Predicción de mínimos en algoritmos paralelos de Optimización Global Intervalar

---

PROYECTO FIN DE CARRERA
INGENIERÍA INFORMÁTICA

Jose Luis Berenguel Gómez
Septiembre 2007

Dirigida por:
Leocadio González Casado

Dpto. de Arquitectura de Computadores y Electrónica
Universidad de Almería

Índice general

Índice de figuras

Índice de tablas

Lista de Algoritmos

Prólogo

La optimización persigue encontrar la mejor solución a un problema, de forma que se minimice (o maximice) el valor de una función objetivo, posiblemente sujeta a una serie de restricciones sobre las posibles soluciones.

En la búsqueda de esta solución óptima que minimice nuestra función objetivo, nos encontraremos (normalmente) con numerosos mínimos locales. Estas soluciones locales no son de utilidad en la mayoría de los problemas a los que nos enfrentamos puesto que, las diferencias de la función objetivo en los mínimos locales y globales pueden ser muy grandes.

En la bibliografía existen numerosos algoritmos de Optimización Global basados en Aritmética de Intervalos que resuelven el problema de la búsqueda de una solución óptima, incluso existen algoritmos paralelos para tratar de optimizar la ejecución de los algoritmos anteriores mediante el uso de más de un procesador.

El objetivo de este proyecto es predecir cuántos mínimos globales existen en el problema tratado y determinar las regiones o zonas de atracción que pertenecen a los mismos. De este modo, podremos asignar estas regiones a diferentes procesadores para tratar de mejorar la eficiencia de nuestro algoritmo paralelo de Optimización Global. Como los algoritmos de Optimización Global paralelos no tienen información acerca del número de mínimos que puede haber en el problema a evaluar, podemos estar utilizando más procesadores de los necesarios, con el consiguiente aumento en el tráfico de información paralela. Si somos capaces de predecir, de forma más o menos aproximada, el número de mínimos del problema tratado, sabremos cuántos procesadores debemos usar durante la resolución del problema, para minimizar el tiempo consumido en el envío de información y datos entre los diferentes procesadores mejorando el rendimiento de los algoritmos de optimización.

Por tanto, la predicción de mínimos se puede considerar un complemento a los algoritmos paralelos de Optimización Global para tratar de mejorar la eficiencia de los mismos, determinando en cada momento y para cada problema, cuál es el número óptimo de procesadores que deben ser utilizados en paralelo.

Parte I

Introducción general

Capítulo 1

Introducción

1.1. Descripción del proyecto

Este proyecto persigue la implementación de un algoritmo paralelo de Optimización Global que base el balanceo de la carga en la predicción de mínimos. Esta predicción consiste en crear regiones alrededor de los mínimos globales y locales de la función objetivo, cuya localización no conocemos a priori. La principal dificultad consistirá en establecer el mecanismo que nos permita decidir cuáles son las zonas donde es más probable que se encuentren los mínimos, y por tanto, establecer el número de regiones que habrá en un momento determinado de la ejecución, tratando de aproximar las regiones a los mínimos existentes con el objeto de que cada procesador evalúe una de las regiones predichas.

Las características de las **funciones test** utilizadas para comprobar la eficiencia de los algoritmos de Optimización Global son diversas y heterogéneas. En este proyecto utilizaremos únicamente funciones bidimensionales, debido a que se puede obtener una representación gráfica de las mismas, lo que facilita su estudio. Por tanto, la implementación se ha centrado en dar soporte a este tipo concreto de funciones (ver Sección 6.5).

Una vez determinadas las regiones donde potencialmente se encuentran los mínimos, debemos repartir los subproblemas contenidos por esas regiones entre los procesadores disponibles. Aquí reside la principal diferencia con respecto a los algoritmos paralelos clásicos de Optimización Global, ya que éstos no tienen en cuenta la localidad de las cajas a la hora de balancear la carga. En cambio, con nuestro algoritmo, las cajas que pertenecen a una misma región estarán en el mismo procesador, lo que nos permitirá hacer una mejor estimación de la carga computacional de la región, ayudándonos a decidir en los siguientes balanceos si realizar el envío de nuevo trabajo.

En definitiva, en este proyecto implementaremos un algoritmo de balanceo de la carga con predicción de mínimos, basado en la creación de regiones, y usaremos sus resultados en un algoritmo paralelo sencillo. La implementación, dado que requiere de más de un procesador, se llevará acabo construyendo un modelo paralelo simulado por software (ver Capítulo 6) para simplificar el problema, ya de por sí complejo, de encontrar un modelo de predicción eficiente.

1.2. Motivaciones y objetivos

La principal motivación a la hora de realizar este proyecto es comprobar si al utilizar un método de balanceo basado en la localidad de las cajas (subproblemas), podemos obtener algún tipo de beneficio al disponer de cajas con características homogéneas en un mismo procesador; frente a balancear la carga sin tener en cuenta la procedencia o características locales de esas cajas. Al basar nuestro algoritmo de balanceo en la existencia de regiones, puede ocurrir que procesadores libres no reciban cajas si el algoritmo de predicción determina que no hay nuevas regiones, por lo que estaremos utilizando menos procesadores de los que tenemos disponibles. Como resultado, cabe esperar alguno de los siguientes supuestos:

- Los tiempos de comunicación entre procesadores se reducirán. Debido a la información que nos proporcionan las regiones, nuestro algoritmo de balanceo por regiones comprueba si es óptimo el envío de cajas a otro procesador para su evaluación, o por contra, es mejor evaluarlas en el propio procesador puesto que no tienen suficiente carga computacional. Si logramos ahorrar suficiente tiempo en el envío de cajas con muy poco trabajo, mejoraremos la eficiencia de la búsqueda de mínimos.

- Una consecuencia del punto anterior es que podemos disponer de más procesadores libres cuando la carga computacional del problema no requiere de todos los procesadores del sistema. Al realizar el balanceo de la carga por regiones, podemos determinar cuántos procesadores necesitamos en cada momento, tratando de utilizar los estrictamente necesarios para la resolución del problema.

La conclusión que podemos obtener de los supuestos anteriores es que, a priori no podemos afirmar que utilizar todos los procesadores disponibles en el sistema, garantice la mayor eficiencia de los algoritmos, puesto que podemos estar gastando tiempo en enviar cajas a otros procesadores cuando el tiempo de envío de esas cajas al procesador será mayor que evaluarlas en el procesador en el que se encontraban. Por tanto, la principal dificultad reside en decidir cuándo hay que repartir las cajas.

La aportación más importante que este proyecto puede realizar tras su finalización es la de abrir una **nueva línea de investigación** dirigida a mejorar la eficiencia de los algoritmos paralelos de Optimización Global basados en Aritmética de Intervalos **utilizando un nuevo modelo de balanceo de la carga basado en la predicción de mínimos mediante regiones**, ya que creemos que las regiones nos permitirán obtener valores más precisos de la carga computacional asociada a cada región.

Capítulo 2

Plan de Trabajo

2.1. Estructura del Plan de Trabajo

El Plan de Trabajo se ha elaborado teniendo en cuenta la situación personal del autor, que se encuentra trabajando como profesor de Informática en Institutos de Secundaria de la provincia de Almería. Esto ha hecho que en numerosas ocasiones el tiempo de dedicación para la realización del proyecto se viera reducido, debido a la realización de cursos de formación, jornadas, viajes, y otras actividades relacionadas con la enseñanza.

Así, dado que es muy difícil establecer un tiempo de dedicación estable, para realizar la estimación de trabajo y tiempo que nos tomará el proyecto, hemos escogido como medida estándar la de una persona trabajando 5 días a la semana (de lunes a viernes) durante 5 horas al día (de 9 a 13 de la mañana).

2.2. Plan de Trabajo detallado

La tabla 2.1 muestra la planificación del trabajo previsto al inicio de este proyecto. Como se puede ver en ella, el número estimado de días de trabajo que ocupará la realización del proyecto es de **222 días**. Esto supone casi un año de trabajo, ya que trabajando cinco días a la semana tendríamos una media de 20 días mensuales, por lo que los 222 días estimados nos llevarían algo más de 11 meses.

Tabla 2.1: Plan de trabajo

Tarea	**Tiempo**
Lectura de documentación sobre Optimización Global y paralelismo	**15 días**
Implementación de un algoritmo simple de Optimización Global	**60 días**
Implementación de la clase caja2D	15 días
Implementación de una lista enlazada de cajas	30 días
Implementación de las funciones test	5 días
Implementación del test de monotonía y test de corte	5 días
Implementación del programa principal	5 días
Implementación de un algoritmo de reparto de cajas en baraja	**5 días**
Implementación de la predicción de regiones	**50 días**
Implementación de la clase regiones	15 días
Diseño del método de creación de regiones	30 días
Coloreado de regiones para análisis visual	1 día
Modificar programa principal: realizar las predicciones	1 día
Análisis de resultados visuales	3 días
Implementación de la simulación paralela	**60 días**
Implementación de la clase procesador	5 días
Modificación del programa principal: elección del procesador en ejecución	5 días
Diseño de la decisión sobre cuándo predecir regiones	30 días
Reparto de las regiones creadas entre los procesadores	5 días
Medición de tiempos y toma de datos de la simulación	15 días
Recogida de los resultados de la simulación para los supuestos previstos	**2 días**
Escritura de la memoria del proyecto	**30 días**
Tiempo total estimado para la realización del proyecto	*222 días*

Parte II

Introducción a la Optimización Global

Capítulo 3

Aritmética de Intervalos aplicada a la computación

3.1. Breve Historia de la Computación

La Aritmética ha ido siempre ligada a la Historia de la Computación, antes incluso de que surgieran los primeros ordenadores. Éstos, los ordenadores, se crearon para reducir el tiempo que una persona empleaba en resolver operaciones aritméticas complejas. Antes de que aparecieran las primeras máquinas que realizaban cálculos de forma **automática**, el ábaco era el instrumento utilizado para facilitar todo tipo de cálculos.

En los siglos XVI y XVII, científicos como Napier, Pascal o Leibniz pusieron su granito de arena tratando de fabricar una máquina que pudiera realizar cálculos de manera automática. Pero no fue hasta 1821, cuando Charles Babbage presentó a la Royal Society una máquina capaz de resolver ecuaciones polinómicas mediante el cálculo de diferencias sucesivas entre conjuntos de números. Esta máquina fue llamada **Máquina Diferencial**. Babbage obtuvo la medalla de oro de la Sociedad en 1822 por este logro.

Tras este importante hito, Babbage, ayudado por Ada Augusta Byron, hija de Lord Byron; trató de construir una máquina calculadora de propósito general controlada por una secuencia de instrucciones. La Máquina Analítica, como fue llamada, no pudo llegar a construirse por no disponer del dinero suficiente ni de la tecnología adecuada.

La Historia que le sigue está llena de importantes acontecimientos y personajes que lograron grandes avances tecnológicos, como la invención de los transistores, los circuitos integrados, los principios de la Inteligencia Artificial, los supercomputadores, y un largo etcétera, que han hecho posible que La Humanidad alcance metas inimaginables; y todo en un brevísimo periodo de tiempo.

Por tanto, la búsqueda de métodos automáticos de cálculo, de lo que se encarga la Ciencia de la Computación, fue el origen de la tecnología que hoy disfrutamos.

3.2. La Aritmética en los ordenadores

Los ordenadores disponen en su hardware de soporte para trabajar con números enteros y números en punto flotante. La aritmética de números enteros contiene un abanico ilimitado de enteros. Los ordenadores pueden realizar operaciones aritméticas con números de este tipo ampliando por software el número de dígitos necesarios para obtener un resultado exacto. Por tanto, la mayoría de cálculos que realizamos en los ordenadores, basada en Aritmética con números enteros está libre de errores.

Por otro lado, las aplicaciones científicas normalmente trabajan con números reales. Los números reales pueden tener infinitas cifras decimales, y para trabajar con ellos en un ordenador, tienen que ser aproximados mediante números en punto flotante. En consecuencia, debido a esta aproximación, algunas operaciones aritméticas con números en punto flotante pueden presentar errores en sus resultados.

3.3. Motivaciones de la verificación numérica

Como consecuencia de la imposibilidad de representar los números reales con exactitud en nuestro ordenador, se pueden producir errores en los resultados de cálculos realizados con la Aritmética de números en punto flotante. Ilustremos este problema con un ejemplo presentado por Rump [18]. La siguiente ecuación:

$$f(x,y) = 333{,}75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5{,}5y^8 + \frac{x}{2y} \tag{3.1}$$

fue evaluada para $x = 77617$ e $y = 33096$. El programa fue escrito en Fortran y compilado y ejecutado sobre una SparcStation SLC. Las precisiones probadas fueron simple, doble y extendida con 6, 14 y 35 dígitos decimales cada una. Los resultados obtenidos fueron los siguientes:

$$\begin{aligned} \text{(precisión simple)} f &= 6{,}33825 \cdot 10^{29} \\ \text{(precisión doble)} f &= 1{,}1726039400532 \\ \text{(precisión extendida)} f &= 1{,}17260394005317863185883490452018 38 \end{aligned}$$

El resultado en precisión extendida puede parecer correcto ya que concuerda hasta en 13 dígitos con el resultado en precisión doble, pero ninguno de estos resultados es correcto aunque pueda parecer sorprendente (véase el resultado correcto en la página 11).

Un ejemplo del peligro y la gravedad de cometer este tipo de fallos en operaciones aritméticas podemos verlo en la explosión del cohete Ariana, lanzado por la Agencia Espacial Europea el 4 de junio de 1996. El cohete explotó transcurridos cuarenta segundos de su despegue. Era el primer viaje del Ariana después de una década de desarrollo que costó 7 billones de dólares. Una investigación posterior determinó que el fallo que motivó la explosión del cohete, tuvo su origen en un error software en el sistema de referencia inercial. En concreto, al convertir un número en punto flotante de 64 bits a un entero con signo de 16 bits. El número era mayor que 32768, que es el mayor número representable por un entero con signo de 16 bits. Este ejemplo y el del error de un misil Patriot en la I Guerra del Golfo se pueden encontrar en B.1.1.

3.4. Aritmética de Intervalos

La Aritmética de Intervalos surgió para evitar los errores de redondeo producidos en el ordenador como el presentado en la Ecuación 3.1. El propósito de la Aritmética de Intervalos es obtener unos límites superior e inferior de los errores cometidos en la Aritmética Computacional. La Aritmética de Intervalos se atribuye a Moore con la aparición de su libro *Interval Analysis* en 1966 [17].

El problema ocurrido en la Ecuación 3.1 fue solucionado con Aritmética de Intervalos [13]. El resultado se encuentra incluido en el intervalo:

[-0.827396059946821368114116509547981629200 5,
-0.827396059946821368114116509547981629198 6]

Si el resultado se hubiera calculado utilizando Aritmética de Intervalos con menos precisión, hubiéramos obtenido un intervalo menos preciso, pero que contiene el resultado correcto.

3.4.1. Aritmética de Intervalos real

Podemos utilizar la Aritmética de Intervalos para trabajar con números reales o números complejos, entre otros. Nosotros nos centraremos en la Aritmética de Intervalos con números reales.

Antes de continuar, explicaremos la notación utilizada en este proyecto para la Aritmética Real y la Aritmética de Intervalos.

Notaciones de la Aritmética Real

Se usará la siguiente notación para la Aritmética Real:

$\mathbb{R}$	El conjunto de los números reales.
$\mathbb{R}^n$	El conjunto de los vectores de reales.
$\mathbb{R}^{n\times m}$	El conjunto de las matrices $n \times m$ de reales.

Se usarán minúsculas para definir números reales y vectores de números reales, y mayúsculas para definir matrices. Los elementos de un vector $x \in \mathbb{R}$ o de una matriz $M \in \mathbb{R}^{n\times m}$ se enumerarán por subíndices. Se usarán superíndices para enumerar un vector o una matriz dentro de una secuencia. Por ejemplo:

$x_i \in \mathbb{R}$	Elemento i del vector x.
$M_{i*} \in \mathbb{R}^n$	Fila i de la matriz M.
$M_{*j} \in \mathbb{R}^m$	Columna j de la matriz M.
$M_{i,j} = M_{ij} \in \mathbb{R}^{n\times m}$	Elemento de la fila i de la columna j de la matriz M.
$x^k \in \mathbb{R}^n$	Vector k dentro de una enumeración o secuencia.
$M^k \in \mathbb{R}^n$	Matriz k dentro de una enumeración o secuencia.

Notaciones de la Aritmética de Intervalos

Usaremos letras mayúsculas para notar los intervalos. Un intervalo X se define como: $X = [\underline{x}, \overline{x}]$, $\underline{x} \leq \overline{x} \in \mathbb{R}$, donde $\underline{x}$ es el límite inferior de X y $\overline{x}$ es el límite superior de X. Un número real x es análogo al intervalo $[x, x]$.

Además usaremos las siguientes notaciones:

$\mathbb{I}$	El conjunto de los intervalos.
$\mathbb{I}^n$	El conjunto de los vectores de intervalos.
$\mathbb{I}^{n\times m}$	El conjunto de las matrices $n \times m$ de intervalos.

Los elementos de $\mathbb{I}^n$ son llamados cajas (n-dimensionales). La notación usada para indexar intervalos será similar a la usada para indexar números reales. Para $X \in \mathbb{I}$ e $Y \in \mathbb{I}^n$ se define:

$w(X) = \overline{x} - \underline{x}$	Ancho del intervalo X.
$w(Y) = \max\limits_{1\leq i\leq n} w(Y_i)$	Ancho de la caja Y.
$m(X) = (\underline{x} + \overline{x})/2$	Punto medio de X.
$m(Y) = (m(Y_1), m(Y_2), \ldots, m(Y_n))^T$	Punto medio de la caja Y.

Extensión natural a intervalos

Las operaciones elementales de la Aritmética de números reales, representadas por $\circ \in \{+, -, \cdot, /\}$ se extienden para obtener un resultado sobre intervalos. El cálculo se

realiza de la siguiente manera:

$$X + Y = [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \quad (3.2)$$

$$X - Y = [\underline{x} + \overline{y}, \overline{x} - \underline{y}] \quad (3.3)$$

$$X \cdot Y = [a, b] \quad (3.4)$$

$$a = \text{mín}\{\underline{x}\underline{y}, \overline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\overline{y}\}$$

$$b = \text{máx}\{\underline{x}\underline{y}, \overline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\overline{y}\}$$

$$\frac{1}{Y} = [\frac{1}{\overline{y}}, \frac{1}{\underline{y}}],\ 0 \notin Y \quad (3.5)$$

Definición 1 *Si una función $f : \mathbb{R}^n \to \mathbb{R}$ es computable como una expresión, algoritmo o programa de computador, utilizando las cuatro operaciones básicas sobre intervalos, entonces se define la extensión natural de la Aritmética Real a la de Intervalos de f, como aquella en la que los operandos reales se sustituyen por intervalos de reales y las operaciones reales por operaciones de intervalos.*

Para las funciones $f : D \subset \mathbb{R}^n \to \mathbb{R}^m$ se define:

$f(X)$	El rango de f en X.
$F(X)$	Extensión de f a intervalos.
$\overline{F}(X)$	El límite superior de $F(X)$.
$\underline{F}(X)$	El límite inferior de $F(X)$.

Para poder trabajar con funciones en el ámbito de la Optimización Global, disponemos del concepto de función de inclusión.

Definición 2 *Sea $f : D \subseteq \mathbb{R}^n \to \mathbb{R}$. Una función $F : \mathbb{I}^n(D) \to \mathbb{I}$ se llama función de inclusión de f si:*

$$f(X) \subseteq F(X), \forall X \subseteq D. \quad (3.6)$$

La Aritmética de Intervalos es una herramienta con la que obtener funciones de inclusión de manera automática cuando se implementa en ordenadores, teniendo en cuenta los modos de redondeo establecidos en el estándar IEEE 754 [19]. Por ejemplo, cuando se computan límites inferiores (o superiores) del intervalo solución, el modo de redondeo computacional debe ser al inferior (o superior).

3.4.2. Extensiones de funciones estándar

Podemos calcular la extensión a intervalos de funciones monótonas de forma muy sencilla. Por ejemplo:

$$\sqrt{X} = [\sqrt{\underline{x}}, \sqrt{\overline{x}}], X \geq 0 \quad (3.7)$$

$$ln(X) = [ln(\underline{x}), ln(\overline{x})], X \geq 0 \quad (3.8)$$

$$e^X = [e^{\underline{x}}, e^{\overline{x}}] \quad (3.9)$$

Es posible obtener una extensión a intervalos de cualquier función representable por ordenador. Siempre pretendemos obtener unos límites de la función de inclusión lo más cercanos posibles al valor real de la función $f(X)$.

Capítulo 4

Optimización Global basada en Aritmética de Intervalos

4.1. Optimización Global

El problema de la Optimización Global consiste en encontrar los mínimos (o máximos) globales de una función. Para nuestro proyecto, utilizaremos una versión simplificada del algoritmo de Optimización Global presentado en [4], evaluando la función objetivo en el intervalo dado, sin utilizar métodos de eliminación basados en derivadas.

Sea $f : S \subset \mathbb{R}^n \rightarrow \mathbb{R}$ una función continua. Queremos encontrar los puntos x^* tales que:

$$f(x^*) = \min_{x \in S} f(x) \tag{4.1}$$

Los puntos x^* son aquellos puntos para los que la función f tiene un mínimo global. S es la región de búsqueda a la que pertenecen los mínimos.

4.2. Algoritmos de Ramificación y Acotación Secuenciales

Los primeros algoritmos de Ramificación y Acotación aparecieron en los cincuenta para tratar de resolver problemas *NP-Completos*. Los métodos de Ramificación y Acotación son algoritmos basados en árboles de búsqueda. Los primeros que dieron el nombre de Ramificación y Acotación (*Branch and Bound*) a este método fueron Little, Murty, Sweeny y Karel [14].

4.2.1. Reglas básicas de los algoritmos de Ramificación y Acotación

Podemos describir cuatro reglas básicas que caracterizan a los algoritmos de Ramificación y Acotación:

- *Regla de Ramificación o División*: Establece cómo se subdivide el problema original en subproblemas más pequeños.

- *Regla de Acotación*: Calcula un límite inferior de la solución óptima para cada subproblema en problemas de minimización.

- *Regla de Selección*: Establece qué nodo o subproblema será el siguiente en dividirse.

- *Regla de Eliminación*: Establece cómo reconocer y eliminar subproblemas donde no se encuentra la solución óptima del problema original.

Dependiendo del tipo de problema que estemos tratando, añadimos la siguiente regla:

- Regla de Terminación: Determina cuándo un nodo del árbol de búsqueda pertenece a la solución final. Esta regla se utiliza, por ejemplo, en problemas donde la solución viene determinada por la precisión deseada.

4.3. Inexistencia de mínimos globales

Para eliminar del árbol de búsqueda aquellos nodos que no contienen una solución del problema, podemos utilizar alguno de los test propuestos a continuación.

4.3.1. Test del Punto Medio

Supongamos que hemos obtenido un límite superior del mínimo global (f^*) de $f(x)$ al que llamaremos $\overline{f^*}$, en la región de búsqueda S. Para una caja $X \subset S$, si se cumple que $\overline{f^*} < \underline{F}(X)$, entonces no puede existir un mínimo global en X. El valor de $\overline{f^*}$ puede obtenerse mediante la evaluación de f en algún punto x de S. El nombre de este test se debe a que normalmente el punto elegido para obtener $\overline{f^*}$ es el punto medio de los subintervalos generados durante la búsqueda $X \subseteq S$.

4.3.2. Test de Monotonía

El gradiente para la función de inclusión es cero en cualquier máximo, mínimo o punto de inflexión. Si $0 \notin F_i'(X)$, para algún $i \in 1, \ldots, n$, entonces no existe ningún punto estacionario en X. En particular, el mínimo global sólo podría estar en los bordes de S.

4.3.3. Test de Corte

Siendo $\overline{f^*}$ un límite superior del mínimo global f^* obtenido mediante el *Test del Punto Medio*, el *Test de Corte* elimina de la lista de trabajo y de la lista final, las cajas X^i que cumplen que $\overline{f^*} < \underline{F}(X^i)$.

4.4. Algoritmo simple de Optimización Global basado en Intervalos

Se pretende resolver el problema de Optimización Global en donde la única restricción es tener límites implícitos en la región de búsqueda (Ecuación 4.1).

Las reglas de Ramificación y Acotación utilizadas en nuestro algoritmo serán las siguientes:

Regla de División La regla de división utilizada consiste en realizar la bisección de la caja o intervalo n-dimensional por su lado más ancho.

Regla de Acotación Utilizamos la extensión natural a intervalos para obtener los límites de la función objetivo (ver Sección 3.4.1).

Regla de Selección Utilizaremos una regla de selección *Primero el Mejor* basada en seleccionar las cajas con menor valor del límite inferior $\underline{F}(X)$. Para ello, las cajas se mantienen ordenadas en la lista de trabajo de acuerdo con este criterio.

Regla de Eliminación La regla de eliminación básica utilizada es el *Test del Punto Medio* y el *Test de Corte* (ver Sección 4.3.3). En este proyecto, existe la posibilidad de utilizar otro test de eliminación, el *Test de Monotonía* (ver Sección 4.3.2), pero no lo vamos a utilizar ya que nos interesa mantener una cierta carga de trabajo en el procesador.

Regla de Terminación Consideraremos que una caja es final cuando el ancho de la misma ($w(X)$) sea menor que la precisión establecida para la solución (ϵ). Por tanto si $w(X) < \epsilon$, X contendrá una solución del problema.

El algoritmo simple de Optimización Global basado en intervalos se muestra en el Algoritmo 1 y una descripción detallada del mismo puede verse en [4]. El algoritmo computa el intervalo inicial (o caja) $[S]$. Éste se va dividiendo sucesivamente aplicando la Regla de División, y el algoritmo almacena las subcajas $[X] \subset [S]$ en la lista de trabajo L. Aquellas cajas que con certeza sabemos no contienen un mínimo global de f son descartadas de la lista por la Regla de Eliminación. Las cajas que cumplen la Regla de Terminación son insertadas en la lista de cajas finales Q. El algoritmo concluye cuando no hay más cajas que evaluar en L.

4.4.1. El parámetro $p\overline{f^*}(X)$

El parámetro $p\overline{f^*}(X)$ fue presentado por primera vez en [4] y surge ante la necesidad de obtener un estimador de la cantidad de región de atracción hacia un mínimo que contiene una determinada caja. Así, si una caja tiene menor $\underline{F}(X)$ que otra, es posible que ésta no esté más próxima al mínimo, debido a las sobrestimaciones obtenidas por la Aritmética de Intervalos, por lo que el estimador $p\overline{f^*}(X)$ es un buen método para mejorar la eficiencia de los algoritmos de Optimización Global modificando los criterios de la Regla de Selección. Ejemplos de algoritmos paralelos de Ramificación y Acotación

Algoritmo 1 Algoritmo simple de Optimización Global basado en intervalos

Funct OGBásico$(S, F, \varepsilon, L, Q)$

1. $\overline{f^*} = \overline{F}(S)$ — *Límite superior de* f^*
2. $L := \{S\}$ — *Se inicializa la lista de trabajo*
3. $Q := \{\emptyset\}$ — *Lista final vacía*
4. **while** ($L \neq \{\emptyset\}$)
5. $\quad X := SeleccionaCaja(L)$ — $\underline{F}(X) = min\{\underline{F}(X^i), \forall X^i \in L\}$
6. $\quad L := L - \{X\}$;
7. $\quad \overline{f^*} := min\{\overline{f^*}, \overline{F}(m(X))\}$ — *Actualización de* $\overline{f^*}$
8. $\quad$ **if** $(\overline{f^*} = \overline{F}(m(X)))$ — *Se ha mejorado el valor de* $\overline{f^*}$
9. $\quad\quad L := TestdeCorte(L, \overline{f^*})$ — $\forall X^i \in L, \overline{f^*} < \underline{F}(X^i) \Rightarrow L := L - \{X^i\}$
10. $\quad\quad Q := TestdeCorte(Q, \overline{f^*})$
11. $\quad Divide(X, X^1, \ldots, X^s)$
12. $\quad$ **for** $i := 1$ **to** s
13. $\quad\quad$ **if** $(\overline{f^*} < \underline{F}(X^i)$ **next**$_\mathbf{i}$ — *Test del Punto Medio*
14. $\quad\quad$ **elsif** $(w(X^i) < \varepsilon)$ $Q := Q + \{X^i\})$ — *Almacenar* X^i *en* Q
15. $\quad\quad$ **else** $L := L + \{X^i\}$ — *Almacenar* X^i *en* L
16. **return** Q

que utilizan el parámetro $p\overline{f^*}(X)$ como criterio de selección fueron presentados en [4] y [3]. En este último, se presenta un algoritmo rápido de búsqueda de intervalos, FIS (*Fast Interval Search*), donde lo más interesante es el criterio de selección basado en una estrategia *Primero el Mejor* del valor del $p\overline{f^*}(X)$. Este criterio se aplica a todos los nodos que pertenecen al mismo nivel en el árbol de búsqueda. El algoritmo progresa por etapas desde un nivel del árbol hasta el siguiente.

En la Figura 4.1 cada caja representa un intervalo $[\underline{X}, \overline{X}]$ y sus límites superior e inferior $[\underline{F}(X), \overline{F}(X)]$. La región inicial X^1 es seleccionada y dividida, generando los subintervalos X^2 y X^3 que son evaluados por la función de inclusión. El siguiente subintervalo seleccionado es X^3 ya que $\underline{F}(X^3) < \underline{F}(X^2)$, generando dos nuevos subintervalos, X^4 y X^5. En este punto, la lista de intervalos activos es $\{X^2, X^4, X^5\}$. De acuerdo con el criterio de selección basado en el límite inferior, el siguiente subintervalo seleccionado sería X^2, aunque éste no contiene el mínimo global. Podemos observar también, cómo en intervalos con la misma anchura (X^4 y X^5) la sobreestimación puede ser significativamente diferente.

El parámetro $p\overline{f^*}$ fue definido por primera vez en [4] de la forma siguiente:

Definición 3 *Dada una función* $f(x) : S \subset \mathbb{R}^n \to \mathbb{R}$ *y* $\overline{f^*}$ *un límite superior de* f^**, se define el parámetro* $p\overline{f^*}(X)$ *de una caja* $X \subseteq S$ *como:*

$$p\overline{f^*}(X) = \frac{\overline{f^*} - \underline{F}(X)}{w(F(X))}, \tag{4.2}$$

donde $\overline{f^*}$ puede obtenerse mediante el uso de algoritmos de búsqueda local o durante la ejecución del Algoritmo 1, mediante el *Test del Punto Medio*. Como muestra la Ecua-

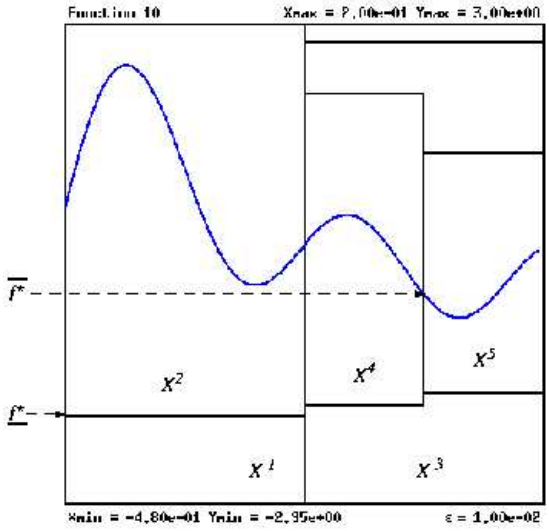


Figura 4.1: Descripción de la regla de selección

ción (4.2), el parámetro $p\overline{f^*}(X)$ determina la distancia de $\overline{f^*}$ a $\underline{\mathrm{F}}(\mathrm{X})$, normalizada por $w(F(X))$.

Aunque este parámetro se usa para reducir el número de evaluaciones de la *función de inclusión* realizadas por el Algoritmo 1, en [5] se utilizó también con éxito para resolver un problema de desbalanceo de carga de trabajo en un algoritmo paralelo, utilizándose como estimador de la carga de trabajo de cualquier caja. En este proyecto también lo usaremos para determinar la carga de trabajo de una determinada caja, aunque la forma de calcular esta cantidad de trabajo se hará de forma diferente (ver Sección 8.2 y siguientes).

Capítulo 5

Paralelización de algoritmos de Ramificación y Acotación

5.1. Introducción

El Algoritmo 1 presenta un método de Ramificación y Acotación secuencial aplicado a un problema de Optimización Global. Lo normal es que ante problemas que requieran una gran carga computacional, los algoritmos secuenciales sean demasiado lentos para encontrar la solución. Por ello, se recurre a la paralelización de los algoritmos de Optimización Global para mejorar el tiempo de ejecución de los algoritmos secuenciales. Un ejemplo de este tipo de problemas podría ser el famoso **Problema del Viajante (simétrico)**, que consiste en visitar todas las ciudades de un mapa exactamente una vez, minimizando el número de kilómetros recorridos. Dos instancias de este problema con más de 10000 ciudades (13509 y 15112 respectivamente) fueron resueltas mediante algoritmos paralelos por Appelgate *et al.* [2].

En este capítulo haremos una introducción básica al paralelismo, clasificando las diferentes arquitecturas paralelas y las principales medidas de rendimiento utilizadas en estos sistemas. Además, describiremos brevemente algunos algoritmos paralelos de Optimización Global.

5.2. Arquitecturas Paralelas

Los elementos que caracterizan un computador paralelo son muy diversos, desde el número de procesadores, p, hasta el tipo de memoria que utilizan, pasando por la topología de interconexión de estos procesadores, los protocolos de comunicación y la velocidad de la red de interconexión, entre otros. El rendimiento de un programa paralelo depende, en general, de estos elementos, pero además, el programa debe tener una serie de características que lo hagan altamente paralelizable. *Amdahl* [1] demostró que el incremento de velocidad de un programa utilizando diversos procesadores está limitado por la cantidad de código secuencial de este programa. Así, aunque cabría esperar que un programa paralelo ejecutado en una arquitectura con p procesadores tarde $1/p$ veces el tiempo de una

ejecución secuencial, normalmente no es cierto. Esta ganancia o incremento de velocidad se denomina *Speed-Up* (ver ecuación en página 24).

Por tanto, hemos visto cómo una arquitectura paralela potente puede no ser suficiente para obtener resultados óptimos, además, no conocer lo suficiente el programa a paralelizar hará que el rendimiento obtenido no sea el esperado. En consecuencia, un buen conocimiento de la arquitectura paralela y del problema que queremos paralelizar hará que los algoritmos creados para este binomio obtengan mejores resultados.

5.2.1. Clasificación de las arquitecturas paralelas

Una de las clasificaciones de las arquitecturas paralelas más extendida es la que propuso Flynn en 1966, basándose en dos tipos de flujos de información: de datos y de instrucciones. El flujo de datos se refiere a la información que circula entre la memoria y la unidad de control, y el flujo de instrucciones se refiere a las instrucciones que se ejecutan en la unidad de proceso. Puesto que los flujos pueden ser simples o múltiples, según la clasificación de Flynn, existen cuatro tipos de arquitecturas:

- *Single-instruction single-data* (SISD). A esta categoría pertenecen los procesadores secuenciales propuestos por Von-Neumman.

- *Single-instruction multiple-data* (SIMD). En esta categoría se encuentran los procesadores matriciales. Todas las unidades de procesamiento reciben la misma instrucción, pero ésta se ejecuta sobre diferentes datos.

- *Multiple-instruction single-data* (MISD). Algunos autores creen que esta clasificación no es viable, mientras otros incluyen a los computadores sistólicos dentro de esta categoría.

- *Multiple-instruction multiple-data* (MIMD). En esta categoría diferentes unidades de control trabajan sobre diferentes fuentes de datos. Es la categoría que explota al máximo las fuentes de paralelismo, y a ella pertenecen los multicomputadores y multiprocesadores.

Dentro de las arquitectura MIMD se encuentran las SPMD (*Single Program Multiple Data*) de la que haremos uso en este proyecto, y que consiste en un único programa trabajando sobre conjuntos de datos distintos. En las siguientes subsecciones analizaremos las características de dos tipos de arquitecturas SPMD, las de memoria compartida y las de memoria distribuida.

5.2.1.1. Arquitecturas de memoria compartida

Las arquitecturas de memoria compartida extienden las características de una arquitectura monoprocesador, ya que disponen de un único espacio de direcciones accesible por todos los procesadores. Es necesario establecer mecanismos para que un procesador pueda acceder a una dirección de memoria concreta de manera exclusiva. Así, la comunicación

o el intercambio de información entre los diferentes procesadores puede hacerse de manera sencilla leyendo o escribiendo en una dirección de memoria. El principal problema de estas arquitecturas es que presentan dificultades a la hora de escalar el número de procesadores, porque el bus o red de acceso a memoria se convierte en un cuello de botella al estar compartido por todos los procesadores. Debido a las características de acceso a la memoria compartida del sistema, existen dos subarquitecturas diferentes dentro de las arquitecturas de memoria compartida.

Arquitecturas de procesamiento simétrico

Los procesadores simétricos, también conocidos como SMP, tienen la característica de que el tiempo de acceso a cualquier dirección de memoria es el mismo para todos los procesadores del sistema. A este tipo de arquitectura se le conoce como UMA (*Uniform Memory Access*).

Arquitecturas de memoria compartida-distribuida

Estas arquitecturas se caracterizan porque cada unidad de procesamiento dispone de su bloque de memoria. Así, cuando un procesador accede a una dirección de memoria que está en su mismo módulo será mucho más rápido que cuando necesita acceder a la dirección de memoria de un módulo que está en otro procesador. Este tipo de arquitecturas se denominan NUMA (*Non Uniform Memory Access*). Cada procesador, puede disponer de copias de un mismo dato en sus memorias caché, debido a que cada procesador dispone de su módulo de memoria. Si este dato es modificado por alguno de los procesadores, hay que establecer el mecanismo adecuado para garantizar que todos los procesadores dispongan del dato actualizado en sus memorias. Estos sistemas se denominan cc-NUMA (*cache-coherent Non Uniform Memory Access*). Por contra, los sistemas que no disponen del mecanismo que mantiene la coherencia de datos en la caché se denominan ncc-NUMA (*non cache-coherent Non Uniform Memory Access*).

5.2.1.2. Arquitecturas de memoria distribuida

En las arquitecturas de memoria distribuida cada uno de los procesadores dispone de su propio espacio de direcciones, por lo que el acceso a cualquier dirección local se hace de forma rápida. En cambio, las comunicaciones con el resto de procesadores del sistema son más lentas que en las arquitecturas de memoria compartida, debido a la red de interconexión. Una ventaja de este tipo de arquitecturas es que son fácilmente escalables. Los intercambios de información entre los diferentes procesadores se realiza mediante paso de mensajes. Un ejemplo de este tipo de sistemas serían los *cluster* de estaciones de trabajo (COW, *Cluster of Workstations*).

5.2.2. Medidas de rendimiento

En esta sección presentaremos los conceptos necesarios para evaluar la eficiencia de un algoritmo paralelo.

Para resolver un problema utilizando una arquitectura paralela es deseable que el tiempo total de resolución de este problema se vea reducido conforme se incrementa el número de procesadores. En general, este hecho se produce cuando:

1. El tamaño del problema es suficientemente grande.

2. La porción de código paralelo es superior a la porción de código secuencial.

Está claro que no tiene sentido paralelizar un problema cuya resolución es sencilla, ya que se empleará mucho más tiempo en gestionar la parte paralela del algoritmo que en la propia resolución del problema. El segundo punto es consecuencia directa de la *Ley de Amdahl*.

La primera medida en la que nos debemos fijar en un algoritmo paralelo se denomina *Speed-Up*, que muestra la ganancia o aceleración de velocidad de un algoritmo paralelo frente a su homólogo secuencial. Se calcula cómo:

$$S(p) = \frac{t_{sec}}{t_{par}(p)} \tag{5.1}$$

donde p es número de procesadores, t_{sec} es el tiempo total del mejor algoritmo secuencial conocido, y $t_{par}(p)$ es el tiempo total empleado por el algoritmo en un sistema paralelo con p procesadores. En general, se cumple que $1 \leq S(p) \leq p$ aunque en algunos casos se pueden obtener ganancias de velocidad superlineales ($S(p) > p$).

La relación entre la ganancia de velocidad y el número de procesadores, se conoce como Eficiencia, y permite conocer la velocidad a la que crece el rendimiento del algoritmo paralelo al incrementar el número de procesadores. La Eficiencia se define así:

$$E(p) = \frac{S(p)}{p} \tag{5.2}$$

Si la gráfica obtenida por la función de Eficiencia decrece a medida que aumentamos el número de procesadores, p, significará que el aumento de procesadores no mejora el rendimiento del algoritmo paralelo.

Uno de los factores que pueden influir positiva o negativamente en el rendimiento de un programa paralelo es el grado de desbalanceo entre los procesadores del sistema. Si podemos medir la cantidad de trabajo computacional que han realizado los procesadores desde el inicio del programa hasta un determinado momento, podemos calcular entonces el grado de desbalanceo del sistema en ese momento. Nosotros consideramemos como unidad de trabajo la evaluación de una caja. Así, la fórmula del desbalanceo se calcula de la siguiente manera:

$$Desbalanceo = \frac{\sigma(Esf)}{\overline{Esf}} \tag{5.3}$$

donde $\sigma(Esf)$ es la desviación típica del esfuerzo de todos los procesadores del sistema (ver Ecuación 5.4) y $\overline{Esf}$ es el promedio del esfuerzo de los procesadores.

$$\sigma(Esf) = \sqrt{\frac{1}{p}\sum_{i=1}^{p}(Esf(p_i) - \overline{Esf})^2} \tag{5.4}$$

El Esfuerzo de un procesador del sistema, $Esf(p_i), i \in [1, p]$, para los algoritmos de Ramificación y Acotación basados en intervalos, se calcula como el número de evaluaciones de la función de inclusión y de la derivada:

$$Esf(p_i) = FE + n \cdot GE \tag{5.5}$$

donde FE es el número de evaluaciones de la extensión a intervalos de la función objetivo, n es el número de dimensiones del problema, y GE el número de evaluaciones de la derivada. En nuestro caso, al no utilizar la evaluación de la derivada, el Esfuerzo será únicamente el número de evaluaciones de la función de inclusión.

5.3. Algoritmos de Ramificación y Acotación paralelos

El objetivo de este capítulo es obtener los datos que nos permitan construir un algoritmo paralelo de Ramificación y Acotación eficiente. Para ello, en las secciones anteriores se han presentado las características generales de las arquitecturas paralelas. Ahora, describiremos las características generales de los algoritmos paralelos de Ramificación y Acotación, así como las principales estrategias de diseño de estos algoritmos. Concluiremos con una breve descripción de algunos algoritmos paralelos de Ramificación y Acotación.

5.3.1. Clasificación de los algoritmos de Ramificación y Acotación paralelos

Vamos a seguir la clasificación propuesta por Gendron y Crainic [7] de algoritmos paralelos de Ramificación y Acotación. Ellos distinguen tres tipos:

- Paralelismo del tipo 1. Se realizan operaciones en los subproblemas creados de forma paralela, por ejemplo, realizar la regla de acotación en paralelo para cada subproblema.
- Paralelismo del tipo 2. Consiste en construir un único árbol de trabajo en paralelo, realizando operaciones simultaneas sobre los subproblemas almacenados en el árbol. Los algoritmos de este tipo tienen que modificar su diseño.
- Paralelismo del tipo 3. En este caso, numerosos árboles de trabajo se construyen en paralelo.

Los algoritmos del tipo 3 han sido objeto de numerosos estudios, y los algoritmos presentados en este proyecto, en los Capítulos 7 y 8, pertenecen a este tipo.

5.3.2. Anomalías en algoritmos paralelos de Ramificación y Acotación

En los algoritmos paralelos de Ramificación y Acotación, se espera que la ganancia de velocidad o *Speed-Up* sea mejor cuanto mayor es el número de procesadores del sistema. En general, esta ganancia de velocidad, $S(p)$, es $1 < S(p) < p$, y puede estar relacionada con el número de iteraciones o evaluaciones que hace cada procesador del sistema. Así, se redefine la ganancia de velocidad con respecto al número de iteraciones, $S_I(p) = I(1)/I(p)$ [6].

Algunos autores [11, 12] observaron que el número de iteraciones realizadas en un algoritmo de Ramificación y Acotación podían aumentar o disminuir en función de la regla de selección utilizada, incluso en algoritmos secuenciales debido a las características heurísticas de los criterios de selección. Estas anomalías se presentan, por ejemplo, cuando hay que elegir entre varios subproblemas con el mismo límite inferior $\underline{F}(X)$, y se debe a que según el problema elegido para la evaluación, podemos llegar antes a la solución, es decir, escoger el **camino crítico mínimo** hacia la solución del problema.

Los tipos de anomalías que pueden presentarse son: *Anomalías Aceleratorias*, cuando $S_I(p) > p$; y *Anomalías Detrimentales* cuando $S_I(p) < p$. Está claro que los algoritmos de Ramificación y Acotación deben preservar las anomalías aceleratorias y evitar las detrimentales.

5.3.3. Ejemplos de algoritmos paralelos de Ramificación y Acotación existentes

En [15] y [4] se hace una revisión bibliográfica de diversas implementaciones de algoritmos paralelos de Ramificación y Acotación. La arquitectura de estos algoritmos suelen ser de tres tipos: centralizada, descentralizada o híbrida.

Son implementaciones centralizadas las realizadas por Henritsen *et al.* [8, 9] en donde un procesador maestro envía problemas a los procesadores esclavo, y Wiethoff [22] similar a la anterior, pero cada procesador se comunica sólo con los cuatro vecinos más cercanos.

Ejemplos de implementaciones descentralizadas son las de Erikson *et al.* en donde cada procesador tiene un proceso organizador y otro esclavo, en donde el organizador selecciona los problemas que debe resolver el esclavo. Implementaron diversos modos de comunicación entre ambos procesos para obtener un modelo de balanceo de carga óptimo. También la implementación de Moore *et al.* [16] en donde un procesador sin trabajo elige otro aleatoriamente para pedir problemas, si éste contesta negativamente, se envía una petición al procesador con índice de procesador siguiente. Otro modelo descentralizado siguiendo una estructura maestro-esclavo es el presentado en [10] donde se introduce el concepto de *leader*. El procesador elegido como *leader* será el que haga las veces de maestro. El criterio de selección del procesador *leader* es el último que encontró la mejor solución.

Por último, en [5, 3] se hacen nuevas aportaciones en el campo de los algoritmos paralelos de Optimización Global. En [5] se presenta un experimento de un algoritmo de búsqueda rápida (*FIS, Fast Interval Search*) en donde se establece un nuevo criterio de reparto eficiente de la carga computacional basado en el parámetro $\overline{pf^*}$ (ver Sección 4.4.1)

junto a un modelo híbrido de búsqueda en el árbol de trabajo. Y en [15] se presentan dos nuevos algoritmos paralelos de Optimización Global implementados con *threads*, una versión global en la que las estructuras de datos son compartidas por todos los *threads* y una versión local en la que las estructuras de datos son locales a cada *thread*. Además incorpora una nueva regla de eliminación que a su vez modifica la regla de división tradicionalmente utilizada.

5.3.4. Estrategias de balanceo de la carga de trabajo

Un buen balanceador debe pretender que todos los procesadores estén realizando un trabajo útil. El problema de los algoritmos de Ramificación y Acotación es que la carga computacional es dinámica, es decir, se crea y destruye en tiempo de ejecución. Por tanto, es difícil determinar la carga computacional de un problema de forma sencilla. Este tipo de algoritmos paralelos requieren de técnicas de balanceo dinámico de la carga, cuyo objetivo es que todos los procesadores realicen aproximadamente el mismo trabajo computacional. Para ello, hay que minimizar el tiempo que un procesador está sin trabajo mientras otros tienen una gran cantidad de trabajo pendiente.

En este sentido, vamos a analizar el diseño de un buen balanceador de carga de trabajo siguiendo las estrategias propuestas en [7]. El primer problema al que nos enfrentamos al crear el algoritmo paralelo es la generación inicial de trabajo para todos los procesadores, ya que en el inicio, sólo existe el problema raíz. Usar el paralelismo tan pronto como sea posible puede no ser una buena estrategia por diversas causas. Una de ellas es que disponer de subproblemas en todos los procesadores puede hacer que se realicen más evaluaciones de problemas, que en otros casos podrían ser rechazadas al actualizar el límite superior f^* (anomalía detrimental). Por otro lado, al evaluar varios subproblemas de forma simultanea podemos encontrar más rápidamente una cota superior de la solución (anomalía incremental).

Por tanto, el objetivo de un balanceador de trabajo puede formularse de la siguiente forma [7]: *Utilizar todos los procesadores disponibles tan pronto como sea posible, siempre y cuando se les proporcionen subproblemas prometedores.*

Se han propuesto numerosas estrategias para lograr este objetivo:

1. Repartir gradualmente los subproblemas que se van creando.
2. Generar numerosos subproblemas con una operación de división especial.
3. Realizar una ejecución secuencial del algoritmo hasta que se generen un *número adecuado* de subproblemas sin evaluar.
4. Realizar una ejecución secuencial en cada procesador hasta alcanzar un número determinado de subproblemas sin evaluar. En este momento cada procesador elige la rama de subproblemas que evaluará.

Una vez todos los procesadores disponen de unidades de trabajo, la siguiente cuestión de diseño es cómo realizar el intercambio de trabajo entre procesadores. En este sentido, el objetivo que persiguen los balanceadores es que todos los procesos realicen una cantidad de trabajo aproximadamente igual. Podemos considerar tres tipos de estrategias:

1. **Estrategia bajo demanda**. En este caso, un procesador que no dispone de problemas en su lista de trabajo envía una petición de trabajo a otro procesador. La petición puede ser aceptada (recibiendo parte del trabajo del procesador al que realizó la petición) o rechazada, por lo que tendría que realizar la petición a otro procesador.

2. **Estrategia sin demanda**. Al contrario que en el primer caso, en esta estrategia un procesador decide enviar trabajo sin recibir ninguna petición previa. Antes de enviar el trabajo, habría que tomar algunas decisiones: cuánto trabajo enviar, a qué procesador, etc.

3. **Estrategia combinada**. En esta estrategia se combinan las dos anteriores. Los procesos envían trabajo sin que se lo soliciten, y además, los procesos envían peticiones cuando se quedan sin trabajo.

5.4. Conclusión

Como conclusión a este capítulo, podemos decir que las estrategias de diseño de un algoritmo paralelo son numerosas, y en lo concerniente a los algoritmos paralelos de Ramificación y Acotación, puede afectar al rendimiento de éstos, bien de manera positiva mediante la aparición de anomalías aceleratorias, bien de manera negativa con la aparición de anomalías decrementales. Entre otras causas, la aparición de estas anomalías se debe al criterio de selección de problemas y a la estrategia de balanceo usada por el algoritmo paralelo. En este proyecto nos centraremos exclusivamente en analizar las consecuencias de elegir entre dos balanceadores de carga distintos. En el Capítulo 7 se presenta una estrategia de balanceo de carga bajo demanda para un algoritmo de Ramificación y Acotación paralelo, por otro lado, en el Capítulo 8 se presenta un algoritmo paralelo de Ramificación y Acotación que sigue una estrategia de balanceo de carga sin demanda, en el que un procesador crea regiones de cajas y las envía a los procesadores libres (sin que estos las hayan solicitado previamente).

Parte III

Simulación paralela

Capítulo 6

Simulación paralela de los algoritmos de Optimización Global Intervalar

6.1. Introducción

Como se explicó en la descripción de este proyecto (ver Sección 1.1), los algoritmos de balanceo que se han implementado para un algoritmo de Optimización Global, tanto el de reparto de cajas en baraja (ver Capítulo 7) como el de creación de regiones con predicción de mínimos (ver Capítulo 8), se ejecutarán sobre un entorno que simulará una arquitectura paralela. Para ello, el sistema que permite la **simulación paralela** ha sido implementado utilizando una estructura de clases (ver Sección 6.5) que nos permitirá controlar y gestionar los datos de las ejecuciones de los algoritmos en los diferentes procesadores.

6.2. Justificación

El algoritmo de Optimización Global con un balanceador basado en creación de regiones es una aproximación de un algoritmo paralelo, ya que su uso está orientado a la ejecución en más de un procesador. Por este motivo, para estudiar y conocer su funcionamiento, se ha implementado un sistema de clases que simula la estructura de una arquitectura multicomputador. Esta arquitectura simulada facilita la toma y recogida de datos para su estudio, así como la implementación de los propios algoritmos, ya que al no tener que paralelizarlos, ahorramos tiempo de desarrollo y simplificamos su implementación. Este sistema permite, entre otros parámetros, elegir el número de procesadores disponibles o configurar la tasa de transferencia de la red, latencia, tiempo de creación de un proceso, etc., cuyos valores de entrada condicionarán los resultados de la simulación.

La elevada dificultad y complejidad a la que nos hemos enfrentado a la hora de crear y desarrollar el algoritmo de creación de regiones con predicción de mínimos, hicieron que desde un principio su implementación se hiciera de forma secuencial. La versión definitiva (ya que ha sufrido numerosas variaciones y modificaciones hasta llegar al resultado que se presentará en este proyecto) simplifica el problema de la máquina paralela y permite obtener resultados independientes de la máquina final.

El trabajo de este proyecto se centra en obtener un nuevo algoritmo de Optimización Global paralelo que base el balanceo de la carga en la creación de regiones en torno a los mínimos locales y globales de la función objetivo (ver Sección 1.2). Como hemos dicho, este desarrollo no ha sido fácil, por lo que era primordial que la forma de comprobar si íbamos por el buen camino fuera lo más práctica posible.

Está claro que la simulación del algoritmo, tal y como está construido, no refleja ni puede reflejar con exactitud los resultados de una ejecución en un sistema paralelo real, pero sí es cierto que nos permite estudiar el comportamiento de nuestro algoritmo y aproximarlo a una arquitectura paralela de forma sencilla; más aún, nos permite compararlo con otros algoritmos implementados para ser ejecutados en nuestro sistema paralelo simulado.

La lista de nuevas investigaciones que proporciona este proyecto es larga y variada, como puede verse en el Capítulo 11, y entre ellas, una de las propuestas que se plantean es la de implementar la versión paralela de nuestro algoritmo y estudiar los resultados de su ejecución en una arquitectura paralela real.

6.3. Parámetros del programa

El programa de **Optimización Global mediante Simulación Paralela** (ogsp) se encarga de ejecutar el Algoritmo 1 presentado en la Sección 4.4, de acuerdo a los parámetros y opciones elegidos para su ejecución. El programa `ogsp` proporciona el soporte necesario para realizar la ejecución con el número de procesadores deseado, el tipo de algoritmo de balanceo, y otros parámetros y opciones que se detallan a continuación.

La ejecución del programa `ogsp` tiene la siguiente estructura:

```
$ ./ogsp [opciones] nombreFunc xmin xmax ymin ymax epsilon
```

donde el significado de los parámetros es el siguiente:

nombreFunc Es el nombre de la función test utilizada. Ver la lista de funciones test utilizadas en el Apéndice A.

xmin, xmax Dominio de la primera coordenada de la función test.

ymin, ymax Dominio de la segunda coordenada de la función test.

epsilon Determina la precisión de la solución. Las cajas X cuya anchura sea menor que este valor ($w(X) < \epsilon$), se considerarán cajas finales.

La lista de opciones que podemos utilizar es la siguiente:

-h Muestra la ayuda del programa

-v Activa el modo verbose (desactivado por defecto), por lo que se muestra información de depuración por la salida de error estándar.

-mt Activa el *Test de Monotonía*(desactivado por defecto). Esta opción no ha sido utilizado para el cálculo de los resultados del proyecto.

-o Muestra la información de salida para el programa de visualización `escalaw.tcl`. Este programa muestra en una ventana gráfica el proceso de evaluación de las cajas. Se ha implementado utilizando tcl/tk.

-c Colorea las cajas cada vez que se hace un reparto. Se visualiza con el programa `escalaw.tcl`. Las cajas de un mismo procesador se colorean de igual color, así se pueden identificar qué cajas pertenecen a un mismo procesador.

-w valor Tamaño de la ventana de visualización del programa `escalaw.tcl`. Por defecto es 400 pixels.

-p valor Número de procesadores del sistema. Su valor por defecto es 1.

-ttr valor Tasa de transferencia de la red en Mbps. Por defecto 100 Mbps.

-l valor Latencia de la red en segundos. Por defecto 0,1 segundos.

-tc valor Tiempo de creación de un nuevo proceso en segundos. Por defecto 0,1 segundos.

-f valor Indica la función test a utilizar. Las funciones disponibles son *Goldsteins-Price* (valor 0, por defecto), *Levy3* (valor 1) y *SixHumpCamelBack* (valor 2).

-b Distribución de cajas mediante el algoritmo de reparto de cajas en baraja (ver Capítulo 7).

-r Distribución de cajas mediante el algoritmo de creación de regiones con predicción de mínimos (ver Capítulo 8). Éste es el valor por defecto de la simulación.

-y Distribución de cajas mediante un nuevo algoritmo de creación de regiones sin predicción de mínimos (ver Capítulo 10).

-d Incluye la toma de decisión antes de realizar una distribución de cajas (ver Sección 8.6).

-nb valor Número de cajas que se evalúan para calcular el tiempo de proceso de una caja. Por defecto 200 cajas.

-s strcode Genera archivos de estadísticas cuyo nombre comienzan por *strcode*. El nombre completo de los archivos identifica el problema y con qué parámetros y opciones ha sido ejecutado el programa.

-g grfcode Genera archivos de gráficos cuyo nombre comienzan por *grfcode*. El nombre completo de los archivos identifica el problema y con qué parámetros y opciones ha sido ejecutado el programa.

6.4. Estructura de la simulación

Para comprender cómo funciona nuestro programa **ogsp**, es necesario explicar cómo se ha realizado la implementación del programa de simulación paralela. Explicaremos cómo se realiza la selección del procesador que evaluará una caja en cada iteración, y lo más importante, cómo se realiza la medición de tiempos de ejecución y proceso para que los resultados obtenidos sean del todo fiables.

6.4.1. Tiempos de ejecución en los procesadores

Tal y como se ha implementado la simulación paralela, es necesario elegir el procesador que vamos a poner en ejecución de entre todos los que tienen cajas, ya que en realidad, estamos trabajando en secuencial.

Al comenzar la ejecución, sólo habrá un procesador trabajando, pero en el momento en que realicemos un reparto de cajas a otro u otros procesadores, tendremos que decidir cuál de ellos seleccionar. La solución escogida es la de elegir el procesador con menor tiempo real de ejecución. De esta forma, nos aseguramos de que no hay un procesador que se adelanta demasiado en el tiempo real de ejecución. Esto es así debido a que no se tarda el mismo tiempo en todas las iteraciones, puesto que un procesador hará ciertas tareas o comprobaciones que otro no hará.

Supongamos la ejecución que se muestra en la Figura 6.1. Podemos ver cómo el procesador p_0 inicia la evaluación de las cajas y en el momento t_0 se realiza un reparto de cajas a los procesadores p_1 y p_2. En este momento, cuando hay más de un procesador que dispone de cajas para evaluar, es cuando debemos decidir qué procesador debe ser el siguiente que continúe ejecutándose en la simulación. Para tomar esta decisión, comprobamos el tiempo de ejecución real de cada procesador con trabajo, y escogemos aquel que vaya más atrasado en la ejecución. En este caso, p_1 sería el escogido.

Esta aproximación garantiza que ningún procesador se adelantará en exceso, por lo que se mantiene la característica de un entorno paralelo real. Además, como siempre habrá al menos un procesador que disponga de cajas, no habrá ningún instante de tiempo en el que no haya procesadores en ejecución y el tiempo real siempre avanzará.

6.4.2. Cálculo de tiempos

En la ejecución de un programa de Optimización Global entran en juego una serie de tiempos que marcarán los resultados recogidos tras su finalización.

La idea fundamental de nuestro algoritmo de Optimización Global con balanceo de la carga basado en regiones es determinar qué es más costoso: ¿evaluar las cajas en el procesador o enviar una parte de ellas y evaluarlas en otro procesador dividiendo así el trabajo? Para dar respuesta a esta pregunta, debemos disponer de una serie de variables, entre las que se encuentran los tiempos que vamos a presentar en esta sección.

Para realizar el cálculo de tiempos en el programa, utilizaremos la función `clock()` que se define así:

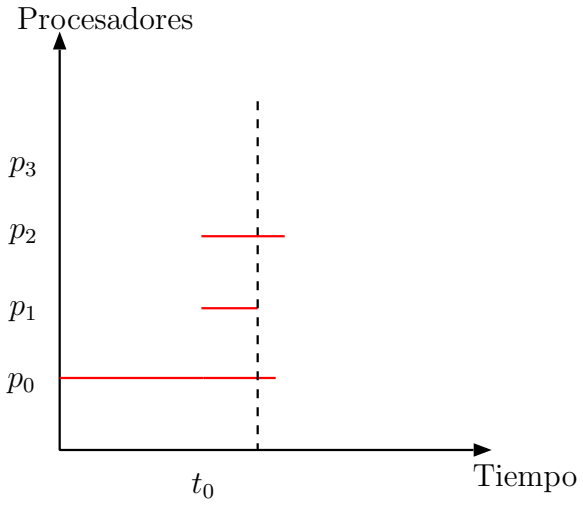


Figura 6.1: Selección del procesador que debe entrar en ejecución

```
clock_t clock(void)
```

La medición se debe realizar al principio y al final del bloque de código que queremos evaluar. El tiempo de evaluación será la diferencia de tiempos tomada al inicio y al final de ese bloque:

//comenzamos la medición de tiempo
$c_1 = clock()$
...
código a evaluar
...
//fin de la medición de tiempo
$c_2 = clock()$
//tiempo de proceso del bloque de código
$c = c_2 - c_1$

Para obtener el tiempo en segundos, utilizamos la constante `CLOCKS_PER_SEC`.

A continuación, explicaremos qué tiempos hemos tenido en cuenta durante el desarrollo de este proyecto, el por qué de su medición y uso, y la forma de obtenerlos o calcularlos.

6.4.2.1. El tiempo de creación de un proceso

Cuando estamos ejecutando un programa en un computador paralelo, éste divide parte de su trabajo para que sea ejecutado en los procesadores libres. Los procesadores libres, una vez han recibido el trabajo, tienen que iniciar un nuevo proceso para evaluar dicho trabajo. El tiempo que tarda un procesador en crear un nuevo proceso, depende en gran medida de la arquitectura sobre la que estemos trabajando. Es posible también que el proceso ya exista y esté esperando a recibir datos, por lo que el tiempo de creación del proceso sería 0. Para nuestro sistema paralelo simulado, hemos tenido en cuenta este tiempo a la hora de crear nuevos procesos en procesadores que estaban ociosos y lo hemos

notado por t_{cp}. El valor por defecto que hemos utilizado para el tiempo de creación de un proceso es de 100 *ms*, aunque podemos modificarlo con la opción `-tc` de nuestro programa (ver Sección 6.3).

6.4.2.2. El tiempo de proceso de una caja

El tiempo de proceso de una caja es el tiempo empleado por el procesador en evaluar una caja de la lista de trabajo. Los pasos que se siguen en el proceso de evaluación de una caja X son los siguientes:

1. Se extrae X de la lista.
2. Evalúo el punto medio de X para tratar de actualizar $\overline{f^*}$.
3. Si he actualizado $\overline{f^*}$ realizo el *Test de Corte*.
4. Divido X.
5. Para cada una de las cajas obtenidas tras la división:
   a) Calculo $F(X)$.
   b) Compruebo si es una caja final, si debo desechar la caja, y en otro caso, si hay que insertar la nueva caja a la lista.
   c) Calculo el trabajo de la caja si hay que insertarla en la lista (ver Sección 8.2.2).

Está claro que el tiempo de proceso de la caja dependerá en gran medida de la máquina en la que estemos trabajando, pero además, también dependerá de las condiciones en las que se encuentre la ejecución y de los parámetros utilizados en la misma. Por ejemplo, no se tardará lo mismo en procesar y dividir una caja para la función *Goldstein-Price* que para la función *Levy3*; o si en una ejecución utilizamos el Test de Monotonía y en otra no. Un valor fundamental que puede influir en este tiempo es el número de cajas que hay en la lista, puesto que no será lo mismo insertar una caja en una lista de 100 elementos que insertarla en una lista en la que hay 10000 cajas.

En este proyecto, se utiliza el tiempo de proceso de una caja para decidir si balancear la carga de trabajo (ver Sección 8.6). Para realizar este cálculo, hacemos la evaluación de un número determinado de cajas al comienzo de la ejecución. Así, obtendremos un tiempo de proceso diferente para cada tipo de función, midiendo el tiempo que empleamos en realizar dicha evaluación. El tiempo de proceso de una única caja se obtendrá dividiendo el tiempo de proceso de las cajas entre el número de cajas evaluadas:

$$t_{pc} = \frac{tiempo\_en\_procesar\_n\_cajas}{nb} \tag{6.1}$$

El número de cajas que se evaluarán por defecto es de 200 ($nb = 200$), aunque se puede cambiar este valor con la opción `-nb` (ver Sección 6.3). El cálculo se realiza al comienzo de la ejecución, antes de realizar algún reparto.

6.4.2.3. El tiempo de envío de una caja a otro procesador

Otro de los tiempos que debemos tener en cuenta en nuestro algoritmo de decisión de balanceo de la carga (ver Algoritmo 7) es el tiempo que tardamos en enviar un número determinado de cajas a otro procesador. Este tiempo depende de la tasa de transferencia de la red (*ttr*), de la *latencia* y del tamaño de las cajas que enviemos. Así, el tiempo de envío de n cajas a otro procesador en un sistema paralelo vendrá determinado por la Ecuación 6.2:

$$t_{ec}(n) = latencia + \frac{n \cdot sizeof(caja)}{ttr} \tag{6.2}$$

6.4.2.4. El tiempo de ejecución total de un procesador

El tiempo de ejecución de un procesador es, ni más ni menos, el tiempo que un procesador ha estado trabajando. Para realizar la medición de este tiempo, lo único que tendremos que hacer es medir el tiempo que el procesador ha estado en activo en la simulación. Utilizaremos este tiempo para conocer cuál es el tiempo de proceso real de nuestro algoritmo.

6.4.2.5. El tiempo de ejecución global de la simulación

El tiempo de ejecución global es el tiempo que tardaría nuestro algoritmo desde que empieza a ejecutarse hasta que el último procesador termina su trabajo, si nuestro algoritmo corriera en una máquina paralela. Debemos tener en cuenta que en un instante de tiempo cualquiera, pueden estar trabajando más de un procesador, por lo que nuestra máquina de simulación mantiene el tiempo real de ejecución de cada proceso en la línea de tiempo.

Vamos a ilustrarlo con un ejemplo. En la Figura 6.2 podemos ver un ejemplo de una línea de tiempo de una ejecución en paralelo. En el instante t_0 se realiza un reparto de cajas a los procesadores p_1 y p_2. En este momento, se actualiza el tiempo de ejecución real de los nuevos procesadores a t_0 que es el tiempo real en el que se ha producido el reparto. Este tiempo real se actualiza siempre que el procesador mantenga su ejecución, o en otro caso, cuando se queda sin trabajo y vuelve a recibir cajas por parte de otro procesador, como le ocurre al procesador p_1 en t_1. Para saber cuál es el tiempo de ejecución global de la simulación, sólo tenemos que comprobar qué procesador tiene el mayor tiempo de ejecución final.

Un error que no debemos cometer es que un procesador que quedó libre en un instante determinado, reciba cajas de un procesador cuyo tiempo de ejecución real sea menor, puesto que en teoría, no ha podido saber que ha quedado libre. Para solucionar esto, lo que hacemos es que a la hora de repartir, buscamos el procesador libre con menor tiempo de ejecución real, y comprobamos que ese tiempo es menor que el tiempo de ejecución real del procesador que reparte. Si no es así, el reparto de cajas no se realiza y proseguimos con la ejecución.

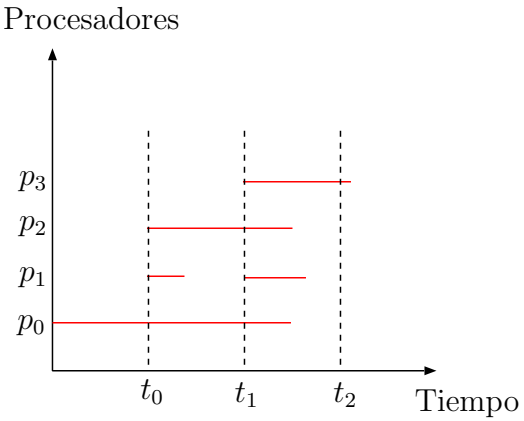


Figura 6.2: Cálculo del tiempo de ejecución global de la simulación

6.5. Estructura de la implementación

La implementación del proyecto se ha llevado a cabo utilizando el lenguaje de programación C++, la biblioteca de funciones XSC (*eXtended Scientific Computation*) para el lenguaje C (C-XSC 2.1.1), y el compilador GCC (versión 4.0.2).

La biblioteca XSC es un software utilizado para realizar cálculos científicos que requieren de validación. Además, pone a disposición un gran número de tipos de datos, operadores y funciones. Las principales características de C-XSC son:

- Aritmética de números reales, complejos, intervalos, y de intervalos de números complejos.
- Vectores y matrices dinámicas.
- Subarrays de vectores y matrices.
- Operadores aritméticos con alta precisión.
- Control de redondeo para datos de entrada/salida.

Vamos a explicar ahora los contenidos de los diferentes archivos y directorios en los que se ha estructurado el proyecto. Los directorios que nos encontramos son los siguientes:

doc En este directorio se encuentra la documentación del proyecto.

stats En este directorio se guardarán las estadísticas de las ejecuciones, y los gráficos generados para cada una de ellas.

include Aquí se encuentran los archivos de cabecera de la biblioteca XSC.

lib Aquí se encuentra la biblioteca XSC.

Por último, estos son los archivos en los que se ha estructurado el proyecto:

ogsp.cpp Es el archivo principal del proyecto. Se encarga de obtener los parámetros de entrada y realizar la simulación paralela del algoritmo de Optimización Global con el balanceador escogido.

estadisticas.hpp Contiene los atributos y funciones necesarios para gestionar las estadísticas del proyecto (ver Sección 6.6).

dwb.hpp y dwb.cpp Contiene los algoritmos de balanceo de la carga (ver Capítulos 7 y 8).

problems.h y problems.cpp Contiene la implementación de las funciones test.

functions.h y functions.cpp Aquí se encuentran funciones de utilidad desarrolladas para el proyecto.

procesador.hpp y procesador.cpp Contiene la clase `procesador` que gestiona las características de éstos en la simulación paralela (ver Sección 6.5.1).

regiones.hpp y regiones.cpp Contiene la clase `regiones` y las funciones para el manejo de las mismas (ver Sección 6.5.2).

lista2D.hpp y lista2D.cpp Contiene la clase `lista2D` que maneja la lista doblemente enlazada, utilizada para el mantenimiento de la lista de cajas (ver Sección 6.5.4).

caja2D.hpp y caja2D.cpp Contiene la clase `caja2D` que gestiona la estructura de las cajas (ver Sección 6.5.4).

punto2D.hpp y punto2D.cpp Contiene la clase `punto2D` que implementa las características de un punto en dos dimensiones.

makefile Archivo con las órdenes para compilar el proyecto.

Las siguientes secciones muestran con mayor detalle la implementación de las clases.

6.5.1. La clase Procesador

Para conseguir un entorno en el que el algorimo de Optimización Global pudiera ser probado sin necesidad de realizar una implementación paralela, fue necesario crear una estructura que se adecuara a nuestro problema, basada en la implementación de una clase que simulara a un procesador real. Esta clase, llamada `procesador`, permite la ejecución independiente del algoritmo de optimización, así como la comunicación que se realiza entre todos los procesadores presentes durante la ejecución.

Los atributos que contiene la clase son los siguientes:

idle Este atributo indica que el procesador está ocioso, es decir, no se encuentra en ejecución al no tener cajas que evaluar.

proc_id Es el número de identificación del procesador, donde $proc\_id \geq 0$.

max_proc Es el número máximo de procesadores que hay disponibles en la simulación, donde: $max\_proc \geq 1$. Si el valor de max_proc es uno, el problema se ejecuta como en el caso secuencial. No hay límite para el número de procesadores que podemos utilizar en la simulación.

***boxlist** Es un puntero a una lista que contiene las cajas que el procesador tendrá que evaluar. La implementación de la lista se puede ver en la Sección 6.5.4.

***finalist** Es un puntero a la lista de cajas finales, es decir, aquellas cajas cuyo ancho es menor que ϵ. La implementación de la lista se puede ver en la Sección 6.5.4.

****pregiones** Esta estructura es la que se utiliza para generar las regiones en el procesador. La Sección 6.5.2 explica la estructura y funcionamiento de esta clase, y más adelante, en la Sección 8.3 se explica el proceso seguido para crear las regiones en el procesador.

stats Es un objeto de la clase `stat_proc` que se encarga de guardar y gestionar todos los datos de la simulación relacionados con el procesador (ver Sección 6.6).

6.5.2. La clase Regiones

La clase `regiones` permite la creación, en un procesador, de las regiones de atracción de mínimos detectadas por el algoritmo. Estas regiones se crean a partir de la lista de cajas del procesador. Esta clase contiene un vector de punteros a cajas. Cada uno de los elementos de este vector contendrá la lista de cajas pertenecientes a una región.

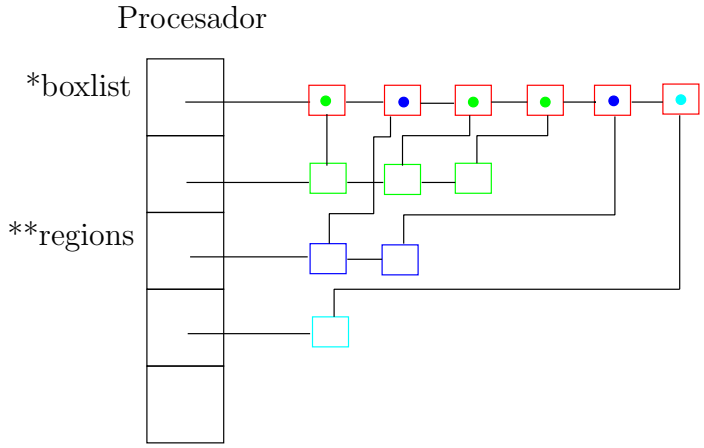


Figura 6.3: Estructura de las regiones y lista de cajas

La Figura 6.3 ilustra el diseño de las regiones. En ella se pueden ver tres regiones, la primera con tres cajas, la segunda con dos y la última con una caja. Estas cajas son

punteros a las cajas de la lista, por lo que podemos identificar qué cajas pertenecen a qué regiones recorriendo la lista una sola vez.

La clase regiones contiene sólo tres atributos:

max_regions Es el número máximo de regiones que se podrán crear. Este valor coincide con el número máximo de procesadores que hay en el sistema, ya que el algoritmo de generación de regiones no debe crear más regiones que procesadores tenemos.

num_regions Es el número de regiones que hay actualmente en el procesador. Este atributo es útil durante la generación de las regiones para comprobar el número de regiones que llevamos creadas; y posteriormente, para el reparto de las regiones entre los procesadores.

****regions** Matriz dinámica que contiene las cajas estructuradas por regiones (ver Figura 6.3).

Una vez hemos creado las regiones, llega el momento de repartirlas entre los procesadores libres. La primera región se queda en el procesador, y el resto, si las hubiera, se reparten. El proceso de creación de regiones se presenta en la Sección 8.3, y el movimiento de regiones a nuevos procesadores en la Sección 8.4.

6.5.3. La clase DWB

La clase `DWB` (*Dynamic Workload Balance*) contiene el algoritmo de balanceo MPR (*Minimus Prediction by Regions*) basado en la creación de regiones alrededor de los mínimos locales y globales. Este algoritmo, y todo lo que a él concierne se explica con todo detalle en el Capítulo 8.

6.5.4. Listas y cajas

Vamos a explicar a continuación, cómo se ha realizado la implementación de las cajas y la lista de cajas para este proyecto.

Cajas

Denominamos caja a una estructura que almacena los valores necesarios para el cálculo del rango de la función objetivo a partir de sus coordenadas. Estas coordenadas son el dominio donde la función será evaluada, $X \in \mathbb{I}^n$; y el rango de la función en ese dominio será la extensión a intervalos de la función, $f(X)$, estará delimitado por un límite superior al que llamaremos `Lsup` ($\overline{F}(X)$), y un límite inferior que denotaremos por `Linf` ($\underline{F}(X)$).

La implementación de la estructura que gestiona las cajas se ha realizado en la clase `caja2D`. Como ya dijimos, el proyecto ha sido implementado para probar funciones test bidimensionales, por lo que la implementación de esta clase se ha hecho para almacenar cajas bidimensionales únicamente. La clase `caja2D` contiene los siguientes atributos:

Lsup Es el límite superior del rango de la función objetivo en su extensión a intervalos, $\overline{F}(X)$.

Linf Es el límite inferior del rango de la función objetivo en su extensión a intervalos, $\underline{F}(X)$.

x1, x2 Son las coordenadas (dadas en intervalos) para las que la función objetivo será o ha sido evaluada.

trabajo Es el número de cajas que podrán generarse como máximo a partir de ésta. Es obvio que para cajas cuyo dominio sea mayor, este valor será más grande comparado con cajas cuyo ancho esté próximo al de una caja final. La definición exacta de trabajo de una caja y el cálculo de éste se verá más adelante en la Sección 8.2.

Lista de cajas

Una caja se divide por su lado más ancho, dando lugar a dos nuevas cajas. En todo este proceso, el número de cajas aumenta, por lo que se hace necesario almacenarlas en alguna estructura con prioridad. Nosotros hemos elegido almacenar las cajas en una lista doblemente enlazada, con punteros a la cabecera y a la cola de la lista. La lista, que ha sido implementada en la clase `lista2D`, mantiene ordenadas las cajas en orden creciente de `Linf` ($\underline{F}(X)$) de modo que siempre evaluaremos primero la caja con menor $\underline{F}(X)$ (la primera caja de la lista), como se describe en el Algoritmo 1.

Además, la lista mantiene actualizados los siguientes atributos:

Linfmin Almacena el menor valor de `Linf` $\underline{F}(X)$) de las cajas que hay en la lista. Como la lista se mantiene ordenada, éste corresponde al de la primera caja.

Linfmax Almacena el mayor valor de `Linf` $\underline{F}(X)$) de las cajas que hay en la lista. Como la lista se mantiene ordenada, éste corresponde al de la última caja.

trabajo Contiene el trabajo total en potencia de toda la lista (ver Sección 8.2.3).

numItems Indica el número de elementos que hay en la lista.

***cabecera** Es un puntero al primer elemento de la lista. Apunta a un objeto de la clase `itemlista2D`, que es la encargada de gestionar los punteros de la lista.

***cola** Apunta al último elemento de la lista.

La clase itemlista2D Esta clase se implementó para facilitar el mantenimiento de la lista doblemente enlazada. Los atributos de esta clase son los siguientes:

***ant,*sig** Son punteros a objetos de la clase `itemlista2D` (definición recursiva). Se encargan de mantener la lista doblemente enlazada.

***box** Apunta al objeto que contiene los datos de la caja (clase `caja2D`).

Listas y árboles Sabemos que una estructura en árbol proporciona una eficacia mayor con respecto a las listas. Cuando el número de elementos aumenta enormemente, el número de comprobaciones que hay que realizar para encontrar un elemento en el árbol es mínimo. En cambio, cuando la estructura elegida es una lista, el número de comprobaciones crece de forma polinómica. La elección de una u otra estructura influye en los resultados obtenidos ya que las operaciones sobre árboles son mucho más eficientes. Nosotros hemos escogido la estructura de lista por su mayor sencillez en la implementación de todas las operaciones de mantenimiento.

6.6. Resultados de la simulación

Todo proyecto, no importa su finalidad (investigación, innovación, de trabajo, ...), debe mostrar unos resultados a partir de los cuales podremos obtener una serie de conclusiones que nos hagan reflexionar y recapacitar sobre su viabilidad y utilidad.

En nuestro caso, ha sido necesario obtener una serie de datos que hagan esta reflexión posible. En primer lugar, hemos recogido datos sobre las ejecuciones simuladas de los procesadores que intervienen en ella:

Esfuerzo Es una medida de la cantidad de trabajo ejecutado. Se calcula como $Esf(p) = FE + n \cdot GE$, donde FE es el número de evaluaciones de la función de inclusión, n las dimensiones del problema y GE el número de evaluaciones de la derivada de la función de inclusión.

Número de cajas evaluadas Indica el número de cajas que ha evaluado cada procesador.

Máximo número de cajas Indica el número máximo de cajas que ha habido en el procesador durante la ejecución. Es importante conocer este número ya que indica los requisitos de memoria del programa.

Tiempo de proceso Indica el número de segundos que el procesador ha estado trabajando.

Fin del proceso Indica el momento en el que el procesador acabó su ejecución desde que comenzó la ejecución del programa si hubiera estado ejecutándose de forma paralela. Es lo que hemos llamado tiempo real de ejecución.

Tiempo de envío de cajas Indica los segundos que el procesador ha empleado en enviar cajas a otro procesador.

Tiempo en crear nuevos procesos Indica los segundos que el procesador ha empleado en crear nuevos procesos. Esto ocurre cuando un procesador reparte cajas a un procesador libre y éste tiene que crear los nuevos procesos ($tc > 0$).

Además de estos valores por procesador, el programa también muestra una serie de valores globales de la ejecución del programa:

Número máximo de procesadores utilizados Indica, de todos los procesadores disponibles, cuántos hemos utilizado.

Esfuerzo total Es el sumatorio del esfuerzo realizado por todos los procesadores.

Desbalanceo Indica el nivel de balanceo de la carga que ha habido en la ejecución.

Número total de cajas evaluadas Indica el número total de cajas que se ha evaluado en la ejecución.

Promedio del número máximo de cajas Es la media aritmética del número máximo de cajas por procesador.

Tiempo total de ejecución Indica los segundos que ha tardado el programa en finalizar en la línea de tiempo real, es decir, el tiempo que tardaría el programa en finalizar si los procesadores trabajaran en un sistema paralelo real.

Tiempo total de proceso Es el sumatorio de los tiempos de proceso de todos los procesadores.

Número total de predicciones Si se ejecuta el programa con el balanceo mediante regiones activado (opción `-r` o `-y`), indica el número de veces que se ha ejecutado la creación de regiones. Si se ejecuta con el método de reparto en baraja (opción `-b`), indica el número total de repartos de cajas que se han realizado.

Número de predicciones inútiles Sólo es válido para la ejecución del algoritmo de predicción. Indica en cuántas ocasiones el algoritmo de predicción no ha sido capaz de detectar más de una región. Este dato es útil, ya que ejecutar un número elevado de veces el algoritmo de predicción sin éxito, hace que la eficiencia del programa disminuya.

6.7. Algoritmo de simulación paralela

En este capítulo se han explicado las características de nuestro programa de Optimización Global mediante una ejecución paralela simulada, **ogsp**. Ahora, mostraremos la descripción del funcionamiento del programa en el Algoritmo 2.

Los datos de entrada del programa son:

- P: Los procesadores disponibles en el sistema.
- S: Espacio de búsqueda de la función objetivo.
- F: Extensión natural a intervalos de la función objetivo.
- ϵ: Criterio de terminación ($w(X) < \epsilon$).
- ttr: Tasa de transferencia de la red.

- l: Latencia de la red.
- nb: Número de cajas a evaluar para el cálculo del tiempo de proceso de una caja.
- tcp: Tiempo de creación de un proceso.
- $balanceador$: Algoritmo de balanceo de carga escogido.
- $decision$: Indica si se realiza decisión antes del balanceo.

Algoritmo 2 Algoritmo de Optimización Global mediante simulación paralela

Funct ogsp$(P, S, F, \varepsilon, balanceador, ttr, l, nb, tc, tpc)$

1. $P_e := P^1$ — *El primer procesador entra en ejecución*
2. $InitProcesador(P, S)$ — *Se inicializa la lista de trabajo del procesador*
3. **while** ($nb \neq 0$) — *Calculamos el tiempo de proceso de una caja*
4. $\quad P_e \rightarrow OGBasico$ — *Se evalúa una caja en P_e según el Algoritmo 1*
5. $\quad nb := nb - 1$
6. $t_{pc} := CalcularTprocesoCaja(nb)$ — *ver Ecuación 6.1*
7. **while** ($P \neq \{\emptyset\}$) — *Mientras haya procesadores con trabajo*
8. $\quad P_e := BuscarPrimerOcupado(P)$ — *Procesador con menor tiempo de ejecución real.*
9. $\quad P_e \rightarrow OGBasico$ — *Se evalúa una caja en P_e según el Algoritmo 1*
10. $\quad P_e \rightarrow BalancearCarga(balanceador, decision)$

El Algoritmo 2 comienza seleccionando el primer procesador para su inicialización (líneas 1 y 2). A continuación el procesador evalúa un número de cajas, determinado por el parámetro nb para calcular el tiempo que se tarda en procesar una caja. La evaluación de las cajas se realiza tal y como se describió en el Algoritmo 1 (línea 4). Después de ese cálculo comienza el bucle principal del algoritmo (línea 7) que comprueba si hay procesadores con cajas. Si es así, seleccionaremos aquel que tenga menor tiempo de ejecución (línea 8) (ver Sección 6.4.1). Tras evaluar una caja (línea 9), el procesador ejecuta el algoritmo de balanceo elegido para la simulación (línea 10), que podrá ser el algoritmo de balanceo en baraja (ver Capítulo 7) o el algoritmo de balanceo con predicción de mínimos basado en regiones (ver Capítulo 8). Después de realizar el balanceo de la carga, el algoritmo vuelve a la línea 7 para seguir evaluando cajas de los procesadores activos.

Parte IV

Balanceo dinámico de la carga

Capítulo 7

Balanceo de la carga mediante el reparto de cajas

7.1. Descripción general

El algoritmo de balanceo de la carga mediante el reparto de cajas es una aproximación básica en los algoritmos de Optimización Global paralelos. Estos algoritmos hacen uso de todos los procesadores del sistema, y en cuanto uno queda libre, informa a los demás de su estado y posteriormente recibe cajas por parte de otro. En nuestro modelo de simulación paralela, detectar qué procesador está libre y cuál será el procesador que le envíe cajas es una tarea mucho más simple. Las técnicas de balanceo de carga utilizadas son distintas en cada algoritmo paralelo, pero en general siguen las indicaciones que presentamos en la Sección 5.3.4. El algoritmo de balanceo presentado en este capítulo intenta minimizar el tiempo que un procesador está ocioso.

En el algoritmo que vamos a presentar en este capítulo, el reparto de cajas se realiza de forma muy sencilla: el procesador que reparte recorre su lista de cajas enviando al otro procesador cajas de forma alternativa. Este reparto es similar a repartir una baraja de cartas entre dos personas, una carta para cada uno de forma alternativa. Por ello, lo hemos llamado algoritmo de balanceo de la carga en Baraja. Con este método se consiguen dos cosas: (1) equilibrar el número de cajas de los dos procesadores y (2) equilibrar la carga de trabajo, ya que si enviamos las cajas a partir de la mitad, esas cajas serán menos prometedoras por tener mayores límites inferiores, y probablemente sean eliminadas pronto.

Para contrastar los resultados obtenidos con nuestro algoritmo de balanceo basado en la predicción de mínimos, decidimos implementar este otro algoritmo basado en el reparto de cajas. Ambos algoritmos se ejecutan sobre nuestro sistema de simulación paralela (ver Capítulo 6), de tal forma que los resultados obtenidos con ambos algoritmos serán comparables entre sí.

En primer lugar, explicaremos cuándo el algoritmo debe realizar el reparto de cajas; y en segundo lugar, explicaremos qué procesadores estarán implicados en este reparto. Por último, presentaremos el algoritmo de balanceo en Baraja tal y como ha sido implementado.

7.2. Decisión del reparto de cajas

Tomar la decisión de cuándo tenemos que repartir cajas es muy sencilla para este método. En primer lugar, el procesador que se encuentra ejecutándose en la simulación, tras realizar todo el ciclo de evaluación de una caja, debe comprobar si hay que repartir cajas. La condición principal que debe cumplirse es que haya procesadores libres. Está claro que si todos los procesadores están trabajando, no habrá que repartir cajas, pero dadas las características de nuestra plataforma de simulación, habrá que comprobar ciertos detalles de la ejecución antes de repartir (aún cuando haya procesadores libres).

En la Sección 6.4.1 explicábamos que el procesador que entra en ejecución en cada iteración (recordemos que nuestro programa realmente se ejecuta en secuencial) era aquel que tenía un menor tiempo de ejecución real. En un sistema paralelo real, cuando un procesador queda libre, informa al resto de su estado y recibe cajas por el primer procesador que detecta esta situación. Esto se puede lograr, por ejemplo, a través de una variable compartida en una Arquitectura Paralela de memoria compartida. Para lograr este funcionamiento en nuestra simulación, debemos comprobar los tiempos de ejecución de los procesadores antes de repartir. Por ejemplo, imaginemos que en nuestro sistema simulado el procesador P_1 queda libre en el tiempo de ejecución t_1. A continuación, pasaría a ejecutarse el procesador con menor tiempo de ejecución, que llamaremos P_2. Cuando este procesador evalúa una caja, comprobaría si debe hacer un reparto de cajas. Si sólo se comprobara si hay procesadores libres, inmediatamente se realizaría el reparto, ya que detectaría que el procesador P_1 se encuentra ocioso. Esta situación puede llevarnos a un grave error puesto que, si en el momento del reparto, el procesador P_2 tiene un tiempo de ejecución t_2 menor que t_1; en un sistema paralelo real, no sería posible que ese procesador realizara el reparto, ya que no puede saber de antemano que el procesador quedará libre.

Por tanto, un procesador puede repartir cajas a otro libre, si el tiempo de ejecución real del procesador que reparte es mayor que el tiempo de ejecución real en el que el procesador quedó libre.

7.3. Elección de procesadores

De la sección anterior se deduce que antes de repartir las cajas, debemos iniciar un proceso que seleccione los dos procesadores implicados: uno es el que repartirá las cajas, y el otro el que las recibirá.

Elección del procesador que reparte Para elegir el procesador que reparte las cajas en nuestro sistema, debemos fijarnos en lo que ocurre en un sistema paralelo real. En éste, el procesador que reparte es aquel que primero observa que hay un procesador ocioso. En nuestro caso, como los procesadores tienen tiempos de ejecución distintos, escogeremos aquel que tenga mayor tiempo de ejecución.

Elección del procesador que recibe La elección del procesador que recibe las cajas es muy sencilla: será aquel procesador ocioso cuyo tiempo de ejecución sea menor. La

explicación es bien sencilla: en el caso de que hubiera más de un procesador libre, habría que repartir las cajas a aquél que hubiera quedado libre antes.

Este proceso, en un sistema multiprocesador real es más difícil de conseguir debido a sus características (sincronización, latencia de las comunicaciones, etc.).

7.4. Algoritmo de balanceo de la carga en Baraja

El Algoritmo 3 describe el funcionamiento del algoritmo de balanceo de carga presentado en este capítulo. Este algoritmo se ejecuta en cualquier procesador que detecta que un procesador del sistema ha quedado ocioso. Los parámetros del algoritmo se describen a continuación:

- P_L: Identificador del procesador que está libre.
- L: Lista de cajas activas del procesador que ejecuta el algoritmo.
- L_1: Primera lista auxiliar.
- L_2: Segunda lista auxiliar

Algoritmo 3 Algoritmo de balanceo de la carga en Baraja

Funct Baraja(P_L,L,L_1,L_2)

1. $contador := \emptyset$
2. **if** ($numeroElementos(L) < 2$) — *No hay cajas suficientes para repartir*
3. **return** 0
4. **while** ($L \neq \{\emptyset\}$) — *Repartimos las cajas*
5. $X := seleccionaCaja(L)$ — $\underline{F}(X) = min(\underline{F}(X^i), \forall X^i \in L$
6. $L := L - \{X\}$
7. **if** ($contador \,\%2 = 0$)
8. $L_1 := L_1 + \{X\}$ — *X a la lista* L_1
9. **else**
10. $L_2 := L_2 + \{X\}$ — *X a la lista* L_2
11. $contador := contador + 1$
12. $MoverCajas(P_L, L_2)$ — L_2 *se envía al procesador libre*
13. $L := L_1$ — L_1 *se queda en el procesador que ejecuta el balanceo*
14. **return** $numeroElementos(L_2)$

El Algoritmo 3 comienza inicializando la variable auxiliar *contador* (línea 1) y comprobando si hay cajas suficientes para realizar el balanceo (línea 2). Para poder hacer este balanceo de la carga, es necesario que en la lista de trabajo haya al menos 2 cajas. A continuación entramos en el bucle principal del algoritmo (líneas 4 a 11) que se ejecuta mientras haya cajas en la lista de cajas activas L del procesador. En este bucle se extraen las cajas de L y se insertan alternativamente en L_1 y L_2 en cada iteración. Cuando se

ha terminado de repartir las cajas en ambas listas, se realiza el movimiento de las cajas de la lista L_2 al procesador libre (P_L) (línea 12) mientras que el procesador que ejecuta el algoritmo de balanceo se queda con las cajas de la lista L_1 (línea 13). Finalmente, el algoritmo retorna el número de cajas que han sido enviadas al procesador libre.

Capítulo 8

Balanceo de la carga basada en la predicción de mínimos

8.1. Descripción general

En este capítulo se introducen todos los conceptos utilizados en la creación de un nuevo algoritmo de balanceo de la carga y estimación del número de procesadores necesarios para resolver el problema, en un momento dado de la ejecución, basado en generar regiones de cajas próximas entre sí, cuyos centros de atracción tratarán de estar localizados sobre los mínimos de la función objetivo. Comprobaremos su eficiencia, y la compararemos con un algoritmos de balanceo basado en el reparto de cajas entre procesadores (ver Capítulo 7). Este algoritmo de balanceo de carga con predicción de mínimos basado en regiones lo hemos llamado algoritmo de balanceo MPR (*Minimus Prediction by Regions* o Predicción de Mínimos mediante Regiones).

Este proyecto parte de la base de que los algoritmos paralelos pierden demasiado tiempo en la comunicación con otros procesadores, sobre todo cuando un procesador queda libre y recibe cajas de otro procesador. Decimos ésto porque, en muchas ocasiones, es más costoso enviar esas cajas a otro procesador que evaluarlas en el mismo procesador, debido al coste de las comunicaciones y a que esas cajas contienen poco **trabajo**.

El concepto **trabajo de una caja**, que se explica en la Sección 8.2, se utilizará en este proyecto, y en concreto en el algoritmo MPR (aunque puede ser utilizado por cualquier algoritmo de balanceo de carga), para decidir cuándo existe trabajo suficiente para crear las regiones y moverlas a otro procesador.

La creación de regiones persigue, además de obtener un nuevo algoritmo de balanceo dinámico basado en la localidad de los subproblemas, determinar en trabajos futuros las propiedades y características que pudieran tener en común estas cajas, próximas entre sí. En particular, pensamos que el disponer en un mismo procesador de cajas con una localidad parecida, incrementa la información acerca de dónde se encuentra el mínimo, ya que podríamos analizar la dirección que estamos siguiendo en el proceso de evaluación de las cajas (ver Figura 8.1).

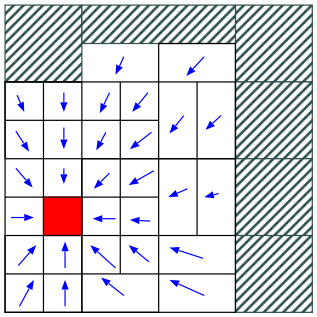


Figura 8.1: Ilustración de una región de atracción de un mínimo.

8.2. El trabajo de una caja

Una de las consideraciones más importantes que vamos a tener en cuenta para decidir si se van a repartir cajas de un procesador al resto de procesadores libres (a tantos como regiones se hayan creado), es si hay suficiente carga de trabajo como para emplear tiempo de proceso en enviar cajas a otro procesador.

Dependiendo del balanceador de carga de un algoritmo de Optimización Global, éstos pueden realizar repartos de cajas sin comprobar el trabajo de la caja, basándose sólo en el número de cajas. Otros algoritmos [5] incorporan estimaciones del trabajo de una caja para tomar la decisión de balanceo.

Con nuestra aproximación, comprobamos si hay suficiente trabajo en el procesador, y si así es, procedemos a crear las regiones, que en caso de que haya más de una (puesto que una región se queda en el procesador que las genera), las enviaremos a procesadores libres.

8.2.1. Definición de trabajo de una caja

Debemos explicar qué entendemos por trabajo de una caja para poder utilizar este concepto. Sabemos que en el proceso de búsqueda de mínimos las cajas son divididas, evaluadas y comprobadas para, su rechazo, inserción en la lista de trabajo o en la lista de cajas finales. Una caja se considerará caja final cuando el ancho de ésta sea menor que la precisión requerida para la solución, notada por ϵ. Por tanto, nosotros podemos saber, a partir de una caja cualquiera X, conocido el ancho de ésta, $w(X)$, cuántas cajas se pueden generar hasta que todas ellas tengan un ancho menor que ϵ, es decir, hasta que todas las cajas sean finales.

Definición 4 *Se define el número de descendientes de una caja X, y se notará por $desc(X)$, como el número máximo de cajas X^i que se pueden generar a partir de ella tras divisiones sucesivas, hasta alcanzar la precisión $w(X^i) < \epsilon$.*

Por tanto, el *número de descendientes* de una caja depende de su anchura $w(X)$, siendo ésta la de su lado mayor; y de la precisión del problema ϵ, que determina la anchura mínima de una caja para considerarla final. Está claro que este número es una cota superior del

número de cajas generadas en realidad, puesto que habrá cajas que sean rechazadas por el *Test de Cota* o por cualquier otro test, con lo que el número de cajas que se generarán se verá reducido. Esto indica que el trabajo de una caja no podrá ser mayor que el número de descendientes que pueda tener, por lo que será necesaria una estimación:

Definición 5 *Se define el trabajo de una caja X, y se notará por $wk(X)$, como el número de cajas estimado que se evaluarán por el algoritmo de Optimización Global, siendo el número de descendientes de X, $desc(X)$, una cota superior de este trabajo.*

Para obtener una aproximación real del trabajo de una caja, debemos utilizar un estimador que acote el *número de descendientes*, ya que, de otro modo, nuestro valor de trabajo estará muy por encima del valor real. El estimador que hemos escogido es el $p\overline{f^*}$, que como se comprobó en la Sección 4.4.1, es un buen indicador de la cercanía o lejanía de la caja a un mínimo, y por tanto, de la cantidad de trabajo de dicha caja.

8.2.2. Cálculo del trabajo de una caja

Para calcular el trabajo de una caja X, tenemos que calcular primero el número de posibles descendientes. Para una caja X cualquiera de dos dimensiones, suponiendo que utilizamos el método de bisección para dividirla, el número total de cajas generadas (*número de descendientes*) viene dado por la fórmula del número de elementos en un árbol binario de nivel n: $2^n - 1$

Sea X una caja bidimensional, y $w(X)$ el ancho de la caja, el número de descendientes de la caja X se calcula como:

$$desc(X) = 2^{i+1} - 1$$

donde i, es el número de divisiones que hay que realizar para que las cajas alcancen la anchura mínima de una caja final ($w(X) < \epsilon$):

$$i = \frac{2 \cdot \log \frac{w(X)}{\epsilon}}{\log 2}$$

Como hemos dicho antes, $desc(X)$ es una cota superior del número de cajas reales que se generarán. Por tanto, necesitamos un estimador que aproxime el trabajo total de manera real. Como dijimos en la sección anterior, el estimador elegido ha sido el $p\overline{f^*}$, con lo que, la fórmula del cálculo de trabajo de una caja, que notaremos por $wk(X)$, quedaría así:

$$wk(X) = \left(2^{\frac{2 \cdot \log \frac{w(X)}{\epsilon}}{\log 2} + 1} - 1 \right) \cdot p\overline{f^*} \tag{8.1}$$

8.2.3. Cálculo del trabajo en la lista de cajas

La estructura en la que se almacenan las cajas en el procesador, es una lista doblemente enlazada en la que las cajas se encuentran ordenadas de menor a mayor valor de $\underline{F}(X)$. El trabajo estimado que un procesador tiene pendiente, debe actualizarse cada vez que se inserten o se eliminen cajas.

Como vimos en la Sección 6.5.4, el atributo **trabajo** de la clase `lista2D` almacenaba el trabajo total de la lista. Lo único que debemos hacer es gestionar este atributo cuando insertamos o extraemos una caja de la lista, o cuando eliminamos cajas mediante el *Test de Cota*, o cualquier otro test. También deberemos modificar el trabajo de la lista cuando hacemos un envío de cajas a otros procesadores.

Definición 6 *Sea $wk(X)$ el trabajo estimado de una caja cualquiera, calculado según la Ecuación 8.1, el trabajo total estimado para una lista de cajas L se notará por $wp(L)$ y se calcula cómo:*

$$wp(L) = \sum_i wk(X^i), \forall i X^i \in L \tag{8.2}$$

Además, hemos de tener en cuenta que el trabajo de las cajas, al utilizar el estimador $p\overline{f^*}$, depende del estado en el que estaba la caja en el momento de hacer ese cálculo, ya que este estimador utiliza valores que varían durante la ejecución, como $\overline{f^*}$ (ver Ecuación 4.2). Por tanto, a medida que la ejecución avanza, habrá cajas cuyo trabajo no refleje la situación actual, y por tanto, el valor de trabajo en la lista tampoco reflejará esa situación.

Para solucionar este problema, y mantener en la lista una estimación del trabajo lo más actualizada posible sin afectar la eficiencia del algoritmo, se recalcula el trabajo de las cajas durante el *Test de Cota*, actualizando el trabajo en la lista de cajas (no en la lista final) aprovechando el recorrido que hacemos por ella.

8.3. Generación de regiones

Cuando nos enfrentamos por primera vez a analizar este problema, vimos que el asunto era, cuanto menos, difícil de abordar. Sobre el papel, la idea parecía sencilla: agrupar cajas de acuerdo a su localidad espacial. Pero en la práctica, había una serie de preguntas a las que era difícil dar respuesta: ¿cuántos grupos o regiones de cajas tendríamos que hacer? o ¿qué cajas pertenecerían a qué regiones? y ¿cuándo generaríamos esas regiones?

Nosotros, presentaremos aquí una respuesta a todas estas preguntas, que posiblemente no es la única ni la mejor, pero que a la vista de los resultados, es una aproximación a tener en cuenta.

8.3.1. Candidatos y líderes de una región

Empezaremos dando respuesta a la primera de las preguntas planteadas: ¿cuántos grupos o regiones de cajas tendríamos que hacer?

De todas las cajas que hay en el procesador, elegiremos aquellas que pueden estar más cerca de un mínimo, esto es, las cajas con menor $\underline{F}(X)$. El problema que se nos plantea ahora es, decidir qué cajas vamos a considerar cercanas a un mínimo (a las que llamaremos **cajas candidatas**), o lo que es lo mismo, para qué valor de $\underline{F}(X)$, una caja va a considerarse candidata a generar una región. A este problema lo hemos llamado **determinación del umbral de candidatos**, y se explica con todo detalle en la Sección 8.3.1.1.

Una vez determinadas cuáles son las cajas candidatas a liderar una región, debemos elegir, de entre ellas, a los auténticos **líderes**. Para realizar esta elección, hemos de tener en cuenta dos datos fundamentales: número máximo de regiones y cercanía de candidatos. La elección de líderes se explica con todo detalle en la Sección 8.3.1.3.

Para concluir el proceso de generación de regiones, una vez escogidas las cajas líderes de cada región, bastará con asignar el resto de cajas a la región adecuada. Este proceso de asignación de cajas a regiones se explica en la Sección 8.3.2.

8.3.1.1. El umbral de candidatos

Como se ha comentado anteriormente, elegiremos las cajas candidatas a ser líderes de una región, de acuerdo con su cercanía a un mínimo. Esta cercanía a un mínimo viene determinada, en mayor o menor medida, por su $\underline{F}(X)$. Por tanto, el problema consiste en establecer una cota o **umbral** para el valor de $\underline{F}(X)$ de la lista de cajas. Todas aquellas cajas cuyo $\underline{F}(X)$ esté por debajo de este umbral, serán **candidatas**.

Para realizar el cálculo del umbral de candidatos, hemos tenido en cuenta los siguientes factores:

- El número de procesadores disponibles: `max_proc`
- El mayor valor de $\underline{F}(X)$ en el procesador (en la lista de cajas): `Lmax`
- El menor valor de $\underline{F}(X)$ en el procesador (en la lista de cajas): `Lmin`

La fórmula que establece el valor umbral es la siguiente:

$$umbral = Lmin + \frac{Lmax - Lmin}{max\_proc} \tag{8.3}$$

Está claro que el valor que establezcamos como umbral, tiene que ser mayor que el menor valor de Linf de la lista ($Lmin = \min_i\{\underline{F}(X^i) : X^i \in L\}$), de otro modo, no habría cajas que pudieran ser candidatas. Se puede ver en la fórmula que el umbral se obtiene mediante la suma de un cierto valor a éste `Lmin`. Ese valor que le sumamos al `Lmin`, proviene de una fracción. Analicemos los dos miembros de esta fracción para comprobar las implicaciones de la misma:

El numerador: Lmax-Lmin La diferencia entre el valor máximo y mínimo de $\underline{F}(X)$ en el procesador tiene una consecuencia clara. Si esta diferencia es grande, es probable que muchas de las cajas que se encuentran en el procesador no hayan sido divididas lo suficiente, por lo que esperamos encontrar regiones en ella. En caso contrario, si la diferencia es pequeña, quiere decir que las cajas tienen valores de $\underline{F}(X)$ similares,

próximos entre sí, lo que puede significar un grado de división y cercanía a los mínimos parecido, por lo que, probablemente, habrá pocas regiones en el procesador.

El denominador: max_proc Este valor trata de reducir la cota del umbral, de tal forma que nos quedemos con las mejores cajas candidatas posibles, aún cuando tengamos más procesadores disponibles. Es decir, se trata de generar regiones con una gran cantidad de trabajo que no se eliminen rápidamente. A simple vista puede parecer que a mayor número de procesadores, menos regiones se van a generar, y es cierto, pero no debemos olvidar que este algoritmo se ejecuta en diferentes procesadores de forma paralela. Por tanto, al principio es fácil generar regiones suficientes para todos los procesadores, ya que el valor del numerador suele ser alto. A medida que avanza la ejecución, la generación de regiones se complica y se diversifica entre los procesadores en ejecución, por lo que sólo se crearán regiones con suficiente carga de trabajo. Es preferible tener cuatro procesadores trabajando y que cada uno genere dos o tres nuevas regiones, con lo que sumarían de ocho a doce nuevos procesadores, a tener cuatro procesadores y que cada uno genere cuatro o cinco nuevas regiones, lo que supondría pasar a ejecutar el programa paralelo en veinte o veinticuatro procesadores.

Por último, queremos destacar que esta fórmula que acabamos de presentar para determinar el umbral de cajas candidatas es totalmente heurística y que los resultados obtenidos a partir de ella han sido aceptables (ver Capítulo 9).

8.3.1.2. Máximo número de regiones que pueden generarse

El número máximo de regiones que puede crear el algoritmo de balanceo MPR viene determinado por el número de procesadores libres.

Hemos de saber que el fin que perseguimos al generar las regiones es enviar todas las cajas de dichas regiones a otros procesadores, cada región a un sólo procesador. Para establecer el número máximo de regiones a generar, podríamos optar por una de estas dos opciones:

1. Generar como máximo tantas regiones como procesadores libres tenemos (más una que se queda el procesador que ejecuta el algoritmo). De esta forma nos aseguramos de que cada región se asigne a un procesador libre.

2. Generar como máximo dos regiones, una que se enviará a un procesador libre y la otra que se quedará en el procesador que crea las regiones.

La solución escogida en este proyecto es la planteada en el punto 2, ya que el crecimiento de esas regiones puede ser exponencial, puesto que cualquier procesador puede ejecutar el algoritmo de balanceo MPR. No descartamos el estudio de otras alternativas en posibles trabajos futuros.

8.3.1.3. Elección de líderes

Ya conocemos qué cajas son candidatas a ser líderes de una región, y también conocemos cuántas regiones como máximo podemos generar. Por tanto, ya sabemos cuántas cajas líderes necesitamos.

La elección del primer líder es sencilla: la primera caja que esté por debajo del umbral establecido será el primer líder de una región. No hacemos ningún otro tipo de comprobación. Por tanto, el primer líder será la primera caja de la lista. Recordemos que la lista está ordenada de menor a mayor valor de $\underline{F}(X)$.

La elección del segundo líder se realiza también comprobando el umbral, pero además, debemos atender a un segundo factor: **la cercanía a los líderes ya seleccionados**. Por tanto, en este punto se introduce un nuevo e importante concepto: **la cercanía entre cajas**. Este nuevo concepto surge de la necesidad de generar regiones que estimen la localización de los mínimos del problema, ya que vamos a dividir el trabajo de acuerdo a esa localidad.

Uno de los problemas que se trata de solucionar con este concepto es el de no escoger dos cajas líderes muy próximas entre sí. Lo habitual, en el proceso de evaluación de las cajas es que, llegado a un cierto nivel de división, las cajas alrededor de un mínimo tengan valores de $\underline{F}(X)$ muy parecidos. Dependiendo de este nivel de división (y de las dimensiones del problema), alrededor de una caja elegida como líder, puede haber diez, cien, mil... cajas con valores de $\underline{F}(X)$ similares a ella. Está claro que si eligiéramos estas cajas como líderes de una región, en seguida completaríamos el número de regiones que podemos generar, sin lograr una división en regiones adecuada.

Así, la elección de los líderes a partir del primero pasa por comprobar, además de su umbral, la cercanía a todos los líderes ya seleccionados. Si la caja que estamos comprobando no es cercana a ninguno de los líderes actuales, será seleccionada como líder para otra región. Este proceso se repite hasta recorrer todas las cajas o completar el número de regiones máximo que podemos generar.

Cercanía de cajas Tenemos que establecer cuándo una caja es cercana a otra. Esta distinción es clave para obtener resultados óptimos en el algoritmo de balanceo MPR, ya que de ella depende todo el proceso de creación de regiones, tanto en la elección de líderes visto en esta misma sección, como en la asignación de las cajas a las regiones como se verá en la Sección 8.3.2. Para determinar si dos cajas son cercanas, hemos tenido en cuenta los siguientes factores:

- Las dimensiones de la región de búsqueda activa en el procesador en el momento de la ejecución del algoritmo de balanceo y,
- El grado de división de las cajas.

En las primeras pruebas realizadas del algoritmo de balanceo MPR, observamos que si sólo teníamos en cuenta las dimensiones del problema inicial S, no se encontraban líderes cuando la ejecución estaba avanzada y cuando la precisión requerida para la solución era alta. De esta forma, la ejecución mediante la predicción de regiones era poco eficiente ya

que no generaba regiones cuando había gran cantidad de trabajo próximo entre sí. Para solucionar este problema, se pensó en establecer como límites del espacio, los determinados por las cajas que se encuentran actualmente en la lista de trabajo del procesador. De este modo, el tamaño de la región de búsqueda se aproxima más a la realidad. Así, el algoritmo puede encontrar líderes cuando el grado de división de las cajas y la precisión requerida para la solución son elevados, y por tanto generar las regiones de cajas.

En primer lugar, para determinar la cercanía entre una caja líder y otra candidata, necesitamos conocer la relación existente entre el tamaño de la región de cajas activas en el procesador, que notaremos por S' y el ancho de la primera caja líder, que notaremos por X_L. A esta relación la llamaremos factor de cercanía y lo notaremos por f_c:

$$f_c = \frac{w(S')}{w(X_L)} \tag{8.4}$$

El factor de cercanía, f_c, se utiliza para ponderar la distancia euclídea, d_e, entre dos cajas. Esta ponderación es necesaria para ajustar d_e al momento de la ejecución (grado de división de ambas cajas, precisión, ...). El cálculo de f_c presentado en la Ecuación (8.4) ha ofrecido resultados satisfactorios, pero no descartamos una investigación más a fondo sobre nuevas propuestas para este factor.

En segundo lugar, hemos establecido que la distancia mínima, d_{min} que tiene que haber entre la caja líder, X_L y una caja candidata cualquiera, X_C para que ésta no se añada a la región y pueda convertirse en nuevo líder, tiene que cumplir:

$$d_{min} = \frac{f_c}{2} \cdot w(X_C) \tag{8.5}$$

Es decir, la caja candidata X_C debe encontrarse al menos a más de la mitad de distancia según el factor de cercanía calculado, ya que queremos generar dos regiones.

Definición 7 *Sean dos cajas X_1 y X_2, y d_e la distancia euclídea entre los centros de ambas cajas. La caja X_1 es cercana a X_2 cuando se cumple que:*

$$d_e < d_{min} \tag{8.6}$$

Los cálculos de f_c y d_{min} se realizan conforme a las ecuaciones (8.4) y (8.5).

La Figura 8.2 explica de manera visual lo que se pretende con lo explicado anteriormente. En la figura suponemos que la caja X_L ha sido elegida Caja Líder, por lo que el resto de cajas candidatas a ser líderes de la región (todas aquellas cuyo valor de $\underline{F}(X) < umbral$) deberán comprobar su cercanía con la caja líder. Cuando encontremos una caja de entre las candidatas que verifique la Ecuación 8.6, esa será una nueva caja líder. Lo ideal sería lo que se muestra en la Figura 8.2, que las cajas que se encuentran alrededor de la caja X_L sean determinadas como cercanas a ella para formar una región entorno a ese mínimo, y que la caja X_C (u otra entorno a ella) que pertenece a la zona de atracción de otro mínimo sea considerada líder de una nueva región.

El Algoritmo 4 muestra el proceso seguido por el algoritmo de balanceo MPR para elegir los líderes de las regiones. Los parametros utilizados por este algoritmo se describen a continuación:

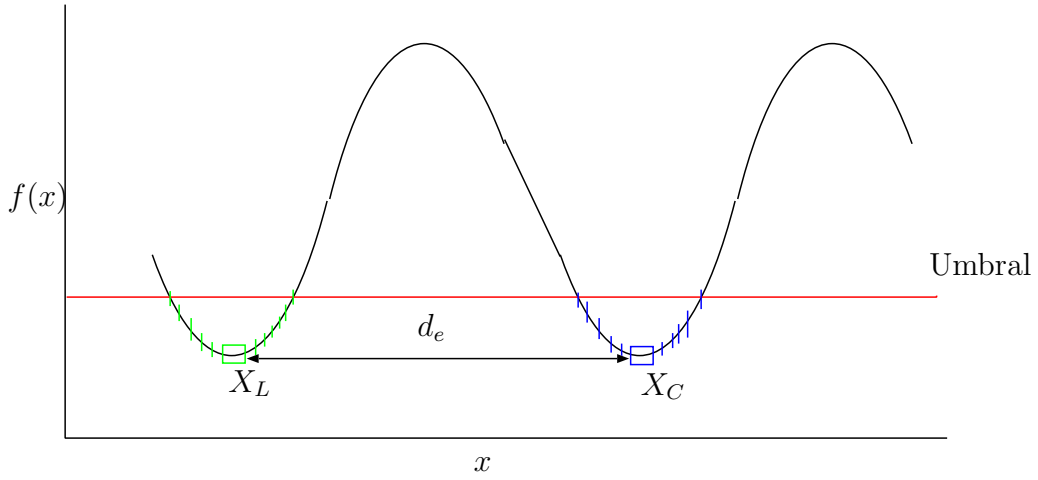


Figura 8.2: Determinación de líderes de una región y cercanía entre cajas

- L: Lista de cajas activas del procesador.
- Z: Vector con los líderes de las regiones.
- *umbral*: Valor del umbral de candidatos calculado según la Ecuación 8.3.
- *lideres*: Indica el número de líderes a encontrar. El valor utilizado para este proyecto será de dos.

Inicialmente la lista de líderes se encuentra vacía (línea 1). A continuación se entra en el bucle principal del algoritmo en el que se recorre la lista de cajas L hasta encontrar al menos 2 líderes (líneas 2 y 3). En la línea 4 se comprueba si el límite inferior de la caja extraida de L es mayor que el valor del *umbral* por lo que ya no sería una caja candidata y saldríamos del bucle *while*. En caso contrario, comprobaríamos si la caja es cercana a alguno de los líderes presentes en la lista de líderes, Z. El algoritmo termina devolviendo la lista con las cajas líderes encontradas.

8.3.2. Asignación de cajas a regiones

Tras seleccionar las cajas que liderarán cada una de las regiones (al menos dos), se deben asignar el resto de cajas de la lista a una de las regiones representadas por su líder. La forma de hacerlo será, de nuevo, mediante cercanía. Se comprueba la distancia que hay entre la caja y cada uno de los líderes. La caja será asignada a la región más cercana, es decir, a la región cuyo líder se encuentre más cerca de la caja. El proceso acaba cuando todas las cajas tienen una región asignada.

Nuevamente, hemos optado por la solución más rápida y sencilla, si bien es cierto que hemos valorado otras opciones como la de generar regiones cuya carga de trabajo esté balanceada, teniendo en cuenta el trabajo de las cajas (ver Sección 8.2).

El Algoritmo 5 describe el proceso de asignación de cajas a regiones utilizado en el algoritmo de balanceo MPR. Los parámetros necesarios por este algoritmo son:

Algoritmo 4 Algoritmo de elección de líderes de regiones

Funct ElegirLideres($L, Z, umbral, lideres$)

1. $i := \emptyset$
2. $Z := \{\emptyset\}$
3. **while** ($NumeroElementos(Z) < lideres$)) *Hasta que haya 2 líderes en Z*
4. $X := RecorrerLista(L)$ *Recorre la lista de principio a final*
5. **if** ($\underline{F}(X) > umbral$) *No hay más candidatos*
6. break;
7. **elsif** (($\underline{F}(X) < umbral$)) **and** (!$esCercana(X, Z)$)
8. $Z^i = X$ *La caja X es un nuevo líder*
9. $i++$
10. $L := L - \{X\}$ *Sacamos X de la lista de trabajo*
11. **return** Z *Devolvemos la lista con los líderes encontrados*

- L: Lista de cajas activas del procesador.
- Z: Vector con los líderes de las regiones devuelto por el Algoritmo 4.
- R_{Z^i}: Región asignada al líder Z^i.

El algoritmo recorre la lista de cajas L completamente (línea 1), asignando cada una de las cajas X a R_{Z^k} donde la distancia euclídea entre la caja X y el líder Z^k es la menor de las distancias entre todos los líderes en Z. Finalmente se devuelve el valor de R que contendrá una matriz con las cajas asignadas a las regiones.

Algoritmo 5 Algoritmo de asignación de cajas a regiones

Funct AsignarCajasRegiones(L, Z, R)

1. **while** ($X := RecorrerLista(L)$)
2. $R_{Z^k} += \{X\}$ $k = \min_i(d_e(X, Z^i)), \forall i \in Z$
3. $L := L - \{X\}$ *Sacamos X de la lista de trabajo*
4. **return** R

8.3.3. Algoritmo de balanceo de la carga MPR

El algoritmo de balanceo de la carga con predicción de mínimos basado en regiones o MPR (Minimus Prediction by Regions) crea las regiones en un procesador cualquiera, quedándose con una de las regiones generadas y enviando el resto a procesadores libres. El Algoritmo MPR, descrito a lo largo de este capítulo se presenta de manera formal en el Algoritmo 6. Los parámetros utilizados son los siguientes:

- L: Lista de cajas activas del procesador.

- Q: Lista de cajas finales del procesador.
- Z: Lista de cajas con los líderes de las regiones.
- R: Regiones de cajas asignados a los líderes en Z.
- *lideres*: Número de líderes a encontrar.
- P: Lista de procesadores libres.

En la línea 1 se realiza el cálculo del umbral de candidatos según se describió en la Ecuación 8.3. A continuación, se comprueba que la primera caja en L cumple con la restricción del umbral (línea 2), que debe cumplirse siempre, y se inicia el proceso de elección de líderes descrito en el Algoritmo 4 (línea 4). Si no se encuentran más de dos líderes, el algoritmo MPR finaliza su ejecución al no poder crear las regiones, en caso contrario, se asignan el resto de cajas de L a alguno de los líderes en Z según el Algoritmo 5 (línea 7). Tras este punto, R contiene las cajas agrupadas en regiones, la primera de ellas permanece en el procesador (línea 8) mientras que el resto son enviadas a procesadores libres (líneas 9 y 10). Como en este proyecto el algoritmo MPR crea únicamente 2 regiones, sólo es necesario que haya un único procesador libre, aunque el algoritmo describe el caso general en el que podrían haberse creado 2 o más regiones. En ese caso, el número de procesadores libres debe ser igual o mayor al número de regiones generadas menos uno.

Algoritmo 6 Algoritmo de balanceo de la carga MPR

Funct MPR$(L, Q, Z, R, lideres, P)$

1. $umbral := calcularUmbral(L, Q)$ — *Ecuación (8.3)*
2. **if** $(umbral < \underline{F}(X_1))$ — $\{\underline{F}(X) = min(\underline{F}(X^i), \forall X^i \in L\}$
3. **return**
4. $Z := ElegirLideres(L, Z, umbral, lideres)$ — *Algoritmo 4*
5. **if** $(numeroElementos(Z) < 2)$
6. **return** — *Si no hay al menos dos regiones*
7. $R := AsignarCajasRegiones(L, Z, R)$ — *Algoritmo 5*
8. $L := R_{Z^1}$ — *La primera región se queda en el procesador*
9. **for** $i := 2$ **to** $lideres$
10. $MoverCajas(P^{i-1}, R_{Z^i})$
11. **return** L

8.4. Movimiento de las regiones generadas a nuevos procesadores

Tal y como se ha descrito en el Algoritmo 6, tras la creación de las regiones, se realiza el movimiento de éstas a procesadores libres. Es importante advertir que **siempre se**

moverán a otros procesadores, **todas las regiones excepto una**, ya que el procesador que ha generado las regiones no puede quedar vacío y sin trabajo.

Aunque en nuestro programa este movimiento es inmediato, en una arquitectura paralela real supondría enviar una gran cantidad de información a través de la red de la arquitectura, lo que supone un coste añadido para el tiempo de ejecución. Para mantener los tiempos de la simulación similares a como estarían en una arquitectura paralela, debemos sumar el tiempo de envío de cajas según la ecuación (6.2). A cada procesador se le sumará un tiempo de envío de acuerdo a esta fórmula, según el número de cajas recibidas. El procesador que envía, debe calcular el tiempo que tarda en enviar todas las cajas. Además, el procesador que crea las regiones debe tener en cuenta el tiempo que tarda en crear cada uno de los procesos nuevos (sección 6.4.2.1).

8.5. Coloreado de regiones

Aprovechamos que el proyecto se centra en el estudio de un balanceador dinámico basado en la predicción de mínimos mediante regiones para **funciones bidimensionales**, para poder ver de forma visual las regiones que crea nuestro algoritmo tras su generación.

Para visualizar las regiones utilizamos el programa `escalaw` que se ha desarrollado en el lenguaje tcl/tk específicamente para este propósito. Las cajas de una misma región se dibujan de un mismo color, así podemos ver las agrupaciones de cajas por colores. Estas serán las cajas que se envíen a nuevos procesadores, salvo una de las regiones que se queda en el procesador que realiza la predicción.

Existen dos formas de visualizar las regiones coloreadas:

1. Con la opción `-o -c`, se creará la salida para que el programa `escalaw` **coloree** las regiones después de cada predicción. Esto requiere que se recorra la lista entera de cajas, y se redibujen todas ellas con el color adecuado, por lo que este proceso es bastante lento, aunque permite ver las regiones de cajas tal y como han quedado al finalizar su generación.

2. Con la opción `-o`, se creará la salida para que el programa `escalaw` muestre el proceso de división de cajas. En este proceso, las cajas van cambiando de color según se encuentren en un procesador u otro a medida que avanza la ejecución. De esta forma, podemos ver qué cajas tiene cada procesador según su color cuando la caja ha sido evaluada. Este proceso es más rápido que el anterior , aunque no se ven las regiones de manera inmediata.

Las Figuras 8.3 y 8.4 muestran un ejemplo de la visualización de las cajas repartidas por los algoritmos de balanceo. La primera muestra cómo han quedado las cajas repartidas tras el primer balanceo de la carga utilizando el algoritmo de balanceo en Baraja (ver Capítulo 7) para la función *Goldstein-Price* con 4 procesadores disponibles y una precisión de $\epsilon = 0{,}01$. La segunda figura muestra cómo se encuentran las cajas en los procesadores tras utilizar el balanceador MPR para el mismo problema. La diferencia en cuanto a la localidad de las cajas en los procesadores al utilizar un algoritmo u otro se aprecia notablemente.

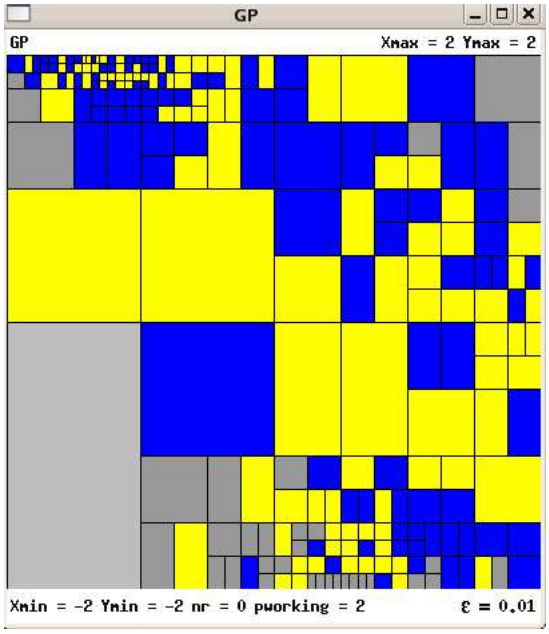


Figura 8.3: Cajas en los procesadores tras el primer balanceo en Baraja para el problema *Goldstein-Price* con 4 procesadores disponibles y $\epsilon = 0,01$.

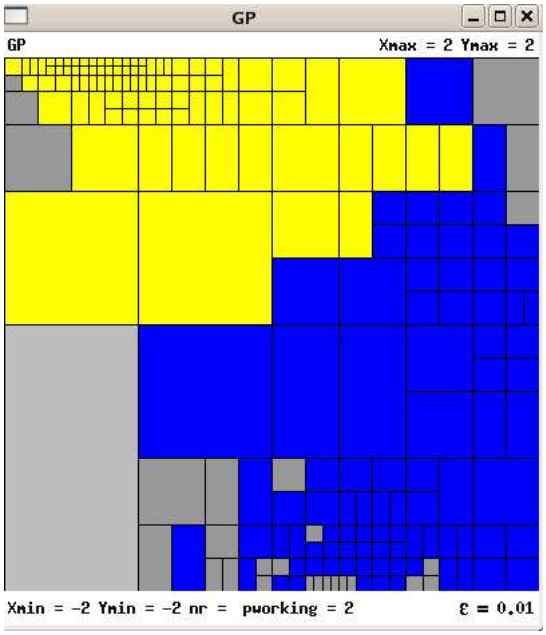


Figura 8.4: Cajas en los procesadores tras el primer balanceo con MPR sin decisión, para el problema *Goldstein-Price* con 4 procesadores disponibles y $\epsilon = 0,01$.

Esta visualización de las cajas nos ha permitido analizar y estudiar el proceso de creación de regiones hasta obtener el resultado final presentado en este proyecto, de ahí que fuera tan importante el uso de funciones test bidimensionales. El siguiente paso será comprobar el rendimiento de este algoritmo de balanceo (ver Capítulos 9, 10 y 11).

8.6. Decisión de la predicción de regiones

Una de las cuestiones que nos planteábamos en la Sección 8.3 (página 56) era la de cuándo debíamos generar las regiones en un procesador. En esta sección veremos todos los elementos que intervienen en la toma de decisión sobre cuándo crear las regiones.

8.6.1. Introducción

El momento en el que un procesador decide ejecutar el algoritmo MPR (ver Algoritmo 6), determina cómo serán las regiones generadas. Si la generación se hiciese antes o después de ese momento, obtendríamos regiones que nada tendrían que ver con las anteriores. Ésto hace que la decisión influya en el resultado final de la ejecución. Hemos de tener en cuenta además, que ejecutar demasiadas veces el algoritmo MPR empeorará la eficiencia del algoritmo, debido al coste computacional que supone analizar las cajas para generar las regiones. Aún más, no siempre que ejecutemos el algoritmo MPR se encontrarán regiones, por lo que hay que minimizar el número de ocasiones en las que ésto sucede. En particular, el problema puede venir cuando hay bastantes cajas candidatas pero no se encuentran al menos 2 líderes (la primera caja siempre va a ser líder). Otro problema que puede surgir al no encontrar regiones es que estemos perdiendo tiempo de cómputo paralelo si hay procesadores libres esperando trabajo. El último problema que se nos plantea es que las regiones que generemos no sean demasiado prometedoras, es decir, no tienen demasiado trabajo computacional, por lo que enviarlas a otro procesador es más costoso que evaluarlas en el mismo procesador.

8.6.2. Elementos que intervienen en la decisión

Para tomar la decisión de crear las regiones o no, debemos tener en cuenta una serie de elementos que influyen en la existencia o no de regiones en el procesador. Estos elementos se analizan en profundidad en los siguientes epígrafes, y entre ellos se encuentran por ejemplo, la cantidad de trabajo en potencia del procesador o la disponibilidad de procesadores libres.

8.6.2.1. Disponibilidad de procesadores libres

Es obvio que, aún cuando el resto de elementos para realizar la predicción sean favorables, si no hay procesadores disponibles no podremos enviar las regiones generadas, por lo que ejecutar el algoritmo sería totalmente inútil. Por tanto, es condición necesaria pero no suficiente que existan procesadores libres cuando se realice la predicción de regiones.

8.6.2.2. Cantidad total de trabajo frente a ejecución en otros procesadores

Una de las cuestiones primordiales para decidir si realizar una predicción de regiones es calcular el trabajo que hay en el procesador, y comprobar si es rentable enviarlo por la red para que lo evalúen otros procesadores. Esto es necesario, ya que hay veces que empleamos tiempo extra en enviar cajas por la red a otros procesadores cuando sería más rápido si las evaluáramos nosotros, bien porque se descartan en seguida, bien porque no se dividen demasiadas veces.

Trabajo en potencia de un procesador

El trabajo en potencia de un procesador es el número potencial de cajas a evaluar a partir del número de cajas en la lista de trabajo del procesador (Secciones 8.2, 8.2.2 y 8.2.3), notado como $wp(L)$ y calculado según la Ecuación 8.2. Esta cifra es una estimación del número de cajas que se esperan evaluar en el procesador, por lo que podemos analizar si es conveniente mantenerlas en nuestro procesador o mandarlas a otro. Para hacer esto, podemos calcular el tiempo que tardaremos en terminar todo el trabajo pendiente en el procesador:

Definición 8 *Se define el tiempo de trabajo en potencia de un procesador, y se notará por t_p, al tiempo de evaluación pendiente en el procesador.*

Sea $wp(L)$ el trabajo en potencia de la lista de cajas activas en el procesador, y t_{pc} el tiempo de proceso de una caja (Ecuación (6.1)). El cálculo de t_p se realiza como:

$$t_p = t_{pc} \cdot wp(L) \tag{8.7}$$

Como puede observarse, t_p es una estimación del tiempo que nos queda aún para terminar la ejecución.

Otra de las ideas para realizar en un trabajo posterior, es la de crear un factor adaptativo que varíe en tiempo de ejecución y aproxime mejor la estimación del trabajo en potencia del procesador.

Envío y evaluación en otro procesador

Frente a mantener las cajas en el mismo procesador, cabe tomar la decisión de enviar cajas para que otro procesador las evalúe. Al igual que calculamos el tiempo de trabajo en potencia de nuestro procesador, podemos calcular el tiempo que se emplearía si enviáramos parte de ese trabajo para que otro procesador lo evaluara.

Para realizar este cálculo, haremos la suposición de que en el proceso de creación de regiones, al menos la mitad del trabajo en potencia será repartido. Esto es así ya que se repartirán cajas siempre y cuando se encuentren dos, esperando que éstas tengan una carga de trabajo similar.

Definición 9 *Se define el tiempo de envío, y se notará como t_{pe}, al tiempo que tardaría en enviarse la mitad de las cajas a otro procesador.*

Sea l la latencia de la red y ttr la tasa de transferencia, $size(X)$ el tamaño de una caja cualquiera en *bytes*, t_{cp} el tiempo que se tarda en crear un nuevo proceso (Sección 6.4.2.1), t_{ec} el tiempo de envío de una caja a otro procesador (Ecuación 6.2), y n el número de cajas que hay en el procesador. El cálculo de t_{pe} se realiza así:

$$t_{pe} = t_{cp} + l + \frac{\frac{n}{2} \cdot size(X)}{ttr} \tag{8.8}$$

Ya tenemos los datos suficientes para decidir, en función de la cantidad de trabajo en potencia del procesador, si es preferible evaluar todo este trabajo en el mismo procesador o enviarlo a otro para que aquel lo evalúe.

Definición 10 *La decisión de generar regiones y ejecutar el algoritmo de balanceo de carga será positiva cuando el tiempo de procesado de la mitad del trabajo en potencia del procesador sea mayor que enviar la mitad de las cajas a otro procesador. La decisión se realizará si y solo si hay al menos un procesador libre.*

$$\frac{t_p}{2} > t_{pe} \tag{8.9}$$

8.6.2.3. Presencia de regiones

Dado que la decisión de la predicción debe tener en cuenta la presencia o no de regiones, lo mejor sería establecer un parámetro que nos indique con fiabilidad si hay o no regiones en el procesador. Esta opción no se ha implementado, pero creemos que un elemento que mejoraría la decisión de realizar la predicción sería controlar en tiempo de ejecución el número de cajas candidatas presentes en la lista de trabajo.

8.6.3. Algoritmo de decisión

El Algoritmo 7 presenta de manera formal la toma de decisión acerca de si realizar o no la predicción de regiones y el balanceo de la carga, es decir, si ejecutar el Algoritmo MPR. Los parámetros que utiliza el algoritmo para tomar la decisión son los siguientes:

- L: Lista de cajas activas del procesador.
- t_{pc}: Tiempo de proceso de una caja.
- t_{cp}: Tiempo de creación de un proceso.
- t_p: Tiempo de trabajo en potencia del procesador.
- t_{pe}: Tiempo de envío de la mitad de las cajas a otro procesador.
- t_{ec}: Tiempo de envío de una caja.
- n: Número de elementos de L.

La decisión se toma tras calcular el tiempo de trabajo en potencia del procesador, t_p según la Ecuación 8.7 (línea 2), y el tiempo de envío de la mitad de las cajas según la Ecuación 8.8 (línea 3). El algoritmo decide que es necesaria la predicción si se cumple la Ecuación 8.9 (línea 4).

Algoritmo 7 Algoritmo de decisión para la predicción de regiones

Funct DecidePredecir$(L, t_{pc}, t_{cp}, t_p, t_{pe}, n, t_{ec})$

1. $n := numeroElementos(L)$
2. $t_p := calcularTiempoTrabajo(t_{pc}, L)$ *Ecuación (8.7)*
3. $t_{pe} := calcularTEnvio(t_{cp}, t_p, n, t_{ec})$ *Ecuación (8.8)*
4. **if** $(\frac{t_p}{2} > t_{pe})$ *Ecuación (8.9)*
5. **return** PREDECIR
6. **else**
7. **return** NO_PREDECIR

Parte V

Resultados

Capítulo 9

Resultados de las simulaciones

9.1. Características del equipo de trabajo

Los resultados de la simulación dependen, en buena parte, del equipo en el que se realiza la ejecución de las simulaciones. Las características del hardware del equipo utilizado son las siguientes:

- Procesador: Intel Pentium 4 Core Duo 2, 8 GHz
- Memoria RAM: 512 MB DDR
- Tarjeta Gráfica: NVIDIA GeForce FX 5200

Por otra parte, el software utilizado ha sido el siguiente:

- Sistema Operativo: Fedora Core 4, Linux Kernel 2.6
- Versión del compilador: g++ 4.0.2
- Librería de verificación númerica: C-XSC 2.2

9.2. Evaluación del rendimiento

Para la evaluación del rendimiento de nuestros algoritmos de balanceo de carga, hemos utilizado un pequeño subconjunto de funciones de prueba utilizadas normalmente en problemas de Optimización Global. Utilizaremos las siguientes abreviaturas para los nombres de las funciones test: GP para *Goldstein-Price*, L3 para *Levy 3* y SH para *Six Hump Camel Back*. También hemos asignado un número a cada una de estas funciones para facilitar la visualización de los gráficos. En la Tabla 9.1 podemos ver el número de cada una de las funciones, así como la precisión utilizada.

Además, hemos planteado dos modelos de prueba: en el primero de ellos, los algoritmos de balanceo reparten cajas cuando detectan que un procesador ha quedado libre; y en el segundo modelo, se evalúa si es conveniente balancear la carga, en función del trabajo en potencia que hay en el procesador (ver Sección 8.6), es decir, se utiliza el algoritmo de

Tabla 9.1: Equivalencia de funciones test y precisión utilizada.

No.	Función	Precisión (ϵ)
0	Goldstein-Price	10^{-4}
1	Levy 3	10^{-4}
2	Six Hump Camel Back	10^{-4}

decisión (ver Algoritmo 7) antes de realizar el balanceo. Aunque el algoritmo de decisión fue diseñado para ser utilizado con el algoritmo de balanceo MPR, hemos considerado oportuno utilizarlo junto al algoritmo de balanceo Baraja para comparar su eficiencia en los dos algoritmos de balanceo, y su incidencia en los resultados de la simulación. Por tanto, en este capítulo analizaremos los resultados de las simulaciones de las siguientes pruebas:

- Modelo de balanceo de la carga en Baraja sin toma de decisión para el reparto, denotado por BAR.
- Modelo de balanceo de la carga en Baraja con toma de decisión para el reparto, denotado por BARD.
- Modelo de balanceo de la carga con Predicción de Mínimos mediante Regiones sin toma de decisión para el reparto, denotado por MPR.
- Modelo de balanceo de la carga con Predicción de Mínimos mediante Regiones con toma de decisión para el reparto, denotado por MPRD.

Todos los modelos se han evaluado utilizando 1, 2, 4, 8 y 16 procesadores virtuales para analizar cómo cambia el rendimiento según el número de procesadores del sistema. La ejecución con un procesador es igual para todos los modelos de prueba, ya que no se produce ningún reparto de cajas.

Para realizar el cálculo de los resultados, se han realizado **cinco ejecuciones de cada problema**, para comprobar que no se producen diferencias notables ante situaciones de ejecución similares. Estas diferencias podrían darse debido a que las decisiones acerca de en qué procesador repartir las cajas, pueden variar en cada ejecución. Los resultados finales se obtienen de eliminar la ejecución con mayor y menor tiempo de ejecución y haciendo la media de los resultados en el resto de ejecuciones. Estos resultados globales de las ejecuciones se muestran en las Tablas 9.3 y 9.4 y se analizarán de forma pormenorizada en la siguiente sección.

9.3. Análisis de resultados

Uno de los parámetros tenidos en cuenta para la evaluación del rendimiento de nuestros algoritmos paralelos ha sido el *Speed-Up*. En la Figura 9.1 se muestran los valores de *Speed-Up* de los modelos de prueba para cada una de las funciones test. En la figura podemos ver

cómo crece el *Speed-Up* en los modelos BAR y BARD a medida que aumenta el número de procesadores. Todos los modelos muestran una eficiencia muy parecida, si bien, BARD

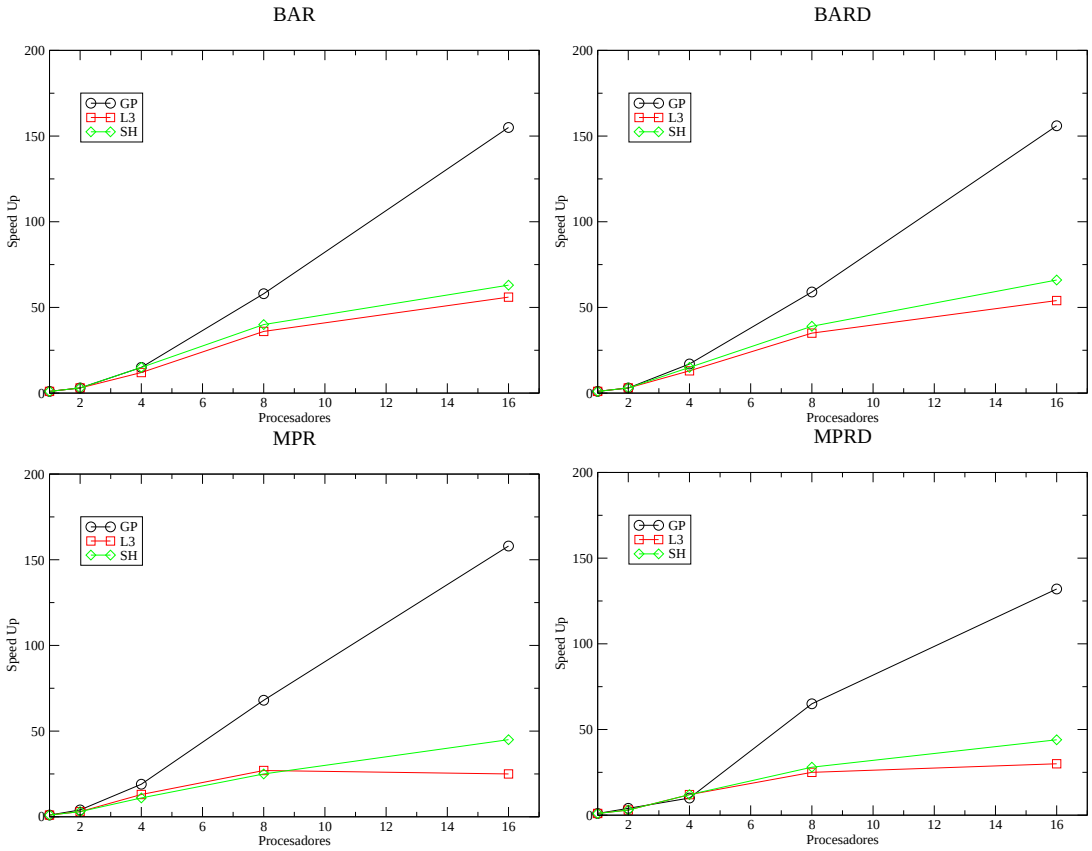


Figura 9.1: Valores del *Speed-Up* de los algoritmos de balanceo de la carga.

destaca ligeramente, siendo superior en la función GP, destacando también en las funciones L3 y SH. Esta mejora en el rendimiento suele coincidir con la disminución del número de repartos de cajas realizados, es decir, el modelo que menos repartos hace se muestra más eficiente. Esto es así ya que la decisión antes del reparto comprueba que hay suficiente trabajo para realizar un balanceo de la carga. Hay algunas excepciones, por ejemplo, en la función SH con 4 procesadores y un modelo de balanceo BAR realiza 25 repartos, mientras que el modelo de balanceo BARD realiza 21 repartos, siendo el *Speed-Up* de 15,51 y 15,19 respectivamente. Debido a que la decisión se realiza teniendo en cuenta el trabajo en potencia del procesador (ver Sección 8.2.3), que es una estimación de la cantidad de carga de trabajo que hay en el procesador, podemos perder algo de tiempo si el algoritmo decide no repartir cajas a otros procesadores y realizamos más trabajo del que habíamos previsto. En cuanto al algoritmo de balanceo MPR y MPRD, podemos ver en la Figura 9.1, en las gráficas inferior izquierda y derecha cómo evoluciona el *Speed-Up* en las ejecuciones sin y con toma de decisión, respectivamente. El dato principal a destacar de estas dos gráficas es que el algoritmo de balanceo MPR para la función GP, supera al algoritmo de reparto BAR y BARD. Además, tanto el algoritmo de balanceo MPR como MPRD, se muestran bastante competitivos en ejecuciones hasta con 4 procesadores, disminuyendo

su eficiencia considerablemente para 8 y 16 procesadores. La explicación la encontramos en que la función GP es la que mayor carga computacional requiere de las tres, por lo que el tiempo empleado en la creación de las regiones tiene menos impacto en la ejecución (de ahí también que hasta con 4 procesadores se mantenga un rendimiento competitivo). Por otro lado, este mejor comportamiento de los algoritmos de balanceo MPR se obtiene gracias a que el reparto de cajas con una localidad similar ayuda a mantener menos cajas en la lista de cajas activa (ver el dato *Nº Cajas Max* en las Tablas 9.3 y 9.4).

En la Tabla 9.2 presentamos la relación entre el promedio de número de cajas máximo que hubo en la lista de cajas de los procesadores durante la ejecución de los algoritmos Baraja y MPR. En aquellos modelos en los que este valor es > 1 indica que el promedio del número de cajas máximo es superior en el algoritmo en Baraja. Podemos observar cómo el algoritmo MPR obtiene una mejor relación que el BAR: en 9 ejecuciones de 11, es decir, en el 75 % de las pruebas. En cambio, cuando realizamos la decisión el porcentaje cae al $41,67\,\%$ por lo que el balanceo en BARD mejora su comportamiento en este caso.

Tabla 9.2: Relación del promedio del número máximo de cajas en las listas de trabajo de los procesadores en los algoritmos de balanceo Baraja y MPR.

Modelo	Sin decisión			Con decisión		
Procesadores	GP	L3	SH	GP	L3	SH
2	1,10	1,20	1	0,99	1,13	1,01
4	1,24	1,13	1,11	0,88	0,90	1,07
8	1,12	1,02	0,94	1,17	1	1,03
16	1,30	0,90	1,01	0,92	0,88	0,92
Media ($\sum$/4)	1,19	1,06	1,02	0,99	0,98	1,01
Media del grupo		1,09			0,99	

En la Figura 9.2 se puede ver cómo evoluciona el número de cajas en cada uno de los procesadores para el problema GP con 8 procesadores disponibles, la línea de tiempo (horizontal) se muestra en milisegundos. La imagen superior muestra la evolución cuando se utiliza el algoritmo de balanceo BAR. Vemos cómo entorno a los 12 segundos los 8 procesadores empiezan a aumentar el número de cajas en su lista de trabajo hasta superar las 8000 cajas en cada una de las listas. Tantos elementos en la lista provoca que su manipulación sea mucho más lenta. En la imagen inferior podemos ver la evolución del número de cajas al utilizar el algoritmo de balanceo MPR. Se puede apreciar cómo el número máximo de cajas alcanzado por la mayoría de los procesadores es bastante inferior, además también vemos cómo las cajas se eliminan mucho más rápidamente. En la parte final de la ejecución, el balanceo BAR utiliza los 8 procesadores mientras que el balanceo MPR disminuye progresivamente el número de procesadores, según el número de regiones presentes.

A la vista de estos datos podemos decir que repartir las cajas con el algoritmo de balanceo MPR facilita que el número de cajas en las listas de trabajo disminuya, sobre

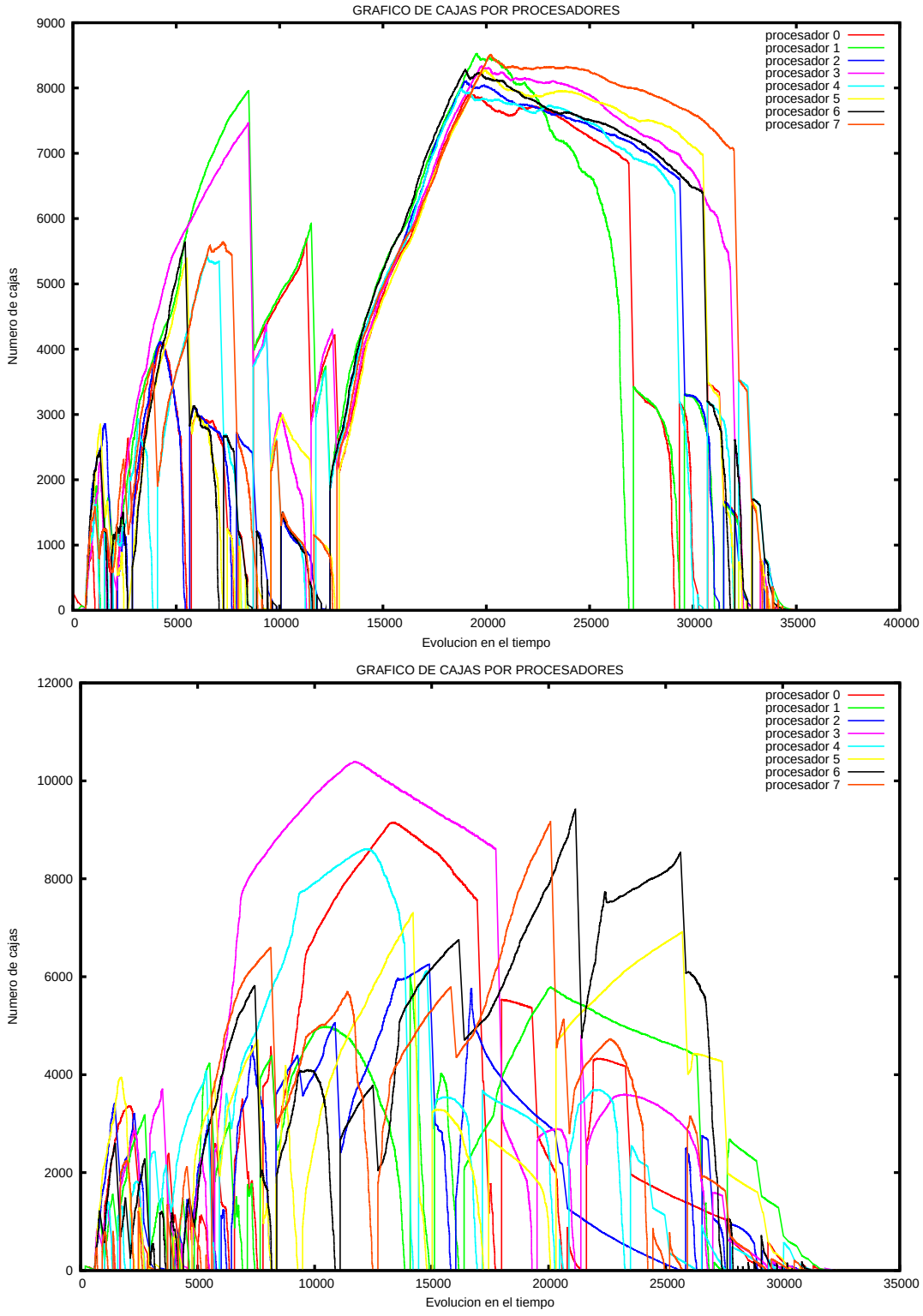


Figura 9.2: Evolución de cajas en los procesadores para el problema GP con 8 procesadores y $\epsilon = 10^{-4}$ sin decisión. Arriba balanceo BAR. Abajo balanceo MPR.

todo cuando no se realiza la decisión, por lo que podemos obtener alguna mejora en el tiempo de manipulación de la lista (inserción, recorrido, eliminación, etc). En este caso, la decisión perjudica la eficiencia del algoritmo de balanceo MPR, sobre todo cuando el número de procesadores del sistema es superior a 4. Esto puede deberse a una mala asignación inicial de regiones que posteriormente da lugar a no encontrar región alguna, aún cuando el algoritmo de decisión establece que hay suficiente trabajo para realizar la predicción. Hay que destacar que el algoritmo de balanceo MPRD, se muestra más eficiente que su homólogo sin decisión en la función SH con 2, 4 y 8 procesadores, y L3 con 16 procesadores. En todos estos casos, la decisión disminuye el número de predicciones hechas. Por tanto, el handicap principal al que nos enfrentamos en el algoritmo de balanceo MPR es el tiempo de creación de las regiones.

El segundo parámetro que hemos analizado en la ejecución de los algoritmos de balanceo de la carga ha sido el Desbalanceo (ver Fórmula 5.3, página 24). La Figura 9.3 muestra los valores de Desbalanceo para cada uno de los modelos propuestos. Se puede apreciar cómo los valores de Desbalanceo máximo se obtienen en las ejecuciones con 16 procesadores en general, y en particular para los algoritmos de balanceo MPR y MPRD y la función L3 en donde este valor supera el 0,5. Con 8 procesadores, los valores se muestran estables en todos los modelos, entorno al 0,1-0,2.

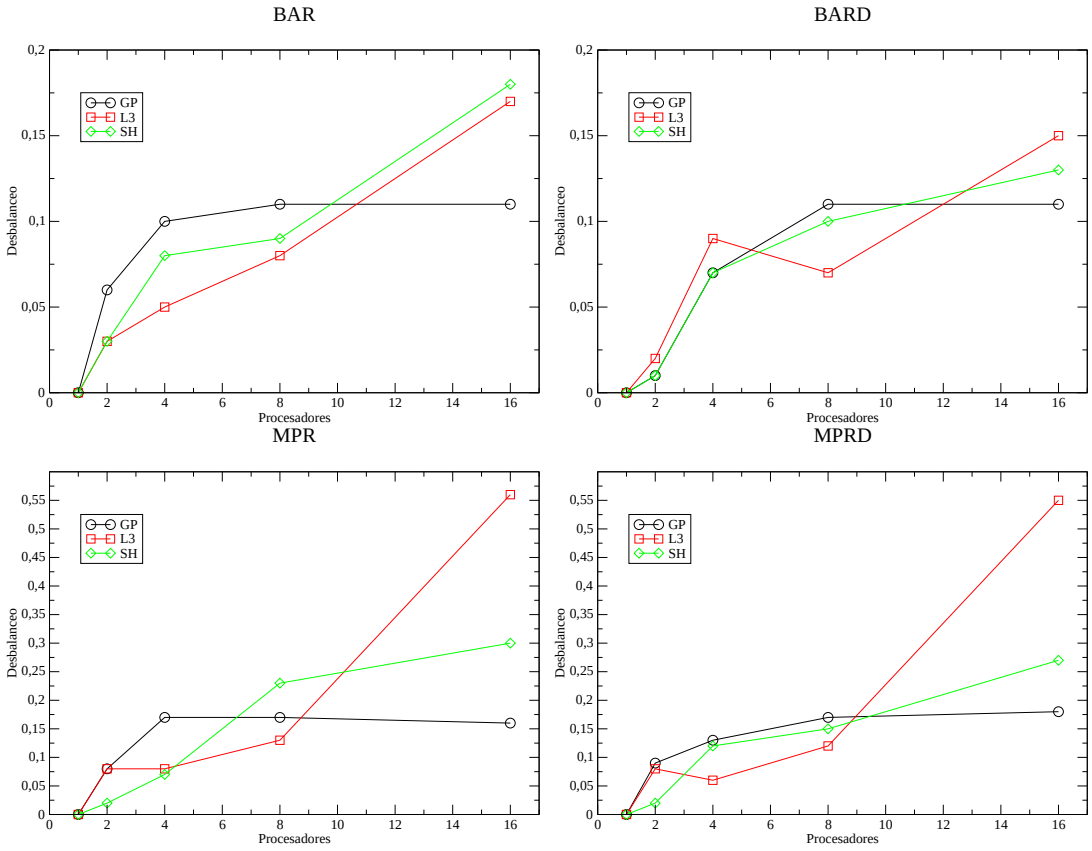


Figura 9.3: Valores de Desbalanceo de los algoritmos paralelos

Además, para los balanceadores BAR y BARD, el valor de Desbalanceo máximo

está por debajo de 0,2. En las diferentes gráficas podemos apreciar también que el desbalanceo aumenta a medida que incrementamos el número de procesadores del sistema. Podemos explicar el incremento notable del Desbalanceo en los algoritmo MPR y MPRD, sobre todo con 16 procesadores, ya que en la última fase de la ejecución, sólo trabaja un número reducido de procesadores, aquellos que el algoritmo estima necesario según el número de regiones encontradas.

Por último, las Figuras 9.4 y 9.5 muestra los tiempos de ejecución y de proceso para cada uno de los modelos propuestos. Entendemos por tiempo de ejecución el tiempo transcurrido desde que comienza la ejecución hasta que ésta finaliza, de acuerdo con las condiciones del sistema paralelo simulado. El tiempo de proceso consiste en la suma de los tiempos en los que los procesadores han estado trabajando, aunque hemos de tener en cuenta que en este tiempo también se incluyen los retrasos por los envíos a través de la red.

Si analizamos los tiempos de ejecución en la Figura 9.4, vemos claramente cómo las ejecuciones con 16 procesadores son más eficientes en todas las funciones de prueba y modelos de ejecución estudiados, si bien esta mejora es menor para la función test L3. No vamos a comentar nada más de estas gráficas ya que toda la información referente a los tiempos de ejecución la hemos analizado en la Figura del *Speed-Up*.

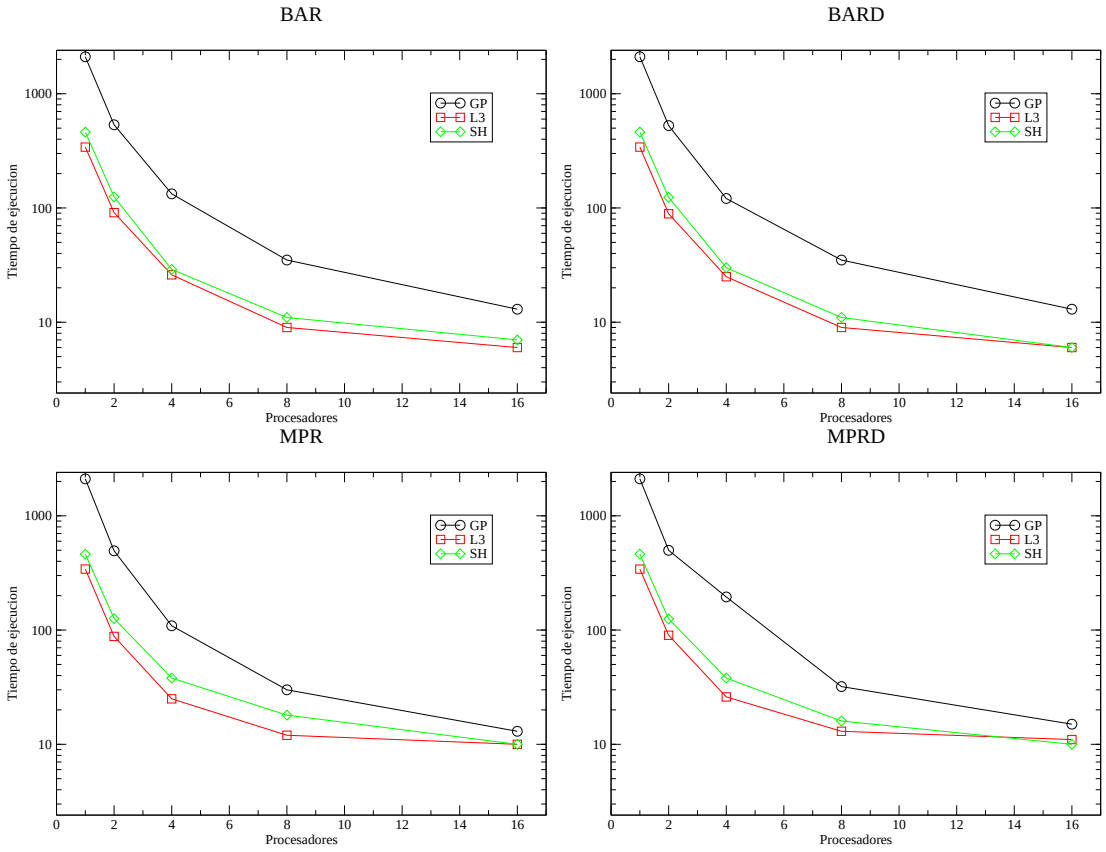


Figura 9.4: Tiempos de ejecución de las funciones test para los algoritmos de balanceo de la carga.

Los tiempos de proceso de los distintos modelos podemos verlos en la Figura 9.5. Aquí podemos ver en todos los modelos, cómo el tiempo de proceso con 16 procesadores en el sistema supera o se aproxima enormemente a los tiempos de proceso de la ejecución con 8 procesadores en las funciones L3 y SH. Esto puede deberse al mayor número de creación de procesos y envíos de cajas que se producen cuando hay más procesadores en el sistema, aunque en los algoritmos de balanceo MPR y MPRD también puede deberse al tiempo empleado en la creación de las regiones. Así, aunque se mejoran los tiempos de ejecución, el tiempo de proceso se ve penalizado debido al aumento del tráfico de información. También podemos decir que comparativamente, los modelos que no incluyen la decisión antes del balanceo tienen ligeramente menores tiempos de proceso que los que sí incluyen esta decisión.

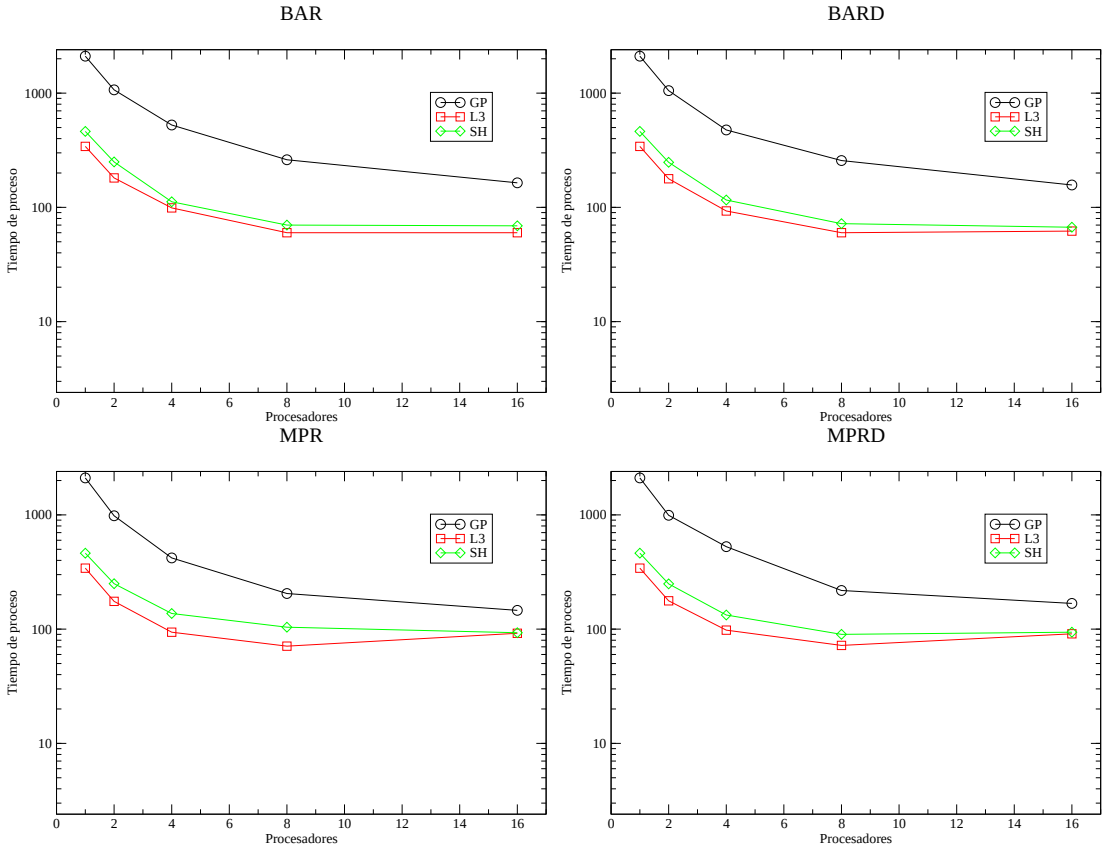


Figura 9.5: Tiempo de proceso de las funciones test para los algoritmos de balanceo de la carga.

Tabla 9.3: Resultados generales del algoritmo de balanceo en Baraja.

Modelo	Sin decisión			Con decisión		
1 procesador	GP	L3	SH	GP	L3	SH
Esfuerzo	1280878	506704	836332	1280878	506704	836332
T. Ejecución	2106,16	341,83	462,21	2106,16	341,83	462,21
2 procesadores	GP	L3	SH	GP	L3	SH
Esfuerzo	1281006	506704	836332	1280878	506704	836332
Desbalanceo	0,06	0,03	0,03	0,01	0,02	0,01
Speed-up	3,93	3,72	3,68	3,99	3,81	3,72
T. Ejecución	535,35	91,69	125,44	526,97	89,8	124,37
T. Proceso	1068,97	181,86	250,05	1052,09	178,33	248,31
Repartos	7,33	7	4	7,33	5,67	2
Nº Cajas Max	52518,67	23382,67	33812	47481	22090	33812
4 procesadores	GP	L3	SH	GP	L3	SH
Esfuerzo	1280883	506704	836338	1280879	506704	836352
Desbalanceo	0,10	0,05	0,08	0,07	0,09	0,07
Speed-up	15,75	12,87	15,51	17,39	13,59	15,19
T. Ejecución	133,74	26,56	29,8	121,11	25,15	30,43
T. Proceso	526,05	99,31	112,37	476,46	93,52	116,27
Repartos	35	26,67	25	32,33	27,33	21
Nº Cajas Max	21398,83	10577,48	16649,47	18424,87	9369,67	15610,73
8 procesadores	GP	L3	SH	GP	L3	SH
Esfuerzo	1281010	506704	836364	1280950	506704	836440
Desbalanceo	0,11	0,08	0,09	0,11	0,07	0,1
Speed-up	58,57	36,40	40,76	59,70	35,91	39,95
T. Ejecución	35,96	9,39	11,34	35,28	9,52	11,57
T. Proceso	261,66	60,4	70,99	257,95	60,79	72,91
Repartos	104,33	49	76	93,67	53,33	77,67
Nº Cajas Max	9895,69	5588,38	7717,42	10114,73	5446,71	7712,75
16 procesadores	GP	L3	SH	GP	L3	SH
Esfuerzo	1280956	506704	836772	1280953	506704	836513
Desbalanceo	0,11	0,17	0,18	0,11	0,15	0,13
Speed-up	155,67	56,31	63,06	156,71	54,09	66,31
T. Ejecución	13,53	6,07	7,33	13,44	6,32	6,97
T. Proceso	164,68	60,64	69,32	157,59	62,27	67,31
Repartos	193,67	119	174,67	224,67	128,67	162,67
Nº Cajas Max	5872,08	3092,39	4098,5	4759,94	3142,58	3808,06

Tabla 9.4: Resultados generales del algoritmo de balanceo MPR.

Modelo	Sin decisión			Con decisión		
1 procesador	GP	L3	SH	GP	L3	SH
Esfuerzo	1280878	506704	836332	1280878	506704	836332
T. Ejecución	2106.16	341.83	462.21	2106.16	341.83	462.21
2 procesador	GP	L3	SH	GP	L3	SH
Esfuerzo	1281725	506704	836332	1281599	506704	836332
Desbalanceo	0,08	0,08	0,02	0,09	0,08	0,02
Speed-up	4,26	3,84	3,65	4,22	3,80	3,67
T. Ejecución	494,13	88,91	126,62	499,03	90,06	125,91
T. Proceso	983,67	175,43	250,21	992,94	177,63	249,59
Predicciones	1742	46,33	342,67	489	31,33	397,33
Válidas	15	10,67	8,33	16	11,33	4,33
Nº Cajas Max	47835,17	19493,83	33812	48073,83	19545	33602
4 procesador	GP	L3	SH	GP	L3	SH
Esfuerzo	1281540	506704	836410	1281146	506704	836452
Desbalanceo	0,17	0,08	0,07	0,13	0,06	0,12
Speed-up	19,31	13,36	11,89	10,75	12,71	12,07
T. Ejecución	109,08	25,59	38,88	195,9	26,9	38,28
T. Proceso	420,37	94,4	137	527,14	98,29	133,49
Predicciones	3233,67	805,67	367,67	18977,67	204,67	414,67
Válidas	64,67	28,33	80	60	38,33	76,67
Nº Cajas Max	17218,83	9354,33	15023,87	20851,2	10383,8	14607,33
8 procesador	GP	L3	SH	GP	L3	SH
Esfuerzo	1280974	506710	836358	1280955	506716	836346
Desbalanceo	0,17	0,13	0,23	0,17	0,12	0,15
Speed-up	68,87	27,77	25,19	65,51	25,08	28,51
T. Ejecución	30,58	12,31	18,35	32,15	13,63	16,21
T. Proceso	205,42	71,76	104,73	218,48	72,57	90,8
Predicciones	5617,67	1185	768,67	1742	2318	668,33
Válidas	141	103	174,33	167,67	95,67	158,33
Nº Cajas Max	8851	5465,88	8184,33	8675,29	5446,45	7477,88
16 procesador	GP	L3	SH	GP	L3	SH
Esfuerzo	1281305	506705	836493	1281242	506705	837927
Desbalanceo	0,16	0,56	0,3	0,18	0,55	0,27
Speed-up	158,48	25,59	45,14	132,63	30,49	44,74
T. Ejecución	13,29	10,05	10,24	15,88	11,21	10,33
T. Proceso	146,04	92,28	93,55	168,62	91,67	94,3
Predicciones	4372,33	7757,33	1074,33	4992	8634	833,67
Válidas	262,67	204,33	270	326,67	199,33	277
Nº Cajas Max	4524,62	3435,25	4068,96	5154,02	3588,54	4150,54

Capítulo 10

Nuevo algoritmo de balanceo de la carga basado en regiones

10.1. Descripción y motivaciones

En el capítulo anterior presentamos los resultados ofrecidos por dos algoritmos de balanceo de la carga para la resolución de problemas de Optimización Global. El nuevo algoritmo de balanceo de carga llamado MPR y presentado en este proyecto, que basa el reparto de cajas en la creación de regiones tratando de predecir la localización de los mínimos, ofrece resultados aceptables cuando hay hasta 4 procesadores en el sistema, y el mejor rendimiento para la función GP, que es la que más carga computacional realiza. Además, la utilización de la decisión antes de realizar el balanceo no mejora la eficiencia del algoritmo. Por otro lado, el balanceo en Baraja se muestra más estable en los test realizados y mantiene un buen nivel de desbalanceo incluso con 16 procesadores. Parte de culpa del bajo rendimiento del balanceador MPR está en el elevado número de predicciones inútiles que realiza, lo que significa que no se generan regiones y por tanto, no se reparten cajas. Esto es malo cuando hay trabajo suficiente para repartir cajas. Así, con estos resultados que se quedan a mitad de camino entre un algoritmo de balanceo eficiente y un algoritmo de balanceo pésimo, decidimos probar un nuevo algoritmo de balanceo basado también en regiones, pero que no utiliza la predicción de mínimos para generar estas regiones.

El nuevo algoritmo de balanceo de la carga basado en la predicción exclusiva de regiones o ERP (*Exclusive Regions Prediction*) pretende mejorar las carencias del balanceador MPR en cuanto al número de predicciones inútiles, tratando de mejorar así su eficiencia. Para hacer esto, sólo hemos tenido que modificar el método de elección de líderes del algoritmo MPR explicado en la Sección 8.3.1.3 y descrito por el Algoritmo 4 (ver página 62). La nueva elección de líderes consistirá en elegir a las dos cajas más prometedores, es decir, las dos cajas en la lista con menor $\underline{F}(X)$, esto implica que los líderes de las regiones sean las dos primeras cajas de la lista de trabajo del procesador. El nuevo método de elección de líderes del balanceador ERP simplifica enormemente el proceso de elección de las cajas líderes de una región, y garantiza que siempre se generen regiones en el procesador, lo que soluciona el problema de las predicciones inútiles.

Para concluir con la descripción de este nuevo algoritmo de balanceo de carga, vamos a mostrar la diferencia de funcionamiento con respecto al balanceador MPR de forma visual. La Figura 10.1 muestra cómo se han repartido las regiones de cajas para la función GP con 4 procesadores en el sistema y una precisión de $\epsilon = 0,01$. Arriba vemos la disposición de las cajas tras un reparto utilizando el balanceador ERP. La imagen de la izquierda muestra el primer reparto realizado al inicio, y la imagen de la derecha muestra la localización de las cajas cuando todos los procesadores disponen de trabajo. En el primer reparto de cajas no hay diferencia con respecto al balanceador MPR (imagen inferior izquierda) pero la diferencia se aprecia en las imágenes de la derecha una vez que todos los procesadores han recibido cajas.

10.2. Evaluación del rendimiento

Nos interesa por tanto, comparar el rendimiento del nuevo balanceador ERP con respecto al anterior MPR y comprobar si se mejoran los resultados presentados en el Capítulo 9. Para ello, se han realizado el mismo conjunto de pruebas que en el capítulo anterior.

10.3. Análisis de resultados

Los resultados completos del balanceador ERP pueden verse en la Tabla 10.1. La Figura 10.2 compara los valores de *Speed-Up* obtenidos por el algoritmo ERP (arriba) con los obtenidos por el algoritmo MPR (abajo). El algoritmo MPR se mantiene como el más eficiente en la función GP cuando no se utiliza decisión, mientras que en las funciones L3 y SH la balanza se inclina del lado del algoritmo ERPD, debido sobre todo a la mejoría notable del rendimiento cuando se realiza decisión antes del balanceo. Además, el algoritmo ERPD mejora el rendimiento del MPRD con 16 procesadores.

En la Figura 10.3 se presenta la diferencia en la evolución de las cajas en los procesadores existente entre los balanceadores ERP (imagen superior) y MPR (imagen inferior) para la función test GP sin decisión. Esta figura muestra a lo largo de la línea de tiempo (en milisegundos) el número total de cajas presentes en cada uno de los procesadores. Como se puede apreciar, el balanceador ERP realiza menos repartos en la parte central de la ejecución. Por contra, el balanceador MPR llega al final de la ejecución con un menor número de procesadores en ejecución debido a que en esta etapa de la ejecución se generan menos regiones. Debido a que el balanceador ERP siempre genera regiones, el rendimiento de este se ve afectado sobre todo en la parte final de la ejecución, por ello, el uso de la decisión antes del reparto mejora en algunos casos significativamente el rendimiento de este algoritmo.

La Figura 10.4 muestra los valores de desbalanceo en los modelos de prueba para los algoritmos ERP y MPR. En el capítulo anterior vimos cómo los valores de desbalanceo del algoritmo MPR se incrementaban de forma notable cuando había 16 procesadores en el sistema. En esta figura vemos cómo el desbalanceo para esta ejecución mejora notablemente con el algoritmo ERP. Además, este balanceador junto a la decisión de la predicción

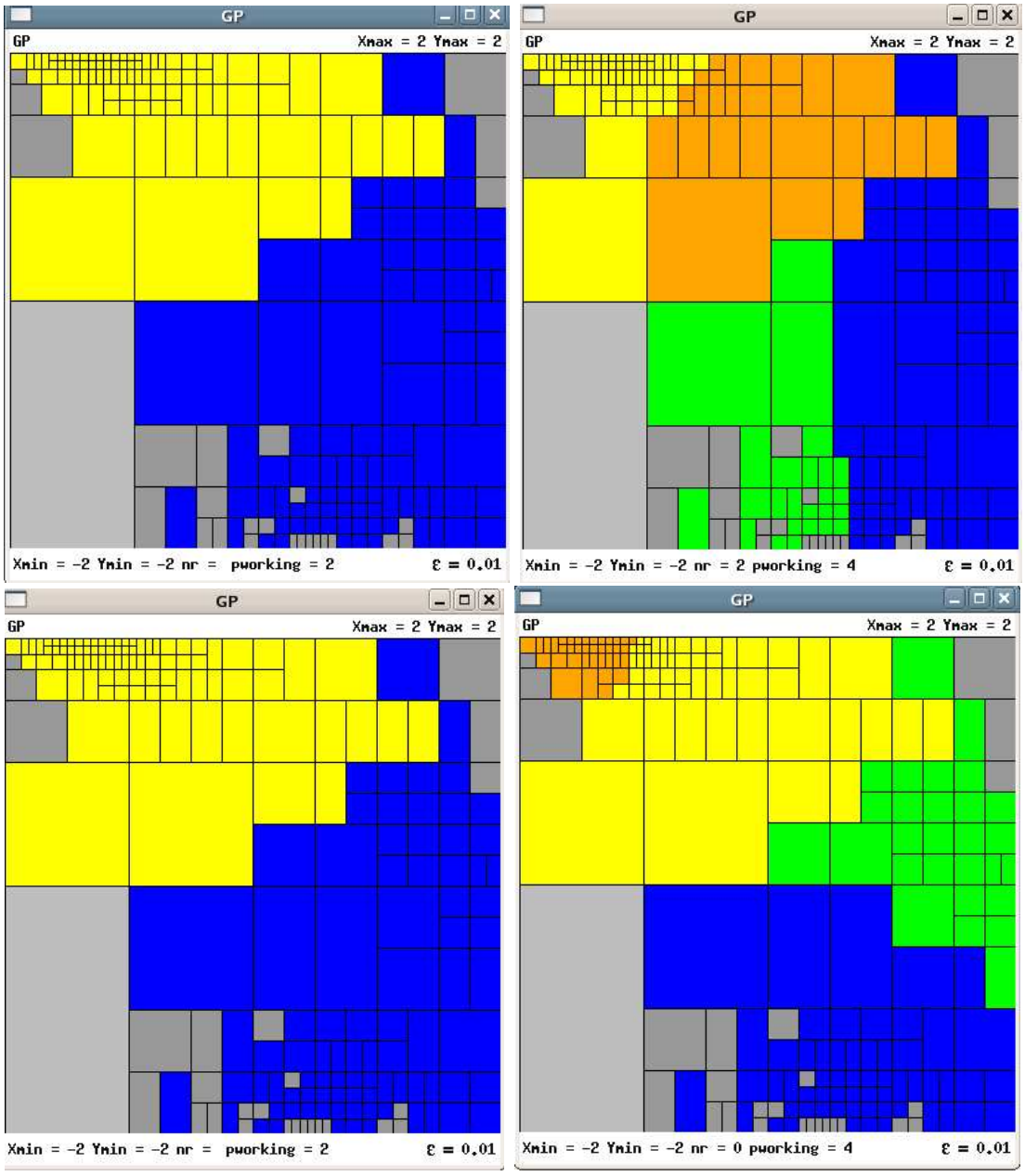


Figura 10.1: Regiones de cajas para la función GP con 4 procesadores en el sistema y una precisión de $\epsilon = 0,01$. Arriba: balanceador ERP. Abajo: balanceador MPR.

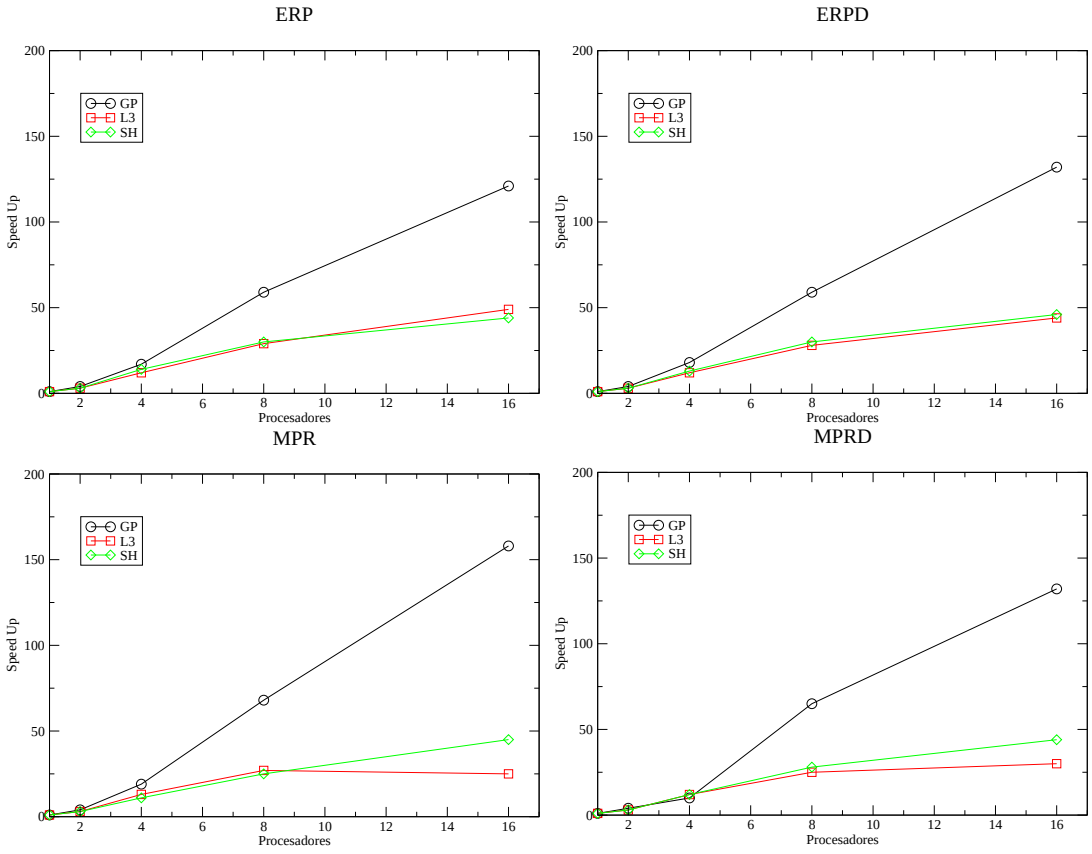


Figura 10.2: Valores del *Speed-Up* de los algoritmos de balanceo de la carga basados en regiones.

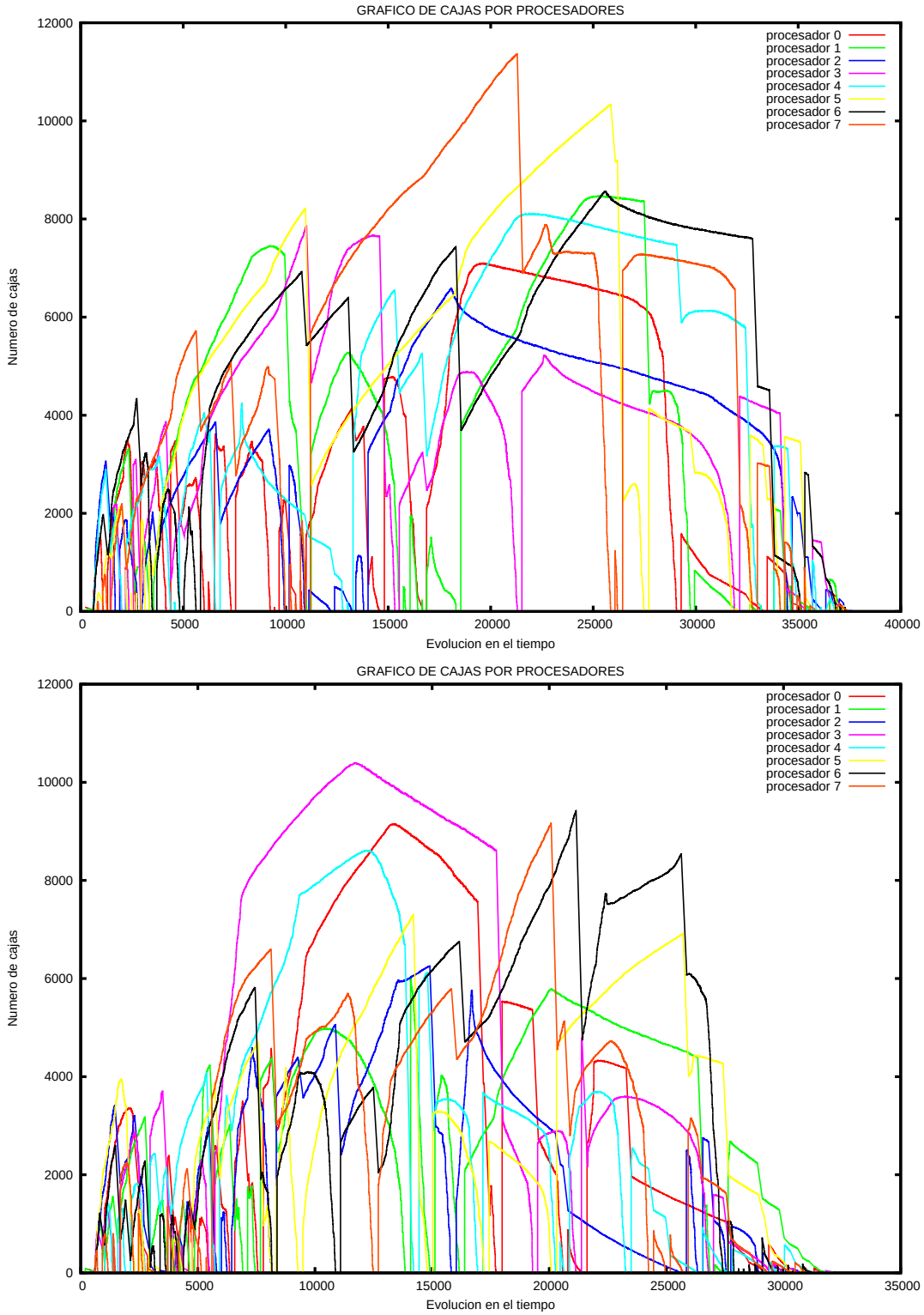


Figura 10.3: Evolución de cajas en los procesadores para el problema GP con 8 procesadores y $\epsilon = 10^{-4}$ sin decisión. Arriba: balanceador ERP. Abajo: balanceador MPR.

mejora significativamente el desbalanceo de las funciones L3 y SH con cualquier número de procesadores y mantiene estable el desabalanceo en la función GP.

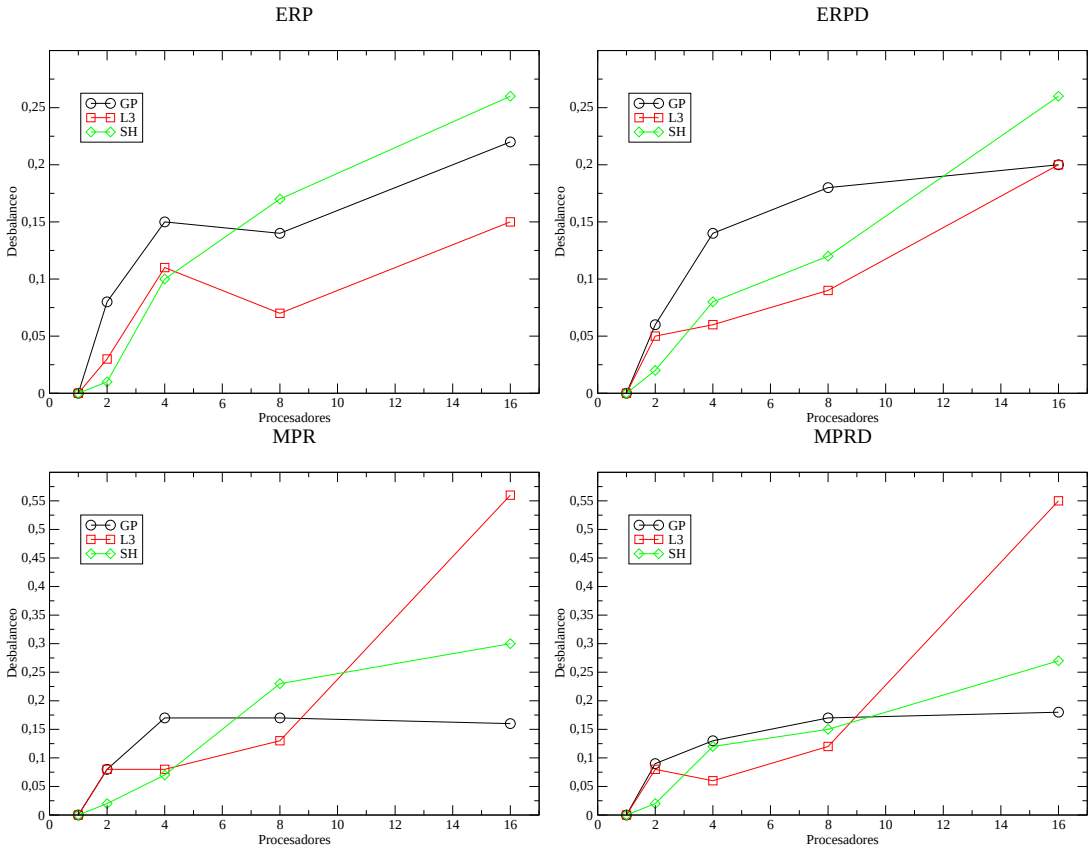


Figura 10.4: Valores de Desbalanceo de los algoritmos de balanceo basados en regiones

Los tiempos de ejecución (ver Figura 10.5) y proceso (ver Figura 10.6) ofrecidos por el balanceador ERP no ofrecen información relevante con respecto a los del balanceador MPR ya que el tiempo de ejecución se analizó junto a los valores de *Speed Up*. En cuanto a los tiempos de proceso, tenemos que decir que utilizar la decisión junto al balanceador ERP mejora los tiempos frente a no utilizarla. Solamente hay 5 casos en los que utilizar la decisión no mejora los tiempos de proceso en el algoritmo ERP: L3 con 2 procesadores, SH con 4 procesadores, GP y L3 con 8 procesadores, y L3 con 16 procesadores (ver datos en Tabla 10.1).

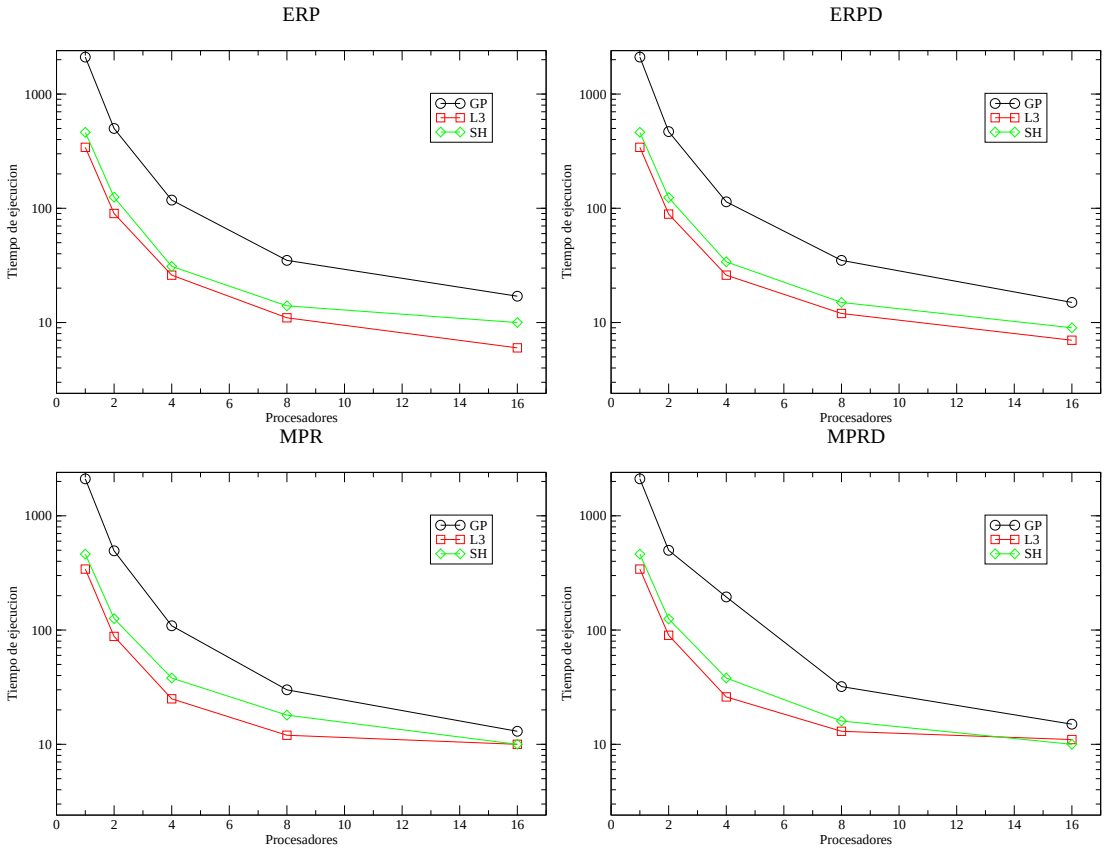


Figura 10.5: Tiempos de ejecución de las funciones test para los algoritmos de balanceo de la carga basados en regiones.

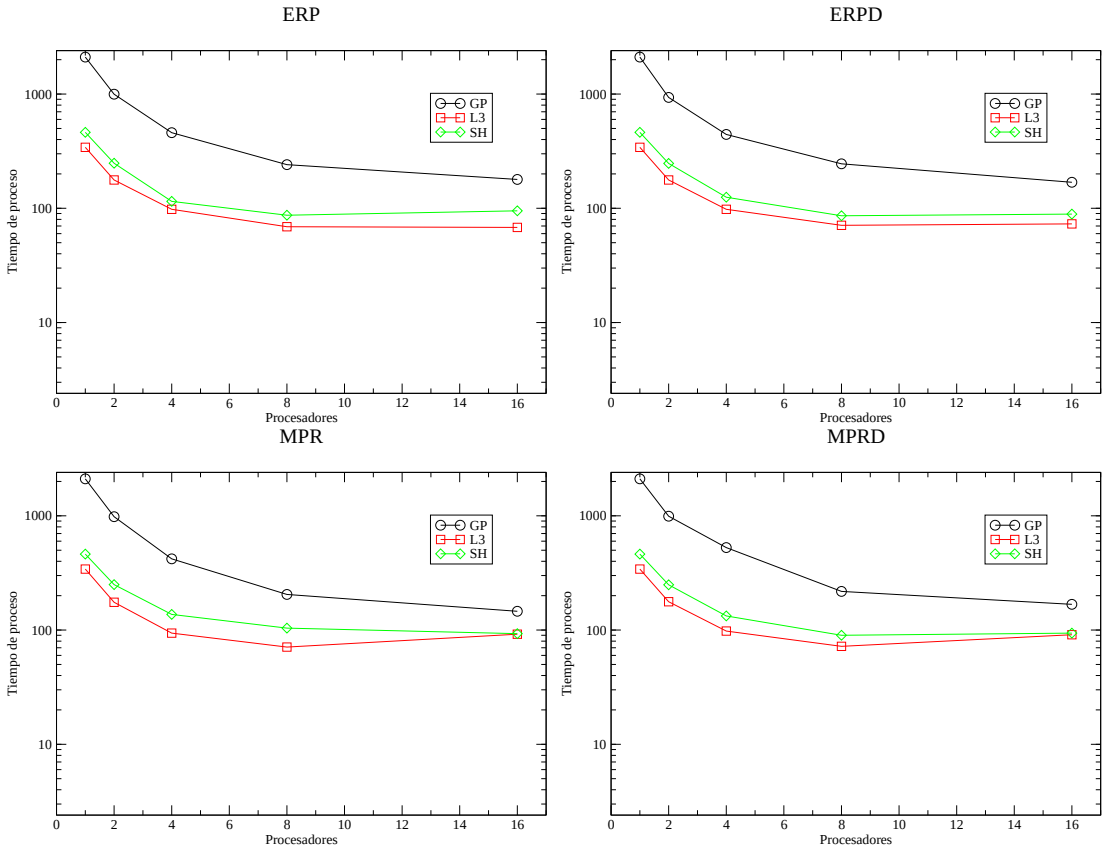


Figura 10.6: Tiempo de proceso de las funciones test para los algoritmos de balanceo de la carga basados en regiones.

Tabla 10.1: Resultados generales del algoritmo de balanceo ERP.

Modelo	Sin decisión			Con decisión		
1 procesador	GP	L3	SH	GP	L3	SH
Esfuerzo	1280878	506704	836332	1280878	506704	836332
T. Ejecución	2106.16	341.83	462.21	2106.16	341.83	462.21
2 procesadores	GP	L3	SH	GP	L3	SH
Esfuerzo	1281577	506704	836332	1281694	506704	836332
Desbalanceo	0,08	0,03	0,01	0,06	0,05	0,02
Speed-up	4,20	3,80	3,69	4,49	3,80	3,72
T. Ejecución	500,46	90,05	125,23	469,43	89,99	124,09
T. Proceso	997,22	177,67	248,97	934,08	177,84	247,56
Predicciones	16	11	7,33	20,67	9,67	3
Nº Cajas Max	43821,83	21220,67	33812	40974,67	20211,67	33812
4 procesadores	GP	L3	SH	GP	L3	SH
Esfuerzo	1281020	506710	836578	1281305	506710	836407
Desbalanceo	0,15	0,11	0,1	0,14	0,06	0,08
Speed-up	17,82	12,74	14,59	18,37	12,85	13,49
T. Ejecución	118,17	26,84	31,69	114,68	26,6	34,27
T. Proceso	459,25	98,8	115,49	443,71	98,39	125,93
Predicciones	68	35	49,33	65,33	34	46,67
Nº Cajas Max	19856,33	9959,42	14820	17757,83	11009,03	14664,9
8 procesadores	GP	L3	SH	GP	L3	SH
Esfuerzo	1281101	506737	836436	1281157	506749	836467,67
Desbalanceo	0,14	0,07	0,17	0,18	0,09	0,12
Speed-up	59,73	29,42	30,94	59,83	28,46	30,65
T. Ejecución	35,26	11,62	14,94	35,2	12,01	15,08
T. Proceso	241,48	69,98	87,52	245,39	71,31	86,76
Predicciones	174,33	91	132	153,33	99,67	133,67
Nº Cajas Max	8598,02	5061,67	8247,46	9450,17	5675,92	6974,17
16 procesadores	GP	L3	SH	GP	L3	SH
Esfuerzo	1280994	506918	836477	1281048	506891	836509
Desbalanceo	0,22	0,15	0,26	0,2	0,2	0,26
Speed-up	121,32	49,04	44,06	132,05	44,22	46,45
T. Ejecución	17,36	6,97	10,49	15,95	7,73	9,95
T. Proceso	179,16	68,31	95,48	169,64	73,02	89,85
Predicciones	367	155,33	281,67	323,67	180	260,67
Nº Cajas Max	4944,79	2970,31	4506,12	5123,46	3048,39	4188

Parte VI

Conclusiones

Capítulo 11

Conclusiones y trabajos futuros

11.1. Conclusiones y principales aportaciones

Este proyecto tenía como principal objetivo la creación de un nuevo modelo de balanceo de la carga que se basara en un reparto de cajas a procesadores libres, de acuerdo con la localidad espacial de las mismas. A lo largo de este proyecto hemos discutido y analizado los principales problemas de este modelo, proponiendo una solución para cada uno de ellos. Finalmente, se implementó un sistema de simulación de una arquitectura paralela configurable a través de parámetros que ejecutara un algoritmo de búsqueda de mínimos utilizando este balanceador. Este programa de simulación paralela nos va a permitir en el futuro, hacer pruebas de nuevas versiones de algoritmos de balanceo y evaluar su rendimiento sin tener que paralelizarlos, ahorrando así bastantes horas de desarrollo. Los resultados ofrecidos por el balanceador MPR han sido comparados con un balanceador que no basa el reparto de cajas en la localidad de las mismas. De los resultados presentados y analizados en el Capítulo 9 podemos obtener las siguientes conclusiones:

1. Los balanceadores basados en regiones realizan el mismo *Esfuerzo* que el basado en Baraja. Por tanto, no se producen anomalías aceleratorias por el simple hecho de tener cajas de una misma región en un mismo procesador, pero el conjunto de funciones de prueba utilizado es pequeño.

2. Los tiempos obtenidos por el balanceador basado en regiones son en general menos eficientes, debido principalmente al tiempo empleado en la creación de estas regiones. Si bien, comparativamente hablando, los resultados son satisfactorios ya que las diferencias de tiempo son mínimas dado el elevado número de predicciones que se realizan.

3. El balanceador basado en regiones ayuda a mantener un menor número de elementos en la lista de trabajo de los procesadores. Esto ayuda a mejorar la eficiencia en las operaciones de manipulación de la lista.

Tras el análisis de estos resultados se propuso un nuevo algoritmo de balanceo basado en regiones en un intento por mejorar la eficiencia del balanceador MPR. El nuevo balanceador ERP se mostró más regular que el anterior en todos los test de pruebas y mejoró la

eficiencia del balanceador MPR cuando utilizaba decisión. Una de las ventajas del balanceador ERP reside en la eliminación de las predicciones inútiles ya que éste siempre va a generar regiones.

Podemos concluir que con un balanceador basado en regiones podemos disponer de procesadores libres cuando no hay regiones disponibles que sean enviadas a los procesadores ociosos, de tal forma que esos procesadores libres se puedan emplear para realizar otras tareas. Este hecho hace que los valores de desbalanceo aumenten, si bien ha demostrado que los tiempos de ejecución y proceso no necesariamente empeoran por este motivo, y cuando así ocurre, el empeoramiento es asumible.

Aunque los resultados obtenidos y las conclusiones aportadas no son todo lo favorecedoras que esperábamos, somos optimistas en cuanto a las ventajas que puede aportar un modelo de balanceo basado en la creación de regiones, ya que tampoco hay grandes diferencias en esos resultados, por lo que tenemos un amplio margen de mejora. Estas mejoras consisten, principalmente, en dos actuaciones:

1. Aplicar métodos de búsqueda local en las regiones, para obtener una cota superior de f^* lo más rápidamente posible. Esperamos que estos métodos de búsqueda local actúen mejor cuando las cajas que hay en un procesador tienen una localidad similar. Así, se conseguiría acelerar el proceso de búsqueda, eliminando muchas más cajas durante este proceso.

2. Mejorar la eficiencia del algoritmo de generación de regiones, tratando de mantener las cajas organizadas en una estructura de acuerdo a su localidad (además del correspondiente valor de $\underline{F}(X)$) en tiempo de ejecución, de forma similar a como se mantiene balanceado un *árbol B*. Esto haría que no tuviéramos que perder tiempo a la hora de repartir las cajas, en cambio, dificulta las operaciones de inserción y eliminación de cajas.

11.2. Trabajos futuros

Como hemos podido comprobar a lo largo de este proyecto, el camino que hemos empezado a recorrer es bastante largo. El camino que nosotros hemos escogido no es el único y seguramente no sea el mejor, por lo que no podemos pensar que el trabajo ya está terminado. Por ello, hemos planteado una serie de trabajos, encaminados a encontrar nuevos métodos que puedan mejorar la eficiencia del trabajo presentado aquí, y por extensión, de los algoritmos de Optimización Global paralelos actuales.

A continuación se exponen aquellos trabajos cuya realización hemos considerado necesaria.

11.2.1. Paralelización del algoritmo de predicción de mínimos

El trabajo principal tras la finalización de este proyecto será la paralelización del algoritmo de balanceo basado en regiones, ampliando el conjunto de funciones test, y dando soporte a problemas de cualquier dimensión. Así, podremos obtener resultados en máquinas paralelas reales y contrastarlos con resultados de otras investigaciones.

11.2.2. Investigar nuevas fórmulas de cálculo del umbral de candidatos

Vimos que uno de los parámetros más importantes que determinaba la presencia o no de regiones era la elección de las cajas candidatas a ser líderes de una región. Las cajas candidatas se elegían en función de un valor umbral de $\underline{F}(X)$. Consideramos necesario analizar en profundidad nuevos métodos y/o fórmulas de elección de líderes para contrastar los resultados con los propuestos en este proyecto.

11.2.3. Nuevos factores de cercanía de cajas

Otro de los valores que influye en la creación de regiones es la elección de líderes, que depende fundamentalmente de la determinación de la cercanía entre las posibles cajas candidatas. Debemos construir y evaluar nuevos métodos que determinen la cercanía entre cajas y comprobar los resultados.

11.2.4. Creación de regiones balanceadas

En este proyecto se ha visto que la creación de regiones consume tiempo de proceso que puede ser vital para mejorar la eficiencia de los algoritmos. El método de creación de regiones que hemos presentado tiene la desventaja de que no se construyen las regiones teniendo en cuenta el balanceo de las mismas, ya que únicamente atendemos a la cercanía entre cajas. La ventaja es que nuestro algoritmo es más rápido y sencillo. El no construir las regiones con una carga balanceada puede entorpecer la mejora de la eficiencia de nuestro algoritmo si se crean regiones con muy poca carga. Otra mejora que puede englobarse dentro de esta es, comprobar la carga de trabajo de las regiones generadas para decidir si es rentable o no enviarlas a otro procesador.

11.2.5. Factor adaptativo del trabajo de un procesador

Consistiría en establecer un nuevo método de estimación del trabajo de un procesador (distinto del utilizado en este proyecto, basado en el $p\overline{f^*}$), que tuviera en cuenta la evolución del trabajo real evaluado en el procesador y el trabajo estimado previamente. Así, la estimación del trabajo sería más precisa, independientemente del problema utilizado.

11.2.6. Control de la presencia de regiones en tiempo de ejecución

Un método que puede garantizar que no se realicen predicciones inútiles de regiones, es controlar cuándo se crean nuevas cajas candidatas o líderes de regiones. Este método se presenta bastante complejo y complicado debido a que la elección de líderes requiere comprobar la lista de trabajo cada vez que se actualiza f^*, además de comprobar la cercanía entre los posibles candidatos, por lo que se harían demasiadas comprobaciones.

Parte VII

Anexos

Apéndice A

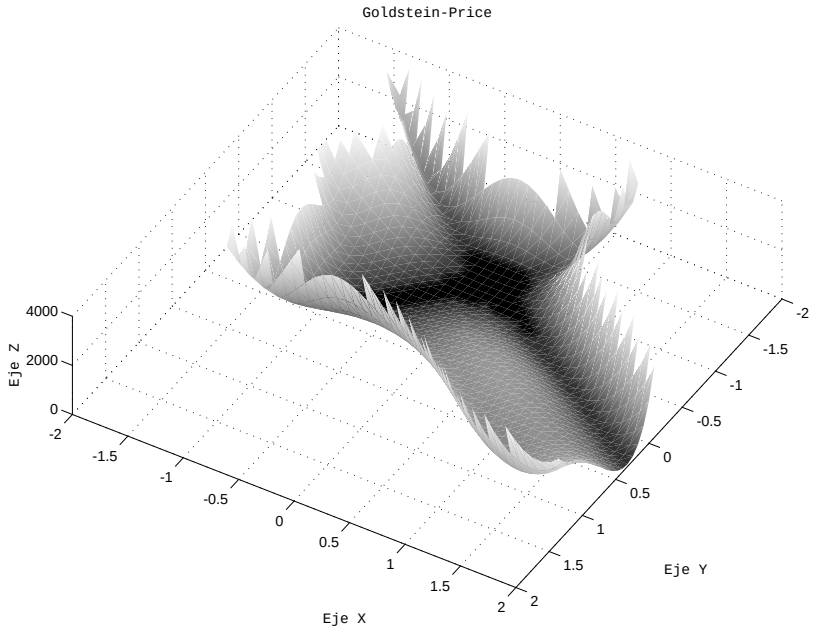
Funciones test

Goldstein-Price (GP)[20]:

Ecuación de la función:

$$f_{GP}(x) = [1 + (x_1 + x_2 + 1)^2(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)] \times [30 + (2x_1 - 3x_2)^2(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)]$$

Dominio (S): $-2 \leq x_i \leq 2$, $i = 1, 2$
Anchura del rango real: $w(f(S)) \simeq 10^6$
Mínimo global: $f^* = 3$, $x^* = (0, -1)^T$



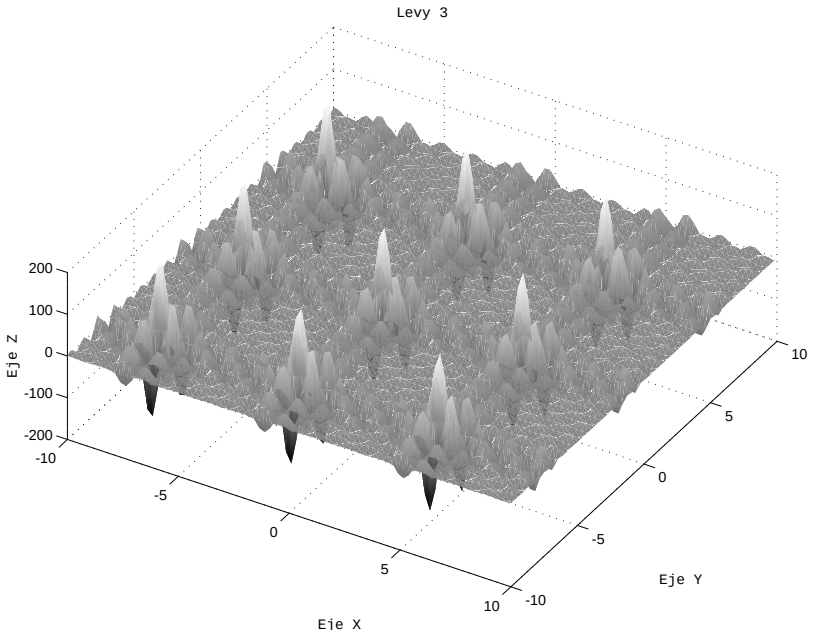
Levy No. 3 (L3)[21]:

Ecuación de la función:

$$f_{L3}(x) = \sum_{i=1}^{5} i\cos((i+1)x_1 + i) \sum_{j=1}^{5} j\cos((j+1)x_2 + j)$$

Dominio (S): $-10 \leq x_i \leq 10$, $i = 1, 2$
Anchura del rango real: $w(f(S)) \simeq 380$
Existen 9 mínimos globales con:
$f^* = -176{,}54179313$

$$X^* = \begin{cases} (4{,}97647760, 4{,}85805687)^T \\ (4{,}97647760, -1{,}42512842)^T \\ (4{,}97647760, -7{,}70831373)^T \\ (-1{,}30670770, 4{,}85805687)^T \\ (-1{,}30670770, -1{,}42512842)^T \\ (-1{,}30670770, -7{,}70831373)^T \\ (-7{,}58989301, 4{,}85805687)^T \\ (-7{,}58989301, -1{,}42512842)^T \\ (-7{,}58989301, -7{,}70831373)^T \end{cases}$$

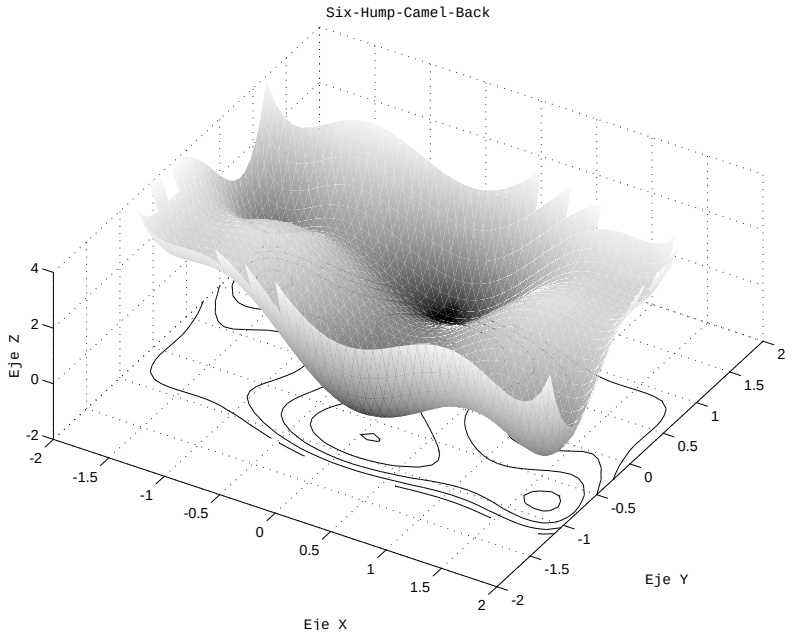


Six Hump Camel Back (SHCB)[20]:

Ecuación de la función:

$$f_{SH}(x) = 4x_1^2 - 2,1x_1^4 + \frac{1}{3}x_1^6 + x1x2 - 4x_2^2 + 4x_2^4$$

Dominio (S): $-5 \leq x_i \leq 5$, $i = 1, 2$
Anchura del rango real: $w(f(S)) \simeq 6400$
Existen 2 mínimos globales con:
$f^* = -1{,}03162845$
$X^* = \left\{ \begin{array}{l} (0{,}08984201, -0{,}71265640)^T \\ (-0{,}08984201, 0{,}71265640)^T \end{array} \right.$



Apéndice B

Direcciones de interés en Internet

B.1. Aritmética de Intervalos

B.1.1. Errores de la Aritmética Computacional:

http://www.ima.umn.edu/∼arnold/455.f96/disasters.html.

B.1.2. Interval Computations:

http://www.cs.utep.edu/interval-comp/

B.1.3. Interval Methods

http://solon.cma.univie.ac.at/interval.html

B.2. Global Optimization

B.2.1. A survey of GO methods:

http://www.cs.sandia.gov/opt/survey/

B.2.2. Global (and Local) Optimization:

http://solon.cma.univie.ac.at/glopt.html

B.2.3. Links of Global Optimization:

http://www.ace.ual.es/ leo/optimizacion.html

Bibliografía

[1] G. Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. In *AFIPS Conf.*, volume 30, page 483. AFIPS Press, 1967.

[2] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. On the solution of traveling salesman problems. *Documenta Mathematica*, Extra Volume Proceedings ICM III (1998):645–656, 1998.

[3] L. G. Casado, J. A. Martínez, and I. García. Experimenting with a new selection criterion in a fast interval search algorithm. *Journal of Global Optimization*, 19(3):247–264, 2001.

[4] L.G. Casado. *Optimización Global basada en Aritmética de Intervalos y Ramificación y Acotacion: Paralelización*. PhD thesis, Universidad de Malaga, Málaga, 1999.

[5] L.G. Casado and I. García. Work load balance approaches for branch and bound algorithms on distributed systems. In IEEE Press, editor, *Proceedings of the 7th Euromicro Workshop on Parallel and Distributed Processing*, pages 155–162, 1999.

[6] R. Corrêa and A. Ferreira. *Parallel Best-First Branch and Bound in Discrete Optimization: a Framework*, pages 145–170. Springer, 1996.

[7] Bernard Gendron and Teodor Gabriel Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, 42(6):1042–1066, 1994.

[8] T. Henriksen and K. Madsen. Parallel algorithms for Global Optimization. *Interval Computations*, 3(5), 1992.

[9] T. Henriksen and K. Madsen. Use of a depth-first strategy in parallel Global Optimization. Technical Report 92-10, Institute for Numerical Analysis, Technical University of Denmark, 1992.

[10] S. Ibraev. *A New Paralle Method for Verified Global Optimization*. PhD thesis, University of Wuppertal, Wuppertal, 2001.

[11] T. Lai and S. Shani. Anomalies in parallel branch-and-bound algorithms. *Communications of the ACM*, 27:594–602, 1984.

[12] T. Lai and A. Sprague. Performance of parallel branch-and-bound algorithms. *IEEE Transactions on Computers*, C-34(10):962–964, 1985.

[13] A.P. Leclerc. *Efficient and Reliable Global Optimization*. PhD thesis, Ohio State University, 1992.

[14] J.D.C. Little, K.G. Murty, D.W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operation Research*, 11:972–989, 1963.

[15] J. A. Martínez. *Métodos de acotación en Algoritmos de Optimización Global Intervalar. Paralelización*. PhD thesis, Universidad de Almería, Almería, 2007.

[16] R. Moore, E. Hansen, and A. P. Leclerc. Rigorous methods for Global Optimization. In C.A. Floudas and P.M. Pardalos, editors, *Recent Advances in Global Optimization*. Princenton University Press, 1992.

[17] R.E. Moore. *Interval analysis*. Prentice-Hall, New Jersey, (USA), 1966.

[18] S.M. Rump. Algorithm for verified inclusions – theory and practice. In R.E. Moore, editor, *Reliability in Computing*, pages 109–126. Academic Press, 1988.

[19] D. Stevenson. IEEE standard for binary floating point arithmetic (IEEE/ANSI 754-1985). Technical report, Floating-Point Working Group, Microprocessor Standards Subcommittee. IEEE, 1985.

[20] A. Törn and A. Žilinskas. *Global Optimization*, volume 3350. Springer-Verlag, Berlin, Germany, 1989.

[21] G. Walster, E. Hansen, and S. Sengupta. Test results for global optimization algorithm. *SIAM Numerical Optimization 1984*, pages 272–287, 1985.

[22] A. Wiethoff. *Verifizierte Globale Optimierung auf Parallelrechnern*. PhD thesis, Universität Karlsruhe, Karlsruhe, 1998.