en el resto de las ramas. El árbol de ejecución se caracteriza por una serie de parámetros que son establecidos por el usuario. A continuación, se describen las distintas heurísticas propuestas, analizando sus ventajas e inconvenientes. Se definen los parámetros que conforman el árbol y el modo en que la selección de los mismos se adapta al problema concreto que se pretende resolver.

6.1. Procedimiento de búsqueda lineal

El procedimiento de búsqueda lineal del método cuasi-Newton computa el tamaño de paso en cada iteración. Determinar el tamaño de paso λ_k en la k-*ésima* iteración implica la solución del siguiente problema de optimización:

$$\varphi(\lambda_k) = f(x_k + \lambda_k p_k) = min \quad f(x_k + \theta p_k), \quad con \quad 0 < \theta \leq \lambda_{max}. \tag{6.1}$$

Verificando las condiciones de Wolfe,

$$f(x_k + \lambda_k p_k) - f(x_k) \leq \mu \lambda_k \nabla f(x_k)^T p_k, \tag{6.2}$$

$$\nabla f(x_k + \lambda_k p_k)^T p_k \geq \eta \nabla f(x_k)^T p_k, \tag{6.3}$$

con μ, $\eta \in (0, 1)$. La primera condición (6.2) fuerza un decremento suficiente en la función objetivo. Sin embargo, esta condición no es suficiente para garantizar la convergencia. Es necesaria la segunda condición o condición de curvatura (6.3) para garantizar que la actualización cuasi-Newton es definida positiva, y que λ_k aproxima el nuevo punto solución a un mínimo local de la función objetivo.

Los métodos más difundidos para resolver este tipo de problemas son los llamados métodos "salvaguarda" [43]. Estos métodos se pueden considerar una combinación de los métodos de bisección y los de interpolación lineal. Su eficiencia se basa en el uso de

información procedente de la función objetivo. Los polinomios se usan con frecuencia para interpolar la función $f(x_k + \theta p_k)$ [29]. De forma que, si la aproximación es inexacta, el procedimiento puede divergir, por lo que un intervalo de incertidumbre $[a, b]$ se usa como salvaguarda.

El procedimiento de búsqueda lineal determina una secuencia de estimaciones mejoradas de un mínimo de $f(x_k + \theta p_k)$ en un cierto intervalo $(0, \lambda_{max}]$. Para computar esta secuencia de estimaciones se utiliza interpolación cúbica con salvaguarda. El procedimiento se muestra en el algoritmo 10. En la iteración k-*ésima* el método parte de una estimación inicial del tamaño de paso λ_{k0}. Se comprueba si esta estimación verifica las condiciones de Wolfe. De ser así, el procedimiento de búsqueda habría concluido. De no ser así, es necesario obtener mediante sucesivas aproximaciones el valor del tamaño de paso óptimo para esa iteración. La rutina implementa un lazo dentro del cual se buscan los dos mejores puntos solución de la función $\varphi(\lambda_k)$. Estos puntos se utilizan para determinar la nueva estimación del tamaño de paso mediante interpolación cúbica. Cada variación δ del tamaño de paso tiene que estar dentro del intervalo de incertidumbre. Además, δ define un nuevo intervalo de incertidumbre en función de si la nueva estimación del tamaño de paso es mejor o no que la anterior. La rutina continúa con la ejecución de este lazo hasta que se encuentre una estimación que verifique las condiciones de Wolfe.

La correcta selección de un tamaño de paso inicial es crítica para la convergencia global del método. En la iteración k-*ésima*, se elige un paso inicial a partir de la ecuación $\lambda_{k0} = min(1, \gamma)$, donde $\gamma = damp \cdot (1 + ||x_k||_1)/||p_k||_1$. El parámetro *damp*, denominado parámetro de amortiguamiento, limita las modificaciones que se pueden producir en el punto solución x_k durante la búsqueda lineal. Así, se evitan evaluaciones de la función objetivo en puntos irrelevantes. El valor adecuado del parámetro de amortiguamiento depende de las características particulares del problema de optimización que se desea resolver.

6.2. Técnicas multipaso basadas en árboles

Las técnicas multipaso se caracterizan por computar varios tamaños de paso en cada iteración de un método cuasi-Newton. En nuestras propuestas [69, 76], los diferentes valores de λ se obtienen de seleccionar distintos valores del parámetro de amortiguamiento. Estas técnicas se basan en una estructura en árbol, de forma que cada rama del árbol computa un método cuasi-Newton diferente. Cada nodo del árbol se corresponde con una iteración del método a la que se le ha asignado un determinado parámetro de amortiguamiento. Por lo tanto, una vez definida la

Algoritmo 10 Procedimiento de búsqueda lineal

1: *Inicializar.*

1.1: Obtener la estimación inicial de λ_k

$\lambda_{k0} = min(1, \gamma)$, $\gamma = damp \cdot (1 + ||x_k||_1)/||p_k||_1$

1.2: Establecer el intervalo de incertidumbre $(a, b] = (0, \lambda_{max}]$,

$\lambda_{max} \propto 1/||p_k||_1$

2: *Chequear el criterio de terminación.*

if las condiciones de Wolfe se verifican **then**

stop

end if

3: *Buscar los puntos de interpolación: 2 mejores tamaños de paso λ_{2best} y λ_{best}.*

if λ_{ki} mejora la solución **then**

$\lambda_{2best} = \lambda_{best}, \quad \lambda_{best} = \lambda_{ki}$

$\varphi_{2best} = \varphi_{best}, \quad \varphi_{best} = \varphi_{ki}$

$\nabla\varphi_{2best} = \nabla\varphi_{best}, \quad \nabla\varphi_{best} = \nabla\varphi_{ki}$

end if

4: *Aplicar interpolación cúbica para computar δ*

$\delta = s/q \cdot \delta_{2best}$, donde $s = \sqrt{|\nabla\varphi_{best}|} \cdot \sqrt{|\nabla\varphi_{2best}|}$,

$q = \sqrt{t^2 - \nabla\varphi_{best} \cdot \nabla\varphi_{2best}}$, $t = 3 \cdot (f_{best} - f_{2best})/\lambda_{2best} + \nabla\varphi_{best} + \nabla\varphi_{2best}$

5: *Computar la nueva estimación de λ_k.*

$\lambda_{ki+1} = \lambda_{ki} + \delta$

6: *Cambiar el intervalo de incertidumbre.*

if λ_{ki+1} mejora la solución **then**

if $\nabla f(x_k + \lambda_{ki+1} p_k)^T \cdot p_k \geq \eta \cdot \nabla f(x_k)^T \cdot p_k$ **then**

$(a, b] = (a - \delta, 0]$

else

$(a, b] = (0, b - \delta]$

end if

else

if $\delta \leq 0$ **then**

$(\delta, b]$

else

$(0, \delta]$

end if

end if

Volver al paso 2.

secuencia de parámetros de amortiguamiento asociada a una determinada rama del árbol, cada rama ejecuta un método cuasi-Newton en un procesador del sistema paralelo. De modo que, si el sistema posee P procesadores, se estarán ejecutando simultáneamente P algoritmos cuasi-Newton. Proponemos que cada cierto número de iteraciones, se seleccione la mejor rama y se utilice como punto de partida para la ejecución de un nuevo árbol. La diferencia entre las distintas técnicas propuestas estriba en el modo en que se define la secuencia de parámetros de amortiguamiento asociados a cada rama del árbol, y en el criterio utilizado para seleccionar el punto solución que sirve de nodo raíz de un nuevo árbol. A continuación, se describen las distintas estrategias, resaltando sus ventajas e inconvenientes.

6.2.1. Estrategia de múltiples parámetros de amortiguamiento por rama del árbol (EMAR)

Este método se basa en la ejecución de las computaciones asociadas a una estructura en árbol. Dicho árbol se caracteriza por dos parámetros: el número de nodos β y el número de ramas. β es un parámetro externo, y el número de ramas del árbol es el de todas las posibles combinaciones que se pueden hacer con los parámetros de amortiguamiento usados, esto es, $(n(damp))^{\beta}$, donde $n(damp)$ es el número de diferentes valores de $damp$ que son usados en cada iteración. El ejemplo de la figura 6.1 muestra como se formaría el árbol para tres valores de $damp$ y $\beta = 2$. En este caso, se generarían nueve ramas.

Una vez que el árbol ha sido definido, en la primera iteración el algoritmo comienza a ejecutar la carga computacional asociada a dicho árbol. Cada rama ejecuta un método cuasi-Newton en un procesador diferente con sus correspondientes valores del parámetro de amortiguamiento. Por lo tanto, serán necesarios tantos procesadores como ramas tenga el árbol. Cada árbol permanece activo durante β iteraciones. Tan pronto como el algoritmo finaliza todas las computaciones asociadas a esas β iteraciones, se determina la rama que ha alcanzado la mejor solución hasta el momento. El criterio de selección consiste en elegir la rama que obtenga el mínimo valor de la función objetivo. Cada vez que se realiza el chequeo para seleccionar la mejor rama, el punto solución actual asociado a esta rama se debe radiar al resto de los procesadores del sistema. En el ejemplo de la figura 6.1, las flechas a la derecha representan operaciones *allreduce*, y las ramas enmarcadas son las que han alcanzado la mejor solución.

Cuando los procesadores reciben el punto solución actual, éste se utiliza como punto inicial en la ejecución de un nuevo árbol. El proceso continúa de este modo

hasta que el método cuasi-Newton converge.

Esta heurística está caracterizada por dos parámetros: el ancho y la profundidad del árbol. El ancho del árbol determina el número de ramas, y su valor depende de la cantidad de distintos parámetros de amortiguamiento empleados. La profundidad del árbol se define como el número de iteraciones existente entre chequeos consecutivos que se realizan para determinar cual es la mejor solución. A medida que este número se incrementa, se alcanzará una mejor solución del problema de optimización. En el límite superior de β cuando se aplica un único árbol a todo el método iterativo, y se computan todas las posibles ramas, se puede garantizar que la heurística obtiene la mejor solución con los parámetros de amortiguamiento considerados.

Sin embargo, en la práctica, el método EMAR presenta algunos inconvenientes. El primero es que eventualmente la rama seleccionada como la mejor en una cierta iteración no es la rama que hará converger el método en el menor número de iteraciones. Por este motivo, el método EMAR no alcanza siempre la mejor solución.

El segundo, y más importante inconveniente de este método, es que el número de procesadores requerido crece rápidamente cuando se incrementa el valor de β o se usa un mayor número de parámetros de amortiguamiento. Sólo pequeños valores de β, como $\beta = 1, 2, 3, \cdots$, conducen a necesidades en número de procesadores realistas.

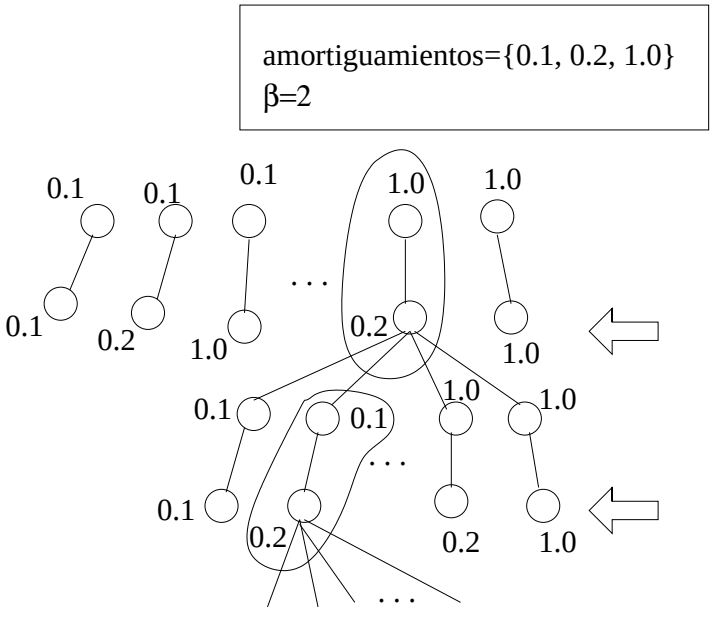


Figura 6.1: Ejemplo de aplicación del método EMAR.

Power		Penalty1		Powellsg	
β	Iter/Efun (%)	β	Iter/Efun (%)	β	Iter/Efun (%)
1	76.0/79.0	1	27.8/24.5	1	15.1/21.6
2	76.0/79.0	2	24.3/21.9	2	-2.0/9.6
3	69.4/72.4	3	21.3/21.7	3	27.3/33.1
4	76.0/79.0	4	35.0/2.3	4	14.1/21.6

Tabla 6.1: Porcentajes de reducción en número de iteraciones y evaluaciones de la función objetivo obtenidos por el método EMAR.

6.2.2. Resultados experimentales del método EMAR

Para evaluar la eficiencia de estas estrategias han sido usados tres problemas de optimización de la librería CUTE ("Power", "Penalty1" y "Powellsg"). Los resultados se comparan con las soluciones alcanzadas por el código de optimización MINOS. La eficiencia se mide en términos de porcentaje de reducción en el número total de iteraciones (*Iter*) y en el número de evaluaciones de la función objetivo (*Efun*) respecto del obtenido por el código MINOS. Los resultados se muestran en la tabla 6.1. Se han utilizado cinco parámetros de amortiguamiento diferentes y valores de β de 1 a 4, lo que implica un número de procesadores de 5 a 625.

En dicha tabla se observa que el rendimiento más alto se alcanza para el problema "Power". Teóricamente, conforme el parámetro β se incrementa, habría más probabilidades de obtener mejores soluciones. Sin embargo, esta conclusión no se puede deducir directamente de la tabla 6.1. De hecho para el problema "Penalty1", y $\beta = 4$ y el problema "Powellsg" y $\beta = 2$, la estrategia EMAR presenta problemas de convergencia. En el primer caso, lo que ocurre es que en alguna iteración el valor inicial de tamaño de paso elegido no resulta adecuado, lo que implica la repetición del proceso de búsqueda, y por tanto, la realización de un mayor número de evaluaciones de la función objetivo. En el último caso, el incremento en el número de iteraciones es debido al primer inconveniente del método EMAR mencionado en la sección anterior, es decir, la rama seleccionada como la mejor no siempre produce la convergencia más rápida. Nótese que, en ambos casos, si el número de iteraciones se incrementa, el número de evaluaciones se reduce, o a la inversa, lo que implicaría una reducción en el tiempo de ejecución.

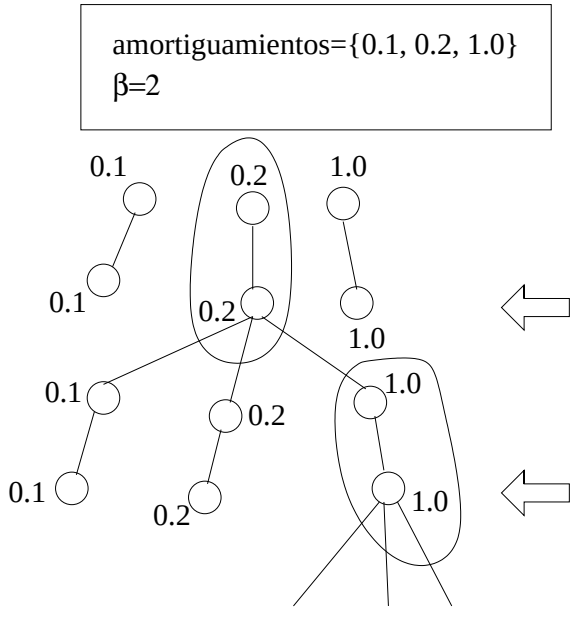


Figura 6.2: Ejemplo de aplicación del método EUAR.

6.2.3. Estrategia de único parámetro de amortiguamiento por rama del árbol (EUAR)

Este método plantea la necesidad de reducir el número de procesadores requerido por el método EMAR. En el método EUAR se considera únicamente un subconjunto de las ramas del árbol definido en el método EMAR, en concreto, sólo las ramas con los mismos valores del parámetro de amortiguamiento en todas las iteraciones. Esta decisión se justifica mediante la observación empírica de que la mejor solución para el método EMAR se alcanza con una frecuencia alta seleccionando los mismos valores de amortiguamiento durante un cierto número de iteraciones consecutivas. Para los problemas evaluados el mismo factor de amortiguamiento es seleccionado más de 5 veces consecutivas una media de un 70 % de las ocasiones. En la figura 6.2 se muestra un ejemplo de aplicación del método EUAR que usa tres parámetros de amortiguamiento y $\beta = 2$.

Cada procesador computa un método cuasi-Newton usando en cada iteración el valor del parámetro de amortiguamiento que tiene asignado. De nuevo, cada β iteraciones, cuando se identifica la mejor rama, se establece una comunicación entre todos los procesadores del sistema. Por lo tanto, como el número de mensajes depende de la profundidad del árbol, se requerirán los mismos en este método que en

el método EMAR. Sin embargo, como ahora el número de procesadores es menor, al tratarse de comunicaciones colectivas, estos mensajes serán menos costosos. El método EUAR es útil en el sentido de que se puede probar el efecto de un mayor número de parámetros de amortiguamiento usando menos procesadores. Eliminamos con ello uno de los dos inconvenientes mencionados del método EMAR, pero no el otro, puesto que el criterio de selección de la mejor rama y la ejecución de las computaciones de la estructura en árbol son los mismos en las dos estrategias.

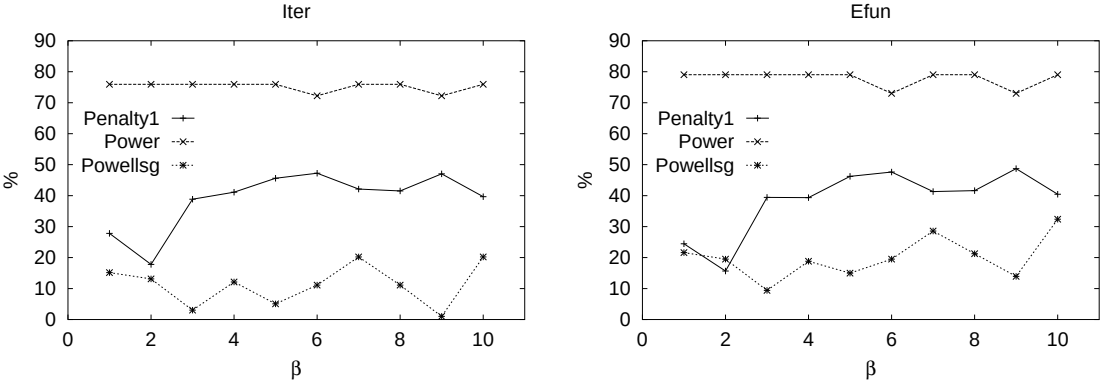


Figura 6.3: Porcentaje de mejora en número de iteraciones y evaluaciones de la función objetivo obtenido por el método EUAR con problemas de tamaño $n = 20$.

6.2.4. Resultados experimentales del método EUAR

Para evaluar la eficiencia del método EUAR utilizamos el mismo banco de pruebas y los mismos valores del parámetro de amortiguamiento que hemos empleado en el método EMAR. Ahora, sólo se necesitarán 5 procesadores, por lo tanto, los resultados se pueden obtener en el *cluster beowulf* del CESGA.

Los resultados para problemas de tamaño $n = 20$ se muestran en la figura 6.3. El problema "Power" obtiene porcentajes similares de mejora para los dos métodos. La eficiencia, en este caso, alcanza un valor de 0.83. Para los otros dos problemas, valores pequeños de β implican resultados similares o peores que en el método EMAR. Sin embargo, al aumentar la profundidad del árbol, el rendimiento del problema "Penalty1" mejora, y el del problema "Powellsg" se mantiene. En algunos casos, el

método EUAR supera al método EMAR en rendimiento, en concreto esta mejora se observa en aquellos casos para los que la estrategia EMAR presenta problemas de convergencia. Por ejemplo, para el problema "Penalty1" y $\beta = 4$ se pasa de un porcentaje de mejora del número de iteraciones/evaluaciones de 35.0/2.3 para la estrategia EMAR a un porcentaje de 41.1/39.7 para la estrategia EUAR. Y para el problema "Powellsg" y $\beta = 2$ se pasa de un porcentaje de -2.0/9.6 del método EMAR a 13.1/19.5 del método EUAR. La justificación de esta mejora se encuentra en el que se ha señalado como primer inconveniente del método EMAR. Además, la reducción en el número de evaluaciones de la función implica un menor coste en la computación del tamaño de paso. Ambas reducciones adquieren mayor importancia cuando el gradiente no está disponible de forma analítica, y tiene que ser calculado mediante diferencias finitas. A partir de estos resultados, se puede concluir que el método EUAR es una alternativa realista y menos costosa al método EMAR. De ahí que a partir de ahora nos centraremos en la estrategia EUAR y sobre ella realizaremos un exhaustivo estudio, evaluando su calidad y comparándola con las otras estrategias propuestas.

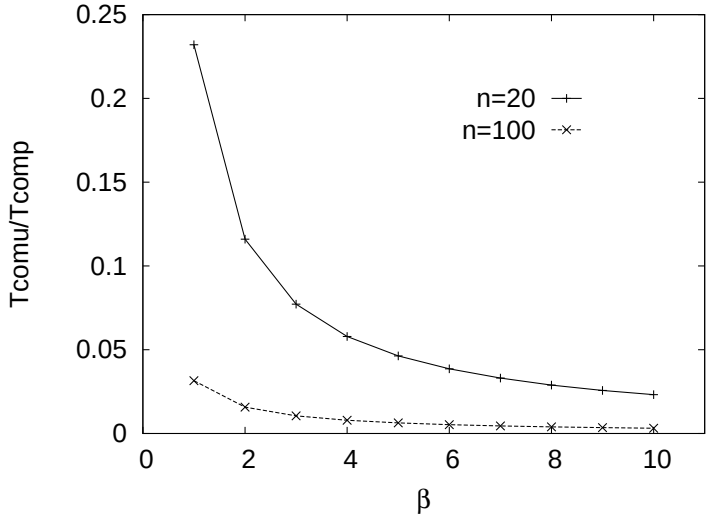


Figura 6.4: Relación T_{comu}/T_{comp} del método EUAR frente a la profundidad del árbol para el problema "Penalty1".

Para estudiar la eficiencia de los algoritmos paralelos propuestos hasta el mo-

mento es necesario analizar la relación entre el tiempo de comunicación y el tiempo de computación. El tiempo de comunicación T_{comu} se define como el coste total de todos los mensajes que es necesario establecer entre todos los procesadores del sistema, y el tiempo de computación T_{comp} como el coste computacional del algoritmo para el nuevo número de iteraciones requerido. Esta relación no sólo depende del *hardware* subyacente, sino también del tamaño del problema de optimización, debido a que un incremento del número de variables del problema implica un coste computacional más elevado por iteración. También depende del tipo de problema que se pretende resolver, ya que el número de mensajes aumenta al incrementarse el número de iteraciones.

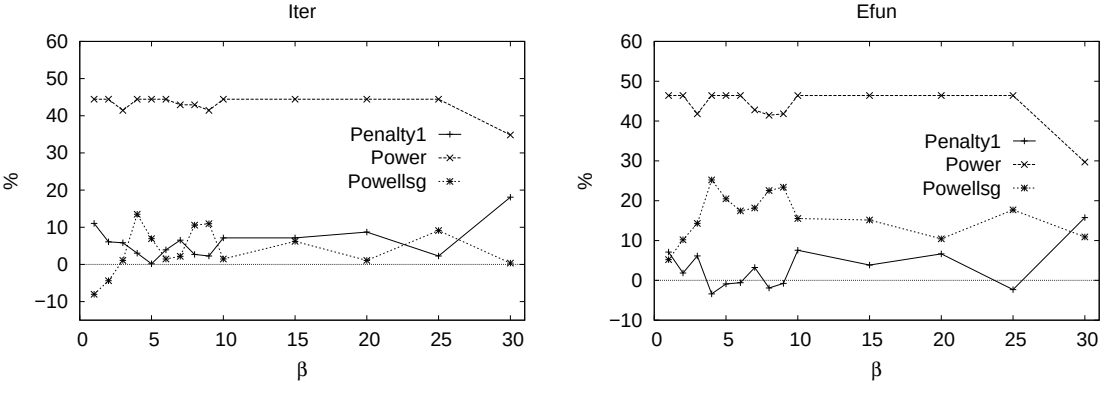


Figura 6.5: Porcentaje de mejora obtenido en número de iteraciones y evaluaciones de la función objetivo por el método EUAR con problemas de tamaño $n = 100$.

Hemos usado el problema "Penalty1" para estudiar esta relación porque es el problema que tarda más iteraciones en converger. Por lo tanto, si $T_{comu}/T_{comp} < 1$ para este problema también lo será para el resto. En la figura 6.4 se muestra el valor de esta razón para distintas profundidades del árbol. Cabe destacar que este cociente es siempre menor que uno, por lo tanto, se puede concluir que la aplicación del método EUAR es siempre eficiente desde el punto de vista de la paralelización. El valor de esta razón decrece cuando el tamaño del problema aumenta al incrementarse el coste computacional.

En la figura 6.5 se muestran los resultados obtenidos para problemas de mayor

tamaño. Se puede observar que el incremento del tamaño del problema empeora el rendimiento del método. Por ejemplo, para el problema "Power" la eficiencia se reduce desde un 0.83, para problemas de 20 variables, hasta 0.36 para problemas de 100 variables. Este deterioro de la eficiencia se observa también en los otros casos.

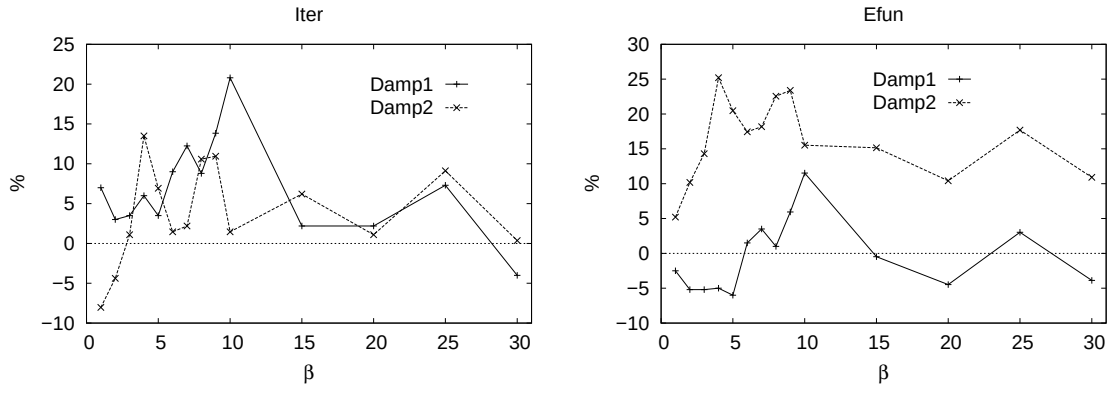


Figura 6.6: Porcentaje de mejora obtenido en número de iteraciones y evaluaciones de la función objetivo por el método EUAR para el problema "Powellsg" con $n = 100$ para dos conjuntos de parámetros de amortiguamiento diferentes.

De las gráficas de la figura 6.3 y de la figura 6.5 se puede concluir que el rendimiento del método EUAR depende de la profundidad del árbol empleado. La figura 6.6 demuestra que la eficiencia también depende de los parámetros de amortiguamiento seleccionados. Los resultados del método EUAR para el problema "Powellsg" obtenidos para dos conjuntos de parámetros de amortiguamiento diferentes, $Damp1 = \{1.0, 2.0, 3.0, 9.0, 10.0\}$ y $Damp2 = \{0.01, 0.1, 0.2, 1.0, 2.0\}$ se muestran en esta figura. Usando el conjunto *Damp1* el método alcanza, en el mejor caso, una mayor reducción del número total de iteraciones, aunque necesita realizar más evaluaciones de la función objetivo para computar el tamaño de paso. Dependiendo de lo costoso que sea evaluar la función objetivo y del tamaño de la matriz Hesiana, que está directamente relacionado con el coste computacional de una iteración, será más beneficioso primar la reducción del número de iteraciones o de evaluaciones de la función objetivo. Lo ideal es conseguir una reducción similar en ambos casos, pero en general, y puesto que al reducir el número de iteraciones el algoritmo converge

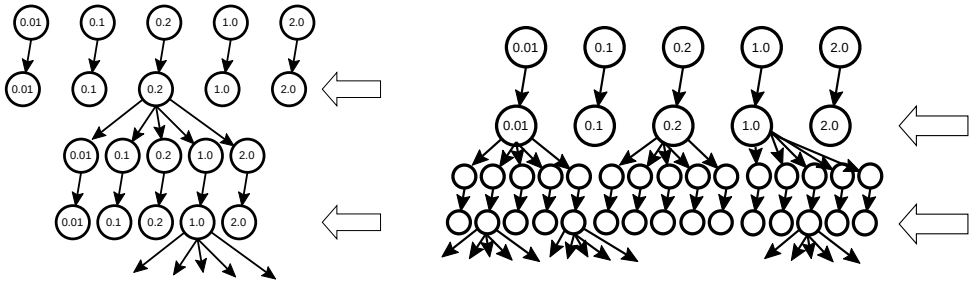


Figura 6.7: (a) Un ejemplo de aplicación del método EUAR. (b) Un ejemplo de aplicación del método ESMR con $mr = 3$. Para ambos casos se usa el conjunto de parámetros de amortiguamiento $\{0.01, 0.1, 0.2, 1.0, 2.0\}$ y $\beta = 2$.

usando un conjunto activo de restricciones más pequeño (lo que implica una Hesiana de menor orden), podemos considerar que la reducción en el número total de iteraciones es más importante en términos de coste computacional.

6.2.5. Estrategia de selección de las mr mejores ramas (ESMR)

Este método se propone para tratar de reducir el inconveniente que aparece en el método EUAR, y más frecuentemente en el método EMAR, de que la rama que en un punto obtiene el mínimo valor de la función objetivo no es la que necesariamente lleva a una convergencia más rápida. El método ESMR consiste en seleccionar más de una rama cada vez que se realiza un chequeo, manteniéndose en ejecución las mr mejores ramas. El valor de mr condiciona el número de procesadores del sistema paralelo necesarios. De este modo, después de la ejecución del primer árbol, se dispone de mr puntos solución distintos, que se usan como puntos iniciales para ejecutar las ramas de mr nuevos árboles. Estos árboles tienen de nuevo tantas ramas como parámetros de amortiguamiento se estén usando, como se ilustra en la figura 6.7(b). El proceso continúa hasta que la solución del problema de optimización haya sido alcanzada.

En este caso, se requerirán $P = mr \cdot n(damp)$ procesos, en lugar de los $n(damp)$ procesadores del método EUAR. Nótese que en la computación del primer árbol $(mr - 1) \cdot n(damp)$ procesadores están desocupados.

6.2.6. Estrategia parámetro de amortiguamiento modificado (EPAM)

Los resultados experimentales vistos en la sección 6.2.4 muestran que la selección de los parámetros de amortiguamiento es importante para alcanzar un alto rendimiento en la ejecución de nuestro algoritmo paralelo. Los tres métodos descritos previamente usan un conjunto fijo de parámetros de amortiguamiento. Sin embargo, se puede añadir un nuevo grado de libertad al método si se deja el valor de los parámetros de amortiguamiento bajo el control del usuario. Dependiendo del tipo de problema, el usuario podrá decidir si usar un conjunto no dirigido de parámetros de amortiguamiento (como sucede con los métodos EMAR, EUAR y ESMR) cuando desconoce el comportamiento del problema, o comenzar con un factor de amortiguamiento adecuado para el problema que se está optimizando, y mejorarlo posteriormente. En este último caso, el método EPAM se basa en modificar el valor del parámetro de amortiguamiento elegido una cierta cantidad x. Si el número de variaciones realizadas es v, se generará un árbol con $2v+1$ ramas. Los parámetros de amortiguamiento propuestos para estas ramas son:

$$damp_i - v \cdot x \ , \ \cdots \ , \ damp_i - x \ , \ damp_i \ , \ damp_i + x \ , \ \cdots \ , \ damp_i + v \cdot x. \quad (6.4)$$

Por lo tanto, el número de procesadores necesarios en este método es:

$$P = mr \cdot (2v + 1). \quad (6.5)$$

El usuario decide, basándose en el tamaño del sistema paralelo que se va a usar, variar mr y v, controlando así el ancho del árbol generado.

El método EPAM se puede usar tras una ejecución inicial del método EUAR. En este caso, el método EUAR se ejecuta con un conjunto de parámetros de amortiguamiento durante algunas iteraciones. Después se utiliza el factor de amortiguamiento que nos ha conducido a la mejor solución usando el método EUAR como amortiguamiento inicial en la ejecución del método EPAM. Este método refinará el valor del parámetro de amortiguamiento mediante variaciones controladas mejorando la eficiencia.

6.3. Resultados comparativos

En esta sección se presenta una comparativa de los resultados obtenidos con las tres últimas heurísticas descritas. Se ha utilizado el mismo banco de pruebas y los mismos valores del parámetro de amortiguamiento que en la validación del método

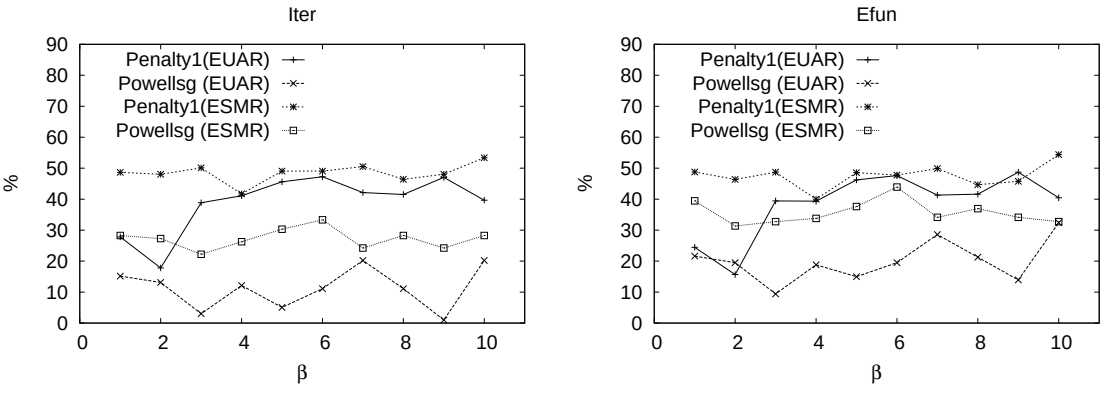


Figura 6.8: Porcentaje de mejora obtenida en número de iteraciones y evaluaciones de la función objetivo por el método EUAR frente al método ESMR para problemas de tamaño $n = 20$.

EUAR. En la figura 6.8 se muestran los resultados alcanzados por el método EUAR frente a los obtenidos por el método ESMR para problemas de tamaño $n = 20$. En este caso, en cada chequeo del método ESMR se han seleccionado las $mr = 10$ ramas con el menor valor de la función objetivo. Los resultados para el problema "Power" no se muestran porque son prácticamente los mismos para los dos métodos. Se trata de un problema tan bien condicionado que la rama seleccionada como la mejor en cada chequeo del método EUAR conduce siempre a la convergencia más rápida. Para los otros problemas, se observa una mejora tanto en la reducción del número total de iteraciones como de evaluaciones de la función objetivo, sobre todo, en el problema "Powellsg", para el que se obtienen los peores resultados con el método EUAR.

Para problemas de tamaño $n = 100$, los peores resultados se obtienen para el problema "Penalty1" como puede observarse en la figura 6.9. En este caso, el único beneficio de seleccionar 10 ramas es que el algoritmo converge en menos iteraciones que las que necesita el código de MINOS; sin embargo, las mejoras obtenidas son muy poco significativas. La razón de que mantener más ramas del árbol vivas no conduzca a una mejor solución está en el hecho de que las ramas seleccionadas son prácticamente iguales y sólo difieren en el valor de sus parámetros de amortiguamiento en los primeros chequeos, de ahí que converjan en un número de iteraciones muy similar. En el problema "Powellsg" se obtienen mejoras más significativas, lo que se explica por el hecho de que en este caso el conjunto de parámetros de amor-

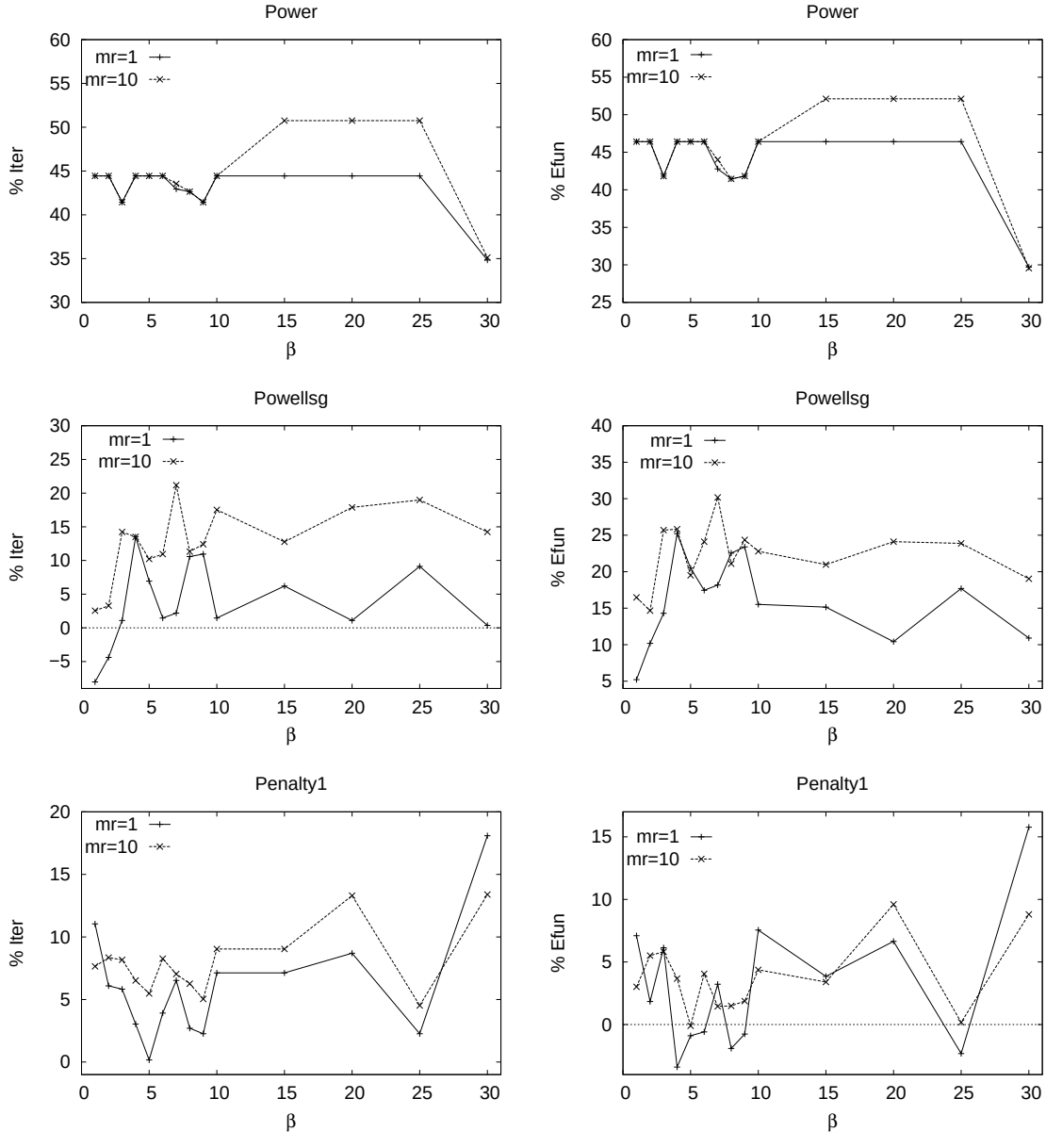


Figura 6.9: Porcentaje de mejora obtenida en número de iteraciones y evaluaciones de la función objetivo por el método EUAR frente al método ESMR para problemas de tamaño $n = 100$.

tiguamiento utilizados lleva a soluciones más dispares en la búsqueda del tamaño de paso, por lo tanto, que las mr ramas seleccionadas son diferentes, produciéndose en el mejor de los casos ($\beta = 7$) una reducción en el número de iteraciones de hasta un orden de magnitud. Por último, para el problema "Power" se observa que se obtiene prácticamente la misma solución con los dos métodos, excepto cuando los chequeos se producen cada 15, 20 ó 25 iteraciones, casos en los que se observa una mejora del rendimiento. El hecho de que para estos chequeos se obtenga una mejor solución es debido a que dos o más parámetros de amortiguamiento de los usados, son igualmente válidos en la selección del tamaño de paso. Sin embargo, sólo uno presenta una mejor convergencia, ya que al mantener el mismo factor de amortiguamiento durante más iteraciones se evitan chequeos intermedios que permitan seguir caminos que finalmente serán más largos. Con $\beta = 30$ se observa que mantener el mismo parámetro de amortiguamiento durante 30 iteraciones seguidas empeora la convergencia del método, en este caso conviene hacer el chequeo con anterioridad para poder ofrecer nuevas ramas al árbol.

La figura 6.10 presenta una comparativa de los tres métodos. En este caso, el método EPAM se ha aplicado manteniendo una única rama viva cada vez que se realiza un chequeo. Inicialmente, se ha utilizado el valor del parámetro de amortiguamiento que conduce a un menor valor de la función objetivo en el método EUAR, y posteriormente este parámetro se ha ido refinando para llegar a los realmente usados en cada caso. La selección de los parámetros x y v, propios del método, se ha realizado empíricamente.

En la figura destaca el diferente comportamiento de los dos problemas. Para el problema "Penalty1" se observan mejoras entre un 40 % y un 50 % tanto en el número total de iteraciones como en el número de evaluaciones de la función objetivo. Cuando la profundidad del árbol es superior a 4 iteraciones los tres métodos conducen a resultados muy similares. El método EPAM muestra un buen comportamiento, muy próximo al del método ESMR. El hecho de que el comportamiento de ambos métodos mejore cuanto mayor sea la profundidad del árbol beneficia al rendimiento del algoritmo paralelo porque implica menos comunicaciones entre los distintos procesadores. El comportamiento en función de la profundidad del árbol es mucho más irregular para el problema "Powellsg". En este caso, el método ESMR presenta un mejor rendimiento en la mayoría de los casos. Sin embargo, cuando la profundidad del árbol es igual a 4, el método EPAM alcanza una mejora superior, por encima del 40 %. Este ejemplo resalta que la determinación de la profundidad del árbol óptima depende del tipo de problema que se quiere optimizar y de la heurística utilizada.

Para resaltar la importancia que tiene la selección de un parámetro de amorti-

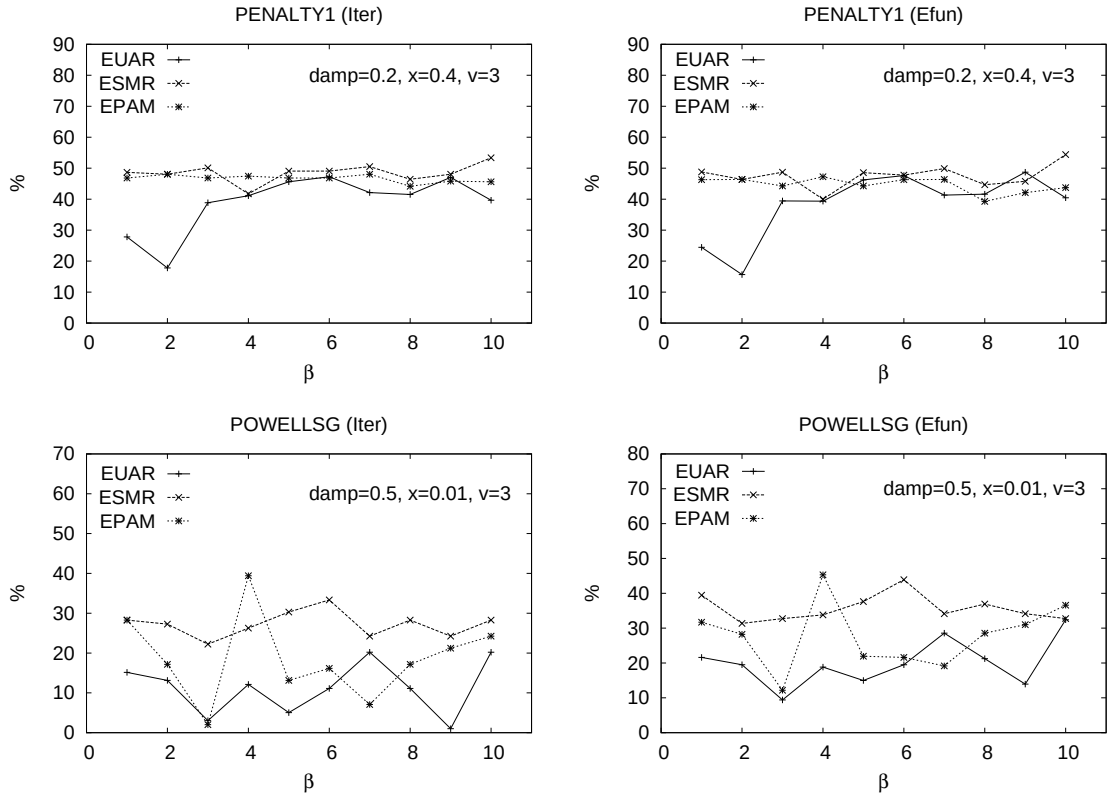


Figura 6.10: Comparativa de los métodos EUAR, ESMR y EPAM para problemas de tamaño $n = 20$.

guamiento adecuado en la convergencia del algoritmo, hemos evaluado el método EPAM para distintos valores de los parámetros v y x, y distintos factores de amortiguamiento para problemas de tamaño $n = 20$. En la figura 6.11 se muestran los resultados obtenidos para el problema "Powellsg", con un factor de amortiguamiento de 0.5 y un parámetro $x = 0.01$, y el problema "Penalty1", con un parámetro de amortiguamiento 0.2 y $x = 0.4$, para distintos valores de v. Para el problema "Penalty1" el factor de amortiguamiento elegido resulta adecuado ya que se obtienen unas reducciones tanto en el número de iteraciones como en el número de evaluaciones de la función objetivo cercanas al 50 % para valores del parámetro v pequeños. Para valores de v superiores a 3 el rendimiento del método empeora significamente, sobre todo, para valores bajos de β. Esto es debido a que cuantas más ramas se generen más posibilidades habrá de que en un determinado chequeo sea seleccio-

nada como óptima una rama que finalmente no sea la más adecuada. En el caso de problema "Powellsg", el comportamiento frente a la profundidad del árbol para distintos valores de v es, de nuevo, muy irregular. Las mejores soluciones se obtienen con $v = 3$ y $v = 4$, pero para cada β la mejor solución varía.

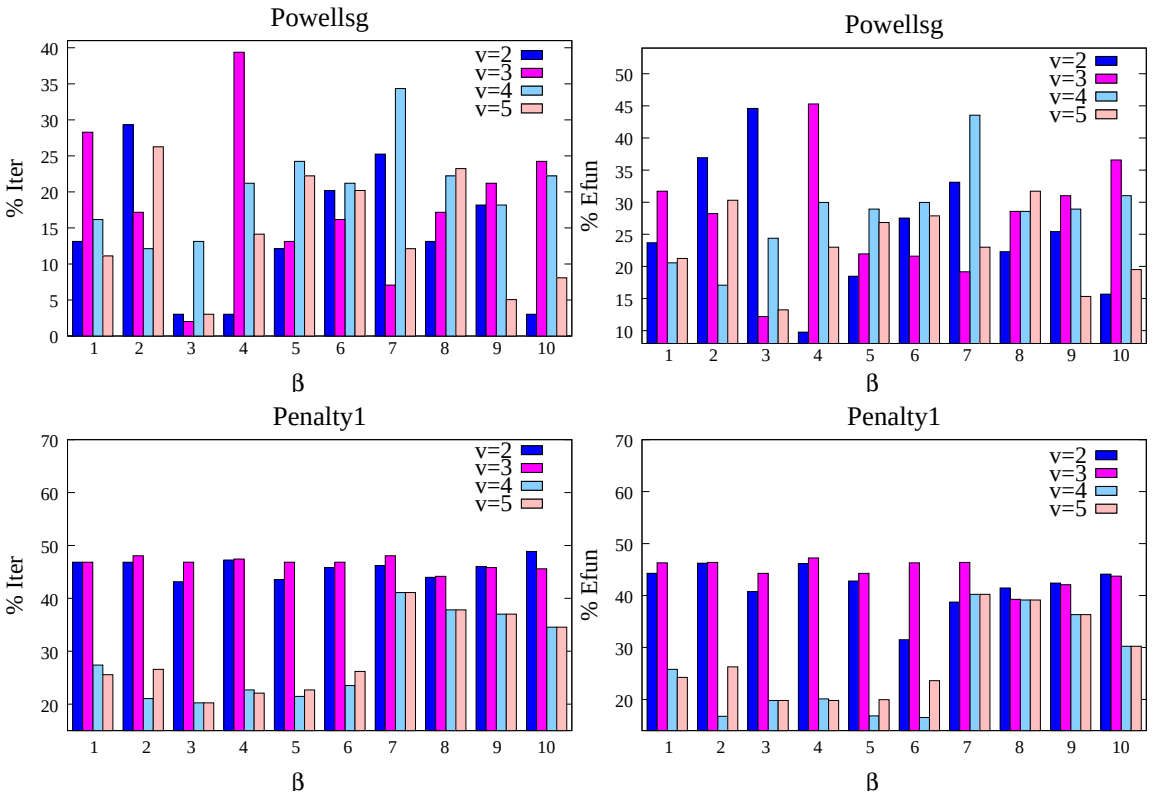


Figura 6.11: Porcentaje de mejora obtenida en número de iteraciones y evaluaciones de la función objetivo por el método EPAM para distintos valores del parámetro v.

En la figura 6.12 se muestran los resultados del método EPAM para el problema "Powellsg", con $v = 3$ y parámetro de amortiguamiento 0.5, y para el problema "Penalty1" con $v = 3$ y factor de amortiguamiento 0.2, utilizando distintos valores del parámetro x con un tamaño de $n = 20$. Los valores de $x = 0.01$ y $x = 0.4$ son la mejor opción para prácticamente todas las profundidades del árbol en el

problema "Penalty1". Comprobando los resultados obtenidos se observa que durante la aplicación de la heurística EPAM, los valores 0.2 ($x = 0.01$) y 0.6 ($x = 0.4$) son los factores de amortiguamiento seleccionados en la mayoría de los chequeos, ofreciendo ambos valores un comportamiento similar en la búsqueda del tamaño de paso. Para valores de β más grandes las diferencias entre las distintas gráficas disminuyen, esto es debido a que el factor de amortiguamiento de 0.2 está presente en todos los casos, independientemente del valor de x. Disminuyendo la frecuencia de los chequeos, se evita la selección de ramas que elijan amortiguamientos diferentes. En el problema "Powellsg" podemos destacar que con $x = 0.01$ se obtienen los mejores resultados cuando la profundidad del árbol es mayor, por lo que este valor de x resulta el más adecuado en este caso.

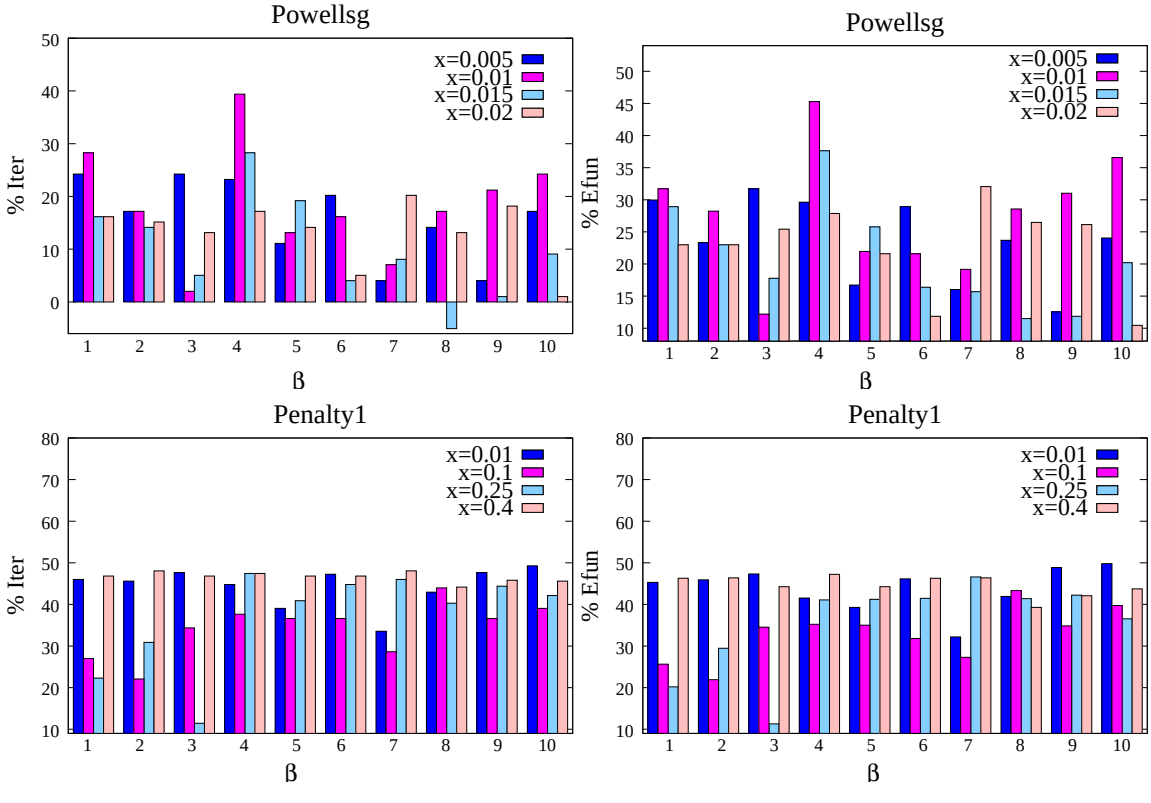


Figura 6.12: Porcentaje de mejora obtenida en número de iteraciones y evaluaciones de la función objetivo por el método EPAM para distintos valores del parámetro x.

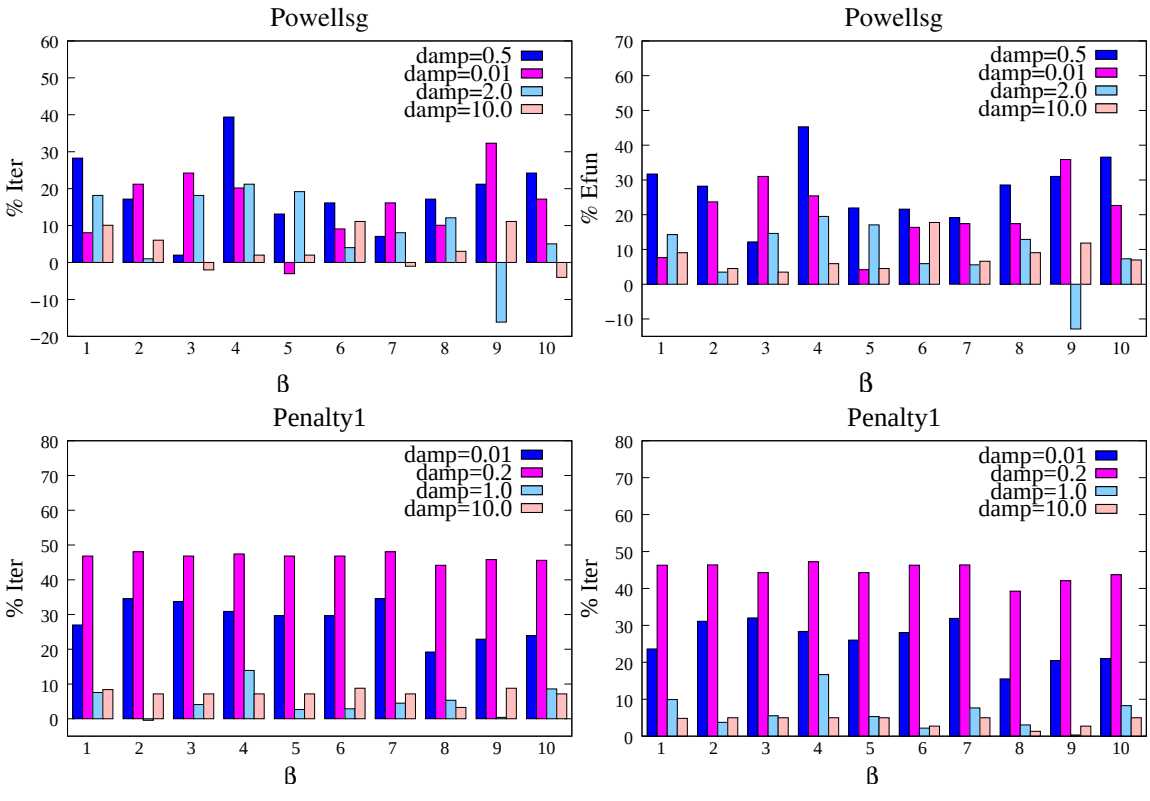


Figura 6.13: Porcentaje de mejora obtenida en número de iteraciones y evaluaciones de la función objetivo por el método EPAM con distintos parámetros de amortiguamiento.

Los resultados obtenidos para distintos parámetros de amortiguamiento usando $x = 0.01$ y $v = 3$ para los problemas "Powellsg" y "Penalty1" se muestran en la figura 6.13. En el problema "Penalty1" se observa que el factor de amortiguamiento de 0.2 que se había elegido inicialmente es el más adecuado, consiguiendo reducciones en torno al 50 %. Si nos alejamos de este valor los resultados empeoran, sobre todo, si se aumenta el valor del parámetro de amortiguamiento. De nuevo, en el caso del problema "Powellsg" los resultados tienen un patrón menos definido. Los factores de amortiguamiento de 0.5 y 0.01 funcionan mejor en la mayor parte de los casos. De lo que sucede con ambos problemas se puede destacar que valores elevados de los parámetros de amortiguamiento conducen a una peor convergencia del algoritmo. Esto es debido a que cuanto mayor es el valor de este parámetro, más cambios

en el tamaño de paso se evitan, pudiéndose descartar tamaños de paso óptimos. Este hecho es más relevante cuanto más abruptamente cambien las soluciones del problema de optimización a resolver.

Finalmente, las figuras 6.14, 6.15 y 6.16 muestran los porcentajes de mejora obtenida por el método EPAM frente al método ESMR para los problemas "Power", "Powellsg" y "Penalty1" respectivamente, con tamaño $n = 100$. En estas gráficas se resalta claramente la importancia que tiene la selección de un factor de amortiguamiento adecuado en la convergencia del algoritmo. En el caso del problema "Power" se consigue un incremento del porcentaje de mejora del número total de iteraciones de un 45 % a un 70 % y del porcentaje de evaluaciones de la función objetivo de un 45 % a un 90 %. Esta reducción en el número de evaluaciones quiere decir que el tamaño de paso ha sido elegido correctamente prácticamente en la primera ocasión en la mayoría de los casos, lo que implica también un menor coste computacional por iteración. En el problema "Powellsg" los resultados de los dos métodos son más parecidos, aunque para mayores profundidades del árbol destaca el método EPAM por conseguir una mayor reducción en el número de evaluaciones. Por último, en el caso del problema "Penalty1" la mejora también es significativa pasando de valores inferiores al 10 % con el método ESMR a valores por encima del 30 % para el método EPAM.

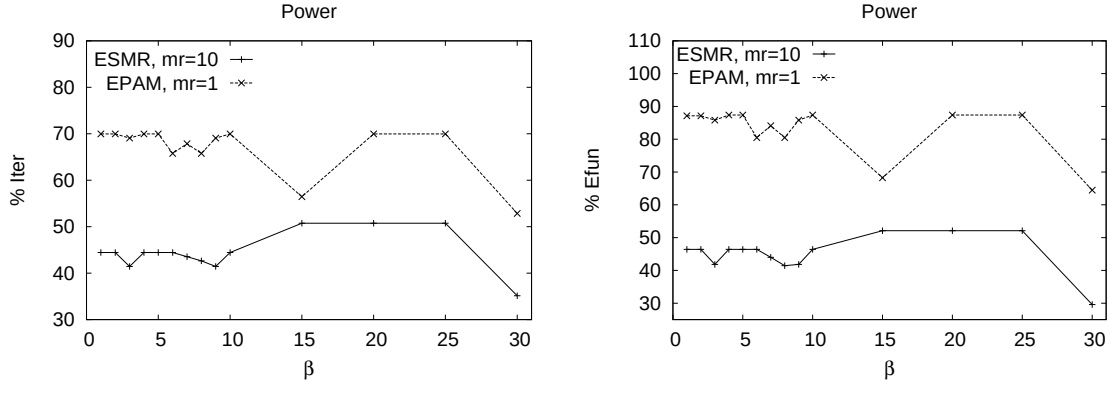


Figura 6.14: Porcentaje de mejora obtenida en número de iteraciones y evaluaciones de la función objetivo por el método EPAM frente al método ESMR para el problema Power con $n = 100$. El parámetro de amortiguamiento usado es 1.0, con $x = 0.5$ y $v = 3$.

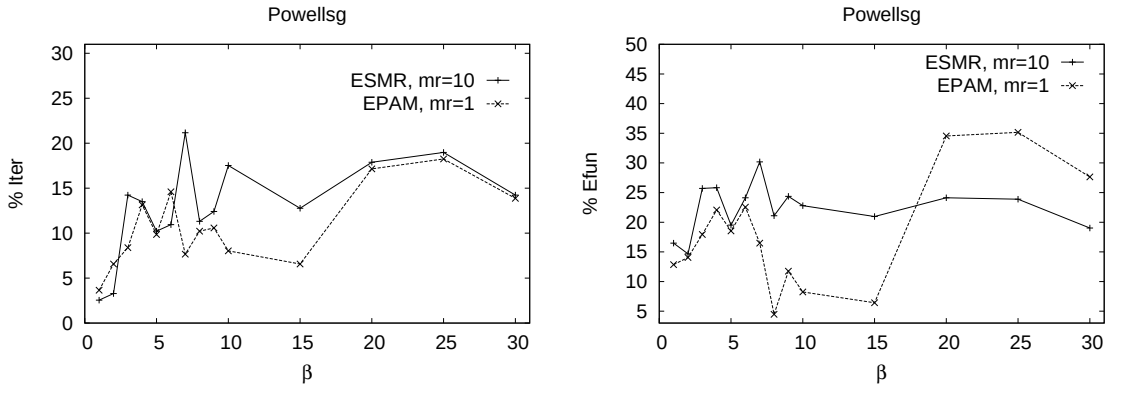


Figura 6.15: Porcentaje de mejora obtenida en número de iteraciones y evaluaciones de la función objetivo por el método EPAM frente al método ESMR para el problema Powellsg con $n = 100$. El parámetro de amortiguamiento usado es 0.2, con $x = 0.05$ y v =3.

6.4. Métodos multidirección y multipaso basados en una métrica variable

Como hemos dicho con anterioridad, recientemente se han propuesto diversas estrategias multipaso y/o multidirección para resolver problemas de optimización no lineal. Es especialmente reseñable el método basado en una métrica variable denominado PVM (*Parallel Variable Metric*) propuesto por Phua [79, 80]. En esta sección lo describiremos en detalle para poder comparar sus resultados con los obtenidos mediante los cuatro métodos propuestos. Hemos elegido la estrategia de Phua por el gran número de puntos en común con la nuestra.

La principal característica de los métodos basados en una métrica variable es la elección de la aproximación de la Hesiana B_k. Estos métodos requieren que esta matriz sea definida positiva y que satisfaga la condición cuasi-Newton

$$B_{k+1}y_k = \xi_k s_k, \quad \xi_k > 0 \tag{6.6}$$

donde $s_k = x_{k+1} - x_k$ y $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$. Uno de los métodos de métrica variable más conocidos es el método BFGS, su actualización viene dada por la

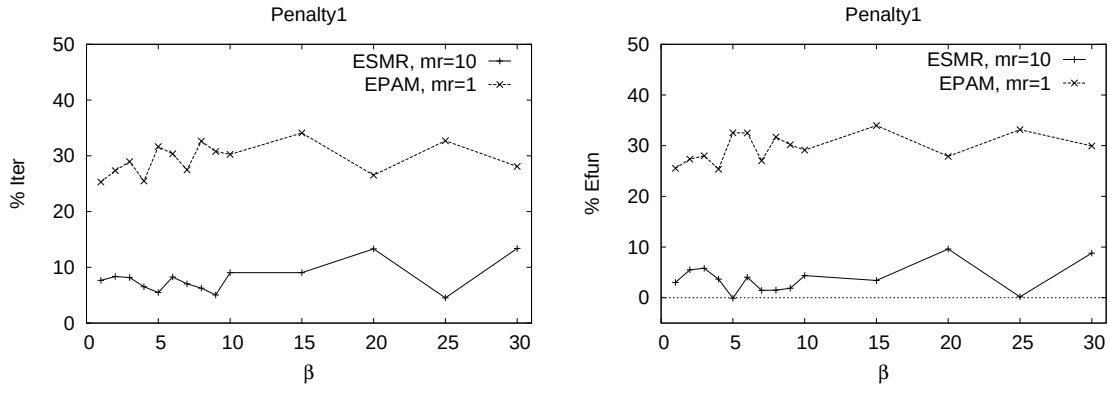


Figura 6.16: Porcentaje de mejora obtenida en número de iteraciones y evaluaciones de la función objetivo por el método EPAM frente al método ESMR para el problema Penalty1 con $n = 100$. El parámetro de amortiguamiento usado es 0.63, con $x = 0.01$ y $v = 2$.

ecuación

$$B_{k+1} = B_k + \frac{1}{s_k^T y_k}((\xi_k + \frac{y_k^T B_k y_k}{s_k^T y_k}) s_k s_k^T - s_k y_k^T B_k - B_k y_k s_k^T) \tag{6.7}$$

con $\xi_k = 1$. Otra métrica alternativa, basada en modelos no cuadráticos, fue propuesta por Biggs [5]. En este caso, el parámetro $\xi_k = 1/t_k$, siendo

$$t_k = \frac{6}{s_k^T y_k}(f(x_k) - f(x_{k+1}) + s_k^T g_{k+1}) - 2 \tag{6.8}$$

Finalmente, otro método para actualizar la aproximación de la matriz Hesiana fue propuesto por Davidon [20]. Este método realiza la actualización añadiendo una matriz simétrica de rango uno, de modo que la actualización de rango uno (SR1) se define como

$$B_{k+1} = B_k + \frac{(s_k - B_k y_k)(s_k - B_k y_k)^T}{(s_k - B_k y_k)^T y_k}. \tag{6.9}$$

Las tres propuestas se pueden generalizar mediante la expresión

$$B_{k+1} = B_k + t_k \frac{s_k s_k^T}{s_k^T y_k} + \frac{[(1-t_k)s_k - B_k y_k][(1-t_k)s_k - B_k y_k]^T}{[(1-t_k)s_k - B_k y_k]^T y_k}, \tag{6.10}$$

donde t_k es un parámetro escalar. Para las actualizaciones SR1 y BFGS $t_k = 0$ y $t_k = \infty$ respectivamente (nótese que estas expresiones de la actualización cuasi-Newton BFGS y SR1 son equivalentes a las dadas en el capítulo 1). El parámetro t_k

es el responsable de la existencia de paralelismo, ya que dependiendo del valor de t_k se define una dirección de búsqueda diferente. Esta es la base de los métodos paralelos multidirección y multipaso basados en una métrica variable. En cada iteración se calculan varias direcciones de búsqueda y para cada dirección de búsqueda distintos tamaños de paso.

El algoritmo PVM propuesto por Phua consta de las siguientes fases:

1. **Inicialización**

   Se elige la matriz identidad como aproximación de la Hesiana ($B_0 = I$).

2. **Cálculo del valor de la función objetivo y del gradiente en el punto solución $\mathbf{x_k}$**

   Definimos $f_k = f(x_k)$ y $g_k = g(x_k)$.

3. **Cálculo de las direcciones de búsqueda en paralelo**

   Supongamos $P_1 > 0$ procesadores disponibles para calcular las direcciones de búsqueda en paralelo.

   $$p_k^{(j)} = -B_k(t_j)g_k, \quad j = 1, 2, \cdots, P_1 \tag{6.11}$$

   donde $B_k(t_j)$ representa cada una de las distintas aproximaciones de la Hesiana, con t_j elegido como $0, \infty, t_k$ en la ecuación 6.10, o cualquier otro valor apropiado.

4. **Cálculo del tamaño de paso en paralelo**

   Se realiza una búsqueda lineal en paralelo en cada una de las direcciones calculadas en el paso anterior. Esta etapa concluye con la selección del mínimo tamaño de paso (λ_k) que verifique las condiciones de Wolfe. Denotamos como p_k^* la dirección de búsqueda para la cual se ha encontrado el paso mínimo.

5. **Cálculo del nuevo punto solución**

   $$\begin{aligned} x_{k+1} &= x_k + \lambda_k p_k^* \\ g_{k+1} &= \nabla f(x_{k+1}) \end{aligned} \tag{6.12}$$

6. **Test de convergencia**

   El algoritmo termina si $||g_{k+1}|| \leq \epsilon \cdot max\{1, ||x_{k+1}||\}$, sino continúa al paso 7.

7. **Cómputo de la actualización de la aproximación de la Hesiana**

$$B_{k+1} = B_{k+1}(\infty) \tag{6.13}$$

Es decir, se aplica la actualización BFGS y se vuelve al paso 2.

Para terminar la descripción de la estrategia propuesta por Phua, es necesario aclarar como se realiza la búsqueda lineal paralela. Supongamos P_2 procesadores disponibles para buscar el tamaño de paso mínimo a lo largo de una dirección $p_k^{(j)}$ $(j = 1, 2, \cdots, P_1)$. El primer paso del algoritmo de búsqueda lineal es elegir los tamaños de paso. Si λ_{max} es el máximo tamaño de paso permitido, se seleccionan los tamaños de tal modo que, para todos ellos

$$0 < \lambda^{(i)} < \lambda_{max}, \tag{6.14}$$

donde $\lambda^{(1)} < \lambda^{(2)} < \cdots < \lambda^{(P_2)}$ son los P_2 tamaños de paso elegidos. A continuación, se computan los correspondientes valores de la función objetivo y del gradiente asociados a cada tamaño de paso. Se busca el tamaño de paso que minimice la función objetivo verificando las condiciones de Wolfe. Una vez encontrado el λ_i óptimo se vuelve al paso 5 del algoritmo PVM. Sino, se aplica una técnica de interpolación cúbica para determinar el tamaño de paso.

6.4.1. Comparativa entre el método PVM y las estrategias multipaso basadas en árboles

Antes de realizar una comparativa entre los métodos descritos, es importante destacar que las estrategias propuestas son únicamente técnicas multipaso, mientras que el método PVM combina técnicas multipaso y multidirección. Para validar su calidad, las estrategias basadas en árboles se comparan únicamente con la técnica multipaso del método PVM, usando como dirección de búsqueda la correspondiente al método BFGS. Nótese que, en cualquier caso, las estrategias multidirección pueden ser aplicadas a nuestra propuesta posteriormente. Además, dichas estrategias no suponen ninguna aportación conceptual importante, pues simplemente se trata de aplicar distintos métodos cuasi-Newton ya existentes.

Para poder realizar una comparación de nuestras estrategias en árbol únicamente con la técnica multipaso del método PVM, hemos aplicado esta técnica sobre el código de optimización MINOS, de modo análogo a como hemos hecho con nuestros códigos multipaso paralelos. La precisión empleada para el criterio de convergencia

Problemas	PVM	EUAR	ESMR	EPAM
	Iter/Efun	Iter/Efun	Iter/Efun	Iter/Efun
Power				
$n = 20$	84/348	26/66	26/66	26/66
$n = 100$	287/2006	185/442	185/442	100/106
Penalty				
$n = 20$	305/987	259/658	228/585	254/688
$n = 100$	1069/3255	942/2431	996/2632	774/1929
Powellsg				
$n = 20$	122/386	79/194	66/161	60/157
$n = 100$	274/2001	237/617	216/576	224/535

Tabla 6.2: Resultados obtenidos por el método PVM multipaso y nuestras propuestas.

es la misma que la usada por las estrategias basadas en árboles, $\epsilon = 10^{-6}$. La comparativa se realiza con el método PVM, que calcula concurrentemente 3 tamaños de paso. Los tamaños de paso elegidos son $\lambda^{(1)} = 1/2$, $\lambda^{(2)} = 1$ y $\lambda^{(3)} = 2$. Los resultados obtenidos se presentan en las tablas 6.2 y 6.3 en términos absolutos, es decir, no mostramos porcentajes de mejora, sino del número de iteraciones y de evaluaciones de la función objetivo necesarias para obtener la solución de cada problema de optimización. La tabla 6.2 muestra una comparativa del método PVM con el resultado obtenido para la secuencia de chequeo óptima en cada una de nuestras propuestas. Asimismo, puesto que en el método PVM se selecciona el mejor tamaño de paso cada iteración, en la tabla 6.3 se presentan los resultados obtenidos por las estrategias EUAR, ESMR y EPAM para $\beta = 1$. Del análisis de ambas tablas hay que destacar que cualquiera de nuestros métodos obtiene mejores resultados que el método PVM multipaso cuando se selecciona la frecuencia de chequeo adecuada, sobre todo, en el caso del problema "Power", para el que se ha utilizado un factor de amortiguamiento óptimo. Además, se observa que en el método PVM multipaso se realizan muchas evaluaciones de la función objetivo para todos los problemas estudiados. Lo que indica que los valores de tamaño de paso seleccionados no son la mejor solución en la mayoría de las iteraciones, siendo necesaria la aplicación de sucesivas interpolaciones hasta determinar un valor de tamaño de paso adecuado. Este hecho resalta el beneficio que supone que nuestras estrategias propongan varios

factores de amortiguamiento distintos, que van a influir positivamente en el cómputo del tamaño de paso. No se proponen distintos tamaños de paso al azar, como ocurre en el método PVM, se trata de ajustar en alguna medida la propuesta del tamaño de paso al problema de optimización a través del parámetro de amortiguamiento. El valor del factor de amortiguamiento limita la búsqueda del tamaño de paso para evitar evaluaciones de la función objetivo en un entorno demasiado próximo del punto solución. Si la solución del problema a optimizar no cambia bruscamente, un valor alto del parámetro de amortiguamiento es adecuado, y se reducirán tanto el número de iteraciones como de evaluaciones de la función objetivo. Si por el contrario, en la solución del problema se producen cambios bruscos, será necesario un factor de amortiguamiento más pequeño. En este trabajo, los parámetros de amortiguamiento usados se han buscado de forma empírica, con unos valores iniciales y posteriores refinamientos hasta obtener los valores finalmente usados.

Problemas	PVM	EUAR	ESMR	EPAM
	Iter/Efun	Iter/Efun	Iter/Efun	Iter/Efun
Power				
$n = 20$	84/348	26/66	26/66	26/66
$n = 100$	287/2006	185/442	185/442	100/106
Penalty				
$n = 20$	305/987	353/969	252/657	260/689
$n = 100$	1069/3255	1023/2681	1062/2799	859/2149
Powellsg				
$n = 20$	122/386	84/225	71/191	71/196
$n = 100$	274/2001	296/782	267/689	261/735

Tabla 6.3: Resultados obtenidos por el método PVM multipaso y nuestras propuestas con $\beta = 1$.

También se debe destacar la importancia que tiene la profundidad del árbol en nuestras estrategias. Dependiendo del problema a optimizar y de los parámetros de amortiguamiento seleccionados una determinada profundidad del árbol será óptima. Así, en la tabla 6.3 se observa que en algunos casos si la profundidad del árbol es 1 la estrategia EUAR obtiene peores resultados que el método PVM multipaso. Esta

deficiencia se solventa en las otras dos estrategias, o bien aumentando el ancho del árbol, seleccionando más ramas en cada iteración, o bien utilizando un parámetro de amortiguamiento más adecuado. En cualquier caso, en todas nuestras propuestas, y en prácticamente todos los casos, se verifica que las mayores reducciones en el número de iteraciones y de evaluaciones de la función objetivo se producen para un valor de la profundidad del árbol relativamente alto, lo que implica un menor número de comunicaciones, y por tanto, un mayor rendimiento del algoritmo paralelo.

Conclusiones, principales aportaciones y líneas futuras

Los problemas de optimización han sido muy estudiados en los últimos años por su aplicación en múltiples áreas. Con frecuencia, sobre todo si se trata de problemas de optimización no lineal, su resolución implica un alto coste computacional. El principal objetivo de esta tesis ha sido el estudio de las estrategias de paralelización que mejor se adecúen a la ejecución eficiente de las rutinas de uso frecuente en métodos de optimización no lineal, y en particular, en algoritmos cuasi-Newton, sobre sistemas de memoria distribuida. Para evaluar la eficiencia de las soluciones propuestas se ha utilizado el código de optimización MINOS como banco de pruebas. Este código se caracteriza por su robustez y por incluir todas las rutinas tratadas.

Las principales aportaciones de esta memoria se pueden estructurar en los siguientes puntos:

- Se ha realizado un análisis computacional detallado del método de gradiente reducido, basado en un conjunto activo de restricciones, utilizado en optimización no lineal. Se ha comprobado que el algoritmo cuasi-Newton, que se usa en este método para tratar las no linealidades, consume más del 90 % del tiempo total de computación en la mayoría de los casos. Dentro de este método, se han determinado las computaciones más costosas en cada iteración: la actualización de la matriz aproximación de la Hesiana, el cálculo de la dirección de búsqueda y el cálculo del tamaño de paso. De todas ellas, la primera destaca por su alto coste computacional, más del 80 % del total, lo que justifica el esfuerzo en su paralelización.

- Se han estudiado las dependencias de datos de los algoritmos que realizan estas computaciones, mostrando su bajo grado de paralelismo. En particular, el algoritmo de actualización de la Hesiana consiste en la aplicación de dos rotaciones planas, o rotaciones de Givens, que están implementadas mediante dos

lazos anidados. La rotación global hacia adelante es la que presenta dependencias más restrictivas, y en ella el único grado de paralelismo se encuentra en el lazo interno. La distribución de datos y la estrategia que se ha implementado para extraer el máximo paralelismo, se aplican eficazmente tanto a la rotación global hacia atrás, como a las resoluciones triangulares, que constituyen el núcleo fundamental del cálculo de la dirección de búsqueda, ya que aunque presentan dependencias más débiles, su flujo de datos es similar.

- Para validar las estrategias paralelas propuestas hemos utilizado dos sistemas de computación: un multiprocesador de memoria distribuida, el AP3000 de Fujitsu, con una red de interconexión propia y dedicada, la AP-Net, y un *cluster*, el *beowulf* del CESGA, elegido para validar su red de interconexión, la Myrinet 2000. Estos dos computadores constituyen dos sistemas muy diferentes, uno de ellos con una red de muy alta capacidad frente al poder computacional de los nodos, y el otro con una red de una capacidad baja frente a la velocidad de los procesadores. Consideramos que la selección de estos dos sistemas constituye una buena opción para caracterizar el comportamiento de nuestros códigos paralelos. Hemos evaluado el comportamiento de las estrategias propuestas en ambos sistemas, comprobando que la relación entre el coste de las comunicaciones y el coste de las computaciones en el *cluster beowulf* es mayor que en el AP3000, de ahí que las fuertes dependencias de datos de las rutinas tratadas adquieran una mayor relevancia en el rendimiento de nuestros códigos paralelos en el *beowulf*.

- En el caso de los sistemas de memoria distribuida los factores que más influyen en la eficiencia del programa paralelo son las comunicaciones y el balanceo de la carga. Proponemos una estrategia mixta en la que algunos datos son computados secuencialmente por todos los procesadores, y otros se calculan de forma concurrente. Esta estrategia minimiza el número de comunicaciones y los tiempos de espera. Además, proponemos una distribución cíclica por columnas que garantiza, al menos inicialmente, el balanceo de los datos.

- En muchos casos tras la ejecución de varias iteraciones del método paralelo, el sistema se encuentra desbalanceado. Para solucionar este problema, hemos propuesto distintas heurísticas de redistribución de la carga que tratan de minimizar el número máximo de envíos o de recepciones por procesador. Se ha estudiado el coste computacional que supondría la existencia de un cierto desbalanceo y el coste de una redistribución de la carga, tanto de forma teórica como experimental, para cada uno de los sistemas de computación utilizados.

Se ha comprobado la fiabilidad de nuestras estimaciones para los posibles casos reales, sobre todo, para el AP3000. La finalidad de este estudio ha sido poder decidir a *priori* cuando compensa realizar una redistribución que balancee la carga computacional.

- Se ha propuesto una alternativa de paralelización que trata de disminuir el tiempo total de computación reduciendo el número total de iteraciones necesarias para la convergencia del método, y el número de evaluaciones de la función objetivo. Las estrategias desarrolladas se basan en métodos multipaso. En cada iteración, se proponen distintos tamaños de paso para obtener el nuevo punto solución. Para definir los diferentes tamaños de paso se seleccionan distintos parámetros de amortiguamiento. El valor de estos parámetros está íntimamente relacionado con la función objetivo del problema a optimizar.

- Los resultados obtenidos por las estrategias propuestas han sido comparados con los de otros métodos o heurísticas. En todos los casos, el rendimiento de nuestras propuestas es superior, ya que han sido diseñadas específicamente para adaptarse a las características de cada problema concreto. En el caso de las rutinas más costosas de un método cuasi-Newton, la eficiencia de nuestras implementaciones es alta para problemas de gran tamaño, sobre todo, para la actualización de la Hesiana, debido a su mayor número de operaciones en punto flotante, y del AP3000, ya que en este sistema el coste de las comunicaciones es menos significativo. En las estrategias multipaso, si se selecciona el parámetro de amortiguamiento adecuado y el problema a optimizar está bien condicionado, los resultados muestran reducciones de hasta un 75 % en el número de iteraciones y un 80 % en el número de evaluaciones de la función objetivo.

Existen diversas líneas de investigación que se pueden derivar del trabajo de esta tesis. A continuación enunciamos algunas de las más relevantes.

- Otra de las arquitecturas representativas en el mundo de la supercomputación son los *clusters* de SMP de tamaño moderado. Para sistemas de este tipo se podrían desarrollar estrategias híbridas, que combinen programación en memoria compartida (usando, por ejemplo, directivas de OpenMP) y programación en memoria distribuida (paso de mensajes), aplicadas a las rutinas más costosas de un método cuasi-Newton.

- Otra posible línea de investigación es la implementación de técnicas multipaso en arquitecturas débilmente acopladas como los *Grids*. Estas arquitecturas se

caracterizan fundamentalmente por:

- Heterogeneidad. Los diferentes sistemas que conforman el *Grid* tienen distintas CPUs, por lo tanto, en nuestro caso, no todos terminarán de computar el nuevo punto solución al mismo tiempo. Asimismo, la existencia de *clusters* dentro del *Grid* acentúa esta diferencia computacional, porque en este caso habría que tener en cuenta no sólo los tiempos de computación propios de la heurística sino también los tiempos de espera en cola.
- Dinamismo. Dependiendo del momento, un sistema estará más o menos cargado, lo que implica que el tiempo de ejecución de una tarea no sea siempre el mismo.
- Tolerancia a fallos. Si un sistema falla, no se pondrán utilizar las computaciones que se le habían asignado.

Las heurísticas multipaso tendrían que adaptarse a estas nuevas condiciones. Por lo tanto, se deberían desarrollar implementaciones perfectamente desacopladas. Una opción es mantener la estructura de un único árbol descrita en este trabajo pero añadiendo estrategias que permitan no tener en cuenta algunas ramas, en caso de fallo o de tiempo de computación excesivo, e incluso la posibilidad de retomar el trabajo asociado a las ramas descartadas en otros sistemas del *Grid* menos cargados. Otra opción, consistiría en buscar muchos más tamaños de paso, pero no dentro del mismo árbol, sino generando múltiples árboles totalmente independientes.

- Finalmente, las estrategias multipaso se pueden completar con técnicas multidirección. Los algoritmos propuestos generarían varias direcciones de búsqueda del nuevo punto solución en cada iteración, entre ellas las direcciones cuasi-Newton usadas en el método PVM. El incremento de las posibilidades de búsqueda del punto solución óptimo en cada iteración conducirá en la mayor parte de los casos a una convergencia más rápida del algoritmo.

Bibliografía

[1] J. Abadie and Carpentier. Generalization of the wolfe reduced gradient method to the case of nonlinear constraints. In R. Fletcher, editor, *Optimization*. Academic Press, New York, 1969.

[2] D. Anderson, J. Chase, S. Gadde, A. Gallatim, K. Yocum, and M. Feeley. Cheating the i/o bottleneck: Network storage with trapeze/myrinet. In *Proceedings of the 1998 Usenix Technical Conference*, June 1998.

[3] M. Baker. Cluster computing white paper. TFCC http://www.ieeetfcc.org/.

[4] Benchmarks subsistema beowulf. http://www.cesga.es/content/view/409/42/lang,en/.

[5] M. C. Biggs. A note on minimization algorithms which make use of non-quadratic properties of the objective function. *J. Inst. Maths. Applics.*, (12):337–338, 1973.

[6] V. Blanco. *Análisis, Predicción y Visualización del Rendimiento de Métodos iterativos en HPF y MPI*. PhD thesis, Facultad de Ciencias Físicas, Universidad de Santiago de Compostela, 2002.

[7] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J.Ñ. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro Chips, Systems, Software and Applications*, 15(1):29–36, 1995.

[8] I. Bongartz, A. R. Conn, N. I. M. Gould, and Ph. L. Toint. Cute: Constrained and unconstrained testing enviroment. *ACM Transactions on Mathematical Software*, 21(1):123–160, 1995.

[9] I. Bongartz, A. R. Conn, N. I. M. Gould, and PH. L. Toint. Cute: Constrained and unconstrained testing enviroment. *ACM Transactions on Mathematical Software*, 21(1):123–160, 1995.

[10] A. Brooke, D. Kendrick, and A. Meeraus. *GAMS: A User's Guide.* The Scientific Press, South San Francisco, 1988.

[11] R. H. Byrd, R. B. Schnabel, and G. A. Shultz. Using parallel function evaluations to improve hessian approximation for unconstrained optimization. Technical Report CU-CS-361-87, University of Colorado at Boulder, Department of Computer Science, 1987.

[12] R. H. Byrd, R. B. Schnabel, and G. A. Shultz. Parallel quasi-newton methods for unconstrained optimization. *Mathematical Programming*, (42):273–306, 1988.

[13] A. Capara, H. Kepler, and U. Pferschy. The multiple subset problem. Technical report, Faculty of Economics, University of Graz, 1998.

[14] Centro de Supercomputación de Galicia. http://www.cesga.es.

[15] E. K. P. Chong and S. H. Zak. *An Introduction to Optimization.* John Wiley and Sons, Inc., New York, second edition, 2001.

[16] A. Cleary. Private communication. Technical report, Dept. of Applied Mathematics; University of Virginia, Charlottesville, Virginia, Octubre 1986.

[17] Jr. E. G. Coffman, M. R. Carey, and D. S. Johson. *Algorithm Design for Computer System Design*, chapter Approximation algorithms for Bin-Packing – An Updated Survey. Springer-Verlag, 1984.

[18] Compaq, Intel, and Microsoft. Virtual interface architecture specification. http://www.rimonbarr.com/repository/cs614/san_10.pdf, December 1997. 1.0 edition.

[19] D. E. Culler, J. Pal Singh, and A. Gupta. *Parallel Computer Architecture. A Hardware/Software Approach.* Morgan Kaufmann Publishers Inc., San Francisco, California, 1999.

[20] W. C. Davidon. Variable metric method for minimization. *Computer Journal*, (10):406–410, 1968.

[21] A. Dekkers and E. Aarts. Global optimization and simulated annealing. *Mathematical Programming*, 50(1):367–393, 1991.

[22] A. Emmen. Compaq introduces new alpha-based parallel servers tuned for high-performance technical computing. In *Primeur The monthly news service for the European HPCN Community.* Houston, June 1999.

[23] D. J. Evans and R. C. Dunbar. The parallel solution of triangular systems of equations. *IEEE Transactions on Computers*, C-32(2):201–204, February 1983.

[24] I. T. Foster. *Designing and Building Parallel Programs. Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Publishing Company, 1995.

[25] R. Fourer, D. M. Gay, and B. W. Kernghan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, 1993.

[26] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. W. Tseng, and M. Wu. Fortran D language specification. Technical report, Dept. Computer Science. Rice University, 1990.

[27] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D.W. Walker. *Solving Problems on Concurrent Proccessors*, volume I. Prentice–Hall, New Jersey, 1988.

[28] M. R. Garey and D. S Johnson. *Analysis and Design of Algorithms in Combinatorial Optimization*, chapter Approximation algorithms for bin-packing problems: A survey, pages 147–172. Springer-Verlag, New York, 1981.

[29] M. Gasca and T. Sauer. Multivariate polynomial interpolation. *Advances in Comp. Math.*, 12:377–410, 2000.

[30] A. Geist, A. Beguellin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 2: User's guide and reference manual. Technical report, Oak Ridge National Laboratory, 1994.

[31] A. Geist, A. Beguellin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM.* The MIT Press, tercera edition, 1996.

[32] K. Ghouas, K. Omang, and H. Bugge. Via over sci- consequences of a zero copy implementation, and comparison with via over myrinet. In *Proceedings of Communication Architectures for Clusters*, pages 1632–1639, San Francisco, April 2001.

[33] P. E. Gill, G. H. Golub, W. Murray, and M. A. Saunders. Methods for modifying matrix factorizations. *Mathematics of Computation*, 28(126):505–535, April 1974.

[34] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization.* Academic Press, 1981.

[35] P. E. Gill, W. Murray, and M. H. Wright. *Numerical Linear Algebra and Optimization*, volume 1. Addison-Wesley Publishing Company, 1991.

[36] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem (part i). *Operational Research*, 9:849–859, 1961.

[37] F. Glover. Tabu search-part i. *ORSA J. on Computing*, (1):190–206, 1989.

[38] F. Glover. Tabu search-part ii. *ORSA J. on Computing*, (2):4–32, 1990.

[39] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley, New York, 1989.

[40] G. H. Golub and C. F. Van Loan. *Matrix Computations (3ª edición).* John Hopkins University Press, 1996.

[41] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* Scientific and Engineering Computation Series. MIT Press, 1994.

[42] D. J. Haglin and R. W. Ford. The message-minimizing load redistribution problem. *Journal of Universal Computer Science*, 7(4):291–306, 2001.

[43] M. T. Heath. *Scientific Computing. An Introductory survey.* McGraw-Hill Higher Education, 1996.

[44] M. T. Heath and C. H. Romine. Parallel solution of triangular systems on distributed-memory multiprocessors. *SIAM J. Sci. Statist. Computing*, 9(3):558–588, May 1988.

[45] High Performance Forum. High Performance Fortran language specification, 1996.

[46] H. Ishihata, M. Takahashi, and H. Sato. Hardware of AP3000 Scalar Parallel Server. *FUJITSU Sci. Tech. J.*, 33(1):24–30, June 1997.

[47] Y. Kee and S. Ha. A robust dynamic load-balancing scheme for data parallel application on multiprocessors systems. *Journal of Electrical Engineering and Information Science*, 4(1):105–114, February 1999.

[48] C. Kenyon and M. Mitzenmacher. Linear waste of best fit bin packing on skewed distributions. *Random Struct. Algorithms*, 20(3):441–464, 2002.

[49] S. Kirkpatrick, D. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 20(1):671–680, 1983.

[50] E. J. Kontoghiorghes. Greedy givens algorithms for computing the rank-k updating of the QR decomposition. *Parallel Computing*, 28(9):1257–1273, 2002.

[51] D. J. Kuck. Parallel processing of ordinary programs. *Advances in Computers*, 15:119–179, 1976.

[52] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc., California, 1994.

[53] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. *Computer Architecture News*, 25(2):241–252, 1997.

[54] G. Li and T. F. Coleman. A new method for solving triangular systems on distributed-memory message-passing multiprocessors. *SIAM J. Sci. Stat. Computing*, 10(2):382–396, March 1989.

[55] I. M. Llorente, F. Tirado, and L. Vázquez. Some aspects about the scalability of scientific applications on parallel computers. *Parallel Computing*, (22):1169–1195, 1997.

[56] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations.* Wiley Interscience Series in Discrete Mathematics and Optimization, Chichester, West Sussex, England, 1990.

[57] M. J. Martín, I. Pardines, and F. F. Rivera. Scheduling of algorithms based on elimination trees on numa systems. *Lecture Notes in Computer Science*, (1685):1068–1072, 1999.

[58] M. J. Martín, I. Pardines, and F. F. Rivera. *Parallel Computing. Fundamentals and Applications*, chapter Left-Looking Strategy for the Sparse Modified

Cholesky Factorization on NUMA Multiprocessors, pages 342–349. Imperial College Press, London, 2000.

[59] Message Passing Interface Forum. MPI: A Message Passing Interface standard. Technical Report CS-94-230, Computer Science Department, University of Tennessee, Knoxville, TN, 1994.

[60] E. Mijangos. On the efficiency of multiplier methods for nonlinear network problems with nonlinear constraints. *IMA Journal of Management Mathematics*, 15(3):211–226, 2004.

[61] P. Mitra, D. Payne, L. Schuler, R. van de Geijn, and J. Watts. Fast collective communication libraries, please. In *Proceedings of the Intel Supercomputing User's Group Meeting*, June 1995. ftp://ftp.cs.utexas.edu/pub/techreports/tr95-22.ps.Z.

[62] J. J. Moré and D. C. Sorensen. Computing a trust region step. *SIAM Journal on Scientific and Statistical Computing*, (4):553–572, 1983.

[63] B. A. Murtagh. *Advanced Linear Programming: Computation and Practice.* McGraw-Hill, 1981.

[64] B. A. Murtagh and M. A. Saunders. Large-scale linearly constrained optimization. *Mathematical Programming*, (14):41–72, 1978.

[65] B. A. Murtagh and M. A. Saunders. A projected lagrangian algorithm and its implementation for sparse nonlinear constraints. *Mathematical Programming*, (16):84–117, 1982.

[66] B. A. Murtagh and M. A. Saunders. Minos 5.4 user's guide (revised). Technical Report SOL 83-20R, Department of operations Research, Stanford University, 1993. Revised 1995.

[67] Myricom Inc. *Guide to Myrinet-2000 Switches and Switch Network*, 2001. Revision 27.

[68] Pallas GmbH. Pallas mpi benchmarks- pmb, part mpi-1. Available at http://www.pallas.com.

[69] I. Pardines. Spanning tree techniques for multi-step parallel optimization methods. In *Proceedings of abstracts of the Matrices in Statistics and Optimization Workshop*, page 3, France, October 2004. CIRM.

[70] I. Pardines, A. J. García-Loureiro, and F. F. Rivera. Algoritmo por bloques para el cálculo paralelo de rotaciones planas. In *Actas de las XI Jornadas de Paralelismo*, pages 283–288, Granada, Septiembre 2000.

[71] I. Pardines, M. Martín, M. Amor, and F. F. Rivera. *Parallel Computing: Fundamentals, Applications and New Directions*, chapter Static Mapping of the Multifrontal Method Applied to the Modified Cholesky Factorization for Sparse Matrices, pages 731–734. Elsevier Science, Ansterdam, 1998.

[72] I. Pardines, J. J. Pombo, and F. F. Rivera. Parallel algorithm for backward and forward sweeps of plane rotations. In *Proceedings of the IASTED International Conference*, pages 418–423, Innsbruck, Austria, February 2001.

[73] I. Pardines and F. F. Rivera. *Parallel Computing: Advances and Current Issues*, chapter Parallel Quasi-Newton Optimization on Distributed Memory Multiprocessors, pages 338–345. Imperial College Press, London, 2002.

[74] I. Pardines and F. F. Rivera. Efficient dynamic load balancing strategies for parallel active set optimization methods. *Euro-Par 2003 Parallel Processing. Lecture Notes in Computer Science*, (290):206–211, August 2003.

[75] I. Pardines and F. F. Rivera. Minimizing the load redistribution cost in cluster architectures. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 326–331, A Coruña, Spain, February 2004.

[76] I. Pardines, D. E. Singh, and F. F. Rivera. Parallel algorithm for nonlinearly unconstrained optimization based on parametric trees. In *Parallel Computing Conference*, September 2005. Pendiente de publicación.

[77] D. M. Pase, T. McDonald, and A. Meltzer. The CRAFT Fortran programming model. *Scientific Programming*, 3:227–253, 1994.

[78] J. Peinado. *Resolución paralela de sistemas de ecuaciones no lineales.* PhD thesis, Universidad Politécnica de Valencia, Departamento de Sistemas Informáticos y Computación, 2003.

[79] K. H. Phua. Multi-directional parallel algorithms for unconstrained optimization. *Optimization*, (38):107–125, 1996.

[80] K. H. Phua and R. Setiono. *Optimization Techniques and Applications*, chapter Multi-step, multi-directional parallel algorithms for unconstrained optimization, pages 481–487. World Scientific, Singapore, 1992.

[81] Relación de los supercomputadores más veloces. http://www.top500.org.

[82] J. B. Rosen, O. L. Mangasarian, and K. Ritter. *Nonlinear Programming.* Academic Press, London and New York, 1970.

[83] R. B. Schnabel. Concurrent function evaluations in local and global optimization. *Computer Methods in Applied Mechanics and Engineering*, 64:537–552, 1987.

[84] S. L. Scott. Synchronization and communication in the T3E multiprocessor. Technical report, Cray Research, 1996.

[85] C. Sun. Parallel solution of sparse linear least squares problems on distributed-memory multiprocessors. Technical Report CTC95TR212, Advanced Computing Research Institute, Cornell Theory Center, Cornell University, Ithaca, NY 14853-3801, May 1995.

[86] Thinking Machine Corporation. CM Fortran language reference manual, 1994.

[87] P. J. M. van Laarhoven. Parallel variable metric methods for unconstrained optimization. *Mathematical Programming*, (33):68–81, 1985.

[88] Web oficial de HP. http://www.hp.com.

[89] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice–Hall, 1999.

[90] P. Wolfe. Methods of nonlinear programming. In R. L. Graves and P. Wolfe, editors, *Recent Advances in Mathematical Programming*, pages 67–86. McGraw-Hill, New York, 1963.

[91] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – a language specification. Technical report, Austrian Center for Parallel Computation. Univ. Vienna, 1992.