UNIVERSIDAD DE ALMERÍA

ESCUELA POLITÉCNICA SUPERIOR Y
FACULTAD DE CIENCIAS EXPERIMENTALES

INGENIERÍA INFORMÁTICA

Mínimo número de símplices en la división por el lado más largo de un n-simplex regular

---

Alumno:
José Manuel García Salmerón

Director:
Leocadio González Casado

1 de septiembre de 2014

Dedicado a
mi familia

Agradecimientos

Quiero comenzar dando mi más sincero agradecimiento a Leocadio González Casado, por haberme dirigido durante este proyecto fin de carrera. Por todas las atenciones, por el tiempo que me ha dedicado y sobre todo por todo su apoyo.

Quiero agradecer también a todos los que me han apoyado a lo largo de este camino y me han ayudado, no solo realizar este proyecto, sino también durante estos cinco años. Sin ellos no habría llegado hasta aquí.

A mi familia, agradezco el apoyo incondicional que he recibido en todo momento, tanto en los buenos como en los malos momentos siempre me han apoyado. Gracias por estar siempre ahí.

A mis compañeros y amigos de clase, que me han ayudado enormemente durante estos años. En cada asignatura, con cada práctica y en cada desayuno. Hemos pasado buenos y malos momentos, pero de todas las situaciones hemos aprendido cosas importantes.

Al Departamento de Informática por hacer posible este proyecto, en especial a Guillermo Aparicio de las Llanderas por ayudarme en todo momento y hacerme comprender mejor el complejo mundo de la Optimización Global y sus algoritmos.

Por último gracias a todos los profesores por todo su esfuerzo y dedicación durante todo este tiempo.

Gracias.

Índice general

1. Introducción al problema **1**

2. Trabajos relacionados **3**

3. Optimización Global y Algoritmos de Ramificación y Acotación **5**
3.1. Introducción . . . 5
3.2. Optimización matemática . . . 6
3.2.1. Problema general . . . 6
3.2.2. Clasificación de los problemas de optimización . . . 8
3.2.3. Clasificación de los algoritmos de optimización . . . 9
3.2.3.1. Métodos estocásticos . . . 10
3.2.3.2. Métodos determinísticos . . . 10
3.3. Algoritmos de Ramificación y Acotación . . . 10
3.3.1. Reglas básicas de los algoritmos de Ramificación y Acotación . . . 11
3.3.2. Heurística en los algoritmos de Ramificación y Acotación . . . 15

4. Algoritmos paralelos **16**
4.1. Introducción . . . 16
4.2. Clasificación de las Arquitecturas Paralelas . . . 17
4.2.1. Secuencia de instrucciones y datos . . . 17
4.2.2. Modelos de memoria . . . 19
4.2.3. Topologías de las redes de interconexión . . . 20
4.2.3.1. Redes estáticas . . . 21
4.2.3.2. Redes dinámicas . . . 23
4.3. Medidas de rendimiento para algoritmos paralelos . . . 24
4.3.1. Ley de Amdahl . . . 25
4.3.2. Ley de Gustafson . . . 26
4.4. Paralelización de algoritmos secuenciales . . . 26
4.5. Balanceo de la carga . . . 28
4.6. Algoritmos de Ramificación y Acotación paralelos . . . 30

5. Descripción del problema **35**
5.1. ¿Qué es un simplex? . . . 35
5.1.1. Politopos . . . 35

5.2. Bisección por el Lado Mayor (BLM) 38
5.3. Particionamiento de un n-símplice regular 40
5.3.1. Símplices con más de un lado mayor. Obtención del árbol completo mínimo 40
5.3.2. Símplices regulares 42
5.3.3. Símplices simétricos 43

6. Soluciones planteadas **45**
6.1. Versión secuencial 45
6.2. Versiones paralelas 48
6.2.1. Versión paralela basada en dos fases (B2F) 50
6.2.2. Versión paralela con creación dinámica de hebras (CDH) 53

7. Resultados **56**
7.1. Versión secuencial 56
7.2. Versión paralela basada en dos fases (B2F) 57
7.2.1. Resultados de B2F para un 3-símplice 57
7.2.1.1. Resultados B2F 57
7.2.1.2. Resultados B2F+ 58
7.2.1.3. Comparación de las alternativas para B2F 59
7.2.2. Resultados de B2F+ para un 4-símplice 62
7.3. Versión paralela con creación dinámica de hebras (CDH) 63
7.3.1. Resultados de CDH para un 3-símplice 63
7.3.2. Resultados de CDH para un 4-símplice 64
7.4. Comparación de los resultados de las distintas versiones 65
7.4.1. Comparación de los resultados para un 3-símplice 65
7.4.2. Comparación de los resultados para un 4-símplice 68

8. Secuencia de lados **72**
8.1. Objetivo de la secuencia 72
8.2. ¿Como obtener la secuencia de lados? 72
8.3. Secuencia del 3-símplice 73
8.3.1. Excepciones en la secuencia 74
8.3.2. Resultados aplicando la secuencia 75

9. Conclusiones y principales aportaciones **77**

Índice de figuras

3.1. Ejemplo de un problema con múltiples mínimos locales. 7

4.1. Taxonomía de Flynn. 18
4.2. Topologías de redes estándar en sistemas paralelos. 22
4.3. Red de lineas cruzadas. 23
4.4. Red genérica multietapa. 24

5.1. Ejemplos de distintos politopos. 36
5.2. Símplices de dimensiones 2, 3 y 4. 37
5.3. Un 2-símplice regular con lados de longitud unidad. 38
5.4. Primeras bisecciones en un 2-simplex regular. 39
5.5. Bisecciones y niveles del árbol binario obtenido mediante BLM de un 2-simplex. 40
5.6. Diferentes tipos de símplices 41
5.7. 2-símplices regulares y no regulares. 42
5.8. 3-símplices regulares y no regulares. 43
5.9. 2-símplices simétricos y no simétricos 43
5.10. 3-símplices simétricos y no simétricos. 44

6.1. Árbol binario para $\epsilon = 0{,}5$. 46
6.2. Esquema del algoritmo secuencial. 48
6.3. Esquema del algoritmo B2F (MaxHebras=4). 51
6.4. Esquema del algoritmo CDH (MaxHebras=4). 55

7.1. Gráfica comparativa de SpeedUp según la carga inicial para B2F. 60
7.2. Gráfica comparativa de balanceo de ramas según la carga inicial para B2F. 61
7.3. Gráfica comparativa de tiempos para los algoritmos propuestos para un 3-símplice con $\epsilon = 0,025$ 66
7.4. Gráfica comparativa de speedup para los algoritmos propuestos para un 3-símplice con $\epsilon = 0,025$ 66
7.5. Gráfica comparativa de tiempos para los algoritmos propuestos para un 3-símplice con $\epsilon = 0,01$ 67
7.6. Gráfica comparativa de speedup para los algoritmos propuestos para un 3-símplice con $\epsilon = 0,01$ 68

7.7. Gráfica comparativa de tiempos para los algoritmos propuestos para un 4-símplice con $\epsilon = 0,2$ 69
7.8. Gráfica comparativa de speedup para los algoritmos propuestos para un 4-símplice con $\epsilon = 0,2$ 69
7.9. Gráfica comparativa de tiempos para los algoritmos propuestos para un 4-símplice con $\epsilon = 0,125$ 70
7.10. Gráfica comparativa de speedup para los algoritmos propuestos para un 4-símplice con $\epsilon = 0,125$ 71

Índice de tablas

3.1. Clasificación de los problemas de optimización. 8

7.1. Resultados secuenciales para un 3-símplice 56
7.2. Resultados secuenciales para un 4-símplice 57
7.3. Tiempos y ganancia en velocidad de la versión B2F para un 3-símplice y $\epsilon = 0{,}025$ 58
7.4. Desbalanceo de la versión B2F para un 3-símplice y $\epsilon = 0{,}025$ 58
7.5. Análogo a la tabla 7.3 usando MinNIniS = 4 · MaxHebras^2 en (6.1) 59
7.6. Análogo a la tabla 7.4 usando MinNIniS=4×MaxHebras^2 en (6.1) 59
7.7. Comparación de tiempos y ganancia en velocidad de la versión B2F para un 3-símplice y $\epsilon = 0{,}025$ según el número de símplices iniciales. 59
7.8. Desbalanceo de la versión B2F para un 3-símplice y $\epsilon = 0{,}025$ según el número inicial de símplices. 60
7.9. Análogo a la tabla 7.5 usando $\epsilon = 0{,}01$ 61
7.10. Análogo a la tabla 7.6 usando $\epsilon = 0{,}01$ 62
7.11. Análogo a la tabla 7.5 para un 4-símplice y $\epsilon = 0{,}2$ 62
7.12. Análogo a la tabla 7.6 para un 4-símplice y $\epsilon = 0{,}2$ 62
7.13. Análogo a la tabla 7.11 usando $\epsilon = 0{,}125$ 63
7.14. Análogo a la tabla 7.12 usando $\epsilon = 0{,}125$ 63
7.15. Tiempos y ganancia en velocidad de la versión CDH para un 3-símplice y $\epsilon = 0{,}025$ 64
7.16. Análogo a la tabla 7.15 usando $\epsilon = 0{,}01$ 64
7.17. Tiempos y ganancia en velocidad de la versión CDH para un 4-símplice y $\epsilon = 0{,}2$ 64
7.18. Análogo a la tabla 7.17 usando $\epsilon = 0{,}125$ 65
7.19. Tiempos y ganancias de velocidad para los algoritmos propuestos para un 3-símplice con $\epsilon = 0,025$ 65
7.20. Tiempos y ganancias de velocidad para los algoritmos propuestos para un 3-símplice con $\epsilon = 0,01$ 67
7.21. Tiempos y ganancias de velocidad para los algoritmos propuestos para un 4-símplice con $\epsilon = 0,2$ 68
7.22. Tiempos y ganancias de velocidad para los algoritmos propuestos para un 4-símplice con $\epsilon = 0,125$ 70

8.1. Secuencia árbol mínimo 3-símplice 74

8.2. Valores de ϵ para los que se cumple la secuencia 75
8.3. Tiempos y tamaños de árbol según la heurística para un 3-símplice. 76

Capítulo 1

Introducción al problema

La división de un símplice regular aparece en varias aplicaciones industriales que usan algoritmos de Optimización Global basados en Ramificación y Acotación. Normalmente la división de un símplice se realiza mediante bisección del lado mayor, debido a que garantiza la convergencia del algoritmo. El objetivo de este trabajo es determinar el árbol mínimo cuando existen varios lados mayores que pueden ser divididos. Esto ocurre para un n-símplice en un espacio d-dimensional con $d = n + 1 > 3$.

El objetivo de este trabajo es el de determinar el tamaño del árbol mínimo generado a partir de un símplice (sección 5.1, página 35). Con ello será posible determinar, o por al menos estimar, la carga computacional que cada símplice a procesar puede generar. Con esta información se pretende mejorar el balanceo dinámico de la carga computacional (sección 4.5, página 28) en las versiones paralelas de algoritmos de Ramificación y Acotación que resuelven problemas de Optimización Global (sección 4.6, página 30).

Por lo tanto, la primera pregunta que se debe realizar es:

Cuestión 1 *¿Cual es el tamaño mínimo del árbol que se genera a partir de un determinado símplice?*

Obtener el árbol mínimo implicará procesar todos los posibles subárboles que se pueden generar a partir de un determinado símplice. Se obtendrá un subárbol por cada uno de los lados por los que sea posible dividir el símplice. Una vez obtenidos todos los posibles subárboles, estos tendrán que ser comparados con el fin de determinar cual es el subárbol mínimo (sección 5.3.1, página 40).

En este trabajo se empleara el método de división por lado mayor (sección 5.2, página 38). Por lo tanto, un símplice generara tantos subárboles como número de lados mayores posea, salvo que el simplex sea regular en cuyo caso se tomara cualquiera de los lados, ya que cualquiera de ellos generará subárboles de igual tamaño (sección 5.3, página 40). Esto quiere decir que será necesario determinar por que lado es necesario dividir el símplice para obtener el árbol mínimo, lo que genera una nueva pregunta:

Cuestión 2 *¿Por que lado mayor es necesario dividir un símplice para obtener el árbol mínimo?*

Para ser capaz de responder esta pregunta, se necesitara recorrer todos los posibles subárboles que se vayan generando y posteriormente comparar el tamaño de estos para determinar cual es el de menor tamaño. Se generara un subárbol por cada lado mayor del símplice, lo que aumentara la cantidad de computo necesaria para resolver el problema y hará necesario la utilización de algoritmos paralelos (ver el capítulo 4).

Este problema puede verse como un problema de optimización (ver el capítulo 3) en el que se desea minimizar el tamaño del árbol a generar. Cuando los problemas de optimización son muy complejos se suelen usar heurísticas a la hora de seleccionar el siguiente nodo a procesar para intentar seguir el camino mínimo, es decir la rama del árbol que lleva a la solución. Esto será estudiado en el capítulo 8, donde se presentara la secuencia de lados mayores a dividir para resolver determinados problemas, explicando el proceso a seguir para obtenerla y aplicarla.

En este trabajo se hace una revisión de los estudios realizados (ver el capítulo 2) y se investiga el efecto del método de división por lado mayor sobre el número de símplices generados. El objetivo final del estudio es diseñar un algoritmo paralelo para la resolución del problema planteado y poder establecer una regla de división que genere el menor árbol, permitiendo así reducir el coste computacional de los algoritmos de Ramificación y Acotación aplicados a un espacio inicial determinado por un n-símplice regular.

El resto del trabajo esta compuesto por el capítulo 5, donde se describe el problema, los elementos que lo componen. Los algoritmos de Ramificación y Acotación utilizados para solucionar los problemas de Optimización Global que generan este tipo de árboles de búsqueda son explicados de forma general en la sección 3.3 del capítulo 3, mientras que la versión paralela de este tipo de algoritmos es descrita en la sección 4.6 del capítulo 4. En el capítulo 6 se presentan los algoritmos propuestos para la resolución del problema y el capítulo 7 se muestran los resultados de los algoritmos propuestos. Por último en el capítulo 9 se muestran las conclusiones y trabajos futuros.

Capítulo 2

Trabajos relacionados

Los algoritmos de Ramificación y Acotación (Branch-and-Bound, B&B) se aplican a la resolución de problemas de Optimización Global mediante la realización de una búsqueda exhaustiva del mínimo de una función objetivo en un dominio dado. En este proyecto se está interesado en problemas donde el espacio de búsqueda está definido por un n-símplice regular, definido en un espacio $(n+1)$-dimensional [6]. Los algoritmos de B&B están caracterizados por las reglas de Acotación, Selección, División, Rechazo y Terminación [7]. Se pretende estudiar como conseguir el árbol de búsqueda generado con tamaño mínimo cuando solo se tienen en cuenta las reglas de División y Terminación, es decir, ningún nodo del árbol es eliminado. Se usará como regla de división la Bisección por el Lado Mayor (BLM) [1, 15, 33] y como regla de terminación la longitud del lado mayor de un subproblema o nodo del árbol.

En general, resolver problemas de Optimización Global usando B&B requiere una capacidad computacional que se incrementa exponencialmente con la dimensión del problema a tratar. Por lo tanto es necesario usar la programación paralela para reducir el tiempo de computo. Como en este tipo de algoritmos la carga computacional cambia dinámicamente durante el tiempo de ejecución, la distribución del trabajo entre los distintos procesadores requiere de un balanceador dinámico de la carga. En [3, 4] se estudian métodos para estimar el número de nodos del árbol que restan por evaluar, en algoritmos de optimización global intervalar donde el espacio inicial es un hiper-rectángulo, así se provee de información sobre el tiempo restante y los recursos necesarios. Estos estudios podrían extenderse a espacios iniciales determinados por un n-símplice, si se pudiera estimar el tamaño del árbol generado a partir de un símplice.

Ya que la cantidad de computo requerido para explorar el árbol de búsqueda puede llegar a ser enorme, se hace necesario el uso de paralelismo para reducir los tiempos de ejecución y poder estudiar arboles de mayor tamaño. En [31, 32] se investiga la eficiencia de los algoritmos B&B sobre un sistema distribuido combinando PThreads y MPI, MPI se emplea para distribuir el trabajo entre los distintos nodos (internodo) y con PThreads se ejecuta el trabajo en nodos multicore (intranodo).

En [8] se propone la creación dinámica de hebras, para adaptar el nivel de paralelismo de la aplicación a los recursos disponibles en el sistema. Si hay que crear una nueva hebra de la aplicación, una hebra con suficiente trabajo, genera una nueva hebra asignándole la

mitad de su trabajo pendiente. Este trabajo estudia como obtener la estimación del trabajo pendiente en la lista de nodos pendientes de procesar, cuando el espacio de búsqueda es un n-símplice. Se conoce el número de nodos a evaluar, pero no cuantos nodos o subárboles se generaran a partir de un nodo cualquiera.

Para mejorar la estimación de la carga hay que determinar el lado por el que debe ser dividido el símplice y estudiar como esta elección afectara al número de subproblemas generados y al tamaño de cada subárbol. La pregunta es como elegir el lado mayor por el que dividir tal que el número de símplices generados en el árbol binario sea el menor posible (véase la cuestión 2) [25, 27, 28].

Debido a la estructura de los símplices y a la forma de construir el árbol de búsqueda, podría ser posible obtener una regla para determinar el lado por el que dividir cada símplice sin necesidad de evaluar todas las opciones [26].

Capítulo 3

Optimización Global y Algoritmos de Ramificación y Acotación

3.1. Introducción

La resolución de un problema de Optimización Global consiste en encontrar el valor mínimo o máximo de una función objetivo f, satisfaciendo ciertas restricciones en el espacio de búsqueda. Para problemas de minimización, el problema de Optimización Global puede escribirse como

$$f^* = f(x^*) = \min_{x \in S} f(x). \quad (3.1)$$

Cuando se requiere una búsqueda exhaustiva del mínimo global, se suelen utilizar algoritmos B&B. Un algoritmo B&B realiza una búsqueda mediante la descomposición sucesiva del problema inicial en subproblemas de menor tamaño hasta alcanzar la(s) solución(es) final(es) o una aproximación a ella(s). Esta aproximación está determinada por la precisión requerida para la solución. El algoritmo crea un árbol en el que se puede podar una rama cuando se determina que un subproblema no contiene una solución global. Para realizar la poda o rechazo de un subproblema se suelen calcular cotas de los valores de la función objetivo para cada subproblema. El algoritmo 1 muestra un ejemplo básico. El comportamiento, y por tanto la eficiencia de un algoritmo B&B, se puede caracterizar por cinco reglas: Selección, Acotación, Rechazo, División y Terminación. En este estudio no se van a aplicar las reglas de Acotación, Rechazo ni de Selección ya que solo estamos interesados en encontrar la regla de División basada en la bisección del lado mayor que genere el menor árbol, cuando el espacio de búsqueda es un n-símplice regular. Por lo tanto, al no existir poda, el árbol se generará completamente. Su tamaño no solo dependerá del lado mayor elegido para ser dividido, sino también de la regla de terminación usada. En nuestro caso, un símplice con un tamaño del lado mayor, o una anchura $w(S)$, menor que un umbral ϵ no requerirá de más procesado.

Algoritmo 1 Ramificación y Acotación

```
1: Λ := {S_1 = S}                                  Conjunto de trabajo
2: Ω := {}                                         Conjunto final
3: ns := 1                                         Número de símplices
4: while Λ ≠ ∅ do
5:    Selecciona S_i de Λ                          Regla de selección
6:    Evalúa S_i                                   Regla de acotación
7:    if S_i no puede ser eliminado then           Regla de rechazo
8:       if w(S_i) ≤ ϵ then                        Regla de terminación
9:          Almacena S_i in Ω
10:      else
11:         {S_2i, S_2i+1} := división(S_i)        Regla de división
12:         Almacena S_2i, S_2i+1 en Λ
13:         ns := ns + 2
14:      end if
15:   end if
16: end while
17: return Ω y f(x*)
```

3.2. Optimización matemática

La Optimización Matemática es el nombre formal de la rama de la ciencia computacional, que busca una respuesta a la pregunta: ¿Qué es mejor?, para problemas donde la calidad de la respuesta puede expresarse mediante un valor. Estos problemas aparecen en áreas científicas como Matemáticas, Física, Química, Biología, Ingeniería, Arquitectura, Economía y Administración, existiendo también un amplio rango de técnicas disponibles para resolverlos.

En esta sección se realizará una breve clasificación de los diferentes problemas de Optimización y de los métodos que existen para resolverlos.

3.2.1. Problema general

El objetivo en un problema de optimización es encontrar la mejor solución, de forma que se minimice o maximice el valor de la función objetivo, posiblemente sujeta a una serie de restricciones sobre los posibles candidatos. La descripción general del problema es:

$$\begin{aligned} minimizar\ & f(x) \\ sujeto\ a:\ & g(x) \leq 0, \\ & h(x) = 0, \\ & \underline{x} \leq x \leq \overline{x}, \\ & x \in \mathbb{R}^n \end{aligned} \tag{3.2}$$

donde $f(x) : \mathbb{R}^n \to \mathbb{R}$ es la función objetivo y $g(x) : \mathbb{R}^n \to \mathbb{R}^m$ y $h(x) : \mathbb{R}^n \to \mathbb{R}^k$ son funciones que restringen la región de búsqueda de la solución.

Se define la región de búsqueda de los posibles candidatos, o dominio del problema (3.2), como:

$$S = \{x \in \mathbb{R}^n : g(x) \leq 0,\ h(x) = 0,\ \underline{x} \leq x \leq \overline{x}\} \tag{3.3}$$

En (3.3), $\underline{x}$ y $\overline{x}$ son límites (finitos) implícitos. Cuando sólo se da el tipo de restricciones por límites implícitos en la región de búsqueda, el problema de optimización se suele incluir en el grupo de problemas denominados sin restricciones.

Una solución global del problema (3.2) viene dada por $x^* \in S$ de forma que $f^* = f(x^*) \leq f(x),\ \forall x \in S$. Una solución local del problema (3.2), se define como $\tilde{f} = f(\tilde{x}) \leq f(x), \forall x \in S \cap N_\epsilon(\tilde{x})$ con $\epsilon > 0$, donde $N_\epsilon(\tilde{x})$ son los ϵ-vecinos del punto $\tilde{x}$, definidos por $N_\epsilon(\tilde{x}) = \{x : \parallel x - \tilde{x} \parallel < \epsilon\}$. Un mínimo local no se considera una solución óptima para (3.2) hasta que no se demuestra que es una solución global. Sólo se considera el problema de minimización, ya que el de maximización puede reescribirse como uno de minimización ($\max\limits_{x \in S}\{f(x)\} = \min\limits_{x \in S}\{-f(x)\}$).

Si todas la funciones definidas anteriormente son continuas, el conjunto de soluciones óptimas para el problema (3.2) es no vacío.

Existe una amplia variedad de problemas donde la existencia de un solo mínimo no puede ser postulada o verificada, con lo que su resolución se vuelve más difícil. Ejemplos de esos problemas pueden encontrarse en [5, 16, 17, 24, 51, 54, 60].

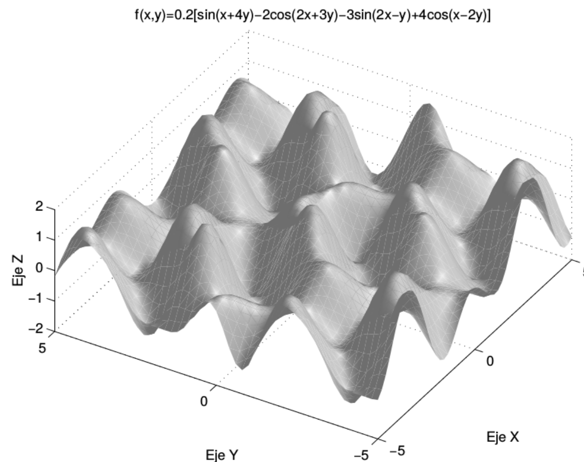


Figura 3.1: Ejemplo de un problema con múltiples mínimos locales.

El campo emergente de la Optimización Global se ocupa de problemas de programación matemática, con la (posible) presencia de múltiples mínimos locales. Este número de mínimos locales es normalmente desconocido y puede ser bastante grande. Además, las diferencias entre los valores de la función objetivo en los mínimos locales y globales pueden ser muy grandes. Debido a esto, las soluciones locales no son válidas en la mayoría de los casos. Incluso para diferencias pequeñas entre los valores en los mínimos locales y globales, el no encontrar el mínimo global puede traducirse, especialmente en

aplicaciones económicas, en millones de euros de pérdidas. Por lo tanto la Optimización Global puede llegar a ser extremadamente importante. Dichas estructuras de mínimos dan lugar a representaciones parecidas a paisajes montañosos. Como ejemplo, la Figura 3.1 muestra una, relativamente simple, composición de funciones trigonométricas con argumentos polinomiales, sobre un espacio de búsqueda bidimensional. Naturalmente, ante tales circunstancias, es esencial usar una estrategia de búsqueda global apropiada.

Los primeros y esporádicos trabajos sobre Optimización Global surgieron a finales de los cincuenta. La evolución desde entonces ha sido muy grande, de forma que el *estado del arte* en la actualidad se caracteriza por varias decenas de monográficos y varios miles de artículos de investigación dedicados exclusivamente a este tema.

3.2.2. Clasificación de los problemas de optimización

Una clasificación general, dependiente del número de variables, de su tipo y del tipo de funciones involucradas en (3.2) se describe en la Tabla 3.1.

Tabla 3.1: Clasificación de los problemas de optimización.

Característica	**Propiedad**	**Clasificación**
Número de variables	Una	Unidimensional
	Varias	Multidimensional
Tipo de variables	Reales continuas	Continua
	Enteras	Entera o discreta
	Reales continuas y enteras	Entera mezclada
	Permutaciones de enteros	Combinatoria
Tipo de Funciones	Funciones lineales	Lineal
	Funciones cuadráticas	Cuadrática
	Otras funciones no lineales	No Lineal.
Formulación del problema	Sujeto a restricciones	Con restricciones
	No sujeto a restricciones	Sin restricciones

Por ejemplo, cuando las variables de los problemas de Optimización son variables enteras, estos problemas se llaman también problemas de *Programación Entera* (IP, Integer Programming). Si las funciones del problema son lineales se llaman problemas de *Programación Lineal* (LP, Linear Programming). Si por el contrario, dichas funciones no son lineales, nos encontramos ante problemas de *Programación No Lineal* (NLP Non Linear Programming), etc.

Una enumeración más concreta puede encontrarse en [55] y en [34], donde además existen numerosas referencias de los siguientes problemas de optimización: Programación Bilineal o Biconvexa, Optimización Combinatoria, Minimización Cóncava, Optimización

Global Continua, Diferencial Convexa (DC), Programación Fraccional, Problemas Lineales y No Lineales Complementarios, Optimización Lipschitz, Problemas MiniMax, Optimización Multinivel, Programación Multiobjetivo, Programación Multiplicativa, Problemas de Redes de Trabajos, Programación Paramétrica No Convexa, Optimización Cuadrática, Programación Inversa Convexa, Optimización Global Separable y otros modelos probabilísticos no convexos.

En cuanto a la complejidad de la resolución de estos problemas, Pardalos y Schnitger [52] mostraron que demostrar que un posible punto candidato es un óptimo local, para problemas de Programación Cuadrática, es un problema NP-Duro. Pardalos y Vavasis [53], demostraron también que la Programación Cuadrática con un autovalor negativo es un problema NP-Duro. Resolver el problema no convexo más simple de Programación No Lineal, es NP-Duro. Esto indica, que el tiempo de resolución de cualquier algoritmo, en el peor caso, se incrementa exponencialmente con el tamaño del problema (teoría acerca de NP-Completitud puede encontrarse en [20]). Consecuentemente, problemas de dimensión cien, cincuenta o incluso diez, pueden considerarse como "grandes", dependiendo del problema de Optimización Global investigado (recuérdese la Figura 3.1).

3.2.3. Clasificación de los algoritmos de optimización

Existen muchas clases de algoritmos de Optimización Global que poseen fuertes propiedades teóricas de convergencia y al menos, en principio, se pueden implementar y aplicar directamente. Todos estos algoritmos rigurosos de Optimización Global, como se comentó anteriormente, tienen una demanda computacional que se incrementa en un orden no polinomial con respecto al tamaño del problema, incluso en las instancias más simples. Cuando se requiere una eficiencia mayor, los algoritmos de Optimización Global pueden hacer uso de fases de optimización local tradicionales, las cuales convergen rápidamente cuando están cerca de una solución o de un óptimo local, estando fuera del ámbito de estos métodos locales el encontrar todas las soluciones, o probar la existencia o ausencia de las mismas. Los métodos tradicionales pueden fallar, no obteniendo una convergencia en problemas difíciles de resolver. Sin embargo, la convergencia global debe garantizarse por medio de los componentes del algoritmo con ámbito global, los cuales, teóricamente, deberían usarse de una forma exhaustiva. Estas necesidades indican la dificultad de desarrollar algoritmos eficientes y robustos de Optimización Global.

A continuación, se resumen las cualidades que debería tener un algoritmo de Optimización Global:

- **Corrección:** No producir resultados incorrectos.
- **Completitud:** Encontrar todas las posibles soluciones.
- **Finitud:** Garantizar la convergencia.
- **Certeza:** Probar la existencia o inexistencia de soluciones.

Sólo unas pocas de las implementaciones existentes garantizan todas las cualidades mostradas.

La clasificación de los algoritmos de optimización suele generalizarse haciendo una distinción entre algoritmos estocásticos y determinísticos.

3.2.3.1. Métodos estocásticos

Los métodos estocásticos poseen un comportamiento no determinista, lo que significa que cada uno de lo estados puede estar formado por elementos aleatorios y la forma de llegar a cada estado también puede ser aleatoria. No obstante, de acuerdo a [39, 50] cualquier desarrollo temporal (sea determinista o probabilístico) que pueda ser analizable en términos de probabilidad puede ser denominado como un método estocástico.

Todos los métodos estocásticos usan algún factor aleatorio en sus algoritmos y la demostración de su convergencia depende de argumentaciones estadísticas [59, 60]. Normalmente los métodos estocásticos se aplican a problemas sin restricciones y no necesitan ningún conocimiento sobre la función objetivo, dando a menudo resultados óptimos o cercanos a los óptimos. Se basan en obtener valores de la función objetivo en puntos de la región de búsqueda elegidos aleatoriamente.

Estos métodos pueden requerir un número aleatorio de iteraciones para alcanzar una solución al problema. Son el recurso a utilizar cuando el tamaño del problema, dimensionalmente hablando, es grande o el problema no está definido analíticamente. Las desventajas de estos métodos es que al emplear un número no determinado de iteraciones, no garantizan que se encuentre la solución en un número finito de pasos, ni que el mínimo encontrado sea el global.

3.2.3.2. Métodos determinísticos

Al contrario que los métodos estocásticos, los métodos determinísticos no toman decisiones aleatorias en sus algoritmos y sus demostraciones de convergencia no dependen de la probabilidad. Es decir, son métodos que alcanzan siempre la misma solución para una entrada particular.

Algunos ofrecen un tiempo de terminación finito para problemas específicos y otros convergen cuando el número de iteraciones se aproxima a infinito. Dentro de los algoritmos determinísticos se incluyen los métodos de Ramificación y Acotación, que se caracterizan por realizar un búsqueda exhaustiva del espacio de búsqueda y de encontrar todas las posibles soluciones con la misma precisión.

3.3. Algoritmos de Ramificación y Acotación

Los primeros algoritmos de Ramificación y Acotación (Branch and Bound, B&B) aparecieron en artículos de los cincuenta y principios de los sesenta, donde los investigadores describieron esquemas enumerativos para resolver, lo que posteriormente se denominaron, problemas NP-Completos. Debido a la generalidad del método y la efectividad que aportaba, fue ampliamente usado. De hecho, todavía es una de las mejores herramientas para resolver problemas difíciles. Los primeros que dieron el nombre de Ramificación y Acotación a este método, fueron Little, Murty, Sweeny y Karel [45], en 1963, en su

artículo sobre el problema del viajante de comercio. Lawer y Wood [44] estudiaron los algoritmos de Ramificación y Acotación, obteniendo una descripción independiente del problema, siendo así el primer artículo que presenta un modelo general de Ramificación y Acotación.

Los métodos de Ramificación y Acotación son algoritmos basados en árboles de búsqueda. Su principio radica en una descomposición sucesiva del problema original en subproblemas más pequeños y disjuntos, hasta encontrar la solución óptima. Un árbol de búsqueda $T = (V, \beta)$, describe estos subproblemas (el conjunto de vértices V, cuya raíz es el problema completo S) y el proceso de descomposición (el conjunto de arcos β). El algoritmo consiste en una búsqueda heurística iterativa en T, que evita visitar subproblemas que no contienen una solución óptima. Los algoritmos de Backtracking, Programación Dinámica y las búsquedas en árboles A^* y AND-OR pueden verse como variaciones de algoritmos de Ramificación y Acotación [22, 42, 48, 49].

Una descripción de los modelos de algoritmos de Ramificación y Acotación que aparecen en los artículos [35, 36, 37, 43, 44, 47, 49], hecha por Bruin, Kindervater y Trienekens puede encontrarse en [13]. Estos autores no discuten modelos más recientes debido a que sólo se diferencian de los anteriores en pequeñas variaciones. Esta revisión de los algoritmos de Ramificación y Acotación está motivada para establecer un esquema común para todas las aplicaciones, mediante la descripción de unas reglas u operadores generales [35, 47]. Para especificar estas reglas nos basaremos en los formalismos utilizados por Corrêa y Ferreira [11], centrándonos en problemas donde se intenta encontrar el valor mínimo de una función objetivo.

3.3.1. Reglas básicas de los algoritmos de Ramificación y Acotación

Según Mitten e Ibaraki [35, 47] los algoritmos de Ramificación y Acotación pueden ser caracterizados por cuatro reglas:

- *Regla de División*: Establece cómo se descomponen los nodos del árbol de búsqueda.
- *Regla de Acotación*: Calcula un *límite inferior* de la solución óptima de un nodo.
- *Regla de Selección*: Define qué nodo será el siguiente a dividir.
- *Regla de Eliminación*: Establece cómo reconocer y eliminar nodos donde no se encuentra la mejor solución del problema original.

Dependiendo del tipo del problema, se añade la siguiente regla:

- *Regla de Terminación*: Determina cuándo un nodo pertenece a la solución final. Esta regla aparece, por ejemplo, en problemas donde la solución del problema está determinada por la precisión deseada. Por otro lado, en algunos problemas tradicionales de Optimización Combinatoria, el criterio de terminación es inherente a obtener una solución factible. Por ejemplo, en el problema del Viajante de Comercio [14, 45, 56, 62], donde hay que encontrar el camino mínimo para visitar todas las ciudades deseadas.

Llamaremos a estas reglas las *reglas básicas* de los algoritmos de Ramificación y Acotación.

Una buena comprensión de la estructura del problema a resolver, es decir, hacer una buena elección de las reglas básicas, da lugar a una reducción del tiempo de ejecución y a poder resolver problemas más complejos en cuanto a su tamaño.

Los algoritmos de Ramificación y Acotación consisten en una secuencia de iteraciones en las que se aplican las reglas básicas a una estructura de datos D, que puede verse como una lista ordenada de subproblemas [9] y a los subproblemas que se extraen de esta estructura. Estas reglas seleccionan un subproblema de D, lo descomponen y eventualmente insertan los subproblemas creados en la estructura de datos D. A cada subproblema se le asocia una solución factible, resolviendo el subproblema si es suficientemente simple o asignándole una solución, elegida entre las posibles dentro del subproblema (no necesariamente la mejor). La mejor solución factible encontrada durante la ejecución del algoritmo de Ramificación y Acotación, será un límite superior de la solución final y se utilizará para eliminar aquellos subproblemas generados que no pueden contener una solución factible mejor que la que ya se ha encontrado [35, 37, 47, 61].

A continuación se describen las reglas de selección heurísticas utilizadas:

Búsqueda en Profundidad (Depth-First): La regla de selección selecciona, de entre aquellos subproblemas que estén a más profundidad en el árbol de búsqueda, el que tenga un menor valor de una función de prioridad heurística h. En este método se usa una estructura de almacenamiento LIFO (Last-In-First-Out) o pila. El árbol asociado a este tipo de selección es aquel en el que uno de los nodos hijo se expande hasta obtener una solución factible del problema o un subproblema no factible. La complejidad espacial de la pila, $O(lw)$, se incrementa linealmente con el nivel del árbol búsqueda alcanzado, l, y el factor de división w [46]. Al descomponer el subproblema elegido, los subproblemas generados a partir de él se introducen en la estructura LIFO, siguiendo un orden decreciente de h.

Desventajas:

- El número de subproblemas inspeccionados es normalmente mayor que el realizado en otros tipos de selecciones, como la de *Primero el Mejor*.
- Si se entra en una rama del árbol de búsqueda que no lleva a la solución óptima, se necesita mucho tiempo para salir de esa ramificación.

Ventajas:

- Es el método de selección más económico desde el punto de vista del espacio de memoria requerido.
- Conduce más rápidamente a la obtención de una mejor cota superior del valor del mínimo global.

Búsqueda en Anchura (Breadth-First): En la búsqueda en anchura, al contrario que en la búsqueda en profundidad, la regla de selección elige, de entre aquellos subproblemas que estén a menor profundidad en el árbol de búsqueda, el que tenga

un menor valor de una función de prioridad heurística h. Este método usa una estructura de almacenamiento FIFO (First-In-First-Out) o cola.

Desventajas:

- Este tipo de estrategia no es recomendable ni desde el punto de vista del tiempo de computación ni desde el del ahorro de memoria.

Ventajas:

- Admite la aplicación de un test de dominancia entre los subproblemas que se encuentran a la misma profundidad del árbol de búsqueda, para algunos tipos de problemas.

Primero el Mejor (Best-First): La regla de selección elige aquel subproblema, con un menor valor de una función de prioridad heurística, independientemente del nivel de profundidad del árbol de búsqueda en el que se encuentre. La estructura más simple de almacenamiento es una lista ordenada por el valor de $li(v),\ v \in D$.

Desventajas:

- La mejor cota superior del valor mínimo global, suele obtenerse en las fases finales del algoritmo.
- Crecimiento exponencial en el espacio de memoria necesario cuando se incrementa la profundidad del árbol de búsqueda. La complejidad espacial es $O(w^l)$.

Ventajas:

- Difícilmente un subproblema es descompuesto innecesariamente. En [21] se demuestra que usando *Best-First* se exploran menos nodos que usando otra estrategia, cuando hay que encontrar todas las soluciones óptimas, no siendo cierto cuando sólo se requiere encontrar una.

Además de los tipos de selección descritos, existen los llamados *heurísticos* que son análogos al *Primero el Mejor*, donde la función heurística puede ser diferente de la cota inferior de la función para un subproblema [38].

Después de la selección de un subproblema, la *Regla de División* crea un conjunto de nuevos subproblemas a partir de v.

Definición 1 *La Regla de División de un nodo, en un algoritmo de Ramificación y Acotación corresponde a:*

- *Una partición del nodo, si v no es un nodo solución, es decir, no es lo suficientemente pequeño para ser resuelto.*
- *La solución del problema asociado al nodo, en otro caso.*

Sea $f^*(v_i)$ el valor óptimo de la solución del nodo v_i, que será descompuesto en los nodos $v_{i1}, v_{i2}, \ldots, v_{im}$ por la regla de división. Entonces, se tiene que:

$$f^*(v_i) = \min_{k=1,\ldots,m} f^*(v_{ik})$$

Es decir, el valor óptimo de la solución de un nodo puede obtenerse mediante la evaluación de sus nodos hijos. En la práctica, las reglas de división cumplen que cada solución factible de un nodo padre es también solución de al menos uno de sus hijos. Estas reglas de división dependerán del problema concreto en el que se aplique el algoritmo de Ramificación y Acotación.

En cada iteración se puede generar un nuevo límite superior de la mejor solución, $\overline{f^*}$, de dos formas: porque el nodo generado por descomposición es un nodo solución, o porque se encuentra una solución factible mejor durante la computación de una partición en una descomposición.

La *Regla de Eliminación* permite hacer una búsqueda inteligente en S que evite considerar subproblemas que se sabe que no conducirán a una solución óptima del problema original.

Definición 2 *La Regla de Eliminación, en cada iteración, elimina todos los subproblemas activos y solución, cuya cota inferior sea mayor que el límite superior que tiene de la solución [10, 35].*

Como puede observarse no existe una regla más importante que las demás, ya que todas ellas cooperan conjuntamente para la resolución del problema. El objetivo principal es reducir el árbol de búsqueda para obtener rápidamente la mejor solución. Aunque la *Regla de Eliminación* es la que realiza la poda del árbol en última instancia, será aplicada en mayor o menor medida, dependiendo de las demás reglas. Una *Regla de Acotación* que obtenga un buen límite inferior de la posible solución de un subproblema, permitirá caracterizar mejor a los subproblemas que no contienen la mejor solución, para eliminarlos. Para poder eliminar subproblemas, también hay que encontrar un buen límite superior de la solución. Este límite superior se consigue inspeccionando primero los nodos más prometedores, ya que son los que pueden aportar mejores soluciones. Este orden se establece mediante la *Regla de Selección*, de la que también dependen los requerimientos de memoria del problema. Hay que distinguir entre el número total de subproblemas inspeccionados y el número máximo de subproblemas almacenados en la estructura de datos con los nodos pendientes de evaluar. La gestión de esta estructura de datos ordenada será más costosa cuanto mayor sea el número de subproblemas que contenga. La *Regla de División* también juega un papel importante ya que de ella dependerá el número de ramas generadas a partir de un nodo del árbol de búsqueda. Por un lado, generar muchas ramas permite obtener información más precisa debido a que los subproblemas generados son menores en tamaño, pero por otro lado, hay que inspeccionar más nodos del árbol en cada división. No sólo es importante el nivel de división realizado, sino también cómo se realiza la división. Por lo tanto, la *Regla de División* debe estar orientada a reducir el espacio de búsqueda pero sin generar un número excesivo de nodos en cada división.

3.3.2. Heurística en los algoritmos de Ramificación y Acotación

La naturaleza heurística de los algoritmos de Ramificación y Acotación aparece cuando en la selección del siguiente subproblema a evaluar hay que decidir, entre varios con la misma prioridad, ¿cuál tiene mayor probabilidad de ser camino mínimo? Debido a la naturaleza de los problemas es difícil predecir que dirección de la búsqueda se debe escoger para reducir el trabajo a realizar.

Capítulo 4

Algoritmos paralelos

En este capítulo se realizara una introducción a las arquitecturas paralelas y se describirán algunas de las características más destacables de los distintos modelos, exponiendo las medidas de rendimiento que normalmente se usan para evaluar la eficiencia de los algoritmos paralelos.

Se realizara una introducción de la metodología a seguir para paralelizar algoritmos secuenciales, donde uno de los problemas que pueden aparecer es el desbalanceo de la carga computacional asociada a cada uno de los procesadores. En algunos casos, estos problemas deben ser tratados mediante técnicas de balanceo dinámico de la carga.

Finalmente se tendrán en cuenta las posibles decisiones a tomar para la implementación de algoritmos paralelos de Ramificación y Acotación y las anomalías que se pueden encontrar.

4.1. Introducción

El rendimiento de los computadores convencionales se ha incrementado considerablemente en muy poco tiempo, aumentando el número de procesadores. Con estas mejoras, el número de problemas resueltos mediante un computador convencional es mayor, pero aun existe un gran número de problemas computacionalmente complejos que no pueden ser resueltos mediante computadores tradicionales, o cuyo tiempo de resolución seria tan alto que no se obtendría la respuesta en un tiempo razonable. Un ejemplo de este tipo de problemas lo forman los algoritmos de Optimización Global basados en Ramificación y Acotación.

Para intentar resolver un mayor número de problemas complejos existen dos tipos de computadoras de alto rendimiento, los computadores vectoriales y los paralelos.

En los computadores vectoriales se obtiene una ganancia en velocidad mediante el uso de multitud de unidades funcionales, de ejecución y caminos de datos. El objetivo de estos computadores es desglosar las operaciones aritméticas, la obtención de datos, la comparación, el desplazamiento, y ejecutar de forma paralela distintas operaciones en diferentes componentes. Por ejemplo, la suma de los componentes de dos vectores. Usando computadores vectoriales se pueden obtener ganancias en velocidad considerables

para muchos problemas, pero en general, este tipo de computadora no es adecuada para su uso en la resolución de problemas de Optimización Global ya que, generalmente las funciones a evaluar no son lineales.

Por otra parte, los computadores paralelos son sistemas que integran varias unidades de procesamiento que puede trabajar concurrentemente sobre distintos conjuntos de datos, además el sistema posee una memoria que es compartida entre todos los procesadores. A partir de aquí se hablara únicamente de computadores paralelos y su uso eficiente en la resolución de problemas computacionalmente costosos.

Esta sección describe algunas de las nociones fundamentales sobre las arquitecturas paralelas, como evaluar el rendimiento de los algoritmos paralelos y los modelos de paralelización de algoritmos.

4.2. Clasificación de las Arquitecturas Paralelas

Un computador paralelo está compuesto por un número de procesares, p, que colaboran en la resolución de un problema. El usuario espera que el tiempo de ejecución obtenido en un computador paralelo con p procesadores sea $1/p$ veces el tiempo de ejecución usando un único procesador. Esta aceleración o ganancia de velocidad no se alcanza en todos los casos, ya que la administración de los procesadores, así como la comunicación entre ellos, restan tiempo. Las medidas de rendimiento de los algoritmos paralelos serán tratadas en el punto 4.3.

Existen diversos esquemas de clasificación de los sistemas de computación basados en sus estructura o comportamiento. Los principales métodos de clasificación consideran el número de conjuntos de instrucciones y/o operadores que pueden ser procesados simultáneamente, la organización interna de los procesadores, la estructura de la conexión interprocesador, o los métodos empleados para calcular los flujos de instrucciones y datos dentro del sistema. A continuación se muestran algunas de estas clasificaciones, de acuerdo a diferentes criterios:

4.2.1. Secuencia de instrucciones y datos

En 1966, Michael J. Flynn [18] propuso uno de los sistemas más sencillos para clasificar ordenadores en función del análisis de los flujos de control y de datos. Estos flujos pueden ser simples o múltiples. La clasificación propuesta por Flynn se basa en el número de instrucciones concurrentes y en los flujos de datos disponibles en la arquitectura. Según está clasificación se diferencian las siguientes clases de computadores:

Simple Instrucción, Simple Dato (SISD): Los computadores de este tipo tiene una secuencia simple de instrucciones que operan sobre una secuencia simple de datos. Son computadores secuenciales que no explotan el paralelismo en las instrucciones ni en flujos de datos (siguiendo el modelo de máquina secuencial propuesto por Von Neumann). Solo se dispone de una CPU y los algoritmos usados este tipo de computador son algoritmos secuenciales, aunque puede existir superposición de

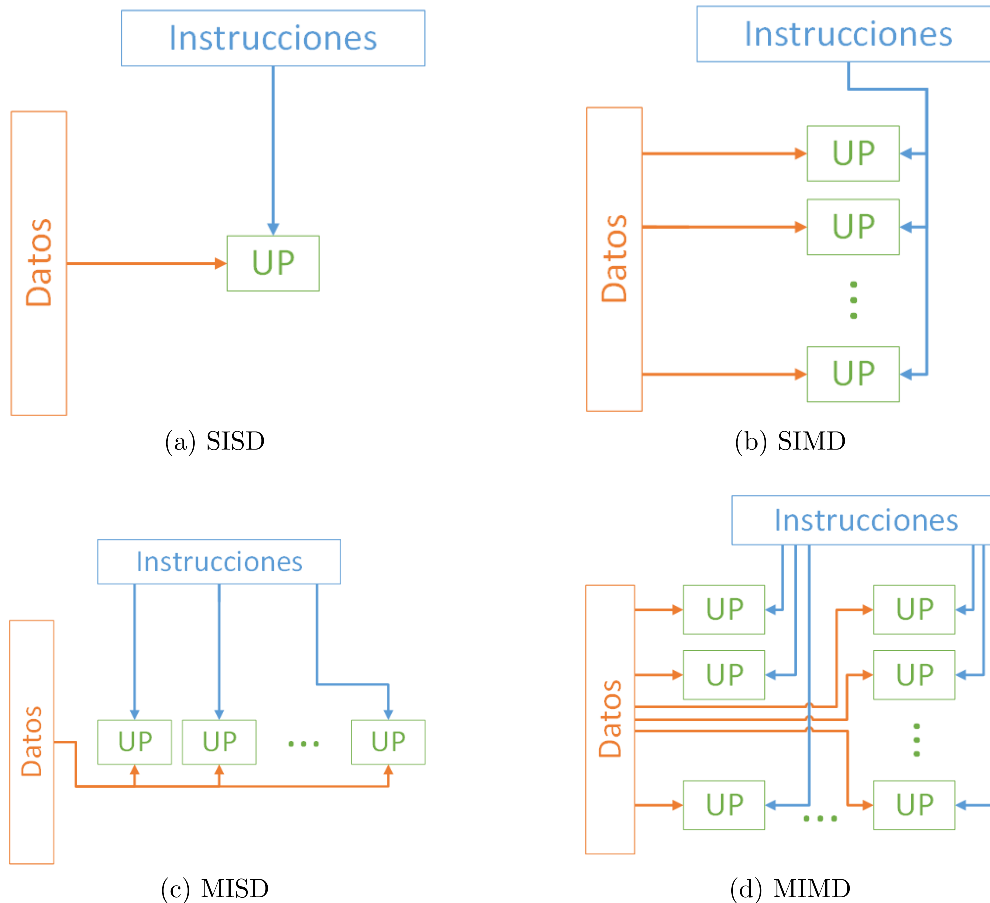


Figura 4.1: Taxonomía de Flynn.

varias instrucciones sí estas se encontrasen en diferentes etapas de la ejecución. (Figura 4.1a).

Simple Instrucción, Múltiples Datos (SIMD): Todos los procesadores del computador paralelo ejecutan las mismas instrucciones sobre diferentes datos de entrada (Figura 4.1b). Es poco común debido al hecho de que la efectividad de los múltiples flujos de instrucciones suele precisar de múltiples flujos de datos. Sin embargo, este tipo se usa en situaciones de paralelismo redundante, donde se necesitan varios sistemas de respaldo en caso de que uno falle.

Múltiples Instrucciones, Simple Dato (MISD): Este modelo está basado en la ejecución de diferentes instrucciones sobre el mismo conjunto de datos. Es decir, los resultados de un procesador se convierten en los operandos del siguiente (Figura 4.1c). Este es el caso de los computadores vectoriales, donde se emplea un único flujo de instrucciones para procesar varios flujos de datos.

Múltiples Instrucciones, Múltiples Datos (MIMD): Cada uno de los procesadores

del computador paralelo ejecuta su propio programa y por lo tanto pueden operar, independientemente unos de otros, sobre datos diferentes. Es decir, los procesadores ejecutan de forma simultanea y de forma autónoma diferentes instrucciones sobre distintos datos. Un computador MIMD también puede funcionar como cualquiera de las maquinas de las otras categorías (Figura 4.1d).

Los sistemas distribuidos suelen clasificarse como arquitecturas MIMD, bien sea explotando un único espacio compartido de memoria, o uno distribuido. Esta categoría cubre a los multiprocesadores y a los multicomputadores.

4.2.2. Modelos de memoria

Los computadores paralelos se pueden clasificar dependiendo de como esté estructurada la memoria:

Memoria compartida: Los procesadores del computador paralelo comparten una memoria central de almacenamiento que es accesible por todos los procesadores mediante una red de interconexión. Podrían aparecer conflictos de acceso a memoria con un alto número de procesadores. Tanto la memoria como el bus tienen un ancho fijo que debe ser repartido entre los distintos procesadores, lo que limita la escalabilidad, además de disminuir la velocidad de acceso a memoria. Para reducir el acceso a memoria principal se introduce una nueva capa de memoria, la memoria cache que es local para cada procesador.

Un factor importante que se debe mantener en los computadores paralelos de memoria compartida es la coherencia de cache. Para permitir mayor paralelismo en el acceso a memoria principal, la memoria se constituye en bloques que son asignados de forma exclusiva a cada uno de los distintos procesadores. De esta forma son posibles accesos procesador↔memoria en paralelo.

Memoria distribuida: Cada procesador posee su propia memoria local, inaccesible para el resto de procesadores de forma directa. Las principales ventajas de este modelo con respecto al modelo de memoria compartida son: (i) Cada procesador puede usar todo el ancho de banda para el acceso a su memoria local. (ii) Es mas escalable, el tamaño del sistema solo está limitado por el tipo de la red de interconexión utilizada. (iii) No hay problemas de coherencia de cache.

La programación de este tipo de sistemas es más costosa, ya que el intercambio de datos entre procesadores se hace mediante el paso de mensajes, a través de la red de interconexión. El paso de mensajes introduce además dos tipos de sobrecarga: el producido por el tiempo necesario para crear y enviar el mensaje y el producido en el procesador receptor que deberá interrumpir su ejecución para tratar el mensaje.

En este modelo se debe tener en cuenta donde se almacenan los datos, esto introduce el concepto de localidad espacial de los datos al diseño de algoritmos paralelos.

Un computador paralelo de memoria distribuida puede estar compuesto por procesadores heterogéneos, que se diferencien en sus velocidades de procesamiento, y deben ser programados mediante librerías de paso de mensajes.

Memoria Compartida-Distribuida: Es este modelo la memoria está distribuida físicamente en cada uno de los procesadores, pero existe un soporte hardware para permitir un espacio de direcciones globales, e incluso para la gestión de la coherencia entre las caches locales de cada procesador. Este modelo combina la sencillez en la programación del modelo de memoria compartida, con la escalabilidad que proporciona el modelo de memoria distribuida. Los computadores de este tipo de memoria se agrupan en tres bloques:

- **Computadores UMA** (*Uniform Memory Acces, Acceso a memoria uniforme*): En esta categoría, el acceso a memoria compartida se implementa usando una red de interconexión entre los procesadores y la memoria. Todos los procesadores del sistema tienen acceso a la misma memoria.
- **Computadores NUMA** (*Non Uniform Memory Access, Acceso a memoria no uniforme*): Cada procesador tiene su propia memoria local pero también tiene acceso a la memoria de otros procesadores. El nombre de "no uniforme"viene de que un procesador puede acceder más rápido a su propia memoria local que la memoria no local (memoria que está en otro procesador o compartida entre procesadores), por lo tanto el tiempo de acceso no es uniforme entre memorias.
- **Computadores COMA** (*Cache Only Memory Acces, Acceso a memoria solo en cache*): La memoria local de cada procesador es usada como memoria cache, mientras que la memoria compartida se emplea como memoria principal.

4.2.3. Topologías de las redes de interconexión

Uno de los factores más importante en el diseño de sistemas paralelos es el conjunto de enlaces que utilizaran los procesadores, las memorias y el resto de dispositivos para comunicarse. Estos enlaces definen la red de interconexión de la cual dependerá el coste del intercambio de datos entre los distintos procesadores. Tanto a nivel físico como lógico, el aumento del número de procesadores en el sistema deberá ir acompañado de un aumento de las comunicaciones entre ellos, con el fin de permitir más transmisiones de datos. Normalmente es necesario que los algoritmos paralelos se adapten al tipo de topología sobre la que se realizara la ejecución.

Al tener sistema paralelos con diversos procesadores surge la necesidad de comunicar información entre ellos para resolver un problema común. Estas comunicaciones pueden realizarse estableciendo datos comunes en memoria compartida o mediante el paso de mensajes. En ambos casos, será necesario la existencia de una red de interconexión que permita realizar las operaciones descritas. En el primer caso, ya que la memoria está físicamente distribuida en varios bancos, la red de interconexión posibilitará la conexión de

cada procesador con todos los bancos de memoria. En el segundo caso, es necesario unir físicamente los procesadores para poder realizar el envío de mensajes.

Matemáticamente una red de interconexión puede verse como un grafo dirigido donde los elementos a comunicar están localizados en los vértices y se comunican a través de las aristas.

Para tener una medida de la efectividad de las redes en la implementación de algoritmos paralelos eficientes sobre el hardware real se han introducido los siguientes conceptos:

Grado: Número de aristas por vértice. Dos vértices son vecinos si existe una arista que los una. EL grado de un vértice está definido por el número de vecinos que tiene. Para obtener un sistema escalable, es mejor si el número de aristas por vértice es una constante independiente del número de vértices.

Diámetro: El diámetro de una red es la distancia más larga entre dos vértices cualesquiera. Los diámetros bajos son mejores, pues la complejidad de los algoritmos paralelos que requieren comunicaciones entre pares de vértices arbitrarios es menor.

Ancho: Es el mínimo número de aristas que deben ser eliminadas con el fin de dividir la red en dos mitades. Son mejores las redes con un ancho grande, porque permiten una mayor variedad en los caminos de enrutado.

A continuación se van a describir las topologías más usadas como redes de interconexión. Se va a distinguir entre redes estáticas y dinámicas.

4.2.3.1. Redes estáticas

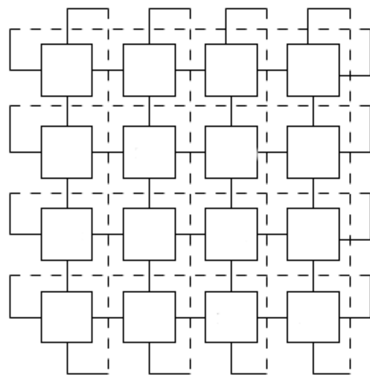
Algunas de las topologías estáticas más utilizadas son las mostradas en la figura 4.2, que serán descritas a continuación:

Malla y toro: En una malla rectangular de n x k procesadores los nodos están organizados en un plano. Cada nodo (i, j) se comunica con los nodos $(i \pm 1, j)$ y $(i, j \pm 1)$. La malla no es un grafo regular debido a los bordes. El grado de los vértices internos es $2n$, mientras que en los vértices exteriores el grado es de n, el diámetro es $n(k-1)$ y el ancho es n.

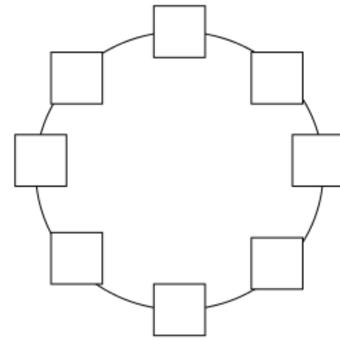
Una de las características de la malla es que es infinitamente extensible, lo que la hace la más modular de las topologías.

El toro de base k y de dimensión n puede considerarse una malla en la que se han cerrado los bordes sobre sí mismos. Contiene $k * n$ vértices, cada uno de ellos conectado a un k-ciclo en cada una de sus n dimensiones. Es un grafo regular de grado $2n$ y de diámetro $n(k/2)$.

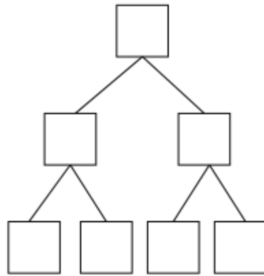
Anillo: En un anillo, cada procesador (i) tiene como vecinos a $((i+1) \mod n)$ y a $((i-1) \mod n)$. Tiene un grado de 2, un diámetro de $n/2$ y un ancho de 2.



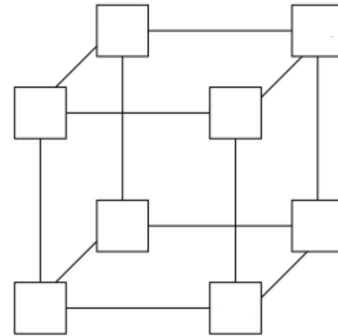
(a) Toro (Malla)



(b) Anillo



(c) Árbol binario



(d) Hipercubo 3-dimensional

Figura 4.2: Topologías de redes estándar en sistemas paralelos.

Árbol binario: La topología de árbol binario de nivel k utiliza $2^{k+1} - 1$ procesadores formando un árbol binario. Cada vértice interior puede comunicarse con sus dos hijos y con su padre (salvo el vértice colocado en la raíz). Su diámetro es $2(k - 1)$ y su ancho es igual a 1.

Hipercubo: Esta topología consta de 2^k procesadores formando un hipercubo de dimensión k. Si se etiquetan los vértices con los índices $0, 1, 2, ..., 2^k - 1$, dos vértices serán adyacentes si sus etiquetas difieren solamente en un bit. El diámetro de un hipercubo de dimensión k es k y el ancho seria 2^{k-1}. Este tipo de configuración tiene un diámetro bajo y un ancho grande. Sus buenos resultados quedan desvirtuados debido a su grado no constante, lo que limita la escalabilidad.

4.2.3.2. Redes dinámicas

Las redes dinámicas son redes que pueden cambiar la topología de comunicación en tiempo de ejecución. Las principales topologías de redes dinámicas son las siguientes:

Buses: Un bus es un conjunto de líneas que permite comunicar de forma selectiva un cierto número de componentes o dispositivos de acuerdo a ciertas normas de conexión. En este caso, los componentes a conectar serán procesadores, memoria, etc.

El inconveniente del bus es que solo permite una conexión al mismo tiempo, por ello si existen varias peticiones de comunicación el *árbitro del bus* deberá establecer un orden entre ellas. Los buses son una forma barata de comunicación que tiene como ventaja la sencillez en la reconfiguración. Sus inconvenientes son el bajo ancho de banca y la latencia debido a las esperas debidas a varias peticiones simultaneas.

Redes de líneas cruzadas: En esta red, cada nodo está conectado con todos los demás a través de un conmutador de líneas cruzadas tal como muestra la figura 4.3. Cada conmutador puede proporcionar una conexión dedicada entre cada para de elementos que se deseen conectar. La posición de cada conmutador se cambia dinámicamente según las necesidades del programa.

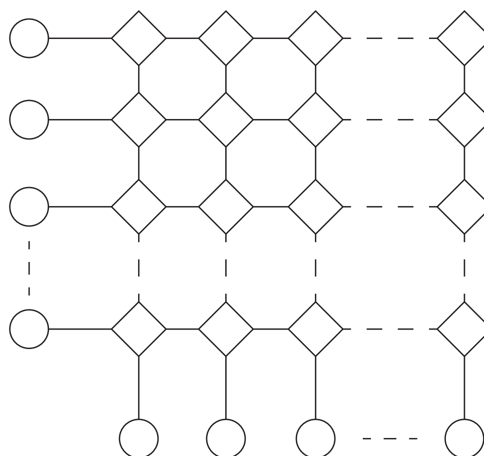


Figura 4.3: Red de lineas cruzadas.

Las principales ventajas de este tipo de redes es que son no bloqueantes y son fácilmente escalables. Su principal inconveniente es la necesidad de emplear una gran cantidad de conmutadores, ya que son necesarios n^2 conmutadores para una red de n x n.

Redes multietapa: Una red de este tipo está formada por una serie de capas de módulos conmutadores n x k. Estos conmutadores pueden cambiar dinámicamente de posición para establecer las conexiones deseadas en cada momento. La figura 4.4 muestra un esquema genérico de este tipo de redes

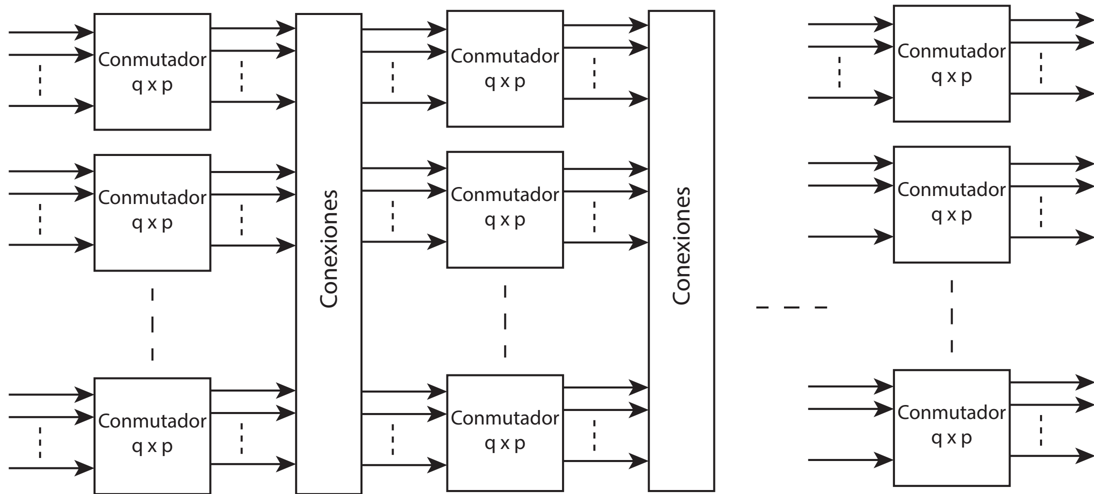


Figura 4.4: Red genérica multietapa.

La ventaja de la red multietapa frente a una red de lineas cruzadas es el menor número de conmutadores. Existen diferentes clases de redes multietapa en función del tipo de modulo conmutador que emplean y la conexión entre los conmutadores:

- Redes Omega
- Redes de linea base
- Redes mariposa
- Redes Delta
- Redes de Closs
- Redes de Benes

4.3. Medidas de rendimiento para algoritmos paralelos

El uso de un computador paralelo para la resolución de un problema numérico solo tiene sentido si el tiempo total de resolución del problema en el computador paralelo se reduce significativamente, en comparación con el tiempo requerido para su resolución en un computador secuencial. Sería deseable que el tiempo secuencial del algoritmo se reduzca alrededor de un factor p, si usan p procesadores para resolverlo en paralelo. Una medida para obtener la aceleración producida por el algoritmo paralelo, frente al secuencial, se denota por la ganancia de velocidad o speedup, que se define como:

Definición 3 *Se define la ganancia de velocidad o speedup, $S(p)$ como la relación entre el tiempo total de la resolución del algoritmo en un computador secuencial, t_{sec}, y el tiempo total requerido para resolverlo en un computador paralelo con p procesadores, $t_{par}(p)$:*

$$S(p) = \frac{t_{sec}}{t_{par}(p)} \tag{4.1}$$

Para el cálculo de la ganancia en velocidad debe usarse el mejor algoritmo serie conocido, por lo tanto, aquel que necesita un menor t_{sec}. Frecuentemente, el mejor algoritmo serie no es único, o no ha sido implementado, por lo que su medida es imposible. Una técnica de uso común es la de usar, como algoritmo serie, el algoritmo paralelo sobre un solo procesador.

Si se calcula la relación entre la ganancia en velocidad obtenida y el número de procesadores, se obtiene la eficiencia del algoritmo:

Definición 4 *La eficiencia, $E(p)$, representa la ganancia de velocidad por procesador:*

$$E(p) = \frac{S(p)}{p} \tag{4.2}$$

Generalmente se cumple $1 \leq S(p) \leq p$. En algunos casos se consigue una ganancia de velocidad superlineal con $S(p) > p$. Para el problema de Ramificación y Acotación que se trata en este trabajo, esto no parece ser posible, pues el algoritmo paralelo evalúa la misma cantidad de símplices en paralelo que los que evalúa la versión secuencial.

La caracterización exacta de un algoritmo paralelo eficiente no es sencilla. Son muchos los factores que pueden influir y limitar el speedup. Una parámetro importante es el tamaño del problema de entrada, ya que si no hay bastante carga computacional para el número de procesadores disponibles, el speedup sera bajo.

4.3.1. Ley de Amdahl

Otro factor decisivo es la cantidad de código secuencial o carga fija, que se describe en la Ley de Amdahl [2]:

Definición 5 *La Ley de Amdahl establece que la parte de código intrínsecamente secuencial limita la ganancia en velocidad de un algoritmo paralelo. Sea s la fracción de operaciones de un algoritmo paralelo que deben realizarse de forma secuencial ($0 \leq s \leq 1$). La ley de Amdhal establece que la máxima ganancia en velocidad $S_m(p)$ que se puede conseguir mediante un modelo paralelo de p procesadores ejecutando dicho algoritmo es:*

$$S_m(p) \leq \frac{1}{s + \frac{1-s}{p}} \tag{4.3}$$

Es decir, para un problema de tamaño fijo, cada algoritmo paralelo es ineficiente asintóticamente con el número de procesadores.

4.3.2. Ley de Gustafson

Uno de los mayores inconvenientes de aplicar la Ley de Amdahl es que el problema (la carga de trabajo) no puede aumentarse para corresponderse con el aumento de la capacidad computacional al aumentar el tamaño de la máquina. Es decir, el tamaño fijo impide la escalabilidad en el rendimiento.

La Ley de Gustafson [29] está muy ligada a la Ley de Amdahl, que pone limite a la mejora del speedup que se puede obtener con la paralelización, debido al tamaño fijo del problema. Gustafson ofrece un nuevo punto de vista y una visión positiva del procesamiento paralelo:

Definición 6 *La Ley de Gustafson establece que cualquier problema suficientemente grande puede ser paralelizado de forma eficiente. Sea α la porción no paralelizable del algoritmo. La ley de Gustafson establece que la ganancia de velocidad $S(p)$ que se obtiene para un problema escalable con p procesadores es:*

$$S(p) = p - \alpha(p - 1) \tag{4.4}$$

La Ley de Gustafson aborda las limitaciones de la Ley de Amdahl, la cual no contempla el aumento de la capacidad de cómputo a medida que el número de máquinas aumenta. Propone que los programadores establezcan el tamaño de los problemas para utilizar el equipamiento disponible en su solución. Por consiguiente, si existe equipamiento más rápido disponible, mayores problemas se pondrán resolver en el mismo tiempo.

Otras causas del aumento del tiempo de ejecución provocado por el código secuencial es la debida al retardo que se produce en la activación de los demás procesadores cuando acaba el código secuencial, el acceso a memoria compartida o la sincronización de los procesadores.

4.4. Paralelización de algoritmos secuenciales

Cuando se aborda la paralelización de un algoritmo secuencial, en algunas ocasiones, el algoritmo paralelo puede ser completamente diferente a la versión secuencial. Una posible metodología a seguir para abordar este problema, la implementación de un algoritmo paralelo, se divide en cuatro etapas [19]:

Partición: Se dividen los datos necesarios y las operaciones en las que se usaran estos datos en tareas. Se pueden seguir dos estrategias:

- **Descomposición de datos** Primero se descomponen los datos y después se les asocian las operaciones.
- **Descomposición funcional** Primero se descomponen las operaciones en distintas tareas y se les asocian a cada tarea los datos asociados.

El objetivo de esta etapa es el de reconocer las partes paralelizables en el algoritmo.

Comunicación: En esta etapa se determinan las comunicaciones que serán necesarias para sincronizar la ejecución de las distintas tareas, escogiendo las estructuras y los algoritmos de comunicación que más se adecuen a cada caso. Los distintos tipos de comunicaciones pueden ser:

Locales: En las comunicaciones locales, cada tarea se comunica con un reducido número de otras tareas.

Globales: En las comunicaciones globales, cada tarea se comunica con la gran mayoría, o con todas, las tareas.

Estructurada: En las comunicaciones estructuradas, una tarea y su vecina forman una estructura regular, como un árbol o una malla.

No estructurada: En las comunicaciones no estructuradas, la representación de las comunicaciones puede ser un grafo cualquiera.

Estáticas: En las comunicaciones estáticas, la tareas que intervienen en la computación no cambian durante la ejecución.

Dinámicas: En las comunicaciones dinámicas, las tareas se comunican entre si mediante estructuras dinámicas que se determinan, mediante los datos computados, en tiempo de ejecución.

Síncronas: En las comunicaciones síncronas, las tareas se ejecutan de forma coordinada, esperándose las unas a las otras para intercambiar información.

Asíncronas: En las comunicaciones asíncronas, no existe coordinación entre tareas y son las propias tareas las que se solicitan información entre si.

Aglomeración: Las tareas y las estructuras de comunicación definidas en las primeras etapas se evalúan respecto a los requisitos de funcionamiento y a los costes de la implementación para una arquitectura particular. Si fuese necesario las tareas podrían combinarse en tareas mayores para mejorar el rendimiento o reducir los costes de desarrollo. El número de tareas obtenido en esta fase debe ser suficiente para el número de procesadores a utilizar.

Proyección: Cada tarea se asigna a un procesador intentando maximizar el uso del procesador y minimizar los costes de comunicación. La proyección puede especificarse estáticamente o determinarse de forma dinámica en tiempo de ejecución, mediante estrategias de balanceo dinámico de la carga. En sistemas de memoria distribuida que emplean paso de mensajes, las tareas con gran cantidad de comunicación se sitúan en procesadores cercanos, o incluso en el mismo procesador. Mientras, las tareas que usan pocas comunicaciones podrían situarse en distintos procesadores, para poder explotar completamente su paralelismo. Ambas estrategias, en ocasiones entran en conflicto.

En la implementación de un algoritmo paralelo a partir de uno serie se pueden usar dos alternativas:

Paralelismo implícito: En este tipo de paralelismo es el compilador, no el programador, el que realiza la paralelización. De esta forma un programa secuencial puede ser paralelizado sin ser casi modificado.

Desafortunadamente la paralelización automática de algoritmos secuenciales no suele resultar en algoritmos paralelos eficientes, porque el compilador debe analizar las dependencias entre las distintas partes del algoritmo secuencial para asegurar un funcionamiento correcto del algoritmo paralelo. El compilador divide el algoritmo secuencial y analiza las dependencias entre las distintas partes. Pero el análisis de las dependencias es complejo debido a los bucles, condiciones o llamadas a subrutinas. Por estos motivos la intervención del programador suele ser necesaria.

Paralelismo explícito: En este modelo, el programado debe determinar las opciones concretas para cada una de las etapas del diseño de algoritmos paralelos. Un escenario común es el del sistema distribuido, en el que el sistema esta compuesto por nodos interconectados. En este tipo de sistema, las comunicaciones entre nodos se suelen realizar mediante paso de mensajes. Por otro lado, los algoritmos se caracterizan por usar una descomposición de datos con, posibles, intercambios de datos entre los procesadores. Además se pueden incluir fases para el balanceo de la carga.

4.5. Balanceo de la carga

El objetivo del un programa paralelo es realizar una buena división del problema para poder repartir la carga computacional entre los distintos procesadores del sistema y reducir la comunicación entre ellos. Las técnicas de balanceo de carga para algoritmos paralelos están basadas en la descomposición de datos, esta descomposición se puede realizar de dos formas distintas:

Balanceo estático: La carga computacional se conoce antes de iniciar la ejecución del programa que resuelve cierto problema, y puede ser dividida en p subproblemas de, aproximadamente, la misma carga computacional. Cada uno de estos subproblemas se resolverá entonces en un procesador diferente. Se intentara también evitar parte de la comunicación entre procesadores, dividiendo los datos en conjuntos disjuntos. La principal ventaja es que este tipo de balanceo es que no produce carga adicional en el tiempo de ejecución del programa.

Balanceo dinámico: Se aplica a problemas donde la carga computacional varia dinámicamente durante la ejecución del programa, o con programas con los cuales no se conoce la carga, a priori, porque se crea según avanza la ejecución. En estos casos hay que buscar formas de distribuir la carga entre los procesadores de una forma equitativa, evitando que haya procesadores con mucho más trabajo que otros.

Para resolver este tipo de problemas se emplean estrategias dinámicas de distribución de carga. Estas estrategias producen un incremento al propio tiempo de ejecución del programa. Por esta razón se debe alcanzar un compromiso entre balanceo de carga y sobrecarga producida.

Los problemas aquí tratados están basados en algoritmos de Ramificación y Acotación, cuyo principal problema es la necesidad de una distribución dinámica de la carga computacional.

Generalmente los algoritmos de balanceo dinámico de carga constan de cuatro componentes:

Medida de carga: La carga computacional se suele cuantificar mediante un índice positivo, que toma el valor cero cuando un procesador está ocioso, e incrementa su valor según aumenta la carga computacional. Este índice es una estimación de la carga computacional y se basa en algunos parámetros medibles, como la cantidad y el tamaño de las operaciones a realizar.

Intercambio de información: Para tomar decisiones sobre la necesidad de balanceo, un procesador debe conocer la carga del resto de procesadores. Este intercambio de información puede hacerse de forma centralizada o distribuida:

- En la versión distribuida cada procesador deber tener información de la carga de los procesadores vecinos, si se establece una estrategia local, o la información de todos los procesadores si se ha establecido una estrategia global.
- En la versión centralizada un procesador es el encargado de recoger y mantener la información de la carga computacional del resto de procesadores. Éste sera el procesador encargado de tomar las decisiones y notificar al resto los cambios para mantener lo más balanceado posible el sistema.

Las estrategias distribuidas globales son impracticables en sistemas muy grandes, pues el tráfico de información para informar a todos los procesadores seria un gran inconveniente para el funcionamiento del sistema. Por otra parte, las estrategias centralizadas obtienen un buen rendimiento en sistemas pequeños. Para sistemas mayores, el procesador dedicado al balanceo suele ser un cuello de botella para las comunicaciones. Para remediar estos problemas se proponen estructuras jerárquicas de este tipo de procesadores dedicados.

Hay que establecer un compromiso entre tener un valor actualizado de la carga del resto de procesadores y el coste de las comunicaciones. Las estrategias más usadas son las siguientes:

Por demanda: Un procesador pregunta al resto sobre la carga antes de iniciar una operación de balanceo.

Periódicamente: Los procesadores intercambian periódicamente información sobre los valores de la carga.

Por cambio de carga: Cuando es estado de la carga de un procesador cambia una cierta cantidad, este informa del cambio al resto de los procesadores del sistema.

Además de emplear la información de la carga actual para realizar el balanceo, se pueden realizar nuevos balanceos de la carga usando las estadísticas extraídas de operaciones de balanceo anteriores.

Otra forma es la distribución aleatoria de la carga, que se realiza en tiempo de ejecución sin tener en cuenta ningún tipo de información sobre el sistema.

Regla de inicialización: Indica el momento en el que debe iniciarse una operación de balanceo. En esta decisión se debe sopesar la ventaja de inicializar el balanceo, frente a la sobrecarga computacional que éste conlleva. Este tipo de decisiones suelen ser heurísticas, debido a que se basan en el último valor conocido de la carga computacional de los demás procesadores, que normalmente es impreciso.

El balanceo de la carga debe inicializarse por un procesador con mucha carga, por un procesador con poco carga o periódicamente. Cuando la operación es iniciada por un procesador se deben establecer los umbrales de carga que inicializaran el proceso.

Operación de balanceo: Se definen las siguientes reglas para el balanceo dinámico de la carga computacional:

Regla de localización: Determina el conjunto de procesadores que van a intervenir en el balanceo de la carga, definiendo de este modo el dominio de balanceo.

Regla de distribución: Determina como distribuir la carga en el dominio de balanceo.

Regla de selección: Determina que trabajos forman la carga que se va a distribuir.

4.6. Algoritmos de Ramificación y Acotación paralelos

Los algoritmos de Ramificación y Acotación (Branch and Bound, B&B) se utilizan para encontrar soluciones óptimas mediante la generación de árboles de búsqueda, donde a menudo una búsqueda exhaustiva en todo un árbol es impracticable. Están basados en una descomposición sucesiva del problema original en subproblemas más pequeños y disjuntos, hasta encontrar la solución óptima, evitando evaluar subproblemas en los que se no encuentra la solución.

El procesamiento paralelo ha sido considerado ampliamente como una fuente adicional de mejora en la eficiencia de la búsqueda, ya que los problemas que se generan durante la ejecución de los algoritmos B&B son disjuntos y se pueden resolver simultáneamente

e independientemente, pudiéndose usar un conjunto de procesadores para resolver varios subproblemas de forma paralela en cada iteración.

Los algoritmos de Ramificación y Acotación se diferencian del otros algoritmos en que:

- Los conjuntos de datos son irregulares y de distinto tamaño.
- La carga de trabajo generada no es uniforme.
- En algoritmos paralelos el trabajo realizado puede depender del número de procesadores que se usen.

Por estos motivos la programación se convierte en un trabajo preciso en el que hay que tener en cuenta las siguientes posibilidades:

Sobrecarga de la búsqueda: Los algoritmos de búsqueda paralelos tiene sobrecargas debido diferentes motivos. Estos incluyen sobrecargas en las comunicaciones y tiempos ociosos debido a desbalanceo de la carga y a conflictos de acceso a memoria, si se usan estructuras de datos compartidas. Así, si las implementaciones paralelas y secuenciales del algoritmo realizan la misma cantidad de trabajo, es normal que la ganancia en velocidad para p procesadores $S(p) < p$. Sin embargo, la cantidad de trabajo realizada en la implementación paralela a menudo difiere de la realizada por la versión secuencial, principalmente, porque se exploran diferentes partes del árbol de búsqueda T.

Definición 7 *Sea W el trabajo realizado por el algoritmo de búsqueda secuencial y W_p el realizado por la versión paralela con p procesadores. Se define el factor de sobrecarga de la búsqueda del sistema paralelo como la relación entre el trabajo realizado por la versión paralelo y la secuencial, W_p/W [23].*

Así, el límite superior de $S(p)$ viene dado por pW/W_p. En la práctica, la ganancia de velocidad puede ser menor, debido a otras sobrecargas de la paralelización. En la mayoría de los algoritmos de búsqueda paralelos el factor de sobrecarga de la búsqueda es mayor a uno. Sin embargo, en algunos casos es menor que uno, dando lugar a ganancias de velocidad superlineales. Si el factor de sobrecarga es menor que uno, entonces el algoritmo secuencial no es el más rápido para resolver el problema.

Decisión entre comunicación y computación: En los algoritmos de Ramificación y Acotación paralelos hay que tomar la decisión en el compromiso entre comunicación y computación. Como se mencionó anteriormente la sobrecarga de la búsqueda para muchos algoritmos es mayor que uno, lo que implica que la implementación paralela realiza más trabajo que la secuencial. Es posible reducir la sobrecarga de la búsqueda para estas implementaciones incrementando las comunicaciones, además también se cumple lo contrario. Por ejemplo, si la región de búsqueda se divide estáticamente entre los procesadores y estos realizan una búsqueda independiente, por lo tanto sin

comunicaciones, estos realizaran en conjunto una búsqueda mayor que en la versión secuencial del algoritmo.

Esta sobrecarga de búsqueda puede reducirse dependiendo de las comunicaciones realizadas entre los procesadores. En los algoritmos de Ramificación y Acotación paralelos, una parámetro importante a comunicar es la mejora encontrada en el límite superior de la solución. Este intercambio de información permitirá a más procesadores hacer uso de la Regla de Eliminación y reducir así la sobrecarga de la búsqueda, pero aumentando la sobrecarga de comunicación.

Escalabilidad y algoritmos escalables: La ganancia de velocidad no se incrementa linealmente con el número de procesadores si se usa la versión paralela para resolver un problema de tamaño fijo. En estos casos, la línea de ganancia de velocidad tiende a saturarse. Este fenómeno es cierto para todos los sistemas paralelos y es descrito mediante la *Ley de Amdahl* (Definición 5).

Para muchos algoritmos paralelos, si se incrementan el tamaño del problema W, con un número fijo de procesadores p, se incrementa la eficiencia, aproximándola a uno, porque la sobrecarga total del proceso paralelo crece menos que W. Para estos algoritmos paralelos la eficiencia puede mantenerse a un determinado valor (entre 0 y 1) con el incremento del número de procesadores, si el tamaño del problema también aumenta. Estos algoritmos se denominan algoritmos escalables [23]. El porcentaje en el que el tamaño del problema debe crecer, con respecto al número de procesadores, para mantener la eficiencia, determina el grado de escalabilidad de los algoritmos paralelos y está determinado por la función de *isoeficiencia* [40, 41].

Como el reparto del trabajo entre los procesadores es una parte importante del algoritmo de Ramificación y Acotación paralelo el modo de almacenamiento de la estructura de datos D, que contiene los subproblemas activos, es una parámetro a considerar. Otro parámetro a determinar es cuándo se accede a la estructura de datos D. Este acceso puede hacerse en momentos arbitrarios durante la ejecución, o los procesadores deben esperarse unos a otros, es decir, un algoritmo síncrono o asíncrono. A continuación se realiza una descripción en detalle de estos parámetros:

Memoria compartida versus memoria distribuida: Un punto importante es la elección de la forma de almacenamiento de los datos de los subproblemas. Las posibles opciones son el modelo de memoria compartida SDOM (Shared Data Object Model) y el modelo de memoria distribuida DDM (Distributed Data Model) [10, 12]. Además existen otros modelos que combinan características de ambos, modelos intermedios.

- En el **modelo de memoria distribuido** cada procesador i mantiene su propia estructura de datos local D_i. Los procesadores se comunican mediante intercambios de subproblemas entre las estructuras de datos locales, que se organizan como listas ordenadas de problemas. Cuando se selecciona un nodo

del subárbol de búsqueda local, éste se obtiene de la estructura de datos local D_i y los generados a partir de él son almacenados en D_i.

La simplicidad de este modelo puede dar lugar a pensar que los algoritmos paralelos de Ramificación y Acotación son sencillos, porque cada procesador realiza una búsqueda secuencial en partes diferentes y disjuntas del árbol de búsqueda T. Sin embargo, no se conoce la estructura de T a priori, y ya que los problemas no son regulares. Al paralelizar estos algoritmos aparecen irregularidades que provocan un gran desbalanceo de la carga computacional entre los distintos subproblemas de los distintos procesadores. Es probable que el algoritmo distribuido produzca una sobrecarga en la búsqueda, si se compara con el algoritmo secuencial, ya que una región que no se considere en el caso secuencial puede ser asignada a un procesador para su evaluación. Además un procesador puede quedarse sin trabajo al quedarse su estructura de datos local vacía. Estos hechos se deberán tenerse en cuenta durante la implementación de algoritmos paralelos de Ramificación y Acotación sobre esta clase de memoria y las medidas a tomar para tener estos problemas están basadas en técnicas de balanceo dinámico de la carga, donde la carga computacional debe dividirse y repartirse de la forma más equitativa posible entre los distintos procesadores (Sección 4.5).

- En el **modelo de memoria compartida** existe una sola estructura de datos D, que contiene el conjunto de subproblemas de T. Los procesadores deben comunicarse a través de la estructura de datos global D y deben mantener, en todo momento, un estado consistente de los subproblemas en ejecución. La ventaja es que como se mantienen todos los subproblemas en la memoria global D y los procesadores pueden acceder a todos ellos, de este modo se realiza una distribución buena de la carga ya que según van acabando los procesadores sus subproblemas pueden coger trabajo asignado, a priori, a otros procesadores. La desventaja de esto es que los procesadores deben acceder con frecuencia a la memoria compartida D y podría causar un cuello de botella si el número de procesadores es alto.

Por lo mencionado en los puntos anteriores, los algoritmos de búsqueda tienen como estructura de almacenamiento lógica las estructuras de datos globales. Éstas se usan normalmente en máquinas de memoria compartida, aunque esto también provoca un aumento de comunicación para acceder a las estructuras compartidas. Para reducir estos costes pueden usarse estructuras distribuidas, aunque esto incrementaría los costes de búsqueda en estas estructuras. Se debe buscar un compromiso entre ambos.

Los sistemas de memoria compartida no son muy escalables, por este motivo la mayoría de sistemas de memoria compartida están basados en un arquitectura de memoria distribuida, con soporte para un direccionamiento global.

Los modelos descritos anteriormente (SDOM y DDM) pueden verse como los extremos, y existen modelos intermedios. Por ejemplo, los procesadores pueden dividirse en grupos, donde cada grupo se comporta de acuerdo a un modelo de memoria

compartida y entre grupos se aplica el modelo de memoria distribuida. Ha esto pueden añadirse memoria locales a cada procesador o otros modelos de memoria intermedios.

Unidad de trabajo: Otro parámetro a elegir es la unidad de trabajo. Hasta ahora se ha asumido implícitamente que D, global o local, era actualizada cada vez que un subproblema ha sido evaluado. También puede permitirsele a un procesador realizar una búsqueda limitada a partir de un subproblema, es decir, generar un subárbol de pocos niveles a partir del nodo seleccionado del árbol de búsqueda, y solo entonces actualizar D con los nodos hojas de ese subárbol. De esta forma decrece la necesidad de realizar comunicaciones, pero al mismo tiempo decrece la calidad de los datos almacenados en D.

Modo de sincronización: Un procesador tiene dos opciones cuando completa su unidad trabajo: (1) esperar a que los demás procesadores completen su unidad de trabajo, (2) o continuar sobre otra unidad de trabajo. Esperar a los demás puede ser más favorable, especialmente en un modelo de memoria compartida, ya que los problemas seccionados por los procesadores en cada iteración son los mejores, debido a que un procesador tiene un conocimiento completo sobre el estado del problema a la hora de seleccionar una nueva unidad de trabajo. De esta forma las implementaciones síncronas limitan la cantidad de trabajo innecesario que se pueda realizar, pero también produce estados de espera en los procesadores. En el caso del asíncrono, un procesador puede no tener conocimiento de la información generada por otro procesador. Esto puede dar lugar a resolver subproblemas cuya resolución no sería necesaria en un modelo síncrono. Por otro lado, las ventajas que ofrece son: un mayor potencial de ganancia de velocidad que el modelo síncrono, se solapan comunicación y computación y no hay puntos de sincronización para acceder a la estructura de datos D.

Modo de interrupción: Para habilitar el uso de la información generada por otros procesadores, un procesador tiene que detectar la actualización de D. Existen dos modos básicos en los que un procesador se da cuenta de una actualización: el procesador comprueba D a intervalos regulares (normalmente en modelos SDOM) o es interrumpido cuando se actualiza D (normalmente en modelos DDM). Si se recibe una interrupción, comprueba D y decide si continua trabajando sobre la unidad actual o si realiza otra tarea. En el último caso, la unidad de trabajo actual se almacena en D.

La elección de estos parámetros no siempre es fácil. Los parámetros dependen de factores como el tipo de problema, es decir, la especificación de las reglas básicas de Ramificación y Acotación, del ejemplo concreto del problema y de la arquitectura utilizada, ya que no todas las arquitecturas soportan unas especificaciones arbitrarias de los parámetros, los protocolos de comunicación pueden ser rápidos o lentos, etc.

Capítulo 5

Descripción del problema

En este capítulo se describirán los componentes que intervienen en el problema plateado, explicando los detalles de cada uno de los componentes, el método de división de símplices escogido y la forma en la que se obtiene el árbol de búsqueda que se usará para resolver los problemas planteados. Además se explicará el método a seguir para determinar cada tipo de símplice (simétricos, con varios lados mayores o regulares) que pueda aparecer durante el proceso de generación del árbol de búsqueda.

5.1. ¿Qué es un simplex?

En este trabajo, el espacio inicial está determinado por un n-simplex regular. Un n-simplex, o n-símplice, está definido en un espacio de $n+1$ dimensiones. Un simplex es el politopo más sencillo posible en un espacio dado.

5.1.1. Politopos

Un politopo es la generalización del concepto de *Polígono* (2 dimensiones) o *Poliedro* (3 dimensiones) a cualquier otra dimensión. Geométricamente un politopo es una región finita de un espacio n-dimensional encerrado por un número finito de hiperplanos.

La figura 5.1 muestra algunos ejemplos de politopos.

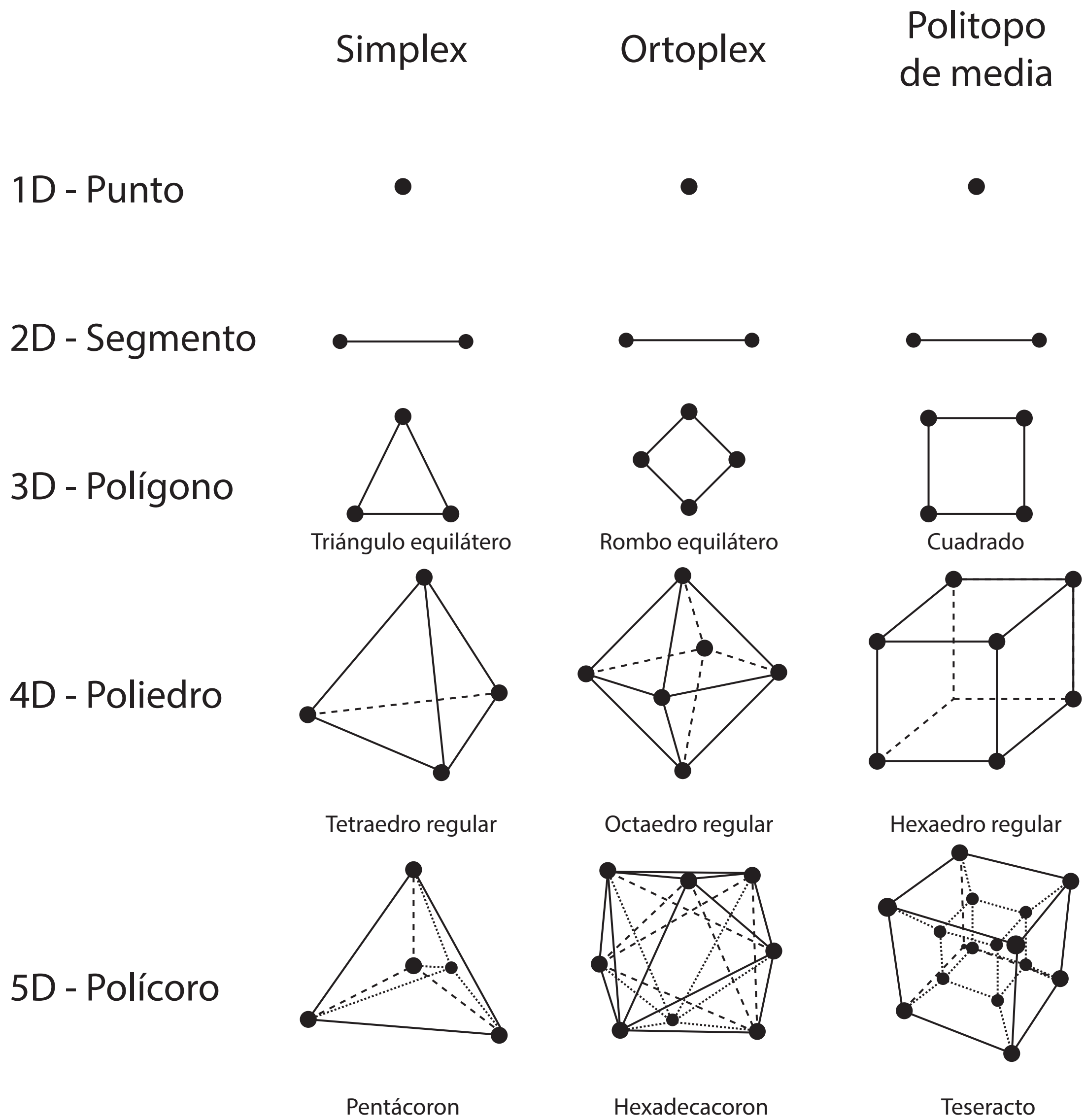


Figura 5.1: Ejemplos de distintos politopos.

Por lo tanto se tiene que un 0-simplex es un punto, un 1-simplex es un segmento de una línea, un 2-simplex es un triángulo, un 3-simplex es un tetraedro, un 4-simplex es un pentácoron, etc. En la Figura 5.2 se muestran los ejemplos de símplices para dimensiones 1, 2 y 3.

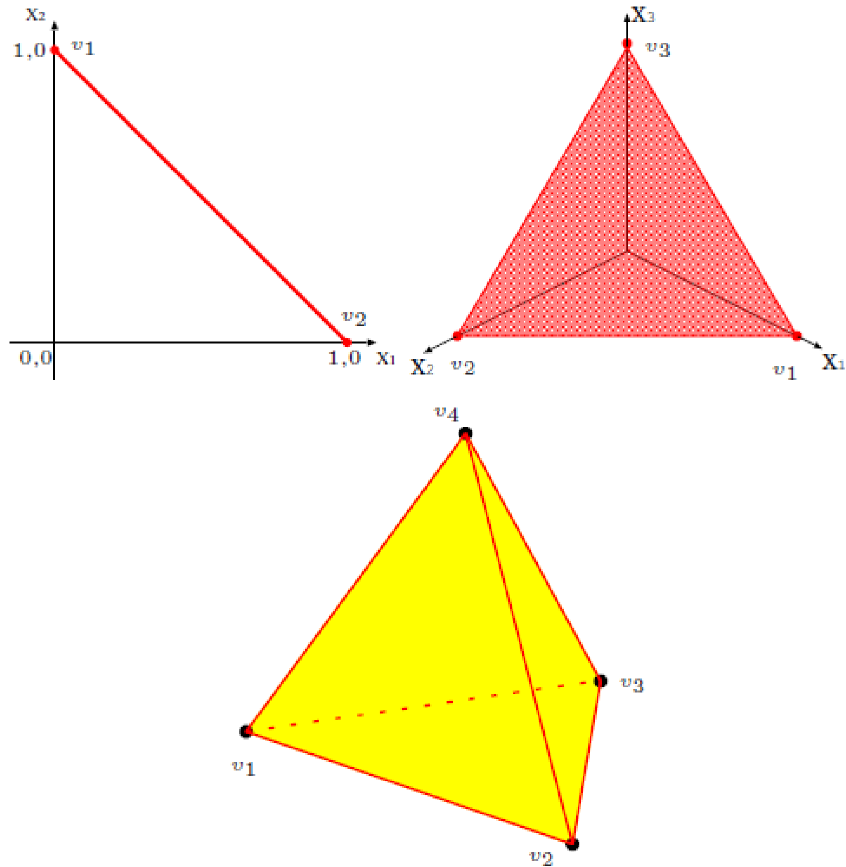


Figura 5.2: Símplices de dimensiones 2, 3 y 4.

Definición 8 *Un simplex se define como la envoltura convexa de un conjunto de $n+1$ puntos independientes afines en un espacio euclídeo de dimensión mayor o igual que n, es decir, el conjunto de puntos tal que ningún m-plano contiene más de $m+1$ de estos puntos con $m \leq n$.*

El n-simplex unidad, o n-simplex estándar, es un subconjunto de $\mathbb{R}^{n+1}$ tal que:

$$S = \left\{ x \in \mathbb{R}^{n+1} \mid \sum_{j=1}^{n+1} x_j = 1;\ x_j \geq 0 \right\} \tag{5.1}$$

Por simplicidad y sin perdida de generalidad, en este trabajo se va a hacer uso de un simplex regular con longitud de lado 1, que se define como:

$$S = \left\{ x \in \mathbb{R}^{n+1} \mid \sum_{j=1}^{n+1} x_j = \frac{\sqrt{2}}{2};\ x_j \geq 0 \right\} \tag{5.2}$$

Definición 9 *Un símplice regular no es más que un símplice en el que todos sus lados tienen la misma longitud.*

La figura 5.3 muestra un 2-símplice regular con lados de tamaño unidad.

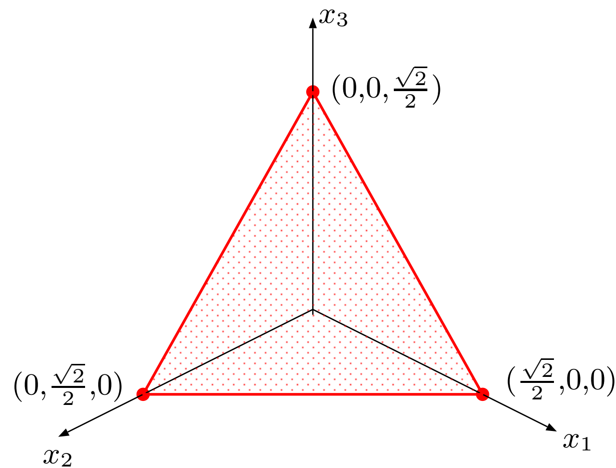


Figura 5.3: Un 2-símplice regular con lados de longitud unidad.

Las características principales de un n-simplex son las siguientes:

- Un n-simplex tiene $n + 1$ vértices y $n(n + 1)/2$ lados.
- Se define la anchura de un simplex, $\omega(S)$, como la longitud del lado mayor del simplex.
- A la envoltura convexa de un conjunto de $m + 1$ puntos independientes afines de un n-simplex $(m < n)$ de denomina m-cara.
- A la $(n - 1)$-cara se la denomina faceta.

5.2. Bisección por el Lado Mayor (BLM)

La Bisección por Lado Mayor o BLM, es un método de refinamiento que puede ser aplicado a problemas de cualquier dimensión. Es uno de los métodos más populares debido a la simplicidad del proceso a seguir. Para generar un árbol se divide cada uno de los símplices por un lado mayor en cada división o nivel.

La figura 5.4 muestra los primeros símplices generados después de tres bisecciones al aplicar este método sobre un 2-simplex cuando se realiza una búsqueda en profundidad.

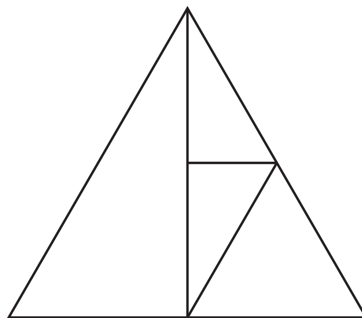


Figura 5.4: Primeras bisecciones en un 2-simplex regular.

La figura 5.3 muestra un 2-símplice regular con lados de tamaño unidad. La selección de un lado mayor en un 2-símplice no es necesaria ya que o el lado mayor es único o el subsímplice a dividir es regular. A partir de $n = 3$ sí pueden existir varias opciones para elegir el lado mayor, como se explica en la sección 5.3.1.

La implementación de la Bisección por el Lado Mayor se muestra en el capítulo 6, donde se describen los diferentes algoritmos planteados y los resultados obtenidos con cada uno de ellos. El algoritmo 2 resume el proceso general de un algoritmo de ramificación y acotación, mientras que el algoritmo 3 detalla el proceso de bisección de un símplice por el lado mayor seleccionado.

Algoritmo 2 RyA(S,ϵ)

Require: S: símplice, ϵ: épsilon
1: $\Lambda := \{S_1 = S\}$ — *Conjunto de trabajo*
2: $\Omega := \{\}$ — *Conjunto final*
3: $ns := 1$ — *Número de símplices*
4: **while** $\Lambda \neq \emptyset$ **do**
5: Selecciona S_i de Λ — *Regla de selección*
6: Evalúa S_i — *Regla de acotación*
7: **if** S_i no puede ser eliminado **then** — *Regla de rechazo*
8: **if** $w(S_i) \leq \epsilon$ **then** — *Regla de terminación*
9: Almacena S_i in Ω
10: **else**
11: L := seleccionLadoMayor(S_i)
12: $\{S_{2i}, S_{2i+1}\}$:= BLM(S_i, L) — *Regla de división (Bisección por lado mayor)*
13: Almacena S_{2i}, S_{2i+1} en Λ
14: $ns := ns + 2$
15: **end if**
16: **end if**
17: **end while**
18: **return** Ω y $f(x^*)$

Algoritmo 3 BLM(S, L)

Require: S: símplice, L: lado mayor

1: Generar nuevo vértice a partir de los vértices v_i, v_j de L: $v_{new} := \frac{v_i+v_j}{2}$
2: Crear dos nuevos símplices S_l, S_r con las características de S
3: Siendo S_{id} el identificador S, $2S_{id}$ y $2S_{id}+1$ son los identificadores de S_l y S_r
4: Sustituir v_j por v_{new} en S_l y calcular distancias entre v_{new} y el resto de vértices
5: Sustituir v_i por v_{new} en S_r y calcular distancias entre v_{new} y el resto de vértices
6: **return** S_l, S_r

5.3. Particionamiento de un n-símplice regular

El procedimiento de particionamiento de un n-símplice para la obtención de un árbol de búsqueda utilizando el método por bisección explicado en el punto 5.2 se detalla en el algoritmo 5 del capítulo 6.

El proceso explicado en este algoritmo se podría resumir en que a partir de un símplice inicial regular se generan sus dos hijos, cada uno de los cuales generara otros dos hijos, etc. Este proceso creará un árbol binario como el mostrado en la figura 5.5 en el que cada una de las ramas finalizara solamente cuando la longitud del lado mayor de su símplice hoja sea mayor que un valor de ϵ definido por el usuario. Esta regla de terminación esta basada en la longitud del lado mayor del símplice.

La figura 5.5 muestra los primeros pasos en el proceso de división del 2-simplex.

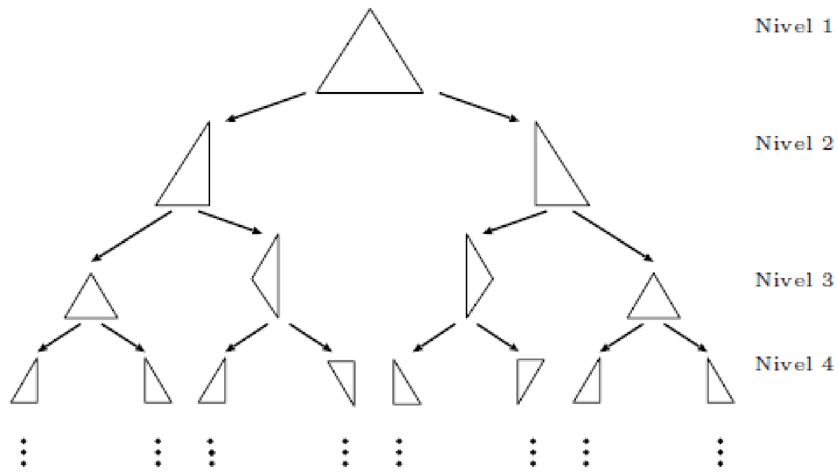


Figura 5.5: Bisecciones y niveles del árbol binario obtenido mediante BLM de un 2-simplex.

5.3.1. Símplices con más de un lado mayor. Obtención del árbol completo mínimo

En este trabajo se ha escogido como método de división la división por el lado mayor, que como su propio nombre indica es un lado tal que ningún otro tiene una longitud

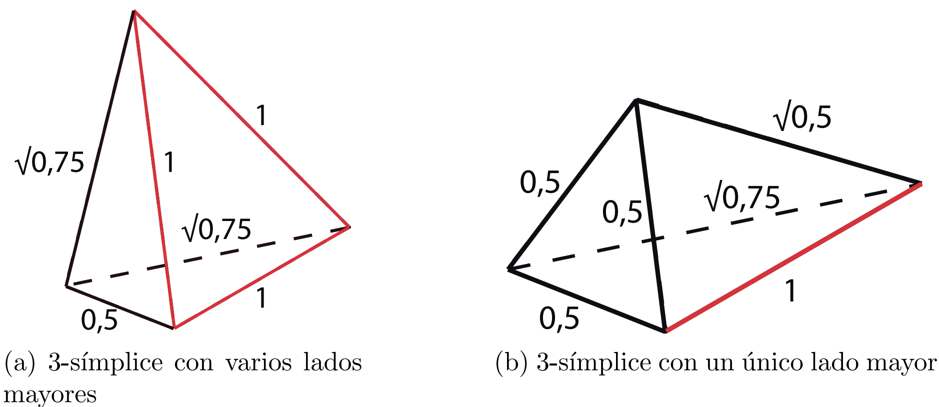
superior.

En un n-símplice con $n < 3$ solo existe un lado mayor en cada símplice, pero a partir de $n = 3$ aparecen símplices con más de un lado mayor. Los posibles métodos para dividir símplices con varios lados mayores son:

- Dividir por el primer lado mayor
- Dividir por un lado mayor al azar
- Dividir todos los posibles lados mayores y posteriormente determinar el que obtiene un árbol más pequeño.
- Aplicar algún tipo de heurística para determinar el lado mayor a dividir

Aquí se estudiaran los dos últimos, ya que son los únicos que pueden asegurar un árbol mínimo como resultado.

La figura 5.6 muestra ejemplos un símplice con más de una lado mayor y otro con solamente un lado mayor para un 3-símplice. Los lados mayores están señalados con color rojo. El símplice de la figura 5.6a tiene 3 posibles lados mayores mientras que el de la figura 5.6b solo tiene uno.



(a) 3-símplice con varios lados mayores

(b) 3-símplice con un único lado mayor

Figura 5.6: Diferentes tipos de símplices

Para obtener el árbol completo mínimo, es necesario explorar todos los posibles subárboles generados a partir de cada símplice con varios lados mayores. Una vez se tienen los resultados de los subárboles generados por cada uno de los lados mayores del símplice, se podrá determinar cual de los lados mayores a generado el menor árbol. Esto aumentara la computación necesaria para resolver los problemas respecto a otros procedimientos (como dividir por un lado al azar). Por lo que surge una nueva cuestión:

Cuestión 3 *¿Es posible evitar el cálculo del tamaño de algunos de los subárboles que aparecen cuando un símplice tiene varios lados mayores?*

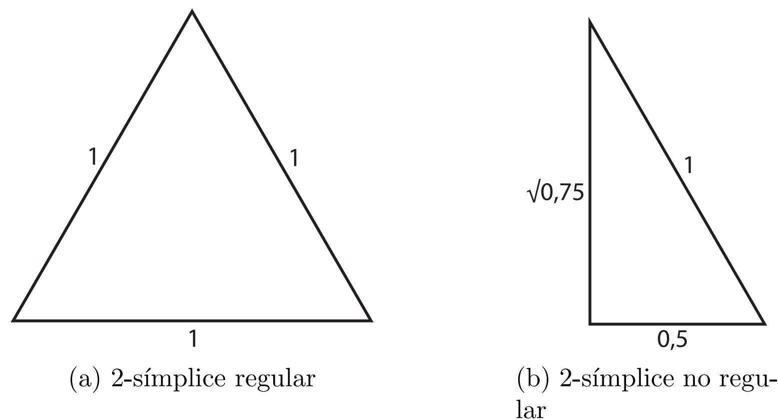
Se podría pensar en un algoritmo B&B que emplee un test de rechazo para resolver el problema de Optimización Combinatoria de encontrar el árbol binario completo mínimo. Se llamará al árbol generado por ese algoritmo árbol OC. Cada nodo del árbol OC, seria un árbol binario en proceso de construcción que dependerá del lado mayor seleccionado para su bisección. Un nodo hoja sería un árbol completo. Un nodo del árbol OC se podría eliminar si ya existe un nodo hoja conteniendo un árbol binario con un menor tamaño.

Por lo tanto la respuesta, a priori, es sí. El test de rechazo explicado anteriormente se puede aplicar también comparando los distintos subárboles generados a partir de un nodo del árbol OC, pues bastara con evaluar uno de los subárboles para lograr un valor de referencia que posteriormente sera transmitido al resto de subárboles, los cuales detendrán su evaluación cuando superen este valor, que hasta ese momento será la mejor cota superior conocida del mínimo.

El problema es el de transmitir este valor entre los diferentes subproblemas cuando estos creen nuevos subproblemas, a que cada subproblema nuevo debería, además de mantener y transmitir su propio valor de subárbol mínimo, actualizar y trasmitir los valores de los posibles subárboles superiores. La implementación expuesta en este trabajo no implementa esta propuesta y se deja planteada como trabajo futuro.

5.3.2. Símplices regulares

Como ya se ha comentando en la definición 9, un símplice regular es aquel que tiene las longitudes de todos sus lados iguales. La figura 5.7 muestra ejemplos para un 2-simplex, de un símplice regular y otro que no lo es.

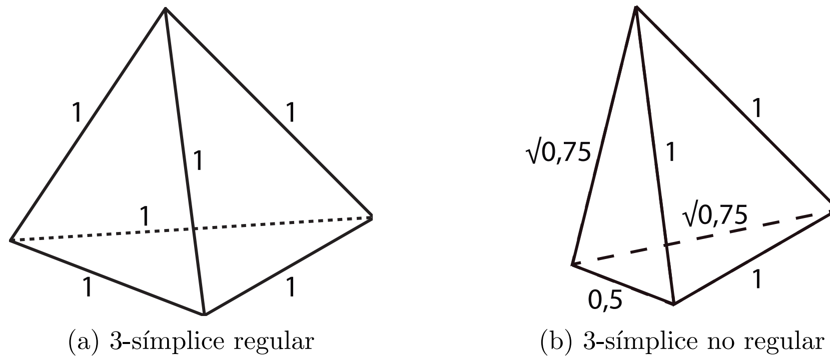


(a) 2-símplice regular

(b) 2-símplice no regular

Figura 5.7: 2-símplices regulares y no regulares.

La figura 5.8 muestra ejemplos para un 3-simplex, de un símplice regular y otro que no lo es.



(a) 3-símplice regular (b) 3-símplice no regular

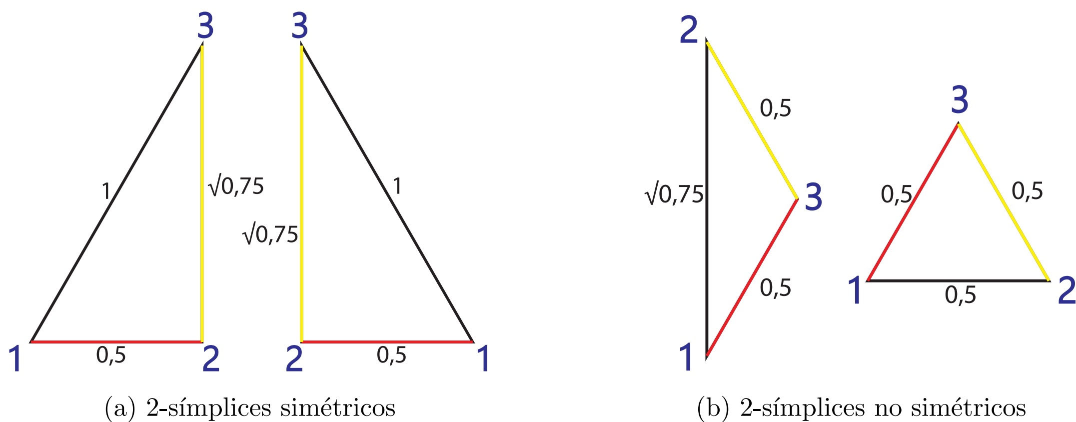
Figura 5.8: 3-símplices regulares y no regulares.

En este caso, para dividir un símplice regular bastara con dividirlo por uno cualquiera de sus lados, ya que los símplices resultantes de la división serán simétricos.

5.3.3. Símplices simétricos

El algoritmo 3 muestra como se crean los dos hijos, pero puede darse el caso de que estos subsímplices hermanos sean simétricos.

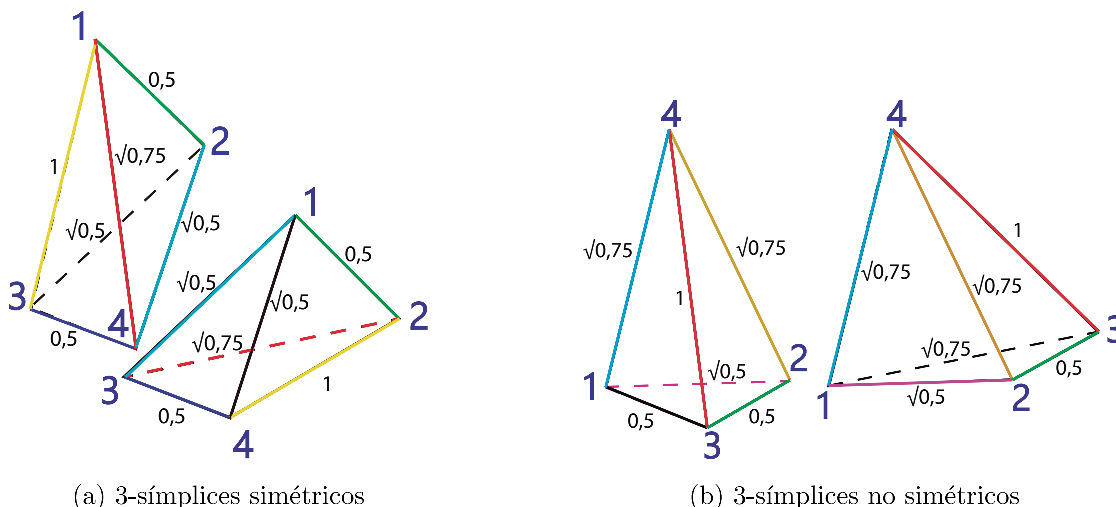
Para averiguar si dos símplices hermanos son simétricos se deben comprobar que existen lados equivalentes (con la misma longitud) en ambos símplices, dispuestos en el mismo orden. La figura 5.9 muestra el proceso seguido para comparar dos símplices y determinar si son o no simétricos. Como se observa en la figura 5.9a el lado rojo unido por los vértices $(1, 2)$ tiene la misma distancia que el lado formado por los vértices $(2, 1)$ en ambos símplices, al igual que el lado amarillo (vértices $(2, 3)$) y el lado negro (vértices $(1, 3)$). Por lo tanto, como los lados de ambos símplices son equivalentes, los símplices son simétricos.



(a) 2-símplices simétricos (b) 2-símplices no simétricos

Figura 5.9: 2-símplices simétricos y no simétricos

Por otra parte, en la figura 5.9b se observan dos símplices que no son simétricos. Estos símplices no tienen todos sus lados iguales y en el mismo orden, por lo que no son simétricos. Otro ejemplo de símplices simétricos es el mostrado en la figura 5.10, donde se observa un 3-símplice simétrico y otro que no lo es.



(a) 3-símplices simétricos

(b) 3-símplices no simétricos

Figura 5.10: 3-símplices simétricos y no simétricos.

El proceso descrito con las imágenes anteriores se puede formalizar con la siguiente proposición, en el que se comprueba si los dos símplices generados por un mismo padre son simétricos:

Proposición 1 *Si el lado mayor lo forman v_i y v_j y tamaño($v_i - v_x$) = tamaño($v_j - v_x$), $\forall v_x \notin \{v_i, v_j\}$, los dos subsímplices son simétricos.*

Tras comprobar que dos símplices que son generados por el mismo símplice padre pueden ser simétricos surge una nueva cuestión relacionando con la mejora del algoritmo.

Cuestión 4 *¿Es posible reducir la evaluación de los subárboles para los símplices simétricos?*

La respuesta es, sí. Pues dos símplices simétricos generaran siempre el mismo subárbol y por lo tanto solo es necesario calcular el tamaño de uno de ellos. Esto reducirá considerablemente el número de subárboles a los que hay que calcular su tamaño.

En este caso solo seria necesario evaluar uno de los hijos y el símplice padre multiplicaría por 2 el resultado obtenido de esta evaluación.

Otra posible mejora sería la de mantener una estructura con todos los símplices que van apareciendo y así cuando se repita alguno no seria necesario volver a evaluarlo. El principal problema es el tiempo requerido para determinar si el tamaño de los subárboles generados a partir de un símplice igual al actual ha sido calculado, ya que el número de símplices puede ser muy grande. Por este motivo se ha descartado esta idea.

Capítulo 6

Soluciones planteadas

En este capítulo se presentan las distintas implementaciones realizadas tras el análisis de los elementos del problema, empleando el algoritmo de bisección planteado en el capítulo 5.

Se comienza con la versión secuencial, a la cual se le realizan las modificaciones planteadas en la sección 5.3 para reducir su tiempo de ejecución e incrementar la eficiencia del algoritmo. Seguidamente se describen dos posibles soluciones paralelas, ambas basadas en la versión secuencial.

Posteriormente, en el capítulo 7 se compararan y comentaran los resultados obtenidos por cada una de la distintas alternativas implementadas.

6.1. Versión secuencial

El algoritmo secuencial debe comprobar las posibles opciones de división de lado mayor, teniendo en cuenta solo aquellas que generan un árbol binario completo de menor tamaño. Esto se puede hacer mediante un algoritmo recursivo. Los algoritmos 4 y 5 realizan el proceso. El algoritmo 4 divide un símplice en dos subsímplices, dividiendo por el lado mayor siempre que este sea mayor que ϵ.

Algoritmo 4 TamañoÁrbol (S, L, ϵ)

Require: S: símplice, L: lado mayor, ϵ: precisión
1: **if** $w(S) \leq \epsilon$ **then**
2:    **return** 1
3: **end if**
4: $\{S_1, S_2\}$=DivideSímplice(S,L)
5: r_1 := TamañoÁrbolMínimo(S_1)
6: r_2 := TamañoÁrbolMínimo(S_2)
7: **return** $r_1 + r_2$

Para cada uno de los dos subsímplices generados se ejecuta el algoritmo 5, donde se calcula el tamaño del subárbol generado por la posible división de cada lado mayor,

devolviendo el tamaño del menor de ellos. Nótese que el algoritmo 5 hace una llamada recursiva al algoritmo 4.

Algoritmo 5 TamañoÁrbolMínimo (S, ϵ)

Require: S: símplice, ϵ: precisión
1: **for** cada lado mayor L_i de S **do**
2: r_i := TamañoÁrbol(S,L_i,ϵ)
3: **end for**
4: **return** $\text{mín}_i\{r_i\}$

El algoritmo 4 realiza una búsqueda en profundidad lo que reduce los requerimientos de memoria. El programa se ejecutaría con el n-símplice regular inicial realizando una llamada al algoritmo 5.

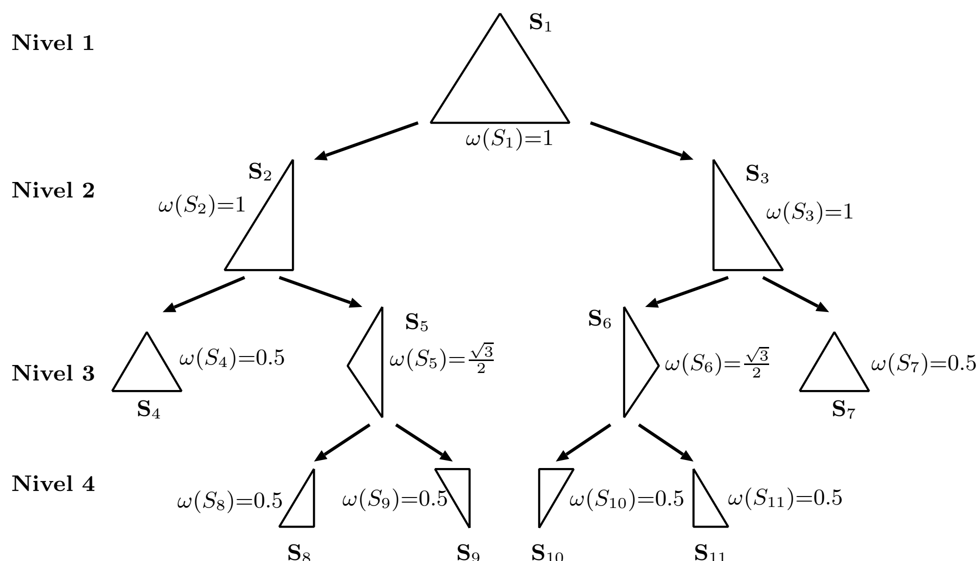


Figura 6.1: Árbol binario para $\epsilon = 0{,}5$.

La figura 6.1 muestra la ejecución del algoritmo 4 para un 2-símplice y $\epsilon = 0{,}5$. El ejemplo no es interesante ya que solo existen subsímplices con un solo lado mayor o subsímplices regulares, dando lugar a un solo posible tamaño del árbol, que para $\epsilon = 0{,}5$ es de 11. Aun así, la figura 6.1 permite resaltar algunos aspectos que, para simplificar, no se han incluido en los algoritmos 4 y 5:

- En el algoritmo 4, si S_1 y S_2 son simétricos, solo es necesario procesar uno de ellos y devolver como resultado el doble del tamaño de su subárbol. Por ejemplo, los símplices hermanos S_2 y S_3 o S_8 y S_9, de la figura 6.1 son simétricos.

- En el algoritmo 5, si el símplice es regular, solo hay que procesar un lado mayor, por ejemplo el primero, ya que las diferentes opciones de lado mayor generan árboles con el mismo tamaño. Los símplices S_1, S_4 y S_7 de la figura 6.1 son un ejemplo.

Los puntos anteriores se pueden añadir a los algoritmos 4 y 5, junto con el método explicado en el punto 5.3.3 para crear dos nuevos métodos más eficientes, los algoritmos 6 y 7 (equivalentes a los algoritmos 4 y 5).

Algoritmo 6 TamañoÁrbol (S, L, ϵ)

Require: S: símplice, L: lado mayor, ϵ: precisión
1: **if** $w(S) \leq \epsilon$ **then**
2:     **return** 1
3: **end if**
4: $\{S_1, S_2\}$=DivideSímplice(S,L)
5: **if** sonSimetricos(S_1, S_2) **then**
6:     r_1 := TamañoÁrbolMínimo(S_1)
7:     **return** $2r_1$
8: **else**
9:     r_1 := TamañoÁrbolMínimo(S_1)
10:     r_2 := TamañoÁrbolMínimo(S_2)
11:     **return** $r_1 + r_2$
12: **end if**

En el caso de que un símplice sea regular bastara con dividir solamente por uno de sus lados en el algoritmo 7.

Algoritmo 7 TamañoÁrbolMínimo (S, ϵ)

Require: S: símplice, ϵ: precisión
1: **if** esRegular(S) **then**
2:     r_0 := TamañoÁrbol(S,L_0,ϵ)
3:     **return** r_0
4: **else**
5:     **for** cada lado mayor L_i de S **do**
6:         r_i := TamañoÁrbol(S,L_i,ϵ)
7:     **end for**
8:     **return** $\min_i\{r_i\}$
9: **end if**

Otro punto interesante a abordar, es el planteado en la sección 5.3.1 del capítulo anterior. En este punto se trataba el caso en el que un símplice posee más de un lado mayor. Este caso ocurrirá con n-símplices con $n > 3$ y la única forma de determinar el árbol mínimo implica evaluar todos los posibles subárboles y devolver solamente el tamaño del subárbol menor.

Además es importante mencionar que es esencial tener en cuenta una precisión a la hora de comparar los valores de los símplices, como distancias o posiciones de los vértices. La precisión usada en este trabajo a sido de 10^{-14} y ha sido usada para comparar los valores de las distancias, que están almacenadas como `double`.

La figura 6.2 muestra el proceso que sigue el algoritmo secuencial para realizar el recorrido en profundidad en el árbol búsqueda. Se puede observar como hasta que no finaliza la evaluación del subárbol izquierdo no se evalúa el subárbol derecho.

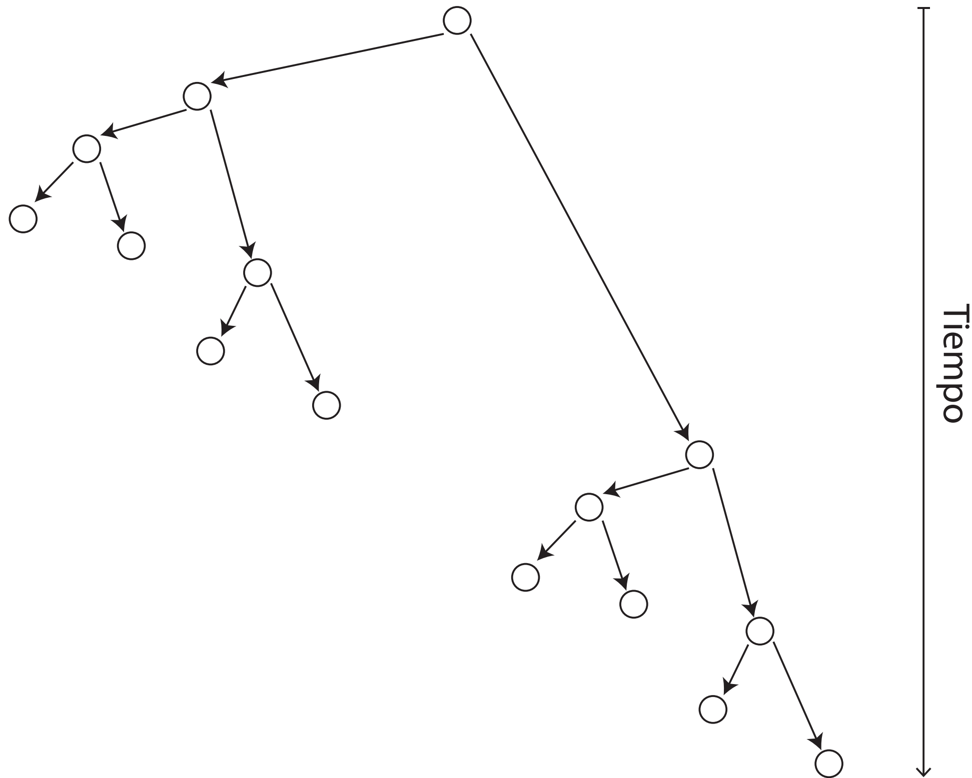


Figura 6.2: Esquema del algoritmo secuencial.

En el siguiente apartado se abordan las dos soluciones paralelas que se han diseñado para poder hacer uso de los distintos cores en un sistema multicore, de forma que se pueda reducir el tiempo de ejecución del algoritmo secuencial significativamente.

6.2. Versiones paralelas

Las arquitecturas y algoritmos paralelos permiten abordar instancias de este problema de combinación geométrica, que por su dimensión o tamaño del árbol generado requieren mucho tiempo de computación en su versión secuencial. Se presentan dos versiones multihebradas en arquitecturas de memoria compartida que podrán integrarse en futuras versiones paralelas sobre arquitecturas de memoria distribuida.

El paralelismo es fácilmente aplicable en la construcción de árboles binarios ya que las ramas del árbol pueden visitarse en paralelo. Uno de los inconvenientes de los árboles que se crean en este trabajo es que existen ramas con distintos niveles, tal como muestra la figura 6.1. La diferencia de niveles aumenta conforme lo hace n y no es conocida, lo que dificulta la distribución de trabajo en los distintos procesadores.

Tras estudiar las distintas alternativas posibles para el diseño de un algoritmo B&B paralelos en el capítulo 4 se han escogido las siguientes características para el diseño e implementación de los algoritmos en un sistema multicore:

- Respecto al balanceo de carga, ya que a priori no se conocerá la carga de cada uno de los subproblemas a evaluar, se opta por implementar un balanceo de carga dinámico. Este distribuirá la carga de trabajo según se vaya generando. Así se pretende obtener una buena distribución de trabajo entre procesadores.

  La forma de distribuir la carga entre los diferentes procesadores determinará el rendimiento del problema, por lo que será un importante tema a tratar durante este capítulo, en el que se plantearan dos posibles alternativas para distribuir la carga de trabajo.

- En cuanto al modelo de memoria escogido, se se hará uso de memoria compartida entre los distintos procesadores en la que se incluirán los elementos necesarios de sincronización entre hebras y los datos que las hebras usaran, aunque cada procesador tendrá una zona en la que trabajar de forma exclusiva en los subproblemas que se les asignen.

  Con el fin de evitar conflictos en el acceso a memoria, la memoria principal del sistema se dividirá en bloques exclusivos asignados a cada una de las hebras. Con esto se evitaran conflictos entre hebras a la hora de asignar y liberar la memoria de forma dinámica.

- Respecto a la comunicación y sincronización entre las hebras, se tendrán en cuenta dos posibles planteamientos:

  - Por un lado podría distribuirse todo el trabajo al comienzo de la ejecución y no sería necesaria una sincronización hasta que todas las hebras terminasen con el trabajo que tengan asignado.
  - El otro planteamiento sería el de distribuir el trabajo según se vaya generando y recoger los resultados según el trabajo se complete.

- Por ultimo, otro parámetro importante a tener en cuenta para la implementación es la unidad de trabajo, es decir, el subproblema que evaluara cada uno de los procesadores. Este será un símplice, que generara parte del subárbol de búsqueda mínimo. El conjunto de trabajo inicial puede estar formado por:

  - Un número lo suficientemente grande de símplices para que todos los procesadores tengan trabajo que ejecutar y para que este trabajo se pueda distribuir de la mejor forma posible tal que la carga final de trabajo entre procesadores este balanceada.
  - La otra opción es la de comenzar con un único símplice y según este se divide y genere nuevos símplices, ejecutar estos nuevos símplices en otros procesadores con el fin de distribuir el trabajo.

Como se verá a continuación cada planteamiento tendrá sus ventajas e inconvenientes.

Teniendo en cuenta las consideraciones anteriores, se desarrollarán dos versiones paralelas del algoritmo secuencial presentado en la sección anterior:

B2F Basada en dos fases. En una primera fase secuencial se generan los posibles subsímplices hasta que su número sea mayor o igual al número de hebras (`MaxHebras`) determinado por el usuario, empleando una búsqueda en anchura. En la segunda fase se ejecuta el algoritmo 4 para cada subsímplice en paralelo.

CDH Creación dinámica de hebras. Este algoritmo es una extensión del presentado en la sección 6.1. En esta nueva versión paralela las hebras se crean siempre que el algoritmo secuencial deba procesar más de un símplice.

6.2.1. Versión paralela basada en dos fases (B2F)

Este algoritmo se divide en dos fases, en la primera se busca particionar el espacio de búsqueda de forma que pueda ser repartido entre los distintos procesadores y en la segunda fase cada hebra calcula el subárbol mínimo para cada símplice de la primera fase.

Para la primera fase es importante realizar una recorrido del árbol en anchura para que los posibles subsímplices estén en niveles superiores de los posibles subárboles. En esta fase, un símplice con varios lados mayores genera varias parejas de subsímplices, una por cada lado mayor. De forma análoga a como se hace en el algoritmo 5 en el que solo la pareja de subárboles de menor tamaño es considerada posteriormente para determinar el tamaño del árbol mínimo.

La primera fase termina cuando el número de símplices en el último nivel del árbol o símplices hoja `NIniS` sea mayor que `MaxHebras`.

En la segunda fase se aplica de forma paralela el algoritmo secuencial a los subsímplices generados en la primera fase.

Una vez generado el árbol de la fase inicial, se crean tantas hebras como el usuario haya definido al ejecutar el programa con el parámetro `MaxHebras`. En la segunda fase, si el número inicial de subsímplices es mayor que `MaxHebras`, una hebra que haya terminado su trabajo evaluaría un subsímplice de los pendientes generados en la fase inicial. Este proceso se repetirá hasta que no existan símplices pendientes, en cuyo caso la hebra finalizara, devolviendo sus resultados.

La Figura 6.3 muestra un ejemplo del proceso de ejecución de este algoritmo. El proceso que sigue el algoritmo se puede describir de un forma sencilla en 3 pasos:

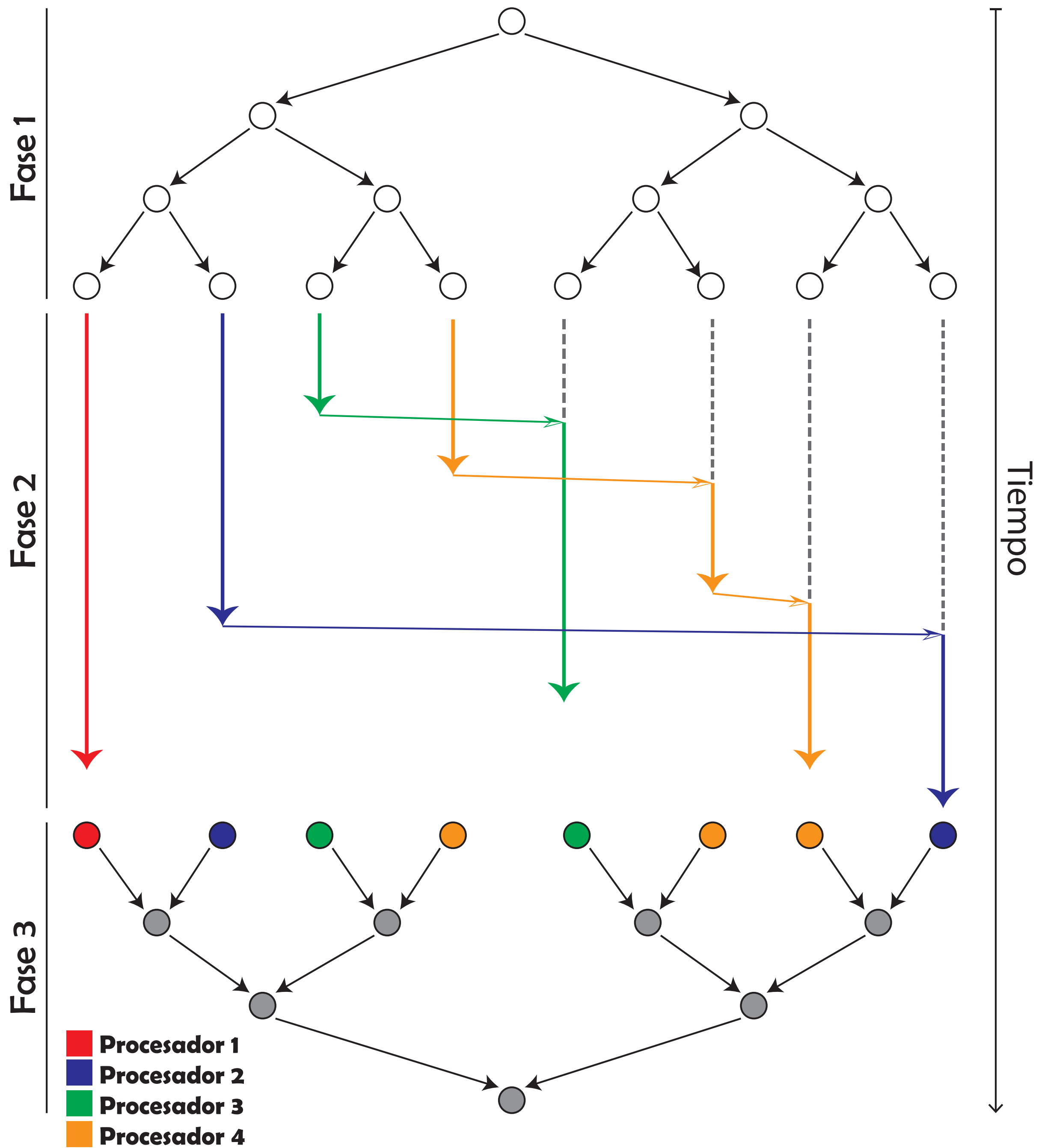


Figura 6.3: Esquema del algoritmo B2F (MaxHebras=4).

Fase 1: El algoritmo construye un árbol con los suficientes símplices (nodos hoja) para que todos los procesadores tenga trabajo.

Fase 2: Con los símplices generados en la fase anterior se asigna un subproblema a cada procesador. El resto de subproblemas permanecerán en una lista en memoria compartida y los procesadores que vayan finalizando sus trabajos tomaran nuevos

subproblemas de esta lista. La fase termina cuando no queda trabajo en la lista de trabajo pendiente.

Las hebras, conforme finalizan la evaluación de un subárbol, almacenan el resultado en el nodo correspondiente antes de evaluar el subárbol mínimo de otro nodo de la fase inicial. Cada hebra que termina su trabajo deberá comprobar si existe más trabajo pendiente y si es así, ejecutarlo. Esto permite un mejor reparto del trabajo, ya que la hebra que antes termine, porque el subárbol a evaluar tiene menos trabajo, será la que evalúe un nuevo subárbol de los pendientes. Así se evita la asignación de todo el trabajo al inicio de la ejecución, momento en el que no se conoce la cantidad de trabajo que implicará cada símplice.

Fase 3: En esta ultima fase se recogen para obtener el tamaño mínimo del árbol completo.

El siguiente algoritmo describe de una forma general el proceso seguido por este método paralelo.

Algoritmo 8 B2F(S, ϵ)

Require: S: símplice, ϵ: épsilon
1: arbol := S *Se inicializa el árbol con el simplex inicial*
2: **while** NIniS < MaxHebras **do**
3: S_t := siguienteSimplexAnchura(arbol) *Amplia el árbol en anchura*
4: S_1, S_2 := dividir()
5: ampliarArbol(arbol, S_1, S_2)
6: **end while**
7: **for** i = 0 a MaxHebras **do**
8: $hebra_i$:= crearHebra(S_i) *Se lanzan tantas hebras como procesadores disponibles*
9: **end for**
10: **for** cada $hebra_i$ en hebras **do**
11: S_i, res_i := recogerHebra($hebra_i$) *Cada hebra devuelve el nodo procesado y su resultado*
12: resultadoSimplex(S_i, res_i) *Se asigna el resultado al nodo correspondiente*
13: **end for**
14: **return** resultadoArbol(arbol) *Combina los resultados de las hebras y devuelve el árbol mínimo*

El número de símplices en la etapa inicial `NIniS` debe ser mayor que `MinNIniS = MaxHebras`, pero para que su número sea cercano a un múltiplo de `MaxHebras` se hace cumplir la siguiente condición:

$$\begin{aligned} &\text{NIniS} \geq \text{MinNIniS } \&\& \\ &(\text{NIniS} \mod \text{MinNIniS} == 0 \parallel \\ &\text{NIniS} \mod \text{MinNIniS} > 0{,}8 \cdot \text{MinNIniS}) \end{aligned} \tag{6.1}$$

donde `MinNIniS` corresponde con el número de hebras `MaxHebras` por un factor pf:

$$\text{MinNIniS} = \text{MaxHebras} * pf \tag{6.2}$$

Este factor pf fue determinado de forma empírica con valores distintos para cada caso y se generalizo en $pf = 4 * \text{MaxHebras}$.

En la sección 7.2 se muestran los resultados del algoritmo aquí descrito (B2F) y se comparan con la versión inicial de este algoritmo en la que no se aplicaba la condición 6.1 para la generación del conjunto inicial de símplices de la fase 1.

6.2.2. Versión paralela con creación dinámica de hebras (CDH)

En la versión paralela B2F puede haber diferencias en la carga de trabajo (no conocida de antemano) asignada a cada hebra, debido a los diferentes tamaños que pueden tener las ramas del árbol y al diferente número de símplices hijos similares en cada subárbol. Para tener un grado mayor de adaptabilidad a la carga computacional en tiempo de ejecución, se va a seguir la estrategia en la paralelización de algoritmos irregulares realizadas en trabajos anteriores [8, 30, 31, 32, 57, 58].

El usuario establece el número máximo de hebras (`MaxHebras`) que normalmente es igual al número de procesadores del sistema. Existe una variable compartida que determina el número de hebras que están trabajando actualmente (`NHebras`). Las hebras son creadas por otras hebras siempre que el valor de `NHebras` sea menor a `MaxHebras` y cuando sea posible realizar trabajo en paralelo. Una hebra al terminar su trabajo devuelve los resultados a su hebra madre, decrementa el valor de `NHebras` y muere.

Las posibilidades de crear una nueva hebra están en la evaluación de la división de varios lados mayores en el símplice actual y en la evaluación de los dos subsímplices generados por división. Debido a que el número de símplices con un solo lado mayor puede ser un porcentaje alto del total, se usarán las dos posibilidades para la creación de hebras.

Los algoritmos 9 y 10 son extensiones con hebras dinámicas de los algoritmos 4 y 5, respectivamente, los cuales se ejecutarían en una hebra. La función `CreaHebra`, genera una nueva hebra y `ResultadoHebra` obtiene el resultado de la hebra creada. La hebra actual solo generará una nueva hebra si el número de hebras es menor que `MaxHebras` y existe trabajo para la hebra actual y la nueva. Una hebra decrementa el valor de `NHebras` si solo le queda esperar los resultados de las hebras que ha creado.

Algoritmo 9 TamañoÁrbolH (S, L, ϵ)

Require: S: símplice, L: lado mayor, ϵ: precisión

1: **if** $w(S) \leq \epsilon$ **then**
2:    **return** 1
3: **end if**
4: $\{S_1, S_2\}$=DivideSímplice(S,L)
5: hebra := false
6: **if** NHebras < MaxHebras **then**
7:    CreaHebra(TamañoÁrbolMínimoH(S_1))
8:    hebra := true
9: **end if**
10: r_2 := TamañoÁrbolMínimoH(S_2)
11: **if** hebra **then**
12:    r_1 := ResultadoHebra()
13: **else**
14:    r_1 := TamañoÁrbolMínimoH(S_1)
15: **end if**
16: **return** $r_1 + r_2$

En el algoritmo 9 se crea una hebra para procesar el subsímplice S_1 mientras la hebra actual evalúa el símplice S_2. En el caso de que ambos fuesen simétricos, no se crearía una nueva hebra y la hebra actual evaluaría solo S_2, devolviendo como resultado el doble del tamaño del subárbol generado a partir de S_2.

Algoritmo 10 TamañoÁrbolMínimoH (S, ϵ)

Require: S:símplice, ϵ: criterio de terminación

0: Inserta cada lado mayor $L_i \in S$ en Λ
1: **while** $|\Lambda| > 0$ **do**
2:    **while** NHebras < MaxHebras **and** $|\Lambda| > 1$ **do**
3:       Extrae L_i de Λ
4:       CreaHebra$_i$(TamañoÁrbolH(S,L_i,ϵ))
5:    **end while**
6:    Extrae L_i de Λ
7:    r_i := TamañoÁrbolH(S,L_i,ϵ)
8: **end while**
9: **for** cada hebra$_i$ creada **do**
10:    r_i := ResultadoHebra$_i$()
11: **end for**
12: **return** $\min_i\{r_i\}$

El algoritmo 10 trata de repartir el trabajo de evaluar las distintas posibilidades de división del lado mayor entre las posibles nuevas hebras. Se deja la evaluación de la división de un lado mayor para la propia hebra. Si no se pueden crear hebras, la propia hebra debe encargarse de realizar el trabajo (línea 7 del algoritmo 10). Una vez realizado el trabajo,

la hebra espera los resultados de las hebras que creó (línea 10). Este tiempo de espera puede limitar la ganancia de velocidad del algoritmo.

La Figura 6.4 muestra un esquema del proceso de ejecución de este algoritmo. Su funcionamiento es muy similar al expuesto en la sección 6.1 de este mismo capítulo, aplicando la división y el reparto en subproblemas que se acaba de exponer.

Como se observa en la imagen el trabajo es dividido y repartido entre los distintos procesadores y estos, de forma paralela, lo pueden seguir repartiendo mientras existan procesadores libres. Por otra parte cuando un procesadores termina su trabajo se lo indica al resto y entonces alguno del resto puede volver a dividir y repartir trabajo y asignárselo al procesador ocioso. Hay que señalar que cuando la distancia del lado mayor es inferior a $1{,}35 \cdot \epsilon$, el algoritmo deja de crear hebras por esa rama, ya que la rama está cerca de finalizar su evaluación. Esto evita que se reparta el trabajo cuando queda poco para llegar al final.

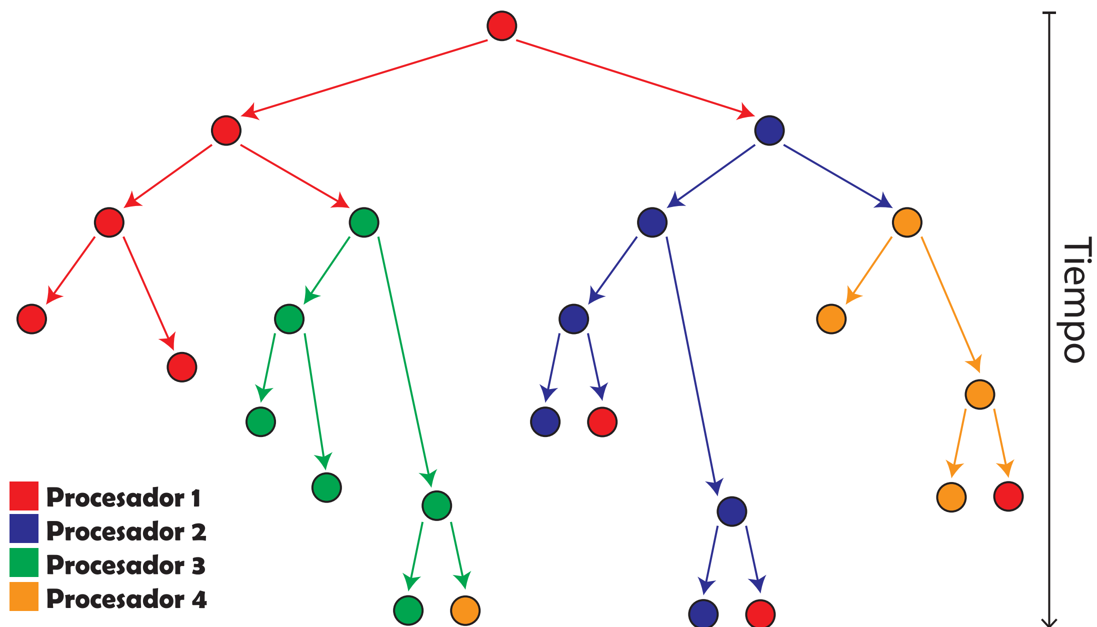


Figura 6.4: Esquema del algoritmo CDH (MaxHebras=4).

Capítulo 7

Resultados

Los algoritmos se han codificado en C/C++, usando *Pthreads API* para las versiones multihebradas y han sido ejecutados en un nodo de *BullX-UAL*, que consiste en dos procesadores Intel® Xeon® E5-2650 de 8 cores a 2,00 GHz y 64 GB de RAM. Todos los algoritmos presentados realizan un elevado número de operaciones de gestión dinámica de la memoria, principalmente en asignar memoria a los nuevos símplices y en su liberación. La liberación de la memoria asignada a un símplice se realiza normalmente en la vuelta de la recursividad. Se ha usado la librería *TCMalloc* de *Google Performance Tools*, ya que mejora considerablemente los tiempos de ejecución, tanto secuenciales como paralelos.

7.1. Versión secuencial

La tabla 7.1 muestra el tamaño mínimo del árbol (`Árbol mínimo`), el tamaño máximo del árbol (`Árbol máximo`), el número de símplices evaluados para obtener el tamaño del árbol mínimo (`Símplices evaluados`) y el tiempo requerido en segundos por el algoritmo secuencial para obtener el tamaño del árbol mínimo (`Tiempo mínimo`) para un 3-símplice y distintos valores de ϵ. Una buena elección del lado mayor a dividir puede reducir considerablemente el número de símplices evaluados respecto al peor caso.

Tabla 7.1: Resultados secuenciales para un 3-símplice

ϵ	Árbol mínimo	Árbol máximo	Símplices evaluados	Tiempo mínimo
0.025	856.103	976.415	$22{,}5 \cdot 10^6$	27,8
0.01	10.835.455	12.176.711	$492 \cdot 10^6$	591,7

La tabla 7.2 muestra los resultados secuenciales análogos a los de la tabla 7.1 pero para un 4-símplice y distintos valores de ϵ, que se han establecido para poder obtener resultados en unos tiempos razonables.

Tabla 7.2: Resultados secuenciales para un 4-símplice

ϵ	Árbol mínimo	Árbol máximo	Símplices evaluados	Tiempo mínimo
0,2	18.103	27.675	$24{,}9 \cdot 10^6$	40,8
0,125	55.455	122.911	$181.7 \cdot 10^6$	297,9

De nuevo, una buena elección del lado mayor a dividir puede reducir considerablemente el número de símplices evaluados respecto al peor caso.

7.2. Versión paralela basada en dos fases (B2F)

En la versión B2F el número de símplices en la etapa inicial `NIniS` debe ser mayor que `MinNIniS`, pero para que su número sea cercano a un múltiplo de `MaxHebras` se hace cumplir la condición 6.1. El objetivo es lograr distribuir la carga de trabajo lo máximo posible entre los procesadores.

7.2.1. Resultados de B2F para un 3-símplice

Se comienza buscando el mejor número de símplices iniciales con el objetivo de lograr una buena distribución del trabajo entre los procesadores. Para ello primero se comprobaran los resultados del algoritmo B2F para un conjunto inicial de símplices fijo, para posteriormente variar el número de símplices en este conjunto inicial y comprobar si la distribución de la carga mejora el rendimiento.

7.2.1.1. Resultados B2F

La tabla 7.3 muestra el tiempo de ejecución del algoritmo paralelo B2F para un 3-símplice y $\epsilon = 0{,}025$. Las columnas `Tiempo B2F` y `SpeedUp B2F` muestran los tiempos de ejecución en segundos y la ganancia en velocidad comparada con la versión secuencial, respectivamente. Puede observarse que la ganancia de velocidad es baja. Una de las causas puede ser el desbalanceo de la carga computacional entre las hebras. Desde el punto de vista paralelo, este tipo de algoritmos son irregulares ya que no se sabe de antemano la carga computacional de un subproblema o subsímplice.

Tabla 7.3: Tiempos y ganancia en velocidad de la versión B2F para un 3-símplice y $\epsilon = 0{,}025$

MaxHebras	Tiempo B2F	SpeedUp B2F
1	27,7	1,00
2	19,5	1,43
4	13,6	2,00
8	8,1	3,43
16	4,3	6,47

La tabla 7.4 muestra el desbalanceo producido por el algoritmo B2F en las condiciones de la tabla 7.3. `NIniS` indica el número de símplices iniciales calculado según (6.1), `Med.EvalS` indica el número medio de símplices evaluados por las hebras y `Desv.` indica el intervalo con el porcentaje de desviación observado respecto a la media. Puede observarse como los subsímplices generados en la etapa inicial tienen unos costes computaciones con grandes diferencias.

Tabla 7.4: Desbalanceo de la versión B2F para un 3-símplice y $\epsilon = 0{,}025$

MaxHebras	NIniS	Med.EvalS	Desv.
1	1	$22{,}5 \cdot 10^6$	
2	6	$11{,}3 \cdot 10^6$	[-12,9 %, 12,9 %]
4	8	$5{,}6 \cdot 10^6$	[-22,9 %, 40,7 %]
8	8	$2{,}8 \cdot 10^6$	[-48,5 %, 54,1 %]
16	30	$1{,}4 \cdot 10^6$	[-27,7 %, 67,2 %]

7.2.1.2. Resultados B2F+

Un método para intentar un mejor reparto de la carga entre las hebras es incrementar el número de símplices iniciales, este método se denominará B2F+. Se ha determinado experimentalmente un valor de `MinNIniS = 4` $\times$ `MaxHebras`^{2} en (6.1).

La tabla 7.5 muestra los mismos datos que la tabla 7.3, pero con el nuevo valor de `MinNIniS`. Puede observarse que la ganancia en velocidad ha mejorado aunque no llega a ser lineal.

Tabla 7.5: Análogo a la tabla 7.3 usando MinNIniS $= 4 \cdot$ MaxHebras^2 en (6.1)

MaxHebras	Tiempo B2F+	SpeedUp B2F+
1	27,5	1,01
2	20,5	1,36
4	12,1	2,30
8	6,1	4,56
16	2,5	11,12

La tabla 7.6 muestra los mismos datos que la tabla 7.6, pero con el nuevo valor de `MinNIniS`. Se puede apreciar que el desbalanceo de la carga ha bajado considerablemente. Aunque se ha intentado obtener un valor de `MinNIniS` dependiente del número de hebras, se ha observado que pequeñas variaciones en su valor afectan significativamente al rendimiento del algoritmo paralelo, debido a su irregularidad.

Tabla 7.6: Análogo a la tabla 7.4 usando MinNIniS=4$\times$MaxHebras^2 en (6.1)

MaxHebras	NIniS	Med.EvalS	Desv.
1	8	$22{,}5 \cdot 10^6$	
2	30	$11{,}3 \cdot 10^6$	[-1,5 %, 1,5 %]
4	64	$5{,}6 \cdot 10^6$	[-4,1 %, 4,3 %]
8	464	$2{,}8 \cdot 10^6$	[-1,5 %, 2,2 %]
16	1844	$1{,}4 \cdot 10^6$	[-2,7 %, 2,6 %]

7.2.1.3. Comparación de las alternativas para B2F

La tabla 7.7 compara los resultados obtenidos en las tablas 7.3 y 7.5. En la tabla se muestran los tiempos en segundos y los speedups obtenidos con respecto a la versión secuencial, empleado la distribución de trabajo descrita en la sección 7.2.1.2 y sin emplear esta distribución (sección 7.2.1.1).

Tabla 7.7: Comparación de tiempos y ganancia en velocidad de la versión B2F para un 3-símplice y $\epsilon = 0{,}025$ según el número de símplices iniciales.

MaxHebras	Tiempo B2F+	SpeedUp B2F+	Tiempo B2F	SpeedUp B2F
1	27,5	1,01	27,7	1,00
2	20,5	1,36	19,5	1,43
4	12,1	2,30	13,6	2,00
8	6,1	4,56	8,1	3,43
16	2,5	11,12	4,3	6,47

La figura 7.1 muestra los resultados de forma gráfica. Como se observa los resultados para 1, 2 y 4 procesadores son muy similares y es solo para un mayor número de procesadores cuando el mayor número de subproblemas implica una ganancia de velocidad.

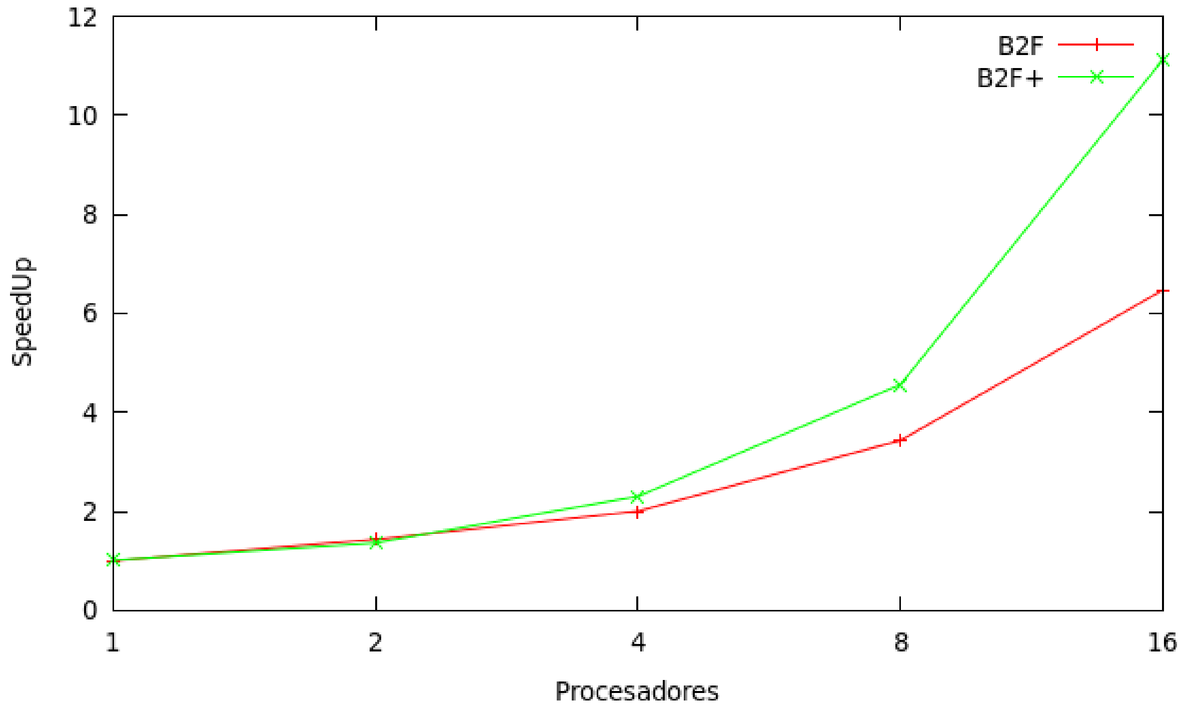


Figura 7.1: Gráfica comparativa de SpeedUp según la carga inicial para B2F.

Por ultimo, la tabla 7.8 y la figura 7.2 comparan el desbalanceo de las ramas para ambos casos, tablas 7.4 y 7.6. Se observa una clara mejora al aumentar el número de símplices de la fase inicial, ya que el desbalanceo se reduce drásticamente.

Tabla 7.8: Desbalanceo de la versión B2F para un 3-símplice y $\epsilon = 0{,}025$ según el número inicial de símplices.

MaxHebras	NIniS B2F+	Desv. B2F+	NIniS B2F	Desv. B2F
1	8		1	
2	30	[-1,5%, 1,5%]	6	[-12,9%, 12,9%]
4	64	[-4,1%, 4,3%]	8	[-22,9%, 40,7%]
8	464	[-1,5%, 2,2%]	8	[-48,5%, 54,1%]
16	1.844	[-2,7%, 2,6%]	30	[-27,7%, 67,2%]

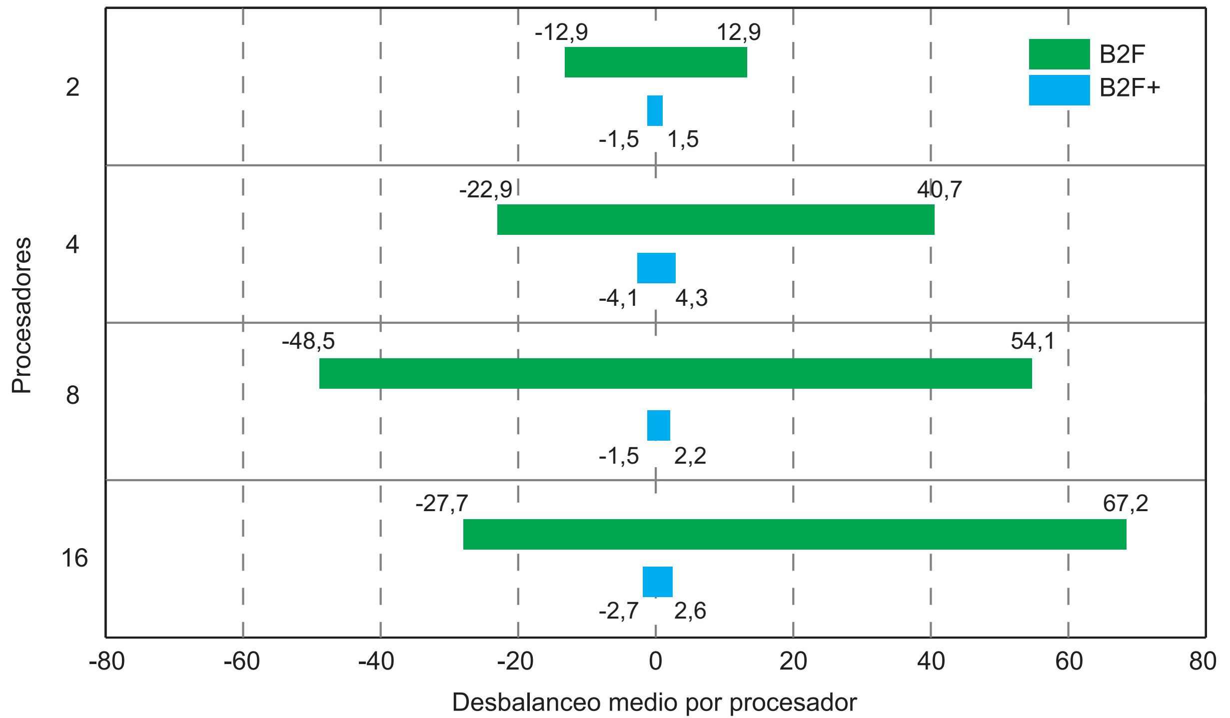


Figura 7.2: Gráfica comparativa de balanceo de ramas según la carga inicial para B2F.

Tras comprobar que se obtienen mejores resultados empleando una esta nueva distribución inicial de la carga, este será el método empleado B2F+ a partir de aquí.

Las tablas 7.9 y 7.10 son análogas a las tablas 7.5 y 7.6 pero para $\epsilon = 0{,}01$. Puede observarse un comportamiento similar del algoritmo.

Tabla 7.9: Análogo a la tabla 7.5 usando $\epsilon = 0{,}01$

MaxHebras	Tiempo B2F+	SpeedUp B2F+
1	590,9	1,00
2	462,8	1,28
4	246,7	2,40
8	115,8	5,11
16	50,7	11,67

Tabla 7.10: Análogo a la tabla 7.6 usando $\epsilon = 0{,}01$

MaxHebras	NIniS	Med.EvalS	Desv.
1	8	$492{,}3 \cdot 10^6$	
2	30	$246{,}1 \cdot 10^6$	[-1,3 %, 1,3 %]
4	64	$123{,}1 \cdot 10^6$	[-5,7 %, 3,4 %]
8	464	$61{,}5 \cdot 10^6$	[-1,6 %, 2,4 %]
16	1.844	$30{,}8 \cdot 10^6$	[-1,1 %, 1,7 %]

7.2.2. Resultados de B2F+ para un 4-símplice

Se han repetido los experimentos con el algoritmo B2F+ para un 4-símplice con los resultados mostrados a continuación:

Tabla 7.11: Análogo a la tabla 7.5 para un 4-símplice y $\epsilon = 0{,}2$

MaxHebras	Tiempo B2F+	SpeedUp B2F+
1	40,8	1,00
2	28,1	1,45
4	16,1	2,53
8	7,3	5,59
16	3,4	12,00

Tabla 7.12: Análogo a la tabla 7.6 para un 4-símplice y $\epsilon = 0{,}2$

MaxHebras	NIniS	Med.EvalS	Desv.
1	48	$24{,}9 \cdot 10^6$	
2	29	$12{,}5 \cdot 10^6$	[-0,8 %, 0,8 %]
4	64	$6{,}2 \cdot 10^6$	[-2,9 %, 5,2 %]
8	462	$3{,}1 \cdot 10^6$	[-2,9 %, 2,9 %]
16	1.844	$1{,}6 \cdot 10^6$	[-2,4 %, 2,6 %]

Tabla 7.13: Análogo a la tabla 7.11 usando $\epsilon = 0{,}125$

MaxHebras	Tiempo B2F+	SpeedUp B2F+
1	304,0	0,98
2	213,8	1,39
4	107,1	2,78
8	50,6	5,89
16	23,6	12,62

Tabla 7.14: Análogo a la tabla 7.12 usando $\epsilon = 0{,}125$

MaxHebras	NIniS	Med.EvalS	Desv.
1	48	$181{,}7 \cdot 10^6$	
2	29	$90{,}9 \cdot 10^6$	[-2,6%, 2,6%]
4	64	$45{,}4 \cdot 10^6$	[-1,5%, 2,2%]
8	462	$22{,}7 \cdot 10^6$	[-3,2%, 1,9%]
16	1.844	$11{,}7 \cdot 10^6$	[-1,4%, 1,1%]

Las tablas 7.11–7.14 muestran que el algoritmo B2F+ tiene un comportamiento similar cuando evalúa un 3-símplice y un 4-símplice.

7.3. Versión paralela con creación dinámica de hebras (CDH)

En la versión CDH se comienza con el símplice inicial en una hebra y se van creando hebras de forma dinámica según se vayan necesitando, véase punto 6.2.2. El objetivo es lograr distribuir el trabajo de forma dinámica según se genera el árbol binario completo.

7.3.1. Resultados de CDH para un 3-símplice

La tabla 7.15 muestra el tiempo de ejecución del algoritmo paralelo CDH para un 3-símplice y $\epsilon = 0{,}025$. Las columnas `Tiempo CDH` y `SpeedUp CDH` muestran los tiempos de ejecución y la ganancia en velocidad comparada con la versión secuencial, respectivamente.

Puede observarse que la ganancia de velocidad es baja. Como se explicó en el capítulo anterior, el algoritmo CDH realiza un balanceo implícito de la carga por lo que la causa de la baja ganancia en velocidad es el retardo introducido en la espera de resultados por parte de las hebras hijas. Desde el punto de vista paralelo, este tipo de algoritmos son irregulares ya que no se sabe de antemano la carga computacional de un subproblema o subsímplice.

Tabla 7.15: Tiempos y ganancia en velocidad de la versión CDH para un 3-símplice y $\epsilon = 0{,}025$

MaxHebras	Tiempo CDH	SpeedUp CDH
1	27,7	1,00
2	15,2	1,83
4	9,1	3,05
8	6,0	4,63
16	5,6	4,96

La tabla 7.16 es análoga a la tabla 7.15 pero para $\epsilon = 0{,}01$. Puede observarse un comportamiento similar del algoritmo.

Tabla 7.16: Análogo a la tabla 7.15 usando $\epsilon = 0{,}01$

MaxHebras	Tiempo CDH	SpeedUp CDH
1	628,5	0,94
2	344,2	1,72
4	203,5	2,91
8	154,2	3,84
16	131,5	4,50

7.3.2. Resultados de CDH para un 4-símplice

Se han repetido los experimentos con el algoritmo CDH para un 4-símplice con los resultados mostrados a continuación:

Tabla 7.17: Tiempos y ganancia en velocidad de la versión CDH para un 4-símplice y $\epsilon = 0{,}2$

MaxHebras	Tiempo CDH	SpeedUp CDH
1	40,1	1,02
2	20,8	1,96
4	11,5	3,55
8	6,3	6,48
16	4,1	9,95

Tabla 7.18: Análogo a la tabla 7.17 usando $\epsilon = 0{,}125$

MaxHebras	Tiempo CDH	SpeedUp CDH
1	282,7	1,05
2	150,9	1,97
4	89,5	3,33
8	48,1	6,20
16	47,6	6,26

Las tablas 7.17 y 7.18 muestran que el algoritmo CDH tiene un comportamiento similar cuando evalúa un 3-símplice y un 4-símplice. Aunque para el caso del 4-símplice la ganancia de velocidad es algo mayor, esto es algo positivo pues significa que el comportamiento mejora cuando se incrementa la cantidad de trabajo.

7.4. Comparación de los resultados de las distintas versiones

A continuación se presenta la comparación de los resultados obtenidos aplicando los distintos algoritmos propuestos a los mismos conjuntos de datos iniciales.

7.4.1. Comparación de los resultados para un 3-símplice

La tabla 7.19 muestran los distintos tiempos y ganancias de velocidad para los algoritmos propuestos para un valor de épsilon de 0,025. Se observa como la versión CDH obtiene mejores resultados para 2, 4 y 8 procesadores, y es a partir de 16 procesadores cuando el reparto inicial del trabajo del algoritmo B2F+ marca la diferencia. Se han tomado los tiempos con incremento inicial de la carga para el algoritmo B2F (Véase B2F+ en la sección 7.2.1.2).

Tabla 7.19: Tiempos y ganancias de velocidad para los algoritmos propuestos para un 3-símplice con $\epsilon = 0,025$

MaxHebras	T.Sec	T.B2F+	S.B2F+	T.CDH	S.CDH
1		27,5	1,01	27,7	1,00
2		20,5	1,36	15,2	1,83
4	27,8	12,1	2,30	9,1	3,05
8		6,1	4,56	6,0	4,63
16		2,5	11,12	5,6	4,96

La figura 7.3 representa de manera gráfica los tiempos mostrados en la tabla 7.19 para los distintos algoritmos propuestos.

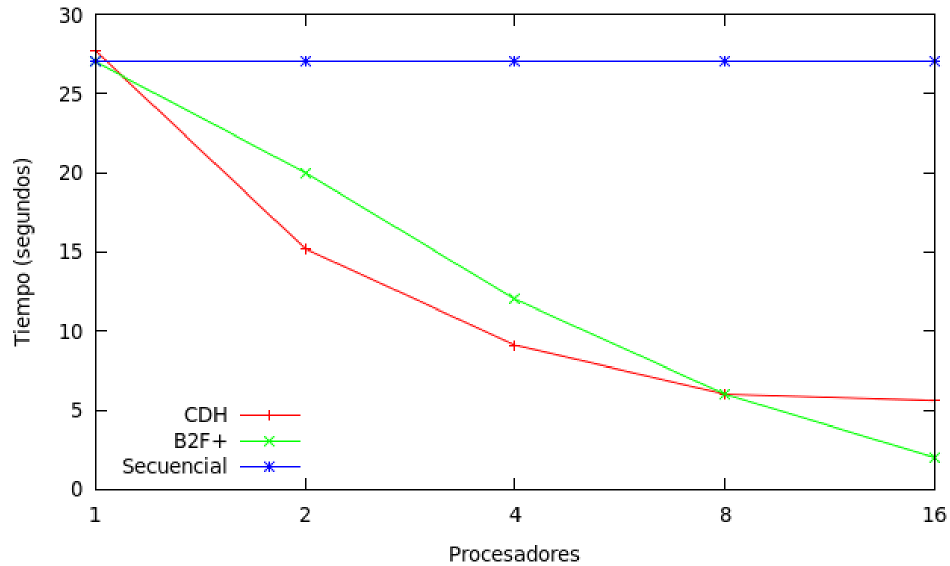


Figura 7.3: Gráfica comparativa de tiempos para los algoritmos propuestos para un 3-símplice con $\epsilon = 0,025$

La figura 7.4 muestra la comparación de ganancia de velocidad de los distintos algoritmos con los valores de la tabla 7.19.

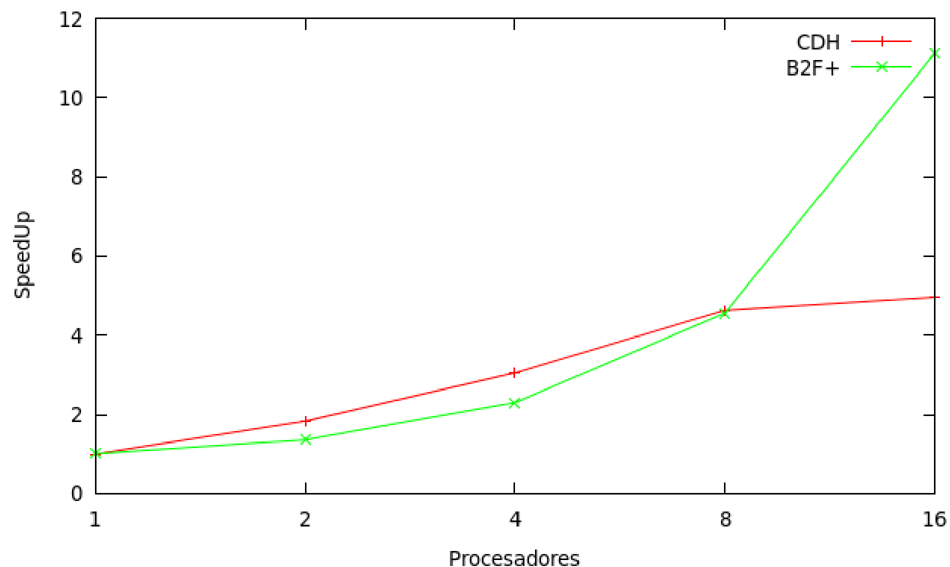


Figura 7.4: Gráfica comparativa de speedup para los algoritmos propuestos para un 3-símplice con $\epsilon = 0,025$

Se muestran los mismos resultados para los experimentos realizados con valor de épsilon de 0,01. Los resultados son similares a los de la tabla 7.19, aunque en este caso las diferencias entre los algoritmos son mayores.

Tabla 7.20: Tiempos y ganancias de velocidad para los algoritmos propuestos para un 3-símplice con $\epsilon = 0,01$

MaxHebras	T.Sec	T.B2F	S.B2F	T.CDH	S.CDH
1		590,9	1,00	628,5	0,94
2		462,8	1,28	344,2	1,72
4	591,7	246,7	2,40	203,5	2,91
8		115,8	5,11	154,2	3,84
16		50,7	11,67	131,5	4,50

La figura 7.5 representa de manera gráfica los tiempos mostrados en la tabla 7.20 para los distintos algoritmos propuestos.

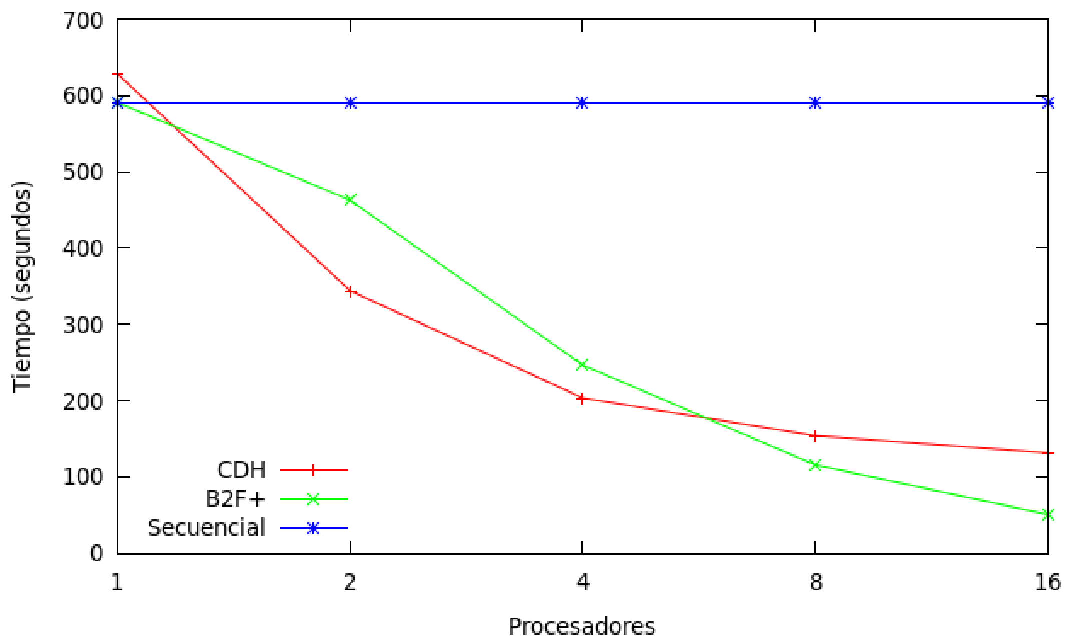


Figura 7.5: Gráfica comparativa de tiempos para los algoritmos propuestos para un 3-símplice con $\epsilon = 0,01$

La figura 7.6 muestra la comparación de ganancia de velocidad de los distintos algoritmos con los valores de la tabla 7.20.

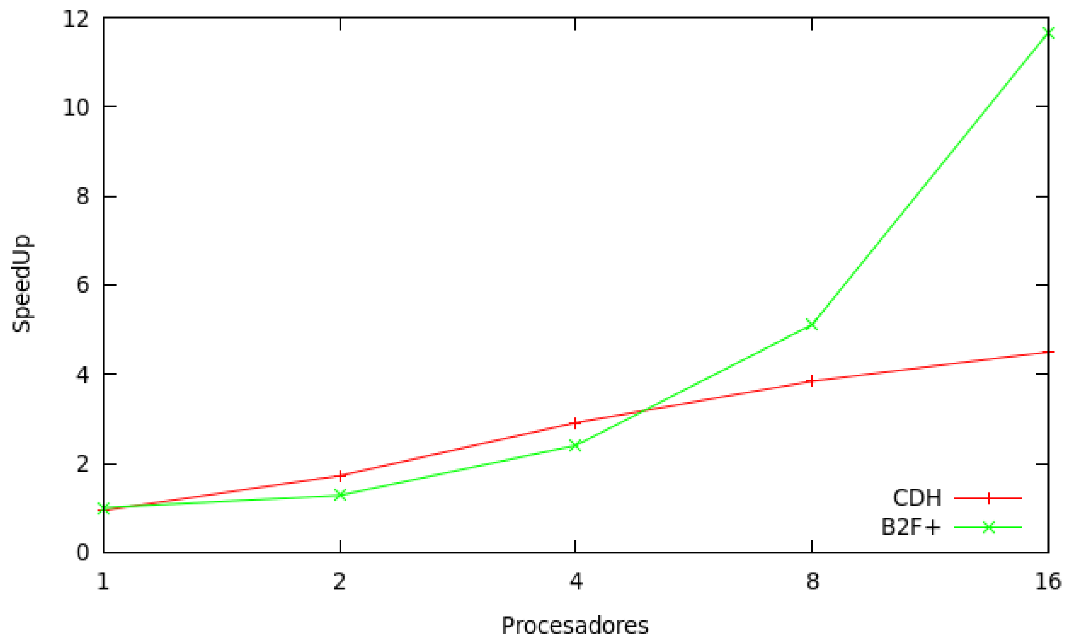


Figura 7.6: Gráfica comparativa de speedup para los algoritmos propuestos para un 3-símplice con $\epsilon = 0,01$

7.4.2. Comparación de los resultados para un 4-símplice

Se han repetido las comparaciones de las distintas versiones para un 4-símplice con los resultados mostrados a continuación.

Se observa como a diferencia de los resultados para un 3-símplice en el que existían mayores diferencias entre las versiones en el caso de un 4-símplice los resultados de ambas versiones están más igualados.

Tabla 7.21: Tiempos y ganancias de velocidad para los algoritmos propuestos para un 4-símplice con $\epsilon = 0,2$

MaxHebras	T.Sec	T.B2F+	S.B2F+	T.CDH	S.CDH
1		40,8	1,00	40,1	1,02
2		28,1	1,45	20,8	1,96
4	40,8	16,1	2,53	11,5	3,55
8		7,3	5,59	6,3	6,48
16		3,4	12,00	4,1	9,95

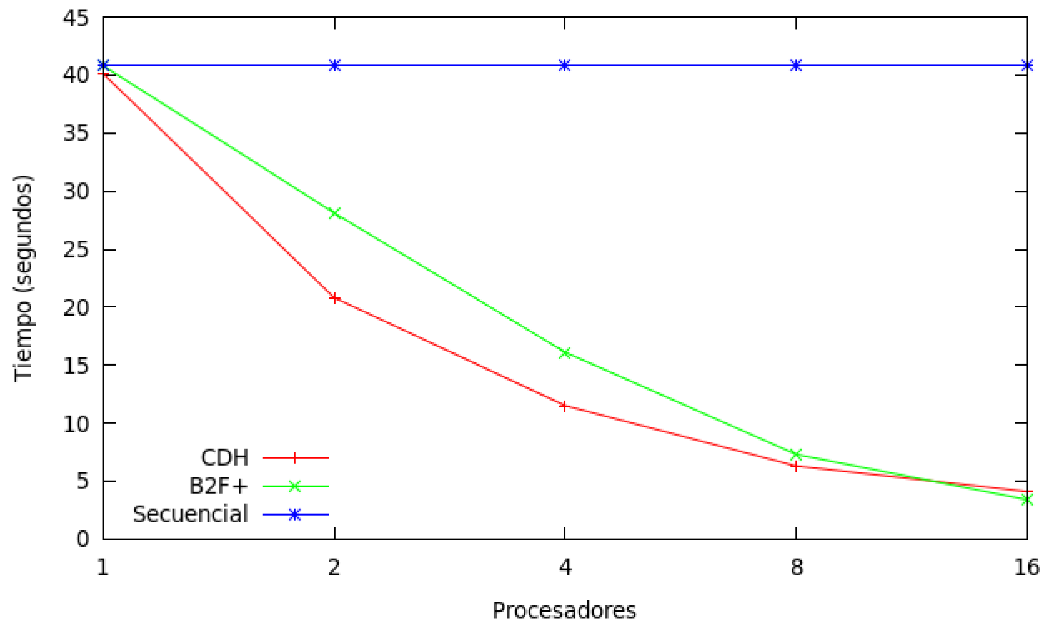


Figura 7.7: Gráfica comparativa de tiempos para los algoritmos propuestos para un 4-símplice con $\epsilon = 0,2$

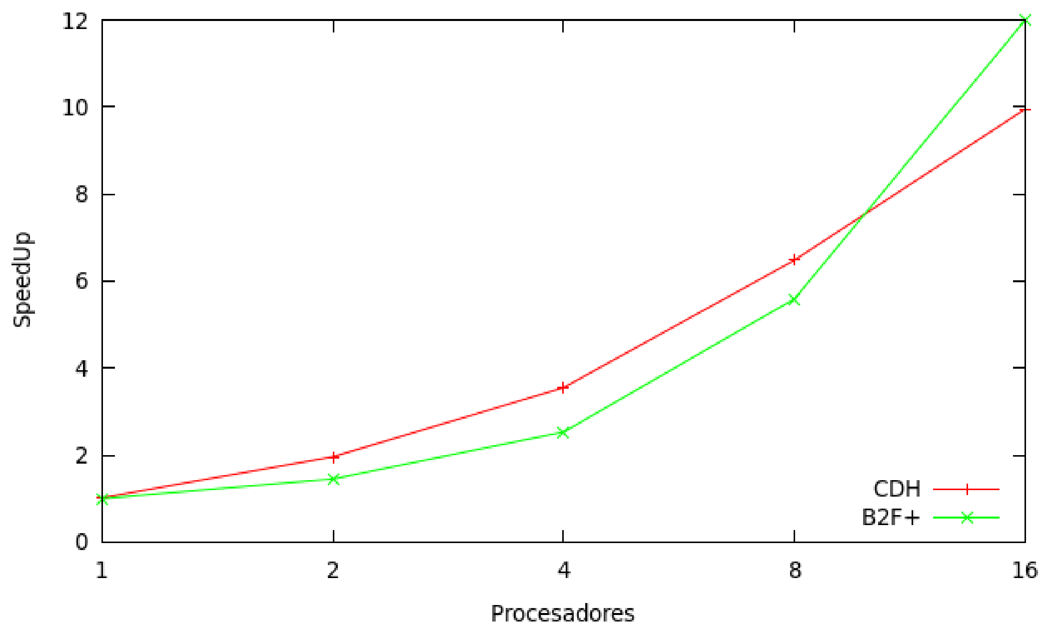


Figura 7.8: Gráfica comparativa de speedup para los algoritmos propuestos para un 4-símplice con $\epsilon = 0,2$

Tabla 7.22: Tiempos y ganancias de velocidad para los algoritmos propuestos para un 4-símplice con $\epsilon = 0,125$

MaxHebras	T.Sec	T.B2F+	S.B2F+	T.CDH	S.CDH
1		304,0	0,98	282,7	1,05
2		213,8	1,39	150,9	1,97
4	297,9	107,1	2,78	86,5	3,33
8		50,6	5,89	48,1	6,20
16		23,6	12,62	47,6	6,26

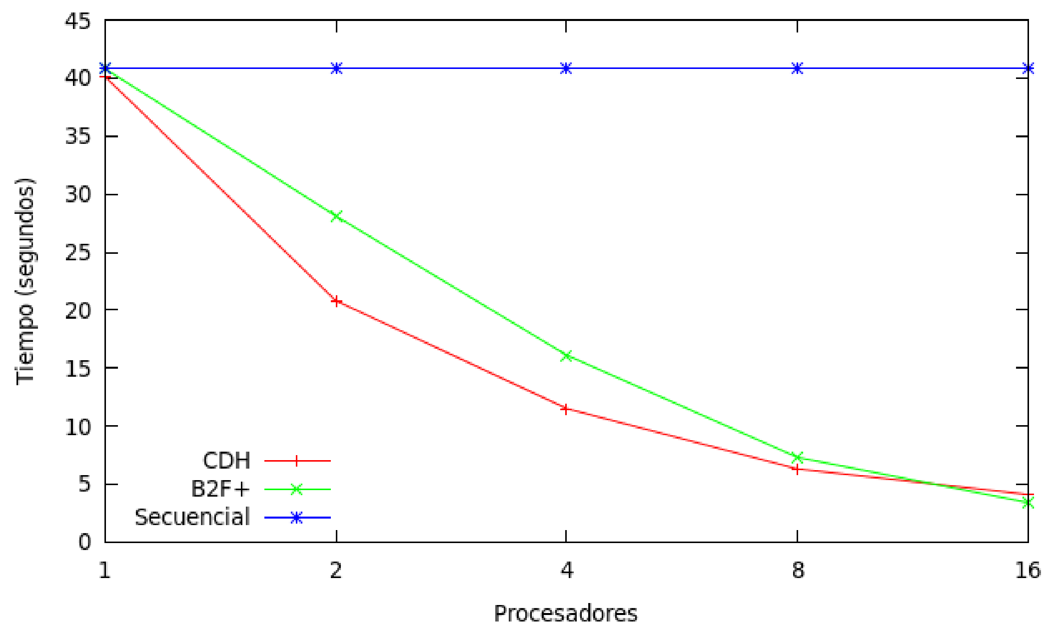


Figura 7.9: Gráfica comparativa de tiempos para los algoritmos propuestos para un 4-símplice con $\epsilon = 0,125$

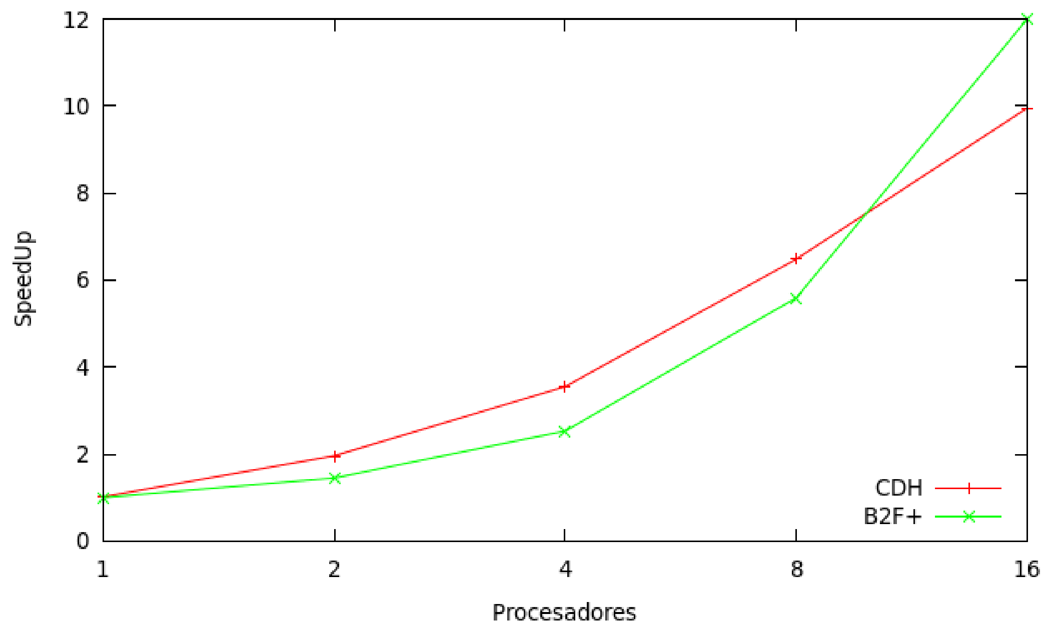


Figura 7.10: Gráfica comparativa de speedup para los algoritmos propuestos para un 4-símplice con $\epsilon = 0,125$

Como se observa en las tablas y gráficas anteriores los resultados de las versiones CDH y B2F+ están mucho más igualados que en el caso del 3-símplice. Esto se debe fundamentalmente a que la mayor carga de trabajo enmascara las diferencias debidas a la espera de las hebras en el algoritmo CDH.

Capítulo 8

Secuencia de lados

En este capítulo se pretende estudiar si existen algún tipo de secuencia de lados que sirva para obtener el árbol binario completo mínimo sin necesidad de realizar los cálculos de lados y sin necesidad de evaluar todos los posibles subproblemas generados a partir de un símplice con varios lados mayores.

Tras el estudio del comportamiento de los símplices al generar el árbol mínimo se ha podido comprobar que suelen tener un comportamiento periódico según el nivel de profundidad en el que se encuentran. Por ello se comenzó a estudiar si existía la posibilidad de obtener una secuencia o patrón que indicara el camino a seguir para obtener el árbol mínimo de una forma mas rápida sin necesidad de buscar entre todos los posibles lados mayores [26].

Se ha aplicado el método expuesto en [26] y se ha intentado replicar el resultado expuesto para el caso de un 3-símplice y buscar el modo de aplicar el mismo método o una modificación del mismo para el caso de un 4-símplice.

8.1. Objetivo de la secuencia

El objetivo de obtener la secuencia de lados a dividir para obtener el árbol mínimo es el de evitar evaluar todos los posibles subproblemas generados por un símplice con varios lados mayores, además de evitar calcular las distancias entre los vértices para determinar el lado mayor. La secuencia escogerá el lado mayor que generará el menor número de símplices sin tener que calcular y comparar las distancias entre los lados. Con esto se evita tener que procesar todos los subproblemas, lo que reduce considerablemente el tiempo que necesita el algoritmo para obtener el árbol mínimo.

8.2. ¿Como obtener la secuencia de lados?

Para obtener la secuencia de lados se ha obtenido el árbol mínimo utilizando una versión modificada del programa secuencial explicado en el punto 6.1. Ahora cada llamada recursiva devuelve el número mínimo de símplices y que lado se ha dividido en cada símplice para obtener este número mínimo. Con estos datos, cuando un símplice tiene

varios lados mayores el programa se queda con el lado mayor que genera un subárbol con menor número de símplices y con la secuencia de lados mayores desde el símplice raíz hasta este.

El programa obtiene la secuencia de lados para el árbol mínimo. Por lo tanto se tendría un árbol en el que se indica el lado a dividir para cada símplice del mismo. Esto en árboles grandes es intratable, por lo que en este trabajo se está interesado en determinar una secuencia que dependa del nivel del árbol y del índice del símplice. La numeración de los símplices se muestra en la figura 6.1 del capítulo 6, donde se puede observar como los símplices comienzan a enumerarse en 1. La linea 3 del algoritmo 3 del capítulo 5 muestra el proceso de generación de los distintos números identificadores de los símplices.

La dificultad de simplificar la secuencia es debida a la existencia de varios lados mayores que generan el mismo árbol mínimo, por lo que una elección u otra cambiará la posible secuencia según el subárbol escogido. Para simplificar esta secuencia y poder reducirla a uno o dos índices de lados por nivel es necesario realizar una reordenación de los vértices que forman el símplice. Esta reordenación se explica en el algoritmo 11 y debe ser ejecutada en vez del algoritmo 3.

El algoritmo 11 es una versión modificada del algoritmo 3 y realiza la división del símplice S por el lado L y reordena los vértices de los hijos generados.

Algoritmo 11 BLM_reordenación(S, L)

Require: S: símplice, L: lado mayor
1: Generar nuevo vértice a partir de los vértices v_i, v_j de L: $v_{new} := \frac{v_i+v_j}{2}$
2: Crear dos nuevos símplices S_l, S_r
3: Asignar v_{new} como último vértice de S_l
4: Asignar v_{new} como último vértice de S_r
5: Copiar los vértices de S en S_l en el mismo orden saltando v_j
6: Copiar los vértices de S en S_r en el mismo orden saltando v_i
7: **return** S_l, S_r

8.3. Secuencia del 3-símplice

Tras ejecutar la versión modificada del algoritmo 3 a un 3-simplex se ha verificado la secuencia para la obtención del árbol mínimo sin necesidad de recorrer todas las ramas ni calcular las distancias entre los vértices (obtenida en [26]).

Tabla 8.1: Secuencia árbol mínimo 3-símplice

Nivel	Lado	Símplices
1	(0, 1)	
2	(1, 2)	
3	(0, 1)	
4	(0, 2)	pares
	(0, 1)	impares
5	(0, 1)	
6	(0, 1)	
7	(0, 1)	pares
	(0, 2)	impares
8	Nivel 5	
9	Nivel 6	
10	Nivel 7	

Para obtener esta secuencia es necesario estudiar todos los subárboles generados a partir de los símplices con varios lados mayores, ya que en algunos de los niveles existían distintas opciones de lado mayor con el mismo resultado. En estas situaciones se ha elegido el lado mayor tal que todos los símplices del mismo nivel y de niveles equivalentes concordasen.

8.3.1. Excepciones en la secuencia

El articulo en el que se basa este trabajo solamente expone el método usado y los resultados obtenidos para ciertos valores de ϵ. Pero tras realizar diferentes pruebas se ha llegado a la conclusión de que existen casos para los que la secuencia descrito anteriormente no es valida, pues usándola no se obtiene el árbol mínimo.

Estos son casos particulares en los que será necesario obtener el árbol mínimo de la forma tradicional ocurren con determinados valores de ϵ. Los valores de ϵ para los que está comprobado el correcto funcionamiento son $\frac{1}{2^n}$, para $n \leq 10$ (para n superiores no se ha comprobado debido al tamaño del problema resultante). Aunque se ha comprobado que para valores de $1 < \epsilon \leq 0{,}1$ al aplicar la secuencia se obtiene el árbol mínimo.

La tabla 8.2 muestra los casos para los que se ha logrado verificar el árbol mínimo aplicando la secuencia y los casos para los que no se ha logrado verificar la secuencia y no se obtiene el árbol mínimo. Para ello, la columna *Árbol mínimo* muestra los resultados obtenidos empleando los algoritmos diseñados en el capítulo 6 y la columna *Árbol secuencia* muestra los árboles resultantes tras incluir la secuencia a los algoritmos del capítulo 6. Se ha marcado en negrita las discrepancias entre valores de árboles mínimos.

Tabla 8.2: Valores de ϵ para los que se cumple la secuencia

Épsilon	Árbol mínimo	Árbol secuencia
$0,8$	31	31
$0,5$ $(1/2^1)$	47	47
$0,35$	351	351
$0,25$ $(1/2^2)$	351	351
$0,2$	1.727	1.727
$0,125$ $(1/2^3)$	2.751	2.751
$0,1$	13.695	13.695
$0,0625$ $(1/2^4)$	21.887	21.887
$0,05$	**108.799**	**109.311**
$0,03125$ $(1/2^5)$	174.847	174.847
$0,02$	**1.354.495**	**1.398.271**
$0,015625$ $(1/2^6)$	1.398.271	1.398.271
$0,01$	**10.835.455**	**11.185.151**
$0,0078125$ $(1/2^7)$	11.185.151	11.185.151
$0,005$	**86.682.623**	**89.479.167**
$0,00390625$ $(1/2^8)$	89.479.167	89.479.167
$0,002$	715.829.247	715.829.247
$0,001953125$ $(1/2^9)$	715.829.247	715.829.247
$0,001$	1.431.658.495	1.431.658.495
$0,000976563$ $(1/2^{10})$	1.431.658.495	1.431.658.495

8.3.2. Resultados aplicando la secuencia

Tras comprobar que la secuencia no funciona para todos los valores posibles de ϵ en esta sección se va comparar el rendimiento de los algoritmos del capítulo 6 con los mismos algoritmos pero generando el árbol mínimo establecido por la secuencia. Para ello se ha escogido una heurística en la que el primer lado mayor será el escogido.

La tabla 8.3 muestra la diferencia de tamaños de árbol y de tiempos de ejecución para distintos valores de ϵ. Las columnas *Tiempo PL* y *SEvals PL* muestra el tiempo de ejecución y el número de símplices evaluados para una heurística en que selecciona el primer lado mayor. Las columnas *Tiempo Sec* y *SEvals Sec* muestra la misma información pero aplicando la secuencia.

Tabla 8.3: Tiempos y tamaños de árbol según la heurística para un 3-símplice.

ϵ	Tiempo Sec	SEvals Sec	Tiempo PL	SEVals PL
0,015625	0,69	1.398.271	0,61	2.067.843
0,0078125	5,54	11.185.151	4,97	16.937.487
0,005	45,1	89.479.167	21,8	93.886.811
0,00195312	365,5	715.829.247	312,4	1.139.267.323

Como se puede observar el algoritmo que escoge el primer lado mayor es más rápido, pero el árbol resultante no es mínimo. Esto se debe a que este algoritmo no necesita realizar ningún calculo, simplemente divide por el primer lado, mientras que el algoritmo con la secuencia reordena los lados en cada iteración y comprueba por que lado se va a dividir aplicando la secuencia.

Capítulo 9

Conclusiones y principales aportaciones

En este proyecto fin de carrera se ha realizado un estudio de los algoritmos de Ramificación y Acotación y de los métodos de paralelización de esta clase de algoritmos.

El objetivo que se perseguía era el de diseñar un algoritmo que obtuviera el tamaño del árbol mínimo generado a partir de un símplice usando una división por el lado mayor. Y a partir de este algoritmo crear una versión paralela con una buena distribución de carga que obtuviese el valor de árbol mínimo más rápidamente.

Las principales cuestiones tratadas en este trabajo han sido:

- ¿Cual es el tamaño mínimo del árbol que se genera a partir de un determinado símplice?
- ¿Por que lado mayor es necesario dividir un símplice para obtener el árbol mínimo?
- ¿Es posible evitar el cálculo del tamaño de algunos de los subárboles que aparecen cuando un símplice tiene varios lados mayores?

Se han logrado contestar, en mayor o menor medida, todas las cuestiones. Aunque algunos puntos han podido quedar no resueltos estos estudios proveerán de una base para seguir intentado resolverlas por completo en trabajos futuros.

Respecto a la cuestión de obtener el árbol mínimo, es importante señalar que para un 2-símplice el tamaño del árbol siempre será mínimo (2-símplice no se tienen varias posibilidades para dividir un mismo símplice). Y es para valores de n superiores cuando esta pregunta cobra sentido, ya que es entonces cuando existen varios subárbol posibles. Los algoritmos planteados han demostrado obtener el árbol mínimo para los símplices con los que se ha experimentado. Estos algoritmos hacen uso una gran cantidad de computo, lo que ha obligado a emplear técnicas de paralelismo para acelerar las ejecuciones y ser capaces de abarcar problemas de mayor tamaño.

Por otra parte, la segunda pregunta solo ha sido resuelta para el caso de un 3-símplice, logrando replicar los resultados ofrecidos por el articulo [26]. Aunque la información contenida en el articulo no era completa, pues no mencionaba que la secuencia solamente

daba como resultado el árbol mínimo para ciertos valores de épsilon. En este aspecto este estudio ha logrado matizar los resultados del artículos, completándolos.

Se ha buscado el modo de aplicar esta secuencia, o una versión alterada, para obtener una versión equivalente para el caso de un 4-símplice, aunque todavía no ha sido posible obtenerla debido a la complejidad del problema.

Por ultimo, la pregunta de reducir las evaluaciones ha podido ser resuelta en gran parte. La complejidad de la implementación, debida a subproblemas dentro de subproblemas, ha imposibilitado una implementación funcional de la idea planteada en la sección 5.3.1. Esto deja una vía de mejora de los algoritmos aquí planteados. Aunque es cierto que se han planteado e implementado con existo otros elementos que han contribuido a reducir una gran cantidad del computo (véase capítulo 5 y 6).

Consideramos que el trabajo realizado en este proyecto es un primer paso de entrada a los problemas de Optimización Global y a los algoritmos de Ramificación y Acotación, que construye una base sobre la que se deberá seguir trabajando en el futuro. Sin se exhaustivos, nuestro interés a corto plazo se centrará en los siguientes aspectos:

- Desde el punto de vista de los algoritmos paralelos planteados en el capítulo 6 se deberá seguir trabajo en aumentar la ganancia de velocidad. Buscando técnicas alternativas que distribuyan mejor la carga o incluso adaptando los algoritmos a modelos distribuidos que puedan ampliar la capacidad de computo del sistema, lo que permitiría tratar problemas mayores.

- En el capítulo 8 se ha demostrado la importancia de una buena elección de los subárboles a evaluar. Por lo tanto un de los puntos a tratar en trabajos futuros sería, sin lugar a dudas, la obtención de nuevas secuencias que agilicen el proceso de evaluación de símplices de gran tamaño

- Por último, sería muy interesante la aplicación a problemas reales, científicos o industriales, de los métodos expuestos en este trabajo. Esto permitiría dar consistencia a los algoritmos planteados y contribuir en otros campos de investigación.

Bibliografía

[1] Andrew Adler. On the Bisection Method for Triangles. *Mathematics of Computation*, 40(162):571–574, 1983. doi: 10.1090/S0025-5718-1983-0689473-5.

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

[3] José L. Berenguel, L. G. Casado, I. García, and E. M. T. Hendrix. On estimating workload in interval branch-and-bound global optimization algorithms. *Journal of Global Optimization*, 56(3):821–844, 2013. doi: 10.1007/s10898-011-9771-5.

[4] José L. Berenguel, L.G. Casado, I. García, and E.M.T. Hendrix. On estimating workload in interval branch-and-bound global optimization algorithms. *Journal of Global Optimization*, In Press. DOI:10.1007/s10898-011-9771-5.

[5] I.M. Bomze, T. Csendes, R. Horst, and P.M. Pardalos, editors. *Developments in Global Optimization*, volume 18 of *Noncovex optimization and its applications*. Kluwer Academic Publishers, 1997.

[6] L. G. Casado, I. García, B. G.-Tóth, and E. M. T. Hendrix. On determining the cover of a simplex by spheres centered at its vertices. *Journal of Global Optimization*, 50(4):645–655, 2011. doi: 10.1007/s10898-010-9524-x.

[7] L. G. Casado, E. M. T. Hendrix, and I. García. Infeasibility spheres for finding robust solutions of blending problems with quadratic constraints. *Journal of Global Optimization*, 39(2):215–236, 2007. doi: 10.1007/s10898-007-9157-x.

[8] L.G. Casado, J.A. Martínez, I. García, and E.M.T. Hendrix. Branch-and-bound interval global optimization on shared memory multiprocessors. *Optimization Methods & Software*, 23(5):689–701, 2008. DOI:10.1080/10556780802086300.

[9] T. Cormen, C. Leiseron, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[10] R. Corrêa and A. Ferreira. On the effectiveness of synchronous branch-and-bound algorithms. *Parallel Processing Letters*, 5(3):375–386, 1995.

[11] R. Corrêa and A. Ferreira. Parallel best-first branch and bound in discrete optimization: a framework. In A. Ferreira and P.M. Pardalos, editors, *IRREGULAR '95, Solving Combinatorial Optimization Problems in Parallel – Methods and Techniques*, pages 145–170. Springer LNCS 1054, 1996.

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

[13] A. de Bruin, G.A.P. Kindervater, and H.W.J.M Trienekens. Towards an abstract parallel branch and bound machine. In A. Ferreira and P.M. Pardalos, editors, *IRREGULAR '95, Solving Combinatorial Optimization Problems in Parallel – Methods and Techniques*, pages 145–170. Springer LNCS 1054, 1996.

[14] C.G. Diderich and M. Gengler. Solving traveling salesman problems using a parallel synchronized branch and bound algorithm. In *International Conference and Exhibition on High-Performance Computing and Networking (HPCN Europe '96)*, pages 633–638, Brussels, Belgium, 1996. Springer-Verlag.

[15] L.G. Casado E.M.T Hendrix and P. Amaral. Global optimization simplex bisection revisited based on considerations by reiner horst. *Proceedings of ICCSA'12*, 7335:159–173, 2010.

[16] C.A. Floudas and P.M. Pardalos. *A collection Test Problems for Constrained Global Optimization Algorithms*, volume 455 of *Lectures Notes in Computer Science*. Springer, 1990.

[17] C.A. Floudas and P.M. Pardalos. *Recent Advances in Global Optimization*. Princeton series in Computer Science. Princeton, 1992.

[18] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comp.*, 1972.

[19] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[20] M.R. Garey and D.S. Johnson. *Computer and Intractability*. W.H. Freeman and Company, 1979.

[21] M. Gengler and G. Coray. A parallel best-first B&B algorithm and its axiomatization. *Journal of Parallel Algorithms and Applications*, 2:61–80, 1994.

[22] S. Gnesi, U. Montanari, and A. Martinelli. Dynamic programming as graph searching: An algebraic approach. *Journal of the ACM*, 28(4):737–751, 1981.

[23] A.Y. Grama and V. Kumar. A survey of parallel search algorithms for discrete optimization problems. *ORSA Journal of Computing*, 7(4):365–385, 1995.

[24] I.E. Grossmann. *Global Optimization in Engineering Design*. Kluwer Academic Publishers, 1996.

[25] Boglárka G-Tóth Eligius M. T. Hendrix Inmaculada García Guillermo Aparicio, Leocadio G. Casado. Heuristics to reduce the number of simplices in longest edge bisection refinement of a regular n-simplex. ICCSA '14, pages 115–125, University of Minho, Guimaraes, Portugal, 2014.

[26] Boglárka G-Tóth Eligius M. T. Hendrix Inmaculada García Guillermo Aparicio, Leocadio G. Casado. On the minimim number of simplex shapes in longest edge bisection refinement of a regular n-simplex. *Informatica*, 2014. In revision.

[27] Boglárka G-Tóth Eligius M. T. Hendrix Inmaculada García Guillermo Aparicio, Leocadio G. Casado. On the minimum number of simplices in a longest edge bisection refinement of a unit simplex. *Proceedings of the XII global optimization workshop. MATHEMATICAL AND APPLIED GLOBAL OPTIMIZATION. MAGO 2014. Málaga, September 2014*, pages 65–68, 2014.

[28] Eligius M. T. Hendrix Inmaculada Garcia Guillermo Aparicio, Leocadio G. Casado and Boglarka G.-Toth. On computational aspects of a regular n-simplex bisection. In *Proceedings of the 2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, 3PGCIC '13, pages 513–518, Compiegne, France, 2013. IEEE Computer Society.

[29] John L. Gustafson. Reevaluating amdahl's law. In *Reevaluating Amdahl's Law*, ACM New York, NY, USA, 1988. Communications of the ACM.

[30] J.F.R. Herrera, L.G. Casado, E.M.T. Hendrix, and I. García. A threaded approach of the quadratic bi-blending algorithm. *Journal of Supercomputing*, 64(1):38–48, 2013. DOI:10.1007/s11227-012-0783-9.

[31] J.F.R. Herrera, L.G. Casado, E.M.T. Hendrix, R. Paulavičius, and J. Žilinskas. Dynamic and hierarchical load-balancing techniques applied to parallel branch-and-bound methods. In *Proceedings of the 2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC 2013)*, pages 497–502. Compiegne, France, October 2013. DOI:10.1109/3PGCIC.2013.85.

[32] J.F.R. Herrera, L.G. Casado, R. Paulavičius, J. Žilinskas, and E.M.T. Hendrix. On a hybrid MPI-Pthread approach for simplicial branch-and-bound. In *Proceedings of the IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPS'13)*, pages 1764–1770. Boston, Massachusetts USA, May 2013. DOI:10.1109/IPDPSW.2013.178.

[33] R. Horst. On generalized bisection of n-simplices. *Mathematics of Computation*, 66(218):691–698, 1997. doi: 10.1090/S0025-5718-97-00809-0.

[34] R. Horst and P.M. Pardalos, editors. *Handbook of Global Optimization*, volume 2 of *Noncovex Optimization and its Applications.* Kluwer Academic Publishers, Dordrecht, Holland, 1995.

[35] T. Ibaraki. Theoretical comparisons of search strategies in branch and bound algorithms. *Int. J. Comput. Inform. Sci.*, 5:315–344, 1976.

[36] T. Ibaraki. On the computational efficiency of branch and bound algorithms. *J. Oper. Res. Soc.*, 20:16–35, 1977.

[37] T. Ibaraki. The power of dominance relations in branch and bound algorithms. *J. Assoc. Comput. Match.*, 24:264–279, 1977.

[38] T. Ibaraki. *Enumerative approaches to combinatorial optimization*, volume I and II. J.C. BALTZER AG, Basel-Switzerland, 1987.

[39] M. Kac and J. Logan. Fluctuation phenomena. In *Fluctuation Phenomena*, North-Holland, Ámsterdam, 1976. E.W. Montroll and J.L. Lebowitz.

[40] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, 1994.

[41] V. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing (special issue on scalability)*, 22(3):379–391, 1994.

[42] V. Kumar and L. Kanal. The CPD: a unifying formulation for heuristic search, dynamic programming and branch and bound. In *National Conference on Artificial Intelligence*, 1983.

[43] V. Kumar and L. Kanal. A general branch and bound formulation for understanding and synthesizing and/or tree search procedures. *Artificial Intelligence*, 21:179–198, 1983.

[44] E.L. Lawer and D.E. Wood. Branch and bound methods: A survey. *Operation Research*, 14:699–719, 1966.

[45] 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.

[46] R. Lüling, B. Monien, A. Reinefeld, and S. Tschöke. Mapping tree-structured combinatorial optimization problems onto parallel computers. In A. Ferreira and P.M. Pardalos, editors, *IRREGULAR '95, Solving Combinatorial Optimization Problems in Parallel – Methods and Techniques*. Springer LNCS 1054, 1996.

[47] L. G. Mitten. Branch and bound methods: general formulation and properties. *Operation Research*, 18:24–34, 1970.

[48] T. Morin and R. Marsten. Branch-and-bound strategies for dynamic programming. *Operations Research*, 24(4):611–627, 1976.

[49] D.S. Nau, V. Kumar, and L.N. Kanal. General branch and bound and its relation to A* and AO*. *Artificial Intelligence*, 23:29–58, 1984.

[50] E. Nelson. Quantum fluctuations. In *Quantum Fluctuations*, Princeton, 1985. Princeton University Press.

[51] P.M. Pardalos and J.B Rosen. *Constrained Global Optimization: Algorithms and Applications*. Lectures Notes in Computer Science. Springer, 1987.

[52] P.M. Pardalos and G. Schnitger. Checking local optimality in constrained quadratic programming is NP-hard. *Operations Research Letters*, 7(1):33–35, 1988.

[53] P.M. Pardalos and S.A. Vavasis. Quadratic programation with one negative eigenvalue is NP-hard. *Journal of Global Optimization*, 1(1):15–22, 1991.

[54] J. D. Pintér. *Global Optimization in Action*, volume 6 of *Noncovex optimization and its applications*. Kluwer Academic Publishers, Dordrecht, The Netherland, 1996.

[55] J.D. Pintér. Continuous global optimization software: A brief review. *Newsletter of the Mathematical Programming Society*, 52:1–8, 1996.

[56] M.J. Rossman, R.J. Twery, and F.D. Stone. A solution to the traveling salesman problem by combinatorial programming. Chicago, 1958.

[57] J. F. Sanjuan-Estrada, L. G. Casado, and I. García. Adaptive parallel interval branch and bound algorithms based on their performance for multicore architectures. *Journal of Supercomputing*, 58:376–384, 2011. doi: 10.1007/s11227-011-0594-4.

[58] J.F. Sanjuan-Estrada, L.G. Casado, I. García, and E.M.T Hendrix. Performance driven cooperation between kernel and auto-tuning multi-threaded interval B&B applications. In *Proceedings of ICCSA 2012*, volume 7333 of *Lecture Notes in Computer Science*, pages 57–70. Springer, Salvador de Bahia, June 18-21 2012. DOI:10.1007/978-3-642-31125-3_5.

[59] F. Schoen. Stochastic techniques for global optimization: A survey of recent advances. *Journal of Global Optimization*, 11(6):207–228, 1991.

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

[61] H. Trienekens. *Parallel Branch and Bound Algorithms*. PhD thesis, Erasmus University, Rotterdam, 1990.

[62] S. Tschöke, R. Lüling, and B. Monien. Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network. In *9th International Parallel Processing Symposium (IPPS '95)*, pages 182–189, ta Barbara, 1995.