UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE CIENCIAS FÍSICAS

Departamento de Arquitectura de Computadores y Automática

TÉCNICAS DE PARTICIÓN Y UBICACIÓN PARA SISTEMAS MULTI-FPGA BASADAS EN ALGORITMOS GENÉTICOS

MEMORIA PARA OPTAR AL GRADO DE DOCTOR
PRESENTADA POR

José Ignacio Hidalgo Pérez

Bajo la dirección del Doctor:

Juan Lanchares Dávila

Madrid, 2001

Técnicas de Partición y Ubicación para Sistemas Multi-FPGA basadas en Algoritmos Genéticos

José Ignacio Hidalgo Pérez

Departamento de Arquitectura de Computadores y Automática

Universidad Complutense de Madrid

e-mail: hidalgo@dacya.ucm.es

Memoria presentada para optar
al grado de Doctor en Ciencias Físicas
Director del Trabajo: Dr. D. Juan Lanchares Dávila

31 de Agosto de 2001

A Juan

Agradecimientos

El noventa por ciento de las tesis comienzan con una frase de algo así cómo *Este trabajo no se hubiera podido llevar a cabo sin la ayuda de....* Bueno, pues yo no voy a ser menos y voy a utilizar un poco de espacio de esta memoria para agradecer la ayuda y apoyo de todas las personas e instituciones que han colaborado en este trabajo. De paso aprovecharé para hacer gala de la exquisita educación y de la capacidades literarias con las que la naturaleza ha tenido a bien obsequiarme.

En primer lugar quiero expresar mi más profundo agradecimiento a mi director de tesis D. Juan Lanchares Dávila. A pesar de la excelente dirección que ha realizado de todo mi trabajo, de su apoyo en la docencia, de haber conseguido que me gane la vida haciendo lo que más me gusta, de haber buscado dinero para financiar los viajes a los congresos y materiales necesarios para poder llevar a cabo la investigación, de revisarse todos mis trabajos varias veces y de corregir los mismos erorres hasta en tres ocasiones, o incluso de ser capaz de llevarme los zapatos a arreglar, a pesar de que todo esto es muy importante, lo que mas le agradezco es su amistad.

También me gustaría agradecer de una forma especial la confianza mostrada por D. Luis Hernández Yañez y D. Carlos Cervigón Rückauer, coordinadores de la Escuela Universitaria de I. T. en Informática de Gestión, durante mi estancia de cuatro buenísimos años en el Colegio Universitario de Segovia.

Durante la investigación he recibido la ayuda de mucha gente pero me gustaría reconocer aquí la especial dedicación y paciencia de varios compañeros. He contado con la ayuda del que para mí ha sido uno de los mejores y más serios investigadores que he conocido, Oscar Garnica. Sin las explicaciones y correcciones de Carlos

Fernández los punteros aún me estarían volviendo loco. Manuel Prieto se encargó de hacerme entender los conceptos básicos del Paralelismo y Eduardo Huedo y Carlos Delgado hicieron la primera implementación del Algoritmo Genético Paralelo. Los archivos XNF se transforman en grafos gracias a la ayuda de José Hícar.

Al departamento de Arquitectura de Computadores y Automática y en especial a D. Francisco Tirado Fernández, D. Róman Hermida Correa y Dña. Milagros Fernández Centeno. A mis compañeros Olga, Luis, Silvia, David, Juan de Vicente, Nacho, Jimt, Rafa, Bonifacio, Jalo, Fernando, Segundo, Rubén, Aitor, Francesco et al, Oscar Polo, Deli y resto de componentes de los departamentos de Informática y Automática y Arquitectura de Computadores y Automática por su buen ambiente tanto dentro del centro de trabajo como en el centro de Madrid.

A todos los componentes del Parallel Processing Group del Centro Nazionale Universitario de Calcolo Elettronico de Pisa (Italia) Ranieri Baraglia, Raffaele Perego, Paolo Palmerini, Salvatore Orlando y Giancarlo Bartoli por su hospitalidad y colaboración durante la última fase de mi trabajo.

Y cómo dicen que los últimos serán los primeros, gracias a mi familia y a mis amigos que dan sentido a todo lo que hago y que muchas veces han demostrado más ilusión que yo en la finalización de este trabajo. Quiero hacer un recuerdo especial a la memoria de Elena y de Pau que siempre estarán con nosotros aunque se hayan ido.

Esta tesis ha sido financiada por los proyectos **TIC 99/0474** de la Comisión Interministerial de Ciencia Y Tecnología, **07 T/001971197** de la Comunidad de Madrid, **PR 181/96-6776** de la Universidad Complutense de Madrid, **CHRX-CT 94-0459** de la Unión Europea y **PQE 2000** del Consorcio Pisa Ricerca del Gobierno Italiano.

Índice General

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

La utilización de las computadoras en todos los aspectos de la vida actual es un hecho evidente y al que todos nos hemos acostumbrado. La amplia difusión de las mismas tiene su base en un desarrollo vertiginoso de las tecnologías, de las herramientas de diseño y de las arquitecturas que hace posible una evolución acorde a las necesidades tanto de la industria como del ciudadano de a pie.

El estudio de las herramientas de diseño implica la resolución de problemas de tipo NP-completo. Este tipo de problemas son aquellos en los que los algoritmos necesarios para su resolución tienen una complejidad no polinómica. Ejemplos de ellos se pueden encontrar en el codiseño hardware-software de diseños empotrados, en la implementación de las distintas herramientas automáticas de diseño asistido por ordenador (CAD), en el control automático de procesos o en multitud de aplicaciones industriales que requieren el uso de algoritmos de optimización.

En la mayoría de estas áreas existen problemas de partición, en los que se ha de distribuir un conjunto de elementos en distintos grupos de tal forma que cumplan una serie de características y restricciones propias del problema que se está tratando.

Por ejemplo, en el codiseño de sistemas empotrados existe el problema de distribuir las distintas partes de un diseño en implementaciones hardware e implementaciones software de tal forma que haya un compromiso entre el coste y el rendimiento del sistema. En las herramientas CAD aparecen problemas de parti-

ción en las distintas fases en las que se divide el proceso de diseño. El problema típico es conseguir una partición del circuito que haga posible la implementación utilizando la tecnología para la que está desarrollada la herramienta CAD. En un proceso de control podemos encontrar el problema de seleccionar qué parte del circuito se debe controlar con un determinado elemento u otro, o qué parte del sistema debe actuar en un momento o en otro del proceso.

Para los problemas descritos anteriormente y, en general, para muchos de los problemas de partición que es necesario resolver en la actualidad, los métodos deterministas o de búsqueda exahustiva no son aplicables, debido fundamentalmente a que para la complejidad de estos problemas el tiempo necesario para encontrar una solución es inaceptable. Por este motivo, durante los últimos años, se ha desarrollado un conjunto de técnicas denominadas heurísticas que obtienen buenas soluciones en tiempos aceptables para el usuario. Las técnicas heurísticas son métodos de búsqueda no exahustiva. Entre ellos destacan los algoritmos de mejora iterativa que obtienen una solución a partir de una inicial o un conjunto de ellas. Dentro de este tipo de técnicas, existe un grupo que basa su funcionamiento en imitar procesos naturales. Cabe mencionar dentro de este apartado el enfriamiento simulado (Simulated Annealing) [109] y los algoritmos genéticos. El enfriamiento simulado emula el proceso de enfriamiento lento en sólidos y se ha utilizado en bastantes problemas de optimización con éxito [118, 120, 101]. Sin embargo, aunque su implementación no es excesivamente complicada, el ajuste de los parámetros que lo controlan puede convertirse en ocasiones en un proceso duro y dificil de resolver.

Los algoritmos genéticos (AGs) son métodos de búsqueda de soluciones que han demostrado su eficiencia en un amplio rango de disciplinas y que se basan en los principios de la evolución natural de supervivencia de los mejores individuos [93]. Se han aplicado a un gran número de problemas de optimización en ciencias e ingeniería [95, 119, 7]. Los AGs suelen dar buenas soluciones en unos tiempos aceptables. A pesar de ello en problemas complejos pueden ser lentos en comparación con otras técnicas. Dado que cada vez se aplican a problemas más complejos y que manejan una mayor cantidad de información, el tiempo necesario para llegar a una buena solución también aumenta. Por ello, se están desarrollando métodos que aceleren y

permitan a los algoritmos genéticos seguir teniendo validez en los campos de aplicación en que son necesarios. Uno de los más utilizados es la paralelización. Los algoritmos evolutivos ofrecen muchas posibilidades para su implementación sobre sistemas paralelos y distribuidos, debido fundamentalmente a que muchas partes de su funcionamiento son independientes. De hecho, los algoritmos genéticos son intrínsecamente paralelos ya que se utiliza un conjunto de soluciones para realizar la búsqueda simultáneamente en varias direcciones.

En esta memoria se estudian los algoritmos genéticos para resolver problemas de partición. Dentro de la partición se ha estudiado el problema concreto de la partición de grafos. Tanto en el campo del diseño como en otras disciplinas existe un conjunto muy amplio de problemas que se pueden tratar mediante la teoría de grafos y que necesitan herramientas de partición para dividir estos grafos que representan la información. La mayoría de los algoritmos de partición de grafos dan buenos resultados para problemas sencillos como el de buscar el mínimo número de aristas de corte entre particiones, pero no demuestran su utilidad en aplicaciones reales.

Los AGs para partición de grafos existentes tienen el grave defecto de romper la estructura del grafo inicial, lo que los hace inaplicables para ciertos problemas de diseño. La mayor parte de los AGs utilizan una codificación centrada en los vértices del grafo y esto es precisamente lo que hace que no conserven la estructura. Podemos citar el trabajo de Bui y Moon [25]. Otras aproximaciones simplemente utilizan los AGs cómo un ajuste fino y no aprovechan su potencial [6]. Una excepción es el algoritmo presentado por Hulin [96] que si conserva la estructura, sin embargo la forma de resolver el problema es mediante una complicada doble codificación.

Entre los problemas que en ocasiones tienen los AGs cabe destacar la convergencia prematura a óptimos locales. Además para problemas complejos, la evaluación de una población grande que asegure una búsqueda con garantías, puede suponer un tiempo de computo inaceptable. Recientemente se han desarrollado técnicas de refinamiento de las soluciones mediante una búsqueda local (por ejemplo la presentada en [108]). Además se han desarrollado otras variantes de los AGs cómo los

AG compactos, que simulan la existencia de una población ahorrando de esta forma memoria para almacenar información relativa a la población [73] y disminuyendo considerablemente el número de evaluaciones en cada iteración. Sin embargo, la búsqueda es menos exhaustiva que con un AG tradicional y puede no dar los resultados esperados. La combinación de los AGs compactos con técnicas de búsqueda local es un campo que no se ha explorado y que ofrece muchas posibilidades para problemas reales. Por ello, proponemos estudiar tanto los AG compactos como su variantes híbridas (combinados con otras técnicas) para resolver problemas de partición de grafos.

El objetivo principal de está tesis es aplicar las ideas que se acaban de mencionar referentes a la partición de grafos, a la obtención de soluciones del problema en sistemas Multi-FPGA [75] (SMFPGAs). Los SMFPGAs están compuestos fundamentalmente por un conjunto de circuitos integrados programables denominados FPGAs (Field Programmable Gate Arrays). Sus campos de aplicación dentro de la microelectrónica son innumerables y podemos mencionar como ejemplo la emulación lógica, la computación númerica o los sistemas reconfigurables dinámicamente [77]. Las FPGAs son dispositivos que tienen un gran número de ventajas cómo la reprogramabilidad o la flexibilidad para realizar distintas funciones en determinados momentos. Pero en ocasiones es necesario utilizar un sistema con varias FPGAs y otros elementos que apoyen a la implementación y que necesitan un proceso de partición del sistema y por lo tanto el desarrollo de algoritmos que ayuden a realizarlo.

Existen dos tipos muy generales de SMFPGAs. Por un lado están aquellos en los que la topología o sistema donde van a ser implementados no está definida a priori y que se conocen como de topología libre. Por otro lado existen tarjetas que tienen definida su estructura y están fabricadas antes de la partición del circuito, es lo que se denomina una topología fija. Dentro de las topologías fijas podemos encontrar dos tipos fundamentales: topología crossbar o de grafo bipartito y topología malla.

Las topologías crossbar tienen dos tipos de FPGAs claramente diferenciados en cuanto a su funcionalidad; de lógica y de rutado. Las primeras implementan la lógica del circuito, mientras que los de rutado implementan las líneas de conexión.

Sin embargo, no tienen porque ser diferentes en lo relativo a su estructura interna, simplemente unos se utilizan para una función y los otros para otra. Los circuitos de lógica están conectados únicamente con los de rutado y viceversa. Este tipo de topologías puede adaptarse bien a ciertos problemas específicos pero no son fáciles de ampliar y suelen desaprovechar los recursos del sistema (tanto de lógica como de rutado).

En una topología malla, las FPGAs están conectadas con sus vecinas más próximas y no existe en principio ninguna distinción entre ellas. La FPGAs están dispuestas sobre la tarjeta en forma de matriz. Este tipo de tarjetas es fácil de rutar y presentan una buena disposición para expandir el sistema con otros similares. Los principales inconvenientes que presentan son relativos a la capacidad lógica y al reducido número de pines de entrada-salida (PE/S) que están disponibles para interconectar las distintas partes del circuito. Estas restricciones hay que tenerlas en cuenta a la hora de desarrollar herramientas de partición. Por ejemplo al evaluar el consumo de PE/S hay que fijarse en que con frecuencia se deben comunicar partes que están en FPGAs no adyacentes por lo que el problema de la escasez de PE/S se agrava aún más. La mayoría de las herramientas realizan una partición y una ubicación separadas en distintas fases de tal forma que no tienen en cuenta el problema que acabamos de exponer. Es necesario por tanto, una técnica de ubicación simultanea a la partición en la que las particiones con un mayor número de conexiones entre sí, deben tener preferencia para ser ubicadas en FPGAs adyacentes y disminuir así el consumo de PE/S.

Podríamos resumir los objetivos generales de esta tesis en los siguientes puntos fundamentales:

- Obtener una técnica de partición que conserve la estructura inicial del grafo.
- Integrar esta técnica de partición en un AG mediante una codificación sencilla que represente soluciones reales sin necesidad de aplicar ope-radores reparadores o cualquier otra técnica de doble codificación que aumente la complejidad del algoritmo.
- Estudiar los algoritmos genéticos compactos como herramienta de partición de grafos e incorporar técnicas de búsqueda local para mejorar las soluciones.

- Estudiar diseñar y paralelizar estos algoritmos de partición de grafos basados en algoritmos genéticos para mejorar su rendimiento.
- Aplicar estos algoritmos a un problema de diseño como el de la partición de circuitos para su implementación sobre SMFPGAs, abarcando las distintas topologías y teniendo en cuenta las restricciones propias de estos sistemas (consumo de pines y capacidad lógica). Las particiones deben respetar la capacidad lógica de los circuitos.
- Incorporar una técnica de ubicación, simultanea a la partición, para SMFPGAs de topología malla que permita reducir el consumo de PE/S, asegurando la rutabilidad del sistema final. Esta ubicación debe dar prioridad para situar más próximas aquellas particiones que tienen una mayor comunicación entre sí. Debe tener en cuenta que en ocasiones estas comunicaciones no son directas, si no a través de particiones no adyacentes.

Una vez realizada la introducción del trabajo presentado en esta tesis y los objetivos, veamos cómo se organizan y cúal es el contenido del resto de capítulos de la memoria.

En el capítulo 2 se hace una revisión del trabajo desarrollado por otros autores en los campos de la partición. En primer lugar se ven las técnicas y algoritmos generales de partición que existen. Dentro de estas técnicas no se exponen los algoritmos genéticos por explicarse extensamente en el capítulo siguiente, pero sí que se hace una revisión de algunas aproximaciones realizadas en la literatura al problema de la partición basadas en algoritmos evolutivos. Este capítulo no podría estar completo sin una vista general de los algoritmos de partición aplicados a SMFPGAs, por ello se ven los más importantes y los que han tenido una mayor influencia en nuestro trabajo. El estado del arte, además de dar una idea de las distintas formas posibles de abordar el problema, nos permite ver de una manera más clara las motivaciones de este trabajo, al destacarse las virtudes y carencias de las ténicas explicadas.

Todos los algoritmos utilizados en este trabajo para resolver los problemas de partición son evolutivos. Por ello, en el capítulo 3 se hace una revisión de este tipo

de algoritmos basados en la naturaleza. Se ha pretendido realizar una exposición bastante didáctica de los algoritmos genéticos, utilizando en la medida de lo posible ejemplos sencillos para facilitar la comprensión y despertar a la curiosidad del lector no introducido en este apasionante mundo de las técnicas evolutivas.

En el siguiente capítulo, el 4, se repasan los conceptos más importantes de los SMFPGAs. Cómo ya se ha dicho, estos sistemas están compuestos por un conjunto de dispositivos programables (FPGAs), además de por otros componentes de apoyo. El fabricante más importante de FPGAs que hay en la actualidad es la firma Xilinx y por ello los SMFPGAs para los que se han particularizado los algoritmos de partición están pensados para dispositivos de Xilinx tanto de la serie 3000 como de la 4000.

Los capítulos 5 y 6 presentan la implementación de los algoritmos (cómo se han diseñado, su codificación, características, etc), los resultados experimentales y las conclusiones obtenidas en cada caso. En primer lugar se resuelve el problema para topologías libres. Después, se han ido incorporando restricciones para tratar las topologías fijas, hasta llegar a la implementación del último algoritmo que mejora los anteriores y trata todas las restricciones impuestas.

Aunque despúes de la explicación de cada uno de los algoritmos y sus resultados se han resumido una serie de conclusiones para cada uno de ellos, en el capítulo 7 se realiza una recopilación de las principales conclusiones y de las aportaciones de la tesis. También se enumeran las publicaciones científicas en las que han sido presentados los resultados y algoritmos de esta tesis, así como los proyectos de investigación y ayudas recibidas para poder financiar este trabajo a lo largo de estos años. Por último incluimos una serie de ideas para continuar la investigación aquí comenzada.

Capítulo 2

Técnicas de partición

Como ya se ha mencionado, son muchas las aplicaciones que requieren un proceso de partición. Por ello existen bastantes algoritmos y técnicas para resolver este tipo de problemas. En este capítulo se hace un repaso de las propuestas más interesantes desarrolladas hasta el momento, deteniéndonos más extensamente en aquellas que han tenido un mayor éxito, así cómo en las que han tenido más influencia sobre nuestro trabajo.

En primer lugar veremos una breve descripción del problema de la partición y cuáles son los algoritmos más utilizados hasta la fecha. Aunque los AGs no se han aplicado hasta el momento para la partición de SMFPGAs, si que existen trabajos anteriores en los que se hacen aproximaciones a la partición de grafos mediante algoritmos evolutivos, por ello también expondremos estas aproximaciones y sus principales diferencias con nuestra propuesta.Los problemas de partición que se tratan en esta memoria son relativos a los SMFPGAs tanto de topología libre como fija. Por este motivo también se expone cómo se han aplicado los distintos algoritmos a la partición de SMFPGAs.

El problema de la partición se puede plantear formalmente como sigue [4]:

Dado un conjunto de bloques funcionales que componen un sistema

$$D = \{B_1, B_2, ..., B_n\}$$

el problema consiste en determinar una partición

$$P = \{P_1, P_2, ..., P_m\}$$

de tal forma que la unión de todas ellas sea equivalente al sistema inicial , que no exista solapamiento entre ellas y que cumpla las restricciones impuestas.

Las restricciones varían con el tipo de problema. Por ejemplo, si estamos tratando es un sistema de codiseño las principales restriccciones son las relativas al tiempo de ejecución y a consideraciones de coste económico. Sin embargo, en el caso de la partición en el diseño de un SMFPGA las restricciones más importantes vienen impuestas por la propia tarjeta donde se desea realizar la implementación y son las relativas a la capacidad lógica y a la disponibilidad de pines de entrada-salida en los dispositivos que la componen.

Cuando el problema de partición se reduce a dividir un sistema en dos partes, se habla de *bipartición o bisección*.

2.1 Algoritmos de partición

Hasta el momento se han realizado varias implementaciones para la resolución del problema de la partición. Kerninghan y Lin (KL) propusieron un algoritmo de bisección que es la base de muchos de los métodos utilizados en la actualidad [105].

El algoritmo de Fiduccia y Matheyses (FM) [60] es una evolución del algoritmo KL. Existen también otras aproximaciones de mejora iterativa aplicadas a la partición de circuitos como la realizada por Sanchis [153] o más recientemente el algoritmo presentado por Dasdan y Aykanat [46]. También se han aplicado en varias ocasiones técnicas espectrales [56]. Hauck [78] y Alpert [5] han publicado dos trabajos excelentes, en los que se puede encontrar una visión general de la mayoría de las técnicas de partición.

2.1.1 Algoritmo de Kernighan-Lin

El algoritmo de Kernighan-Lin (KL) es la base de la mayoría de las heurísticas utilizadas para desarrollar herramientas de partición. El algoritmo KL es una técnica de las llamadas de migración, porque basan la búsqueda en el intercambio de bloques entre los grupos.

El algoritmo funciona realizando una serie de *pasadas*. Durante cada pasada va mejorando una partición inicial mediante el intercambio de pares de vértices entre ambas particiones para crear una nueva partición. Este proceso se repite hasta que no se obtiene ninguna mejora o bien hasta que se han rea-lizado el máximo número de pasadas permitidas.

Formalmente se define como sigue. Sea (A,B) la partición inicial del grafo $G = (V, E)$ (para una introducción a la teoría de grafos ver capítulo 6). Se define g_v como la ganancia en la función de coste de mover el vértice v a la partición opuesta. Dado que se intercambian pares de vértices, definimos también la ganancia de mover $a \in A$ a B cuando se mueve $b \in B$ a A como $g(a, b)$ según la ecuación 2.1:

$$g(a, b) = g_a + g_b - 2 \cdot \delta(a, b) \tag{2.1}$$

donde

$$\delta(a, b) = \begin{cases} 1 & \text{si } (a, b) \in E \\ 0 & \text{en otro caso} \end{cases} \tag{2.2}$$

Al iniciar una pasada todos los vértices son libres, es decir están disponibles para moverlos. Una vez que se ha movido un vértice este queda bloqueado. El algoritmo KL va intercambiando pares de vértices entre los dos grupos hasta que están todos bloqueados. Con esto se calculan las ganancias y una vez que se ha terminado se realiza el intercambio con mayor ganancia, se liberan todos los bloques y se obtiene una nueva partición sobre la que se repite el proceso. Cuando no se consigue un intercambio que mejore la solución el algoritmo se para. En la figura 2.1 se muestra la estructura del algoritmo KL en forma de pseudocódigo.

La selección del mejor par de nodos a intercambiar requiere la búsqueda en el

while Se mejore el resultado **do**
  for $(\forall a \in A)$ **do**
    for $(\forall b \in B)$ **do**
      Calcular g_a, g_b
    end for
  end for
  $Q_A \Leftarrow \emptyset$
  $Q_B \Leftarrow \emptyset$
  for $i = 1$ to $(\frac{V}{2} - 1)$ **do**
    Elejir $a_i \in (A - Q_A)$ y $b_i \in (B - Q_B)/g(a_i, b_i)$ es máximo
    $Q_A \Leftarrow Q_A \ \cup \ a_i$
    $Q_B \Leftarrow Q_B \ \cup \ b_i$
    for $a \in Q_A$ **do**
      $g_a = g_a + 2 \cdot \delta(a, a_i) - 2 \cdot \delta(a, b_i)$
    end for
    for $b \in Q_B$ **do**
      $g_b = g_b + 2 \cdot \delta(b, b_i) - 2 \cdot \delta(b, a_i)$
    end for
  end for
  Seleccionar $k \in \{1, ..., (\frac{V}{2} - 1)\}/ \sum_{i+1}^{k} g(a_i, b_i)$ es máximo
  $A \Leftarrow A - \{a_1, ..., a_k\}$
  $B \Leftarrow B - \{b_1, ..., b_k\}$
  $A \Leftarrow A \cup \{b_1, ..., b_k\}$
  $B \Leftarrow A \cup \{a_1, ..., a_k\}$
end while

Figura 2.1: *Esquema del algoritmo KL*

espacio de soluciones de n^2 elementos. Esta búsqueda puede llegar a repetirse hasta n veces, lo que supone un alto coste computacional y el algoritmo no tiene ningún mecanismo para evitar los óptimos locales si parte de una solución inicial que no sea buena.

2.1.2 Algoritmo de Fiduccia-Matheyses (FM)

Fiduccia y Matheyses propusieron un método de bipartición que es una evolución del algoritmo KL [60]. Partiendo del anterior, Fiduccia y Matheyses sugirieron una serie de modificaciones:

- Solamente es posible mover un nodo en cada pasada.
- Dos movimientos consecutivos han de realizarse siempre en direcciones opuestas. Es decir, si primero se ha pasado un bloque de un grupo a otro, en el siguiente movimiento el origen y el destino deben ser los opuestos.
- El algoritmo mantiene una lista ordenada de bloques candidatos al cambio de grupo y la actualiza en cada movimiento, esto lo implementaron mediante una estructura de datos que permite elegir al mejor candidato siempre en un tiempo constante.

Para asegurar la finalización del algoritmo se crean dos grupos de bloques, los que se han movido y los que no se han utilizado. Cuando un bloque pasa al grupo de los utilizados queda bloqueado y para el siguiente movimiento sólo se podran elegir aquellos bloques que no se hayan movido. Inicialmente todos los bloques están desbloqueados y sólo se pueden mover una vez. El bloque denominado como mejor candidato es aquel cuyo movimiento supone una mejora mayor de la función de coste que se esté utilizando.

Evidentemente el algoritmo FM permite tener biparticiones descompensadas en algún momento, pero la implementación FM también incorpora un *criterio de equilibrio*, de tal forma que se mantenga un control sobre el tamaño de las particiones. Si el nodo que se pretende mover no permite este equilibrio se realiza un cambio en sentido contrario. En la figura 2.2 se puede ver el pseudocódigo del algoritmo FM.

$f_{mejor} \Leftarrow f_{inicial}$
$Part_{mejor} \Leftarrow Part_{inicial}$
while Se mejore el resultado **do**
  Desbloquear todos los elementos
  $f \Leftarrow f_{mejor}$
  $Part \Leftarrow Part_{mejor}$
  while Existen elementos desbloqueados **do**
    Seleccionar i no bloqueado cuya migración produce el mejor Δf_i
    Bloquear el elmento i
    if $f < fmejor$ **then**
      $f_{mejor} \Leftarrow f$
      $Part_{mejor} \Leftarrow Part$
    end if
  end while
end while

Figura 2.2: *Esquema del algoritmo FM*

2.1.3 Partición espectral

Los métodos de partición espectral usan los autovalores de la matriz de adyacencia de un grafo o la matriz Laplaciana para construir una representación geométrica que, posteriormente, se divide utilizando técnicas heurísticas. El problema se plantea como un grafo en el que los nodos representan los bloques del sistema y las aristas las relaciones entre estos bloques. Para la partición de circuitos, los vértices representan módulos y las aristas señales.

Partiendo del grafo se establece la matriz de interconexión y se calculan los autovalores y autovectores de la matriz. Sobre un ejemplo es más fácil seguir el funcionamiento, por ello, supongamos el grafo de la figura 2.3 en el que los nodos v_1, v_2, v_3 y v_4 pueden tener pesos o no, dependiendo del tipo de partición que se quiere resolver. Primero se calcula la matriz adyacencia A, que representa la relación existente entre los distintos bloques del sistema:

$$A = \begin{bmatrix} 0 & e_1 & 0 & 0 \\ e_1 & 0 & e_2 & 0 \\ 0 & e_2 & 0 & e_3 \\ 0 & 0 & e_3 & 0 \end{bmatrix}$$

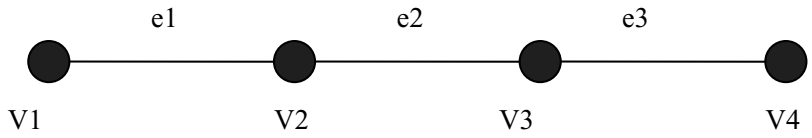

Figura 2.3: *Partición espectral*

a continuación se calcula la matriz de grado definida como $D = [d_{ij}]$: con

$$d_{ij} = \begin{cases} \sum_{k=1}^{n} a_{ik} & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$$

$$D = \begin{bmatrix} e_1 & 0 & 0 & 0 \\ 0 & e_1 + e_2 & 0 & 0 \\ 0 & 0 & e_2 + e_3 & 0 \\ 0 & 0 & 0 & e_3 \end{bmatrix}$$

Luego se calcula el Laplaciano, $L = D - A$

$$D = \begin{bmatrix} e_1 & -e_1 & 0 & 0 \\ -e_1 & e_1 + e_2 & -e_2 & 0 \\ 0 & -e_2 & e_2 + e_3 & -e_3 \\ 0 & 0 & -e_3 & e_3 \end{bmatrix}$$

Una vez obtenido el Laplaciano, se calculan los autovectores y autovalores según el álgebra tradicional, es decir mediante la ecuación 2.3:

$$(L - \lambda \cdot I) \cdot x = 0 \tag{2.3}$$

Esta ecuación tiene una solucion trivial para $x = 0$ que da cómo resultado el autovector expresado en la ecuación 2.4, donde n es la dimensión del Laplaciano:

$$x = \left(\frac{1}{\sqrt{n}}, \quad \frac{1}{\sqrt{n}}, ..., \quad \frac{1}{\sqrt{n}} \right) \tag{2.4}$$

Se calcula el segundo autovalor y su autovector y se lleva x a una partición

heurística. Algunas formas de generar estas particiones son:

- Partir los módulos basándose en el signo de x
- Dividir los módulos alrededor del valor medio de x_i, poniendo cada mitad en una partición
- Clasificar los x_i para obtener un orden lineal de los módulos y luego determinar la partición que lleva asociado un mejor valor de la función de coste

La partición espectral utiliza información global dando buenos resultados. Sin embargo, necesitan de una implementación específica para cada problema. Una explicacion más amplia se puede encontrar en [56]

2.1.4 Enfriamiento simulado (ES)

El algoritmo del ES se basa en la analogía existente entre el enfriamiento en sólidos y la solución de problemas de optimización combinatoria [168] [185]. Mediante la técnica de enfriamiento se consigue que un cuerpo sólido cristalice en una forma pura. Inicialmente, el cuerpo se calienta rápidamente hasta que pasa a la fase líquida, quedando todas sus partículas dispersas aleatoriamente. Posteriormente, mediante un enfriamiento muy lento, y tras pasar por distintos estados de equilibrio, se llega al estado de mínima energía, dónde el sólido es una estructura cristalina sin impurezas.

Esto sólo ocurre si para cada temperatura T, el sólido ha alcanzado un equilibrio térmico caracterizado por la probabilidad de encontrarse en un estado de energía E dado por la distribución de Boltzman (ecuación 2.5):

$$Pr(E) = \frac{1}{Z(T) * e^{-E/K_b \cdot T}} \tag{2.5}$$

donde $Z(T)$ es un factor de normalización denominado función de partición, K_b es la constante de Boltzman y T la temperatura absoluta.

Se observa que al disminuir la temperatura, la distribución de Boltzman se concentra sobre los estados con menor energía y, finalmente, cuando la temperatura se aproxima a cero, sólo los estados de mínima energía tienen una probabilidad de

```
temperatura ⇐ Temperatura Inicial
Coste ⇐ Funcion Objetivo (P)
while (no se alcanze la congelación) do
  while (no se alcanze el equilibrio) do
    P_intento ⇐ Movimiento Aleatorio (P)
    Coste_Nuevo ⇐ Funcion Objetivo (P_intento)
    ΔCoste ⇐ (Coste_Nuevo − Coste)
    if Aceptar(ΔCoste, temperatura) > Aleatorio (0, 1) then
      P ⇐ P_intento
      Coste ⇐ Coste_Nuevo
    end if
  end while
  Decrementar Temperatura
end while
```

Figura 2.4: *Esquema del algoritmo de enfriamiento simulado*

ocurrencia no nula. Si el enfriamiento es demasiado rápido, el sólido no alcanza el equilibrio para cada temperatura, de manera, que se congelan los defectos del sólido y se obtiene una estructura amorfa metaestable en lugar de una estructura cristalina de mínima energía. El algoritmo de ES simula este proceso para buscar soluciones a procesos de optimización.

El primero en aplicar este método para resolver problemas de optimización combinatoria fué Kirkpatrick [109] [45]. Emulando el proceso físico anterior se obtienen soluciones estadísticamente óptimas. Para llevar a cabo la simulación basta considerar las soluciones del problema como los estados de energía, la función de coste asociada a cada solución como la energía de un estado y la solución óptima como el estado de mínima energía [118, 120, 129].

Para aplicar el ES a un problema concreto hay que definir el concepto de solución o estado, el modelo de perturbación para pasar de una solución a otra, la temperatura inicial, la relación de decremento de la temperatura (enfriamiento), la función de coste que mida la energía de la solución, el espacio global de soluciones y el espacio de soluciones vecinas. En la figura 2.4 se puede ver el pseudocódigo del algoritmo ES.

2.1.5 Otras técnicas

Existen otras técnicas de diversos tipos tanto del tipo KL como basadas en procesos naturales. De las primeras comentaremos a modo de ejemplo el KL con ganancia bloqueada y de las segundas hablaremos del *Mean Field Annealing* y de los algoritmos genéticos, aunque la implementaciones que utilizan AGs se ven con más detalle en la sección 2.2.

KL con ganancia bloqueada

Este es un algoritmo bastante reciente [108] que funciona básicamente como el KL pero, en lugar de utilizar g_a, utiliza lo que denomina *ganancia bloqueada* o *lock gain*, l_a. Esta ganancia es la misma que la definida en el apartado 2.1.1 pero considerando solamente los vértices bloqueados. Se basa en una asunción de distribución uniforme de la probabilidad que no es general y que parece indicada sólo para determinados problemas. De todas formas no aporta una variación muy grande con respecto a los métodos presentes en la literatura.

Mean Field Annealing

Es una técnica similar al Enfriamiento Simulado y que se basa en otro proceso natural cómo son los sistemas de particúlas en equilibrio térmico. Van de Bout y Miller [20] aplicaron esta técnica a la partición de grafos. Para ello definen un vector $\vec{x}$ que representa la partición y que tiene tantos elementos como vértices tenga el grafo. Si tenemos una partición $P(A, B)$,

$$x_i = 0 \quad si \quad v_i \in A$$

$$x_i = 1 \quad si \quad v_i \in B$$

cuando x_i no está decidido en qué partición puede estar el elemento su valor es real $\in [0, 1]$. Inicialmente define x_i ligeramente mayor que 0.5. Supongamos que tenemos 5 vértices, el vector inicial sería de la forma:

$$\vec{x} = [\ 0.6 \quad 0.6 \quad 0.6 \quad 0.6 \quad 0.6\ ]$$

se elije un v_i, por ejemplo el tercero, y se evalua la partición con $x_i = 0$ y $x_i = 1$, y el resto de valores aproximados a su entero más próximo. las particiones se denominan $\vec{x}(0)$ y $\vec{x}(1)$ y tienen como valores de la función de coste $FF(\vec{x}(0))$ y $FF(\vec{x}(1))$ respectivamente. Con esto se calcula un nuevo x_i según la ecuación 2.6:

$$x_i = \frac{1}{1 + e^{\frac{FF(\vec{x}(1)) - FF(\vec{x}(0))}{T}}} \tag{2.6}$$

donde T es la temperatura simulada. x_i se sustituye en $\vec{x}$ y se repite el proceso hasta que se obtiene una solución estable. Luego se disminuye el valor de la temperatura y se repite el proceso. A medida que el algoritmo converge linealmente, todos los valores de $\vec{x}$ se aproximan a 0 ó a 1. La bipartición final se obtiene sustituyendo cada v_i por su entero más próximo. al igual que sucede con el enfriamiento simulado tanto la convergencia cómo el control de la temperatura son difíciles de ajustar. Este algoritmo es en cierta forma similar al algoritmo genético compacto (ver sección 6.7) que se ha utilizado para resolver la partición.

2.2 Partición de grafos mediante AGs

Las técnicas utilizadas en el desarrollo de este trabajo están basadas tanto en la utilización de algoritmos genéticos como en la representación de los problemas tratados mediante la teoría de grafos. Los algoritmos genéticos emulan los principios de la evolución de los organismos en la naturaleza para resolver problemas de optimización. Hasta el momento se han implementado varias aproximaciones a la partición de grafos utilizando AGs. La mayoría de estas aproximaciones no tienen en cuenta la estructura del grafo y por ello presentan ciertos problemas para aplicarlas a sistemas específicos como la partición Multi-FPGA, en los que preservar la estructura inicial del circuito es bastante importante para reducir el consumo de pines de entrada/salida y los retardos de las señales. Todas estas aproximaciones codifican el problema basándose en los vértices y la solución propuesta en este trabajo para topologías fijas usa, como veremos en el capítulo 6, las aristas. De esta forma se consigue preservar lo máximo posible la estructura inicial del grafo que representa al circuito. La mayoría de las aproximaciones se han realizado para biparticiones y son pocas las que se aplican a una partición múltiple. A continuación

comentamos los algoritmos más significativos.

2.2.1 Aproximación de Laszewski y Mühlenbein

Laszewski y Mühlenbein [122, 121] realizaron una implementación paralela de un AG para realizar la partición de grafos. El algoritmo realiza una distribución en k partes iguales. Su principal aportación es que maximiza la suma de los pesos de las aristas que hay dentro de una partición. Esto le permite aplicar eficientemente el operador de selección del algoritmo genético, pero sin embargo al realizar las operaciones con los operadores de cruce y mutación necesita hacer una serie de correcciones para evitar codificaciones de soluciones no reales.

Dado un grafo $G = (V, E)$ con n vértices o nodos, las k particiones del mismo:

$$\{P_1, P_2, \cdots, P_k\}$$

vienen representadas por un valor para cada vértice

$$(\ g_1, g_2, \cdots, g_n\ )$$

de tal forma que:

$$v_i \in P_a \Longleftrightarrow g_i = a \in \{1, \cdots, k\}, \forall i \in \{1, \cdots, n\}$$

y la función de coste viene representada por:

$$c(\ g_1, g_2, \cdots, g_n\ ) = \sum_{i<j<n} w(v_i, v_j) \tag{2.7}$$

donde $w(v_i, v_j)$ es el coste o peso asociado a la arista que une v_i y v_j.El método es bastante sencillo pero no tiene en cuenta la estructura del grafo.

2.2.2 Sistema *Genetic Metis*

El sistema Metis [104] es una algoritmo de bipartición que ha dado muy buenos resultados en la partición de grafos. Es una técnica de agrupamiento multinivel.

Cuando trabaja sobre problemas de mínimo corte, busca biparticiones equilibradas que tengan el menor número de conexiones entre las particiones. En 100 ejecuciones del algoritmo, predominan los resultados medios sobre los mínimos. Es decir, que es raro obtener una solución óptima aunque obtiene normalmente una solución aceptable. Alpert et al. [6] utilizan un AG para complementar este método, integrando el algoritmo Metis en un entorno genético. Para ello construyen un código genético concatenando varias soluciones de Metis.

Dado un grafo G de n nodos la bipartición $P = \{X, Y\}$, se representa por un vector de n elementos $\vec{p} = \{p_1, p_2, ..., p_n\}$. Cada valor p_i tiene un valor de 0 si el nodo v_i pertenece a X o un 1 si está en la partición Y. El algoritmo genera s soluciones de Metis y asigna un código, C_i, de s bits a cada nodo v_i uniendo los s valores de p_i en cada una de las soluciones. El sistema genera multi-particiones agrupando dos nodos i y j si $C_i = C_j$. Una vez realizada la codificación aplica un algoritmo genético simple. Veamos un ejemplo de la codificación trabajando con 4 soluciones por individuo sobre un circuito de 6 nodos: Para generar un individuo, se generan 4 soluciones que denominaremos p^i:

$$p^1 = \{0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1\}$$

$$p^2 = \{1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1\}$$

$$p^3 = \{1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1\}$$

$$p^4 = \{1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0\}$$

Al tener 6 nodos y 4 soluciones cada inidividuo tendrá 24 genes:

$$Individuo = C_1 \quad C_2 \quad C_3 \quad C_4 \quad C_5 \quad C_6$$

Para saber el valor de C_1 se coge el primer valor de cada p^i, para C_2 los segundos y así sucesivamente, por lo tanto:

$$C_1 = 0 \quad 1 \quad 1 \quad 1$$

$$C_2 = 1 \quad 0 \quad 0 \quad 1$$

$$C_3 = 0 \quad 1 \quad 1 \quad 1$$

$$C_4 = 1 \quad 1 \quad 1 \quad 0$$

$$C_5 = 0 \quad 0 \quad 0 \quad 0$$

$$C_6 = 1 \quad 1 \quad 1 \quad 0$$

Cada C_i representa un nodo v_i, como dos nodos v_i y v_j están en la misma partición si C_i es igual que C_j, tendremos que la partición P tiene 4 particiones, $P = \{X, Y, Z, W\}$, con:

$$X = \{v_1, \quad v_3\}$$

$$Y = \{v_2\}$$

$$Z = \{v_4, \quad v_6\}$$

$$W = \{v_5\}$$

Este tipo de codificación no realiza ninguna consideración sobre la estructura del circuito y puede presentar problemas a la hora de aplicarlo a SMFPGAs de topología libre. Otro inconveniente es que el número de particiones no es fijo, por lo que también sería dificil de adaptar a una topología fija, pues se puede obtener un número de particiones superior al número de dispositivos disponibles.

2.2.3 Aproximación de Hulin

Una rara excepción es la aproximación a la partición de grafos realizada por Hulin [96] porque conserva la estructura del grafo. Esta técnica se aplica a la partición de circuitos y trata de conservar la estructura del circuito. Para ello utiliza una doble codificación bastante complicada que el mismo denomina AG sofisticado. Esta codificación viene motivada por la obtención de soluciones falsas después de realizar el cruce y la mutación con una codificación sencilla. Este tipo de problemas se comentará más ampliamente en el capítulo dedicado a los algoritmos genéticos.

2.2.4 Algoritmo de Bui y Moon

Una aportación interesante es la proporcionada por Bui y Moon en [25]. En este trabajo se presenta un algoritmo genético híbrido. El algoritmo incluye una heurística

de mejora local rápida.

Como la mayoría de las aproximaciones en este campo, lo aplica al problema de corte mínimo. El tamaño de corte de una partición se define como el número de aristas que unen nodos de distintas particiones. El problema de corte mínimo consiste en encontrar la partición del sistema que minimice el tamaño de corte. La función de coste que utiliza viene dada por F_i (ecuación 2.8):

$$F_i = \frac{(C_w - C_i) + (C_w - C_b)}{3} \tag{2.8}$$

donde:

- C_w: tamaño de corte de la peor solución de la población
- C_b: tamaño de corte de la mejor solución de la población
- C_i: tamaño de corte de la solución i

Primero aplican el problema para la bipartición y posteriormente lo amplian para problemas de partición mayores. La codificación que utilizan no consigue conservar la estructura del grafo. El algoritmo está diseñado para obtener particiones con igual número de elementos en todos los subconjuntos, lo que puede ser una limitación para extenderlo a otros problemas en los que no sea deseable obtener soluciones equilibradas. Los resultados que obtienen son comparables a los que obtiene el algoritmo KL, pero el algoritmo de búsqueda local es una idea bastante interesante como mejora de los AGs y por este motivo lo veremos con un poco más de detalle

Como búsqueda local utiliza una modificación del algoritmo de Kernighan-Lin, después de aplicar los operadores de cruce y mutación (ver capítulo 3). En la figura 2.5 se muestra el pseudocódigo del algoritmo de búsqueda local que utilizan. Las principales variaciones con respecto al algoritmo KL son:

- Limitar el número máximo de elementos a intercambiar, mediante el parámetro $Max - Exch - Size$.
- Permitir sólo una pasada.

for $(\forall a \in A)$ **do**
  for $(\forall b \in B)$ **do**
    Calcular g_a, g_b
  end for
end for
Generar lista $g_a s, g_b s$
$Q_A \Leftarrow \emptyset$
$Q_B \Leftarrow \emptyset$
for $i = 1$ to $(Max - Exch - Size)$ **do**
  Elejir $a_i \in (A - Q_A)$ y $b_i \in (B - Q_B)/g(a_i, b_i)$ es máximo
  $Q_A \Leftarrow Q_A \cup a_i$
  $Q_B \Leftarrow Q_B \cup b_i$
  for $a \in Q_A - \{a_i\}$ adyacente a a_i o b_i **do**
    $g_a = g_a + 2 \cdot \delta(a, a_i) - 2 \cdot \delta(a, b_i)$
    if g_a cambia **then**
      ajustar lista de $g_a s$
    end if
  end for
  for $b \in Q_b - \{b_i\}$ adyacente a a_i o b_i **do**
    $g_b = g_b + 2 \cdot \delta(b, b_i) - 2 \cdot \delta(b, a_i)$
    if g_b cambia **then**
      ajustar lista de $g_b s$
    end if
  end for
end for
Seleccionar $k \in \{1, ..., (\frac{V}{2} - 1)\}/ \sum_{i=1}^{k} g(a_i, b_i)$ es máximo
$A \Leftarrow A - \{a_1, ..., a_k\}$
$B \Leftarrow B - \{b_1, ..., b_k\}$
$A \Leftarrow A \cup \{b_1, ..., b_k\}$
$B \Leftarrow A \cup \{a_1, ..., a_k\}$

Figura 2.5: *Esquema del algoritmo de Bui y Moon*

Recientemente se han realizado extensiones de éste tipo de algoritmos que realizan la búsqueda local con el algoritmo de ganancia bloqueada y con el algoritmo de Kernighan Lin simple [108] [162]. Los resultados obtenidos hasta el momento no son muy significativos, aunque mejoran a los de los algoritmos genéticos sin el proceso de mejora local.

2.3 Técnicas de partición para sistemas Multi-FPGA

Se han realizado varias implementaciones de algoritmos de partición para resolver el problema de los SMFPGAs. Sin embargo la mayoría de ellas o bien no tienen en cuenta las restricciones propias de este tipo de sistemas, o son adaptaciones de otros campos del VLSI. Existen otras técnicas que tienen en cuenta estos factores pero son implementaciones muy específicas para tarjetas concretas.

2.3.1 Algoritmo de Réplica. Sistema PROP

Kuznar et al., han publicado diversos trabajos [114] [115] [116] [117], basados en el algoritmo de FM junto con otras técnicas de optimización, en los que se recogen buenos resultados.

Cuando se realiza la partición, los bloques quedan separados por unas líneas de corte. Los bloques de un lado de la línea pueden tener conexiones con los del otro. La réplica tradicional [53, 54, 55] consiste en seleccionar una celda o bloque de puertas lógicas que está en un lado de una línea de corte y realizar una copia en el otro lado. A continuación se separan sus salidas para intentar reducir el número de conexiones de corte. En ocasiones no se consigue porque tambíen hay que conectar las entradas y éstas pueden atravesar la línea de corte aumentando así el número de conexiones de corte. El sistema PROP realiza una réplica distinta que denomina *réplica funcional*. Si la funcionalidad de la celda lógica es conocida se puede utilizar esta información para evitar conexiones innecesarias. Lo que se hace es dejar al aire (no conectar) aquellas conexiones que no participan en la función que se ha pasado al otro lado de la línea de corte. En la figura 2.6 se muestra un ejemplo de

un movimiento simple (de una partición a otra)(a), de una copia tradicional (b) y de una copia o réplica funcional (c). PROP plantea el problema como se explica a continuación.

Encontrar una partición en K partes que sea factible con el mínimo coste. El coste viene definido por

$$\$_K = \sum_{i=1}^{q} d_i \cdot n_i$$

donde d_i es el coste de cada dispositivo y n_i el número de dispositivos del tipo i utilizados por la partición. El número de particiones k viene dado por:

$$k = \sum_{i=1}^{q} n_i \tag{2.9}$$

Una solución es factible si cumple las restricciones de tamaño (o número de CLBs que utiliza) y de número de pines de entrada-salida ocupados. Si todos los dispositivos son iguales el problema se reduce a encontrar el número de subdivisiones que lo cumplan.

Se representa el problema mediante un hipergrafo donde se diferencia entre nodos terminales y nodos interiores y las aristas representan el conjunto de conexiones que componen el sistema. A continuación lo que hace es buscar nodos interiores que se puedan copiar al otro lado de la línea de corte, para minimizar el número de interconexiones, a esto se le llama copia o réplica funcional de un módulo. En un principio podría parecer que aumenta el tamaño, pero en realidad se compensa y se consiguen eliminar conexiones innecesarias.

Utiliza varias operaciones para aplicar el algoritmo de Fiduccia-Matheyses. Veamos un ejemplo con un sistema secillo como el de la figura 2.6. En un sistema con 5 entradas y dos salidas x_1 y x_2 con:

$$x_1 = f_1(a_1, a_2, a_3, a_4)$$

$$x_2 = f_2(a_4, a_5)$$

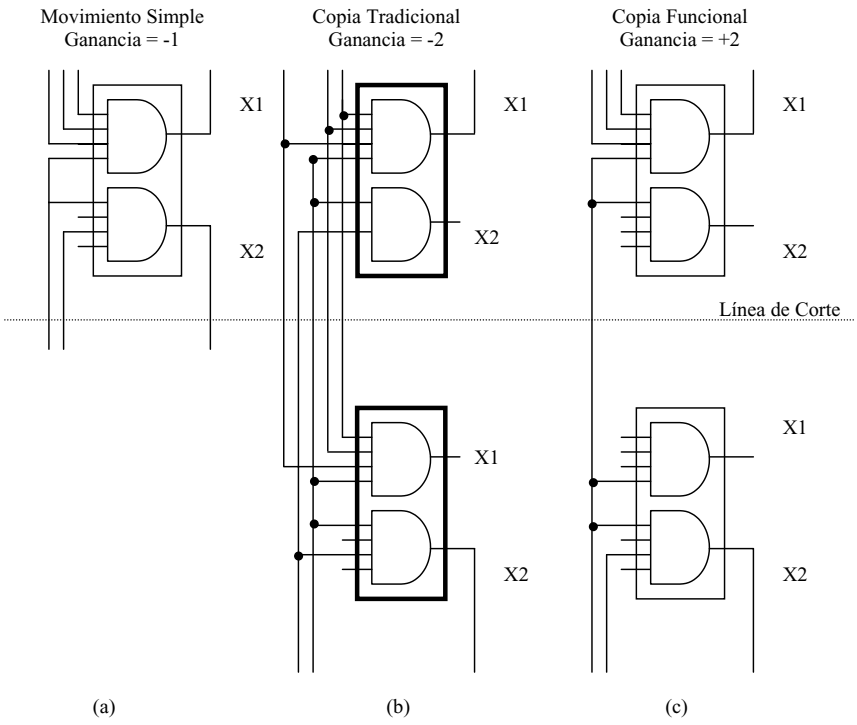

Figura 2.6: *Ganancia de un movimiento simple, una copia tradicional y una copia funcional*

Se definen sus vectores de adyacencia A_{x_1} y A_{x_2} cómo

$$A_{x_1} = [1 \;\; 1 \;\; 1 \;\; 1 \;\; 0]^T$$

$$A_{x_2} = [0 \;\; 0 \;\; 0 \;\; 1 \;\; 1]^T$$

las operaciones que se pueden realizar con esos vectores son:

- Complemento: $\overline{A_{x_1}} = [0 \;\; 0 \;\; 0 \;\; 0 \;\; 1]^T$
- AND lógica: $A_{x_1} \cdot A_{x_2} = [0 \;\; 0 \;\; 0 \;\; 1 \;\; 0]^T$
- Norma: $\|A_{x_2}\| = 2$

Dentro de una celda se asocian varios vectores binarios:

- Los ya mencionados vectores de adyacencia

- Un vector de entrada C_i^I y uno de salida C_i^O para cada bloque en relación con las entradas de la celda, para el ejemplo de la figura 2.6 tendríamos:

$$C_1^I = [1 \;\; 1 \;\; 1 \;\; 1 \;\; 0]^T$$

$$C_1^O = [1 \;\; 0]^T$$

$$C_2^I = [0 \;\; 0 \;\; 0 \;\; 1 \;\; 1]^T$$

$$C_2^O = [0 \;\; 1]^T$$

- Dos vectores críticos de conexión con relación a la celda completa: Q^I y Q^O, que para la figura serían:

$$Q^I = [1 \;\; 1 \;\; 1 \;\; 1 \;\; 1]^T$$

$$Q^O = [1 \;\; 1]^T$$

  Un vector de conexión es crítico si un movimiento cambia su estado

Una vez definidos estos vectores define la ganancia de la réplica funcional G_r cómo:

$$G_r = max(G_{X_1}, G_{X_2})$$

donde

$$\begin{aligned} G_{X_1} &= (\|(C^I \cdot A_{x_1}) \cdot (Q^I \cdot A_{x_1} \cdot \overline{A_{x_2}})\| + (c_1^O \cdot q_1^O)) - \\ &\quad (\|\overline{C}^I \cdot A_{x_2}) \cdot (Q^I \cdot A_{x_1} \cdot \overline{A_{x_2}})\| + (\overline{c}_1^O \cdot q_1^O)) \end{aligned}$$

y

$$\begin{aligned} G_{X_1} &= (\|(C^I \cdot A_{x_2}) \cdot (Q^I \cdot A_{x_2} \cdot \overline{A_{x_1}})\| + (c_2^O \cdot q_2^O)) - \\ &\quad (\|\overline{C}^I \cdot A_{x_2}) \cdot (Q^I \cdot A_{x_2} \cdot \overline{A_{x_1}})\| + (\overline{c}_2^O \cdot q_2^O)) \end{aligned}$$

Estas expresiones derivan de la Ganancia de un movimiento simple G_m, que se calcula contando el número de conexiones de corte y conexiones críticas que se eliminan y el número de conexiones no críticas que se añaden con el movimiento. G_m se puede calcular por la expresión 2.10:

$$G_m = (\|C^I \cdot Q^I\| + \|C^O \cdot Q^O\| - (\|\overline{C^I} \cdot Q^I\| + \|\overline{C^O} \cdot Q^O\|) \qquad (2.10)$$

Finalmemte se aplica el algoritmo FM con biparticiones sucesivas y se obtiene una solución partiendo de una inicial. El principal problema que tiene es que no considera el coste de realizar la disposición final sobre las FPGAs. Además depende claramente de la solució inicial que es una bipartición, aunque este problema lo solventa en parte ejecutando el algoritmo varias veces hasta encontrar una solución final. El método en ciertas ocasiones no obtiene soluciones reales, lo que hace necesaria la ejecución del programa en paralelo para mejorar los resultados. Por otro lado, está diseñado para sistemas heterogéneos (con distintos tipos de FPGAs) y se aplica únicamente a sistemas homogéneos con FPGAs de la serie 3000. Recientemente Krupnova [112, 113] ha presentado otro trabajo con técnicas similares que obtiene resultados muy parecidos a los de Kuznar.

2.3.2 Técnicas de partición espectral

Como ya hemos comentado, una de las técnicas más utilizadas es la partición espectral (PE). La partición espectral calcula los autovalores y autovectores de la matriz de interconexión del circuito. A continuación trabaja sobre ellos de diferentes formas. Los métodos basados en la partición espectral necesitan una implementación específica en función del número de dispositivos que se van a utilizar. Comentaremos brevemente tres aproximaciones que utilizan la PE. El sistema PARABOLI, una implementación para una tarjeta MFPGA específica y una partición orientada a la rutabilidad.

Sistema PARABOLI

El sistema PARABOLI [150] desarrolla una aproximación para resolver problemas de partición en circuitos grandes utilizando un algoritmo de ubicación analítica. Este algoritmo parte de una solución inicial y aprovecha las ventajas de una función objetivo lineal. Este sistema se ha adaptado a SMFPGAs aunque en realidad es un algoritmo general de partición que no tiene en cuenta ciertas caraterísticas propias de este tipo de sistemas.

Trabajando con la matriz de adyacencia A, que da las conexiones entre las distin-

tas particiones, el problema de la ubicacion unidimensional se puede formular como un problema de programacion cuadrático con restricciones cuadráticas (QPPQC) definido como:

$$min \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij}(x_i - x_j)^2 / \sum_{i=1}^{n} x_i^2 = 1$$

donde los x_i son los autovectores. Las componentes de los autovectores se interpretan cómo las coordenadas de las celdas para una ubicación unidimensional.

Su principal problema es que tiene una fuerte dependencia de la partición inicial.

Partición para la tarjeta DRASTIC

Sankarsubramanian y Bhatia [155] desarrollaron un método de partición para la arquitectura DRASTIC basado en la técnicas espectrales. El sistema DRASTIC es una tarjeta compuesta por 3 FPGAS de la serie XC4025 de Xilinx y una memoria SRAM de 1Mb de capacidad.

Para ello modela el circuito como un grafo $G = (V, E)$ donde los vértices representan los módulos y las aristas señales. A contiuación define el problema de la siguiente forma:

Dado un circuito C, una arquitectura de implementación B con un número de dispositivos d de tamaño S_i, con un número de pines por dispositivo P_i y una matriz de conexión M de tamaño d x d (donde $M(i, j)$ representa el número de conexiones disponibles entre el dispositivo i y el j), el objetivo es dividir C en un número de particiones menor o igual a d de tal forma que se cumplan las siguientes restricciones:

- *El tamaño de cada partición j debe ser* $\leq S_j$
- *El número de conexiones de corte de la partición j debe ser* $\leq P_j$
- *El número de aristas entre la partición i y la j debe ser* $\leq M(i, j)$

Sobre esta base construye la matriz de grado del grafo y aplica un algoritmo en dos fases. Primero una partición espectral y luego una mejora mediante el algoritmo

FM. La partición espectral realiza la partición a nivel de la tarjeta, es decir obtiene la solución inicial que posteriormente se mejora con FM. La entrada al sistema incluye el archivo de descripción del circuito y la información de la arquitectura. La salida del sistema es una serie de archivos que describen las partes del diseño. Una vez obtenidos se aplica la partición a nivel de chip para obtener las particiones que cumplan las restricciones.

El sistema obtiene buenos resultados pero no se explica si es adaptable o no a otras topologías y cómo afectaría a la complejidad de obtención de las matrices el aumento en el número de particiones.

Partición espectral orientada a la rutabilidad

Chan, Schlag y Zien [37] han presentado distintos trabajos en los que desarrollan una técnica de partición espectral para SMFPGAs homogéneos cuyo principal objetivo es asegurar la rutabilidad del circuito. Primero realizan una predicción de la rutabilidad y a continuación implementan un partidor espectral que utiliza estas estimaciones.

La técnica utilizada para predecir la rutabilidad es una extensión de la presentada por los autores para FPGAs simples [146]. Es una heurística bastante interesante. El método consiste en estimar la anchura de canal necesaria (W) para completar el rutado del circuito mediante la ecuación 2.11:

$$W = \frac{1}{2} \cdot [\gamma \cdot (1 + \frac{\beta - 2}{\beta}) \cdot L] \tag{2.11}$$

con:

- γ = número medio de pines por CLB
- β número medio de pines utilizado por una conexión
- L Longitud media de una conexión después de la ubicación

Las entradas necesarias para el predictor de rutabilidad son:

- Número total de pines en CLB del circuito inicial.

- Número total de conexiones en el circuito incial.
- Longitud media de las conexiones en cada subcircuito.
- Número de particiones.
- Número de pines de entrada-salida por partición.

De estos datos, el primero y el segundo se pueden obtener del circuito inicial, el tercero se puede estimar y los dos últimos dependen del dispositivo que se vaya a utilizar. Es decir, hay que calcular los valores estimados de γ, β y L, que serán γ_p, β_p y L_p respectivamente.

$$\gamma_p \cong \gamma = \frac{número\ \ total\ \ de\ \ pines\ \ de\ \ CLB}{número\ \ total\ \ de\ \ CLBs} \tag{2.12}$$

$$\overline{\beta_p} = \frac{número\ \ de\ \ pines\ \ de\ \ E/S\ \ y\ \ de\ \ CLB\ \ en\ \ una\ \ partición}{número\ \ de\ \ conexiones\ \ de\ \ una\ \ partición} \tag{2.13}$$

Para calcular $\overline{\beta_p}$ supongamos la siguiente terminología:

- $pins$ = Número total de pines en CLB antes de la partición
- $pins_p$ = Número total de pines en CLB en la partición p
- $nets$ = Número total de conexiones antes de la partición
- $nets_p$ = Número total de conexiones (con uno ó más pines) en la partición p
- $nets_c$ = Número total de conexiones de corte (que divide el partidor)
- IO_b Número total de pines del circuito inicial
- IO_a Número total de pines después de la partición
- io_p Número total de pines en la partición p
- k Número de particiones

Como el número de conexiones en el circuito no varía, se conserva a lo largo del proceso de partición, podemos expresar $nets$ como:

$$nets = \sum_{p=1}^{k} nets_p - (IO_a - IO_b) + nets_c$$

si hallamos la media:

$$\overline{nets} = (\sum_{p=1}^{k} nets_p - (IO_a - IO_b) + nets_c)/k$$

y por lo tanto:

$$\begin{aligned}
\overline{\beta_p} &\cong \frac{\overline{número\ \ de\ \ pines\ \ en\ \ una\ \ partición}}{número\ \ de\ \ conexiones\ \ de\ \ una\ \ partición} \\
&= \frac{\overline{pins_p}+\overline{io_p}}{\overline{nets_p}} \\
&= \frac{\overline{pins_p}+\overline{io_p}}{(nets+IO_a-IO_b-nets_c/k} \\
&= \frac{pins+k\cdot\overline{io_p}}{nets+IO_a-IO_b-nets_c}
\end{aligned}$$

Aproximando $net_c \approx (IO_a - IO_b)/min(k, \beta)$ y $IO_a = k \cdot \overline{io_p}$ y sustituyendo los resultados de las estimaciones en (2.13), obtenemos:

$$\overline{\beta_p} \approx \frac{pins + k \cdot \overline{io_p}}{nets + IO_a - IO_b(1 - 1/min(k, \beta))}$$

Y por otro lado podemos calcular L por la ecuación 2.14

$$L = \sqrt{ocupación\_lógica} \cdot L_{tipo\_FPGA} \tag{2.14}$$

Con las ecuaciones 2.12, 2.13 y 2.14 se puede estimar la rutabilidad del circuito y utillizar esta información a la hora de realizar la partición para manejar tanto restricciones de corte cómo del número de elementos de la partición.

2.3.3 Otras técnicas

Recientemente se han presentado algunas técnicas nuevas de las que mencionamos las propuestas más interesantes a continuación, así como otras menos recientes, pero que presentan alguna variación significativa con las expuestas hasta el momento.

Partición con relajación intermedia de restricciones

Dutt et al. [50] explican dos métodos de partición para obtener soluciones con igual numero de elementos en cada partición que son bastante similares a la mayoría de técnicas de mínimo corte o ratio cut. De hecho son una mejora del sistema PROP. Sin embargo introducen una modificación importante cómo es la aceptar temporalmente soluciones que violen las restricciones pero que una vez finalizado el algoritmo pueden llevar a soluciones óptimas. Dicho de otro modo permite una relajación intermedia de las restricciones.

En la primera aproximación para cada movimiento de la celda que viola las restricciones, se realiza una estimación del coste y del beneficio que supondría el movimiento. Los movimientos que potencialmente dan un mejor resultado, en comparación a no permitirlos, se seleccionan pero siempre teniendo en cuenta que no se puede establecer un punto de corte definitivo hasta que no se corrija la infracción con una serie de movimientos regresivos. Esta implementación presenta un alto coste computacional y por eso proponen la modificación que implementan en el segundo método

El segundo método utiliza una técnica de menor complejidad temporal. Esta técnica calcula un *factor de beneficio* y un *umbral de aceptación* para un movimiento que produce una infracción de las restricciones. Dependiendo de las infracciones realizadas hasta ese momento, de la posición y condiciones de la celda y de la de su vecinas, el movimiento se aceptará o se rechazará. Veamos brevemente cómo se realiza este proceso.

Definimos una celda *infractora* cómo aquella cuyo movimiento hace que la partición resultante no cumpla las restriciones impuestas, es decir, que infringe las

restricciones. Definimos también $B(v)$ cómo el beneficio total ponderado de mover una celda infractora v con tamaño $a(v)$ de V_i a $\cdot V_{\overline{i}}$, que cumple la condición:

$$0 \leq B(v) \leq 1$$

y viene dado por:

$$B(v) = \sum_{i=1}^{c} w_i \cdot b_i$$

donde c es el número total de factores a tener en cuenta, b_i es el beneficio i, w_i es el peso del beneficio i y cumplen las condiciones 2.15 y 2.16.

$$0 \leq w_i \leq 1 \tag{2.15}$$

$$\sum_{i=1}^{c} w_i = 1 \tag{2.16}$$

Para este problema se consideran tres factores:

- $|g_v - g_u|$, norma de la diferencia entre la ganancia máxima g_v en V_i y la ganancia g_u máxima en $V_{\overline{i}}$.
- Proporción disponible de la infracción total fijada como máxima $|relax|$. Si z es la infracción que se ha cometido hasta el momento y $a(v)$ es la infracción que se comete al mover la celda v, la proporción disponible será $|relax| - z - a(v)$.
- $(F_{V_{\overline{i}}})$,Proporción entre las celdas libres que quedaría en V_i después de corregir la infracción, y el número total de celdas libres en V_i en ese momento.

Con todo esto B(v) vendría dada por la ecuación 2.17:

$$\begin{array}{rcllll} B(v) & = & w_1 \cdot & (\phi(g(v) - g(u))) & + & \\ & & w_2 \cdot & (1 - \frac{z+a(v)}{|relax|}) & + & \\ & & w_3 \cdot & (1 - \frac{z+a(v)}{F_{V_{\overline{i}}}}) & & \end{array} \tag{2.17}$$

Donde ϕ es una función que normaliza la ganancia a un valor entre 0 y 1, los mejores resultados son para $w_1 = 0.5$, $w_2 = 0.25$ y $w_3 = 0.25$. La idea de $B(v)$ es que una celda infractora tenga una gran probabilidad de ser aceptada si la diferencia de ganancias entre u y v es grande, si las infracciones cometidas están lejos del límite

y si todavía quedan bastantes celdas por bloquear. El movimiento v se aceptará si su $B(v)$ es mayor que el umbral de aceptación $T(v)$. Este umbral depende de la diferencia ($\triangle cut$) entre el tamaño de corte actual (cut_c) y el mejor encontrado hasta el momento cut_{min}, todo ello modelado por la ecuación 2.18, donde f_v y f_u son variables de control.

$$T(v) = f(\triangle cut) = 1 - \frac{1}{2}(\tanh(\frac{\triangle cut}{f_u} - f_v) + 1) \tag{2.18}$$

El método aplicado es interesante por su minuciosidad y la heurística utilizada, sin embargo su alto costo computacional hace que sólo sea factible aplicarlo estadísticamente y no a todas las celdas infractoras.

Método de bipartición recursiva

Chan y Lewis [37] presentan un algoritmo de bipartición recursiva orientado a un sistema MFPGA jerárquico. Este método se basa en el algoritmo FM y por lo tanto presenta sus problemas típicos entre los que cabe destacar su imposibilidad para evitar los óptimos locales.

Método de Chou

También se han desarrollado técnicas de agrupamiento o clustering. Entre ellas podemos destacar la de Chou et al. [38], que consiste en un proceso de dos fases. Primero aplica un plan de agrupamiento para reducir la complejidad del circuito y a continuación realiza el proceso de partición propiamente dicho. Para el primer paso utiliza un método muy similar al utilizado por la herramienta ESPRESSO [52], que consiste en eliminar redundancias. Su principal problema es que necesita fijar un número mínimo de circuitos para llevar a cabo la implementación final del circuito.

En [38] se propone un algoritmo de mapping sobre un número mínimo de FPGAs. En primer lugar se aplica un esquema de agrupamiento para minimizar el número de conexiones y la complejidad del circuito. A continuación realiza la partición mediante *cubiertas* similares a las del sistema ESPRESSO que se utiliza en optimización lógica. En la primera fase, la de agrupamiento, se elije un tamaño para la partición. A continuación se extrae un subcircuito de esas dimensiones y se

define un conjunto de nodos vecinos. Se aplica una partición intentando minimizar el número de interconexiones y para ello sólo se realizan intercambios con los nodos vecinos. De esta forma se van obteniendo todos los grupos o particiones.

La segunda fase es una partición por cubiertas de conjuntos, en la que se eliminan redundancias y se generan grupos tan pequeños cómo sea posible para posteriormente poder ir asignándolos dentro de las FPGAs y que se cumplan las restricciones.

2.4 Resumen

Una vez repasadas las técnicas y metodologías de partición más importantes podemos resaltar las siguientes carencias que han motivado nuestro trabajo.

En primer lugar las técnicas de partición de grafos no conservan la estructura del sistema, por lo que resulta difícil su aplicación en problemas de diseño. La conservación de la estructura básica es fundamental, ya que asegura que las soluciones tengan rertardos menores y que se optimizen las implementaciones. Además es bastante común utilizar la teoría de grafos para representar circuitos.

Para sistemas Multi-FPGA, las técnicas suelen ser aproximaciones o adaptaciones de otras ya existentes para problemas distintos, lo que hace necesario el desarrollo de algoritmos específicos que tengan en cuenta las características propias de estos sistemas y faciliten el trabajo del diseñador.

Capítulo 3

Algoritmos Genéticos

Los algoritmos evolutivos (AEs) son procedimientos de búsqueda y optimización que tienen sus orígenes e inspiración en el mundo biológico. Se caracterizan por emular los comportamientos evolutivos de la naturaleza y se basan en la supervivencia del mejor individuo, siendo un individuo una solución potencial del problema que se implementa como una estructura de datos. Trabajan sobre poblaciones de soluciones que evolucionan de generación en generación mediante operadores genéticos adaptados al problema. Los parámetros que los controlan son variables en función de como se represente a los genes, del tipo de estructura de datos que implementa una solución, del tipo de operadores y de si estos parámetros son variables o constantes. Según estos factores nos encontramos con un tipo de estrategia u otra.

Dentro de los AEs se engloban normalmente los algoritmos genéticos (AGs), las estrategias evolutivas (EEs) y la programación genética (PG) [166]. Todos ellos tratan de hacer una abstracción del problema para que evolucione y buscar así una mejor solución (figura 3.1).

En esta memoria se hace un estudio de los algoritmos genéticos como herramienta de partición por las siguientes razones:

- Las técnicas de partición explicadas en esta memoria utilizan AGs
- Son los más conocidos y utilizados
- Tienen una importante base matemática y experimental que afianza la teoría

Figura 3.1: *Taxonomía de los algoritmos evolutivos*

- Comprendiendo bien el funcionamiento de los AGs es sencillo entender el resto de programas evolutivos.

En los años 60 John Holland ideó los algoritmos genéticos que fueron posteriormente desarrollados por su grupo de investigación en la Universidad de Michigan [93]. El AG de Holland es un método para pasar de una población (conjunto) de individuos (soluciones) representados por "cromosomas" (cadenas de bits que representan soluciones a un problema) a una nueva población. Para ello se utilizan una especie de "selección natural" y unos operadores inspirados en la evolución que se llaman cruce y mutación. Cada cromosoma está compuesto de genes (bits), donde cada gen puede tener un determinado alelo (valor, que para la representación binaria es 0 ó 1). La selección se encarga de escoger los individuos que participarán en la formación de la nueva población. El operador de cruce intercambia las propiedades de los individuos y el de mutación produce cambios aleatorios en la población de tal forma que se amplie el espacio de búsqueda. Holland también introdujo el operador inversión que invierte el orden de una parte del cromosoma y que nosotros repasaremos.

Al mismo tiempo que Holland desarrolló los AGs, los mismos conceptos fueron utilizados en Alemania por Rechemberg y Schwefel para desarrollar las estrategias evolutivas (EEs) [156]. La forma original de las EEs utilizaba 2 individuos y el operador de mutación como único elemento de búsqueda y cambio de la población. Los dos individuos eran el padre y el descendiente. Posteriormente se han realizado

implementaciones de estrategias evolutivas para poblaciones con múltiples individuos. Por último, la programación evolutiva o genética [110] representa soluciones a los problemas mediante una población de máquinas de estados finitos o mediante programas. La nueva población se crea mediante la mutación aleatoria de los programas padre.

3.1 Algoritmos genéticos simples

Los algoritmos genéticos simples se basan en un principio básico de la evolución: los mejores individuos tienen una mayor probabilidad de reproducirse y sobrevivir que otros individuos menos adaptados al entorno [48, 65, 132]. Para implementar este principio, los algoritmos genéticos mantienen una población que evoluciona a través del tiempo y que al final convergen a una única solución. Los individuos de la población se representan mediante un cromosoma que codifica las variables del problema que se quiere resolver.

Normalmente se utiliza un cromosoma simple compuesto por una cadena de bits de longitud fija, pero existen algoritmos genéticos que codifican el problema utilizando varios cromosomas para representar una solución y otros que utilizan cromosomas de longitud variable. Cada individuo tiene asignado un valor de una función de coste, que mide la calidad de la solución y que es la herramienta que permite simular el concepto de *individuos mejor adaptados*. Aquellos individuos cuyo valor de la función de coste sea mejor tendrán más posibilidades de ser seleccionados para construir la siguiente población y por lo tanto, para pasar sus propiedades a los individuos de la siguiente generación.

Para implementar un algoritmo genético es necesario definir:

- Una función de coste que evalue a los individuos.
- Una codificación que permita representar las soluciones.
- El operador de selección.
- El operador de cruce.

- El operador de mutación.
- El tamaño de la población.
- Los valores de las probabilidades con la que se aplican cada uno de los operadores.

En el resto del capítulo se da una explicación de cada uno de ellos y las formas más utilizadas para su implementación, explicando prioritariamente las utilizadas en esta memoria, pero primero veamos cúal es la estructura general de un algoritmo genético simple.

Para comenzar se genera una población inicial a partir de la que se trabaja. Esta población se suele inicializar de una manera aleatoria, aunque existen implementaciones en las que se obtiene mediante otros métodos cómo una búsqueda local o cualquier otro tipo de algoritmo si se pretende que la búsqueda se inicie en una determinada dirección del espacio de soluciones [64]. Cuando se han obtenido estos individuos iniciales, se repite el proceso que se explica a continuación tantas veces como indique la condición de parada (que puede ser un número máximo de generaciones, la convergencia de la población, etc...). Dicho proceso comienza evaluando la población y seleccionando los individuos que intervendrán en la formación de la siguiente generación. Seguidamente se aplican los operadores de cruce y mutación y se obtiene la nueva población a partir de la que se repiten los mismos pasos para ir avanzando en el proceso de búsqueda. En la figura 3.2 se puede ver el pseudocódigo de un algoritmo genético simple.

```
generación de la población inicial
while no se cumpla la condición de parada do
  evaluación de los individuos
  selección
  cruce
  mutación
end while
```

Figura 3.2: *Esquema de un algoritmo genético simple*

3.2 Representación genética

La elección de la codificación y el diseño de la función de coste son los dos puntos fundamentales en la implementación de un algoritmo genético. Cómo ya se ha dicho las soluciones al problema que se está tratando se codifican mediante cadenas de caracteres denominados cromosomas. Las características que debe cumplir una buena codificación son las siguientes:

- Debe representar todo el espacio de soluciones. Esta primera característica es bastante obvia, pues si no se representa todo el espacio de soluciones, es evidente que se pueden estar ignorando soluciones, entre las que puede estar precisamente la que estamos buscando.

- Debe asegurar que al aplicar los operadores de cruce y mutación no se generan individuos *incorrectos o irreales* es decir, que no representan una verdadera solución al problema. Este punto se verá con más detalle al explicar el funcionamiento del operador de cruce.

- Deben cubrir todo el espacio de búsqueda de una manera continua, ya que si existen discontinuidades el proceso de búsqueda puede ser aleatorio y la efectividad del AG sería bastante difícil de conseguir

La representación más sencilla suele utilizar un código binario es decir $0s$ y $1s$. Al conjunto de carácteres que se utilizan para representar la población se le denomina alfabeto y se representa por Ω. Para el caso de la representación binaria sería: $\Omega = \{0, 1\}$. Cada uno de estos valores se denomina *alelo*. Veamos un ejemplo sencillo de codificación binaria.

Supongamos que tenemos un grupo de 6 personas que trabajan para una empresa. Esta empresa tiene dos oficinas nuevas, una en Madrid y otra en Barcelona. Estas 6 personas deben incorporarse a las oficinas en un determinado orden y deben elegir a que oficina van. Una forma sencilla de codificar binariamente la distribución de los empleados es utilizar un gen (bit) para cada uno de ellos y que el valor de este gen sea 0 si la persona decide irse a Madrid y 1 si elige trabajar en la sede de Barcelona. Así el cromosoma

0 0 0 0 1 1

indicaría que las cuatro primeras personas van a Madrid y las 2 últimas trabajarán en Barcelona.

En determinados problemas no es suficiente con utilizar una codificación binaria y es necesario hacer uso de otra codificación ya sea con cifras o con carácteres alfanúmericos. Supongamos ahora que esta empresa tiene otros 8 empleados que deben elegir entre 4 destinos; Bilbao, Málaga, Santander y Granada. En este caso las posibles soluciones se pueden representar mediante un cromosoma de 8 genes en el que cada uno puede tomar un valor del 1 al 4 para indicar la ciudad a la que se desplazarán. Si asociamos el alelo 1 a Bilbao, el 2 a Málaga, el 3 a Santander y el alelo 4 a Granada, tendremos el alfabeto:

$$\Omega = \{1,\ 2,\ 3,\ 4\}$$

En este caso el cromosoma

$$1\ \ 2\ \ 3\ \ 4\ \ 4\ \ 2\ \ 2\ \ 3$$

indicaría que la elección de los trabajadores es Bilbao, Málaga, Santander, Granada, Granada, Málaga, Málaga y Santander repectivamente. Existen otras formas de representar la población que utilizan estructuras de datos más complejas, pero todas las implementaciones utilizadas en esta memoria utilizan o bien un alfabeto binario o un alfabeto de numeros enteros.

Como ya hemos dicho, la manera en que se codifica un problema es un elemento clave para el buen funcionamiento del algoritmo. La mayoría de las aplicaciones de AG usan una longitud de cromosoma fija y un orden fijo en las cadenas de bits. Sin embargo, en los últimos años hay muchos experiencias en las que se usan otros tipos de codificación. La codificación binaria es la codificación mas extendida, probablemente por razones históricas ya que fue la primera que se uso. La mayoría de la base teórica existente se basa en la longitud fija de la cadena y codificaciones con un orden fijo de bits. Mucha de estas teorías se pueden extender a codificaciones no binarias aunque no están completamente desarrolladas [178, 179].

Holland dió una justificación para el uso de codificación binaria. Comparó dos codificaciones que contenían la misma información, una de ellas con un pequeño

número de alelos y cadenas largas y otra con numero elevado de alelos y cadenas cortas. En el primer caso permitía un mayor grado de paralelismo implícito puesto que una instancia del primero contiene mas instancias que una del segundo. En contra tiene que la codificación binaria es una codificación no natural y poco útil para muchos problemas.

Adaptación del código

La elección de un código fijo tanto en longitud como en orden tiene varios inconvenientes. En lo que a orden se refiere, puede ocurrir que un orden fijo produzca la ruptura de buenos esquemas debido a los efectos del cruce y de la mutación. Este conocimiento no se conoce en tiempo de "compilación" , si así fuera sería tanto como conocer la respuesta al problema. Por lo tanto sería deseable que la codificación fuera cambiando el orden de los bits para evitar la ruptura y desaparición de los mejores esquemas. A este problema se le denomina el problema del "Linkage", se quiere que la funcionalidad relacionada con la posición tenga más probabilidad de permanecer junta bajo cruces y mutación, pero no está muy claro como saber que posiciones deben conservarse en tiempo de compilación. La solución parece ser adaptar el código al tiempo que se avanza en la resolución del problema.

Otra razón para adaptar el código es que la longitud fija del cromosoma limita la complejidad de las soluciones candidatas. Existen estrategias como la programación genética que automáticamente permiten la adaptación del tamaño del código puesto que tras los cruces y las mutaciones los árboles crecen o se reducen. A continuación se describen unas técnicas que permiten resolver estos dos problemas.

Inversión

La inversión es un operador de reordenación que se basa en la genética natural en la que el valor de un gen es independiente de su posición en el cromosoma de manera que al invertirlo se guarda mucha de la información del cromosoma original.

Para usar la inversión se tiene que encontrar alguna forma de realizar la interpretación funcional de la posición del alelo en el cromosoma, de tal forma que su valor sea el mismo independientemente de la posición en la que aparece. Holland

propuso que cada alelo llevara asociado un índice indicando su posición real, que se usaría para la evaluación del cromosoma. Por ejemplo sea la cadena

$$0 \;\; 0 \;\; 0 \;\; 1 \;\; 0 \;\; 1 \;\; 0 \;\; 1$$

que se podría codificar como

$$(1,0) \;\; (2,0) \;\; (3,0) \;\; (4,1) \;\; (5,0) \;\; (6,1) \;\; (7,0) \;\; (8,1)$$

siendo el primer miembro de cada par la posición original en el cromosoma, de manera que este cromosoma es el mismo que

$$(1,0) \;\; (2,0) \;\; (6,1) \;\; (5,0) \;\; (4,1) \;\; (3,0) \;\; (7,0) \;\; (8,1)$$

La inversión trabaja escogiendo dos puntos de la cadena e invirtiendo el orden de los bits entre ellos. Este cambio no modifica el valor de fitness (valor de la función de coste) del cromosoma y sin embargo, ayuda a mantener los esquemas que interesan. La idea detrás de la inversión es producir ordenes entre los bits que beneficien esquemas con más probabilidad de sobrevivir. El operador inversión puede producir problemas a la hora de realizar el cruce que se verán más adelante.

3.3 Función de coste

El otro aspecto fundamental en el desarrollo de un AG es la elección de una buena función de coste. La función de coste debe evaluar a los individuos para indicar cúal es la calidad de la solución que representan y poder realizar el proceso de selección. Un ejemplo típico es el siguiente. Supongamos que tenemos una cadena de 6 bits en la que se quiere maximizar el número de unos que contiene la cadena. En este caso la función de coste sería $f_1(x) = x$, donde x es el número de 1s que contiene la cadena (individuo) evaluada. Así el individuo [0 0 0 0 1 1] tendría asociado un valor de la función de coste igual a 2 y el individuo [0 1 1 0 1 1] un valor de 4.

Normalmente los AGs tratan de maximizar una función, pero si lo que se quiere es minimizar, no hay más que utilizar la inversa de la función objetivo, o bien mul-

tiplicarla por -1. Para el mismo ejemplo si se quiere minimizar el número de $1s$ en la cadena de carácteres, se pueden utlizar las funciones $f_2(x) = -x$ ó $f_3(x) = 1/x$. Otra opción es cambiar el significado de x para que represente el número de ceros que contiene cada cromosoma y utilizar la función $f_1(x)$.

Veamos un ejemplo un poco más complejo. Volviendo al problema de los 8 empleados del apartado anterior, supongamos ahora que el desplazamiento de cada una de las personas tiene un coste para la empresa dependiendo a la ciudad a la que vayan (Tabla 3.1). Si se quiere minimizar el coste de la nueva distribución no

Empleado	C_{Bilbao}	C_{Malaga}	$C_{Granada}$	$C_{Santander}$
1	22	13	14	17
2	8	9	10	26
3	7	35	46	12
4	53	28	96	15
5	61	17	38	17
6	42	32	33	15
7	21	12	21	12
8	41	29	27	35

Tabla 3.1: *Coste por empleado y desplazamiento para el ejemplo de la función de coste*

habría más que utilizar una función de coste de la forma:

$$f(x) = \sum_{i=1}^{8} f_i(x) \tag{3.1}$$

donde $f_i(x)$ es el coste de cada empleado para la distribución evaluada. Una de las distribuciones óptimas sería la representada por el individuo:

2 1 1 4 4 4 2 3

En ocasiones los AGs tienen aparte de una función de coste una función objetivo. Es decir, que aunque la evaluación se realiza de acuerdo a una función se trata de alcanzar un objetivo que se evalua mediante otra función distinta. Esto suele ocurrir cuando se tratan poblemas con restricciones o problemas en los que se pretende optimizar distintos parámetros conocidos cómo problemas multi-objetivo [28, 27, 144]. Está función objetivo no tiene porque ser una función númerica sino simplemente que se cumpla un criterio o no.

3.4 Operadores de Selección

Los algoritmos genéticos tienen un operador de selección que identifica a los mejores individuos de la población actual para utilizarlos como padres de la siguiente generación. Hay muchas formas de realizar la selección, pero siempre se debe asegurar que los mejores individuos tengan una mayor probabilidad de ser seleccionados. Una de las características más importantes de los AGs es que dejan un camino abierto a aquellas soluciones que no pertenecen a las mejores, para que puedan aportar parte de su información a la nueva generación.

Selección por el método de la ruleta

El método de la ruleta es uno de los más utilizados en la implementación de AGs. La idea es dar a cada individuo una probabilidad de ser seleccionado acorde a su función de coste y proporcional a su "calidad"dentro de la población que se está evaluando. Cuanto mejor es su valor de la función de coste mayor es la probabilidad de ser seleccionado.

Para calcular la probabilidad de selección, P_{s_i}, de cada individuo i, primero se debe evaluar la función de coste para cada uno de ellos, FF_i. A continuación se calcúla la suma de todas ellas y se obtiene P_{s_i} de acuerdo a la expresión 3.2:

$$P_{s_i} = \frac{FF_i}{\sum_{i=1}^{n} FF_i} \tag{3.2}$$

donde n es el número de individuos de la población.

A continuación se calcula la probabilidad de selección acumulada P_{a_i} para cada individuo. Para calcularla se van sumando las probabilidades de selección de los individuos:

$$P_{a_i} = \sum_{j=1}^{i} P_{s_j}$$

Se generan tantos números aleatorios como individuos queramos seleccionar. Cada número aleatorio se compara con las probabilidades acumuladas y se escoge el

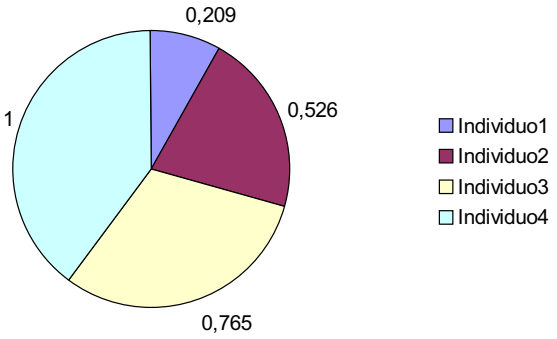

Figura 3.3: *Selección por el método de la ruleta, probabilidad de selección acumalada*

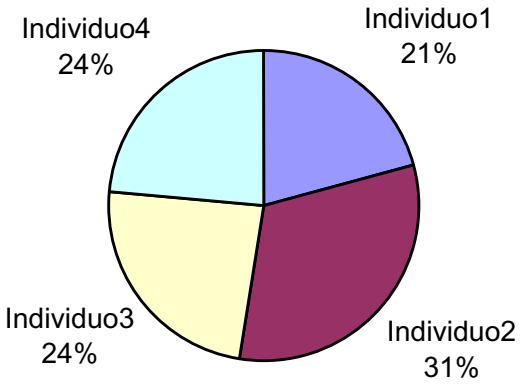

Figura 3.4: *Selección por el método de la ruleta, probabilidad de selección porcentual*

individuo que tenga asociada una probabilidad inmediatamente menor al número aleatorio. Al representar los datos sobre un diagrama circular, figuras 3.3 y 3.4, se ve que al hacer las divisiones correspondientes a la probabilidad de selección acumulada, cada individuo tiene una fracción proporcional a su probabilidad de selección.

Continuando con nuestro ejemplo supongamos la población de 4 individuos con sus correspondientes valores de la función de coste FF_i que aparecen en la tabla 3.2. En la misma tabla se recogen los valores de la probabilidad de selección y el de P_{a_i}. La tabla 3.3 recoge los individuos que se seleccionarían para 4 números aleatorios.

La selección utilizada es muy importante para la correcta convergencia del algo-

N	Individuo	FF_i	P_{s_i}	P_{a_i}
1	11223344	211	0.209	0.209
2	44231221	320	0.317	0.526
3	12341234	241	0.239	0.765
4	22232421	238	0.235	1.000

Tabla 3.2: *Ejemplo de selección por el método de la ruleta*

Num. Aleatorio	N	Selección
0.23	1	44231221
0.07	4	11223344
0.97	3	22232421
0.65	2	12341234

Tabla 3.3: *Ejemplo de selección para 4 números aleatorios*

ritmo. La presión de selección debe ser tal que consiga un equilibrio entre exploración y explotación. Si la presión de selección es muy alta se corre el peligro de que un individuo de la población inicial con un valor de la función de coste superior a la media, que representan óptimos locales pero no globales, se reproduzca en exceso provocando una perdida de diversidad y una convergencia prematura. En este caso se prima la explotación frente a la exploración. El caso contrario, una presión de selección baja puede provocar una búsqueda aleatoria o en el mejor de los casos una enorme ralentización del algoritmo.

Elitismo

Existen diversas variaciones de este proceso de selección simple. Entre ellas podemos destacar el modelo elitista. Este modelo elitista consiste en guardar siempre el mejor individuo de la población para la siguiente generación, normalmente sustituyéndolo por el peor. Hay estudios que indican que un algoritmo con selección elitista asegura la convergencia del AG hacia un óptimo global [152]. Nosotros hemos incorporado el elitismo en nuestros algoritmos una vez realizado el proceso de cruce.

3.5 Operadores de Cruce

Para explorar el espacio de soluciones los algoritmos genéticos utilizan dos operadores que tienen su base en la genética natural: los operadores de cruce y mutación.

El operador de cruce consiste en elegir aleatoriamente un par de individuos de los seleccionados e intercambiar una parte de sus cromosomas entre sí. Este operador se aplica con una determinada probabilidad. El intercambio del código se realiza a partir de unos puntos que se seleccionan aleatoriamente. Existen también muchas formas de realizar el cruce, pero la más usual es seleccionar un único punto e intercambiar el código a partir de ese punto y se denomina *cruce por un punto*.

Veamos un ejemplo sencillo. Supongamos que realizamos el cruce entre los dos primeros individuos seleccionados de la tabla 3.2. Para aplicar el operador se genera un número aleatorio $mod(k)$ siendo k el número de genes, por ejemplo el 3. A partir del gen número 3 se intercambiarían los códigos de los individuos para generar los dos siguientes individuos:

$$4\ 4\ 2 \quad \textit{2 3 3 4 4}$$
$$\textit{1 1 2} \quad 3\ 1\ 2\ 2\ 1$$

Como se dijo al hablar de la codificación, es muy importante asegurar que el cruce no produzca soluciones falsas. Por ejemplo, supongamos el mismo problema pero con 3 ciudades y 4 empleados. Podría utilizarse una codificación binaria con la siguiente correspondencia:

$$0\ \ 0 \rightarrow \text{Madrid}$$
$$0\ \ 1 \rightarrow \text{Málaga}$$
$$1\ \ 0 \rightarrow \text{Granada}$$

Si utilizamos dos bits para el destino de cada empleado dos posibles soluciones son:

$$0\ \ 0 \qquad 0\ \ 1 \qquad 1\ \ 0 \qquad 0\ \ 0$$

$$0\ \ 1 \qquad 1\ \ 0 \qquad 0\ \ 0 \qquad 1\ \ 0$$

Al realizar un cruce por el tercer gen obtendríamos:

$$0\ \ 0 \qquad 0\ \ 0 \qquad 0\ \ 0 \qquad 1\ \ 0$$

$$0\ \ 1 \qquad 1\ \ 1 \qquad 1\ \ 0 \qquad 0\ \ 0$$

Esto indicaría que el segundo empleado va al destino codificado cómo 1 1, pero en nuestra codificación este destino no existe.

Sin embargo si se utiliza una codificación entera con:

$$0 \rightarrow \text{Madrid}$$
$$1 \rightarrow \text{Málaga}$$
$$2 \rightarrow \text{Granada}$$

este problema no existe ya que los individuos padres sería:

$$1 \quad 2 \quad 3 \quad 1$$

$$2 \quad 3 \quad 1 \quad 3$$

y al realizar el cruce obtendríamos:

$$1 \quad 2 \quad 3 \quad 3$$

$$2 \quad 3 \quad 1 \quad 1$$

que no tiene ningún problema de representación. Para resolver este tipo de problemas existen varias soluciones como la doble codificación o mecanismos reparadores. En [132] se pueden encontrar la descripción de las técnicas más utilizadas.

Como se ha dicho el operador inversión puede producir problemas a la hora de realizar el cruce. Vamos a ver un ejemplo, vamos a cruzar los siguientes individuos:

$$(1,0) \quad (2,0) \quad (6,1) \quad (5,0) \quad (4,1) \quad (3,0) \quad (7,0) \quad (8,1)$$

$$(5,1) \quad (2,0) \quad (3,1) \quad (4,1) \quad (1,1) \quad (8,1) \quad (6,0) \quad (7,0)$$

si cruzamos por el tercer bit se obtiene:

$$(1,0) \quad (2,0) \quad (6,1) \quad (4,1) \quad (1,1) \quad (8,1) \quad (6,0) \quad (7,0)$$

$$(5,1) \quad (2,0) \quad (3,1) \quad (5,0) \quad (4,1) \quad (3,0) \quad (7,0) \quad (8,1)$$

Como se puede comprobar, el primer hijo tiene dos copias de el bit 1 y 6 y ninguna de los bits 3 y 5. A este problema Holland le dio dos soluciones, la primera permitir unicamente el cruce entre cromosomas con la misma permutación de posiciones. Esto funciona, pero limita enormemente la capacidad de cruce. La segunda solución es emplear un enfoque maestro/esclavo. Se escoge una cadena para ser el maestro, a continuación se escoge un esclavo que se reordena temporalmente en el mismo orden que el padre, se realiza el cruce y retornamos el segundo padre a su orden anterior. Antes de aplicar este operador se debe sopesar el aumento del tiempo computacional y del consumo de memoria extra.

3.6 Operadores de Mutación

El operador de mutación es el encargado de realizar pequeños cambios en el código con una probabilidad muy pequeña. Se utiliza para reestablecer la diversidad que se haya podido perder con la aplicación sucesiva de los operadores de selección y cruce. Normalmente se considera un operador secundario. Pero esto puede ser un error ya que en ocasiones es muy útil hacer pequeñas modificaciones en la implementación para obtener mejoras importantes en el rendimiento del AG. Ochoa ha realizado un estudio en el que demuestra la importancia de las proporciones de mutación y de selección en la eficiencia del algoritmo [140, 141]. También existen trabajos en los que se utilizan valores de las probabilidades de mutación que varían con la progresión del algoritmo [1, 2].

El método más habitual de realizar la mutácion es seleccionar un gen del individuo y cambiar su alelo por otro de los del alfabeto. La probabilidad con la que estos cambios se producen suele ser baja, para evitar que el AG realice una búsqueda aleatoria. Sin embargo en ocasiones puede ser necesario aumentar esta probabilidad para recuperar la diversidad de la población. En esta memoria se ha diseñado un nuevo operador genético (operador de Regeneración) para evitar problemas de convergencia prematura que se basa en esta afirmación y que ha dado buenos resultados [86].

Para ilustrar el funcionamiento utilicemos el individuo 1 de la tabla 3.2:

1 1 *2* 2 3 3 4 4

si se produce una mutación en el tercer gen los posibles indivduos resultantes serían:

1 1 *1* 2 3 3 4 4
1 1 *3* 2 3 3 4 4
1 1 *4* 2 3 3 4 4

Al igual que ocurría con el cruce, la mutación no debe generar individuos que no representen soluciones reales y esto también hay que tenerlo en cuenta al escoger la codificación.

3.7 Tamaño de la población

Un factor muy importante para la convergencia de los algoritmos genéticos es el tamaño de la población. El tiempo necesario para que un AG converja a una solución única depende del tamaño de la población. Goldberg y Deb publicaron un estudio en 1991 en el que demuestran que el tiempo para que un individuo se propague a toda la población utilizando los métodos más rápidos de selección es $O(n \cdot logn)$ siendo n el tamaño de la población [63]. Aunque los AGs son eficientes, sin embargo no garantizan la obtención de una solución óptima. Su efectividad viene claramente determinada por el tamaño de la población. Es evidente que cuanto mayor sea el número de individuos se explorarán más zonas del espacio de soluciones. Pero también es bastante obvio que esto acarreará un costo computacional mayor. Por eso se debe buscar un compromiso entre el número de individuos utilizados y la calidad que se desea alcanzar.

3.8 Base teórica

Los AGs son fáciles de utilizar, describir e implementar. Sin embargo su formalización puede ser bastante complicada. Se ha realizado bastante trabajo en este campo [65, 149, 174, 183, 184, 176, 175, 102, 177]. En este apartado se hace un repaso de los conceptos más importantes.

La teoría tradicional dice que los AGs trabajan descubriendo, favoreciendo y recombinando bloques de bits que aportan un alto valor a la función de coste de los individuos donde están presentes y que se denominan *Building Blocks* (BBs). Este proceso se realiza de una manera altamente paralela. La idea es que las soluciones buenas tienden a estar compuestas por buenos BBs. Para formalizar este concepto, Holland introdujo el concepto de *esquema*. Un esquema es una cadena de bits que se puede describir por una plantilla de unos (1), ceros (0) y asteriscos ($*$) donde los *s representan "don't cares". Por ejmplo el esquema:

$$H = 1 \quad * \quad * \quad 0 \quad 1$$

representa el conjunto de todas las cadenas de 5 bits que comienzan por 1 y terminan por 0 1, y las cadenas H_i que pertenecen a un esquema se denominan *instancias de H*. El esquema anterior tendría las siguientes instancias:

$$H_1 = 1 \quad 0 \quad 0 \quad 0 \quad 1$$
$$H_2 = 1 \quad 0 \quad 1 \quad 0 \quad 1$$
$$H_3 = 1 \quad 1 \quad 0 \quad 0 \quad 1$$
$$H_4 = 1 \quad 1 \quad 1 \quad 0 \quad 1$$

La forma en que un AG procesa un esquema es la siguiente. Cualquier cadena de bits de longitud l es una instancia de 2^l esquemas diferentes. Por lo que una población de n individuos contiene instancias de entre 2^l y $n \cdot 2^l$ esquemas diferentes (ya que puede haber individuos repetidos). Esto significa que en una determinada generación, mientras que el AG está evaluando explícitamente la función de coste de las n cadenas de bits que componen la población, está en realidad estimando implíctamente el valor medio de la función de coste de un número mucho mayor de esquemas. Es cierto que los esquemas no están explícitamente representados y evaluados en un AG. Sin embargo el comportamiento del AG, en términos de aumento o disminución del número de instancias de los esquemas de la población, se puede describir cómo si realmente estuviera calculando y almacenando los mencionados valores medios.

Se puede calcular la forma en que aumentan o disminuyen las instancias de un esquema de la forma siguiente. Sea H un esquema con al menos una instancia

presente en la población en el instante de tiempo (generación) t. Sea $m(H,t)$ el número de instancias de H en el instante t, y $\hat{u}(H,t)$ el valor medio de la función de coste observado para la instancias de H presentes en la población en el instante t. Para ver como evoluciona, debemos calcular el número de instancias de H esperado en el instante $t+1$.

$$E(m(h,t+1))$$

El número esperado de descendientes de una solución x será:

$$f(x)\ /\overline{f}(t)$$

donde $f(x)$ es el valor de la función de coste para el indivduo x y $\overline{f}(t)$ es el valor medio de la función de coste en el instante t. Si se ignoran los efectos del cruce y la mutación y suponemos que x es una instancia del esquema H:

$$E(m(h,t+1)) = \sum_{x \in H} \frac{f(x)}{\overline{f}(t)} \tag{3.3}$$

es decir:

$$E(m(h,t+1)) = m(H,t)\frac{\hat{u}(H,t)}{\overline{f}(t)} \tag{3.4}$$

la ecuación 3.4 muestra que incluso aunque el AG no calcula *û(H,t)* explícitamente, la variación de la instancias de un esquema de una población a otra depende de ésta cantidad.

Estos cálculos se han realizado sin tener en cuenta los efectos de los operadores de cruce y de mutación. Es evidente que la aplicación de éstos puede crear y destruir instancias de H. Supongamos que aplicamos el operador de cruce por un punto (ver apartado 3.5) con una probabilidad p_c y que se ha seleccionado una instancia del esquema H como padre. Diremos que el esquema H ha sobrevivido al cruce por un punto si al menos uno de los descendientes también es una instancia de H. Si H tiene una lóngitud $d(H)$ en bits y el cromosoma tiene l bits, la probabilidad de escoger un bit del esquema para realizar el cruce y por lo tanto de destruir H es

$d(H)/l$, obviamente multiplicado por la probabilidad de que se produzca el cruce:

$$P(destruirH) = p_c \cdot \frac{longitud(H)}{longitud(individuo)} = p_c \cdot \frac{d(H)}{l} \tag{3.5}$$

y con esto la probabilidad de que H sobreviva al cruce, $S_c(H)$, será cómo mínimo:

$$S_c(H) \geq 1 - p_c \cdot \frac{d(H)}{l} \tag{3.6}$$

la ecuación 3.6 indica que hay una mayor probabilidad de sobrevivir para valores más pequeños de $d(H)$, es decir para los esquemas más cortos.

Consideremos ahora que la mutación se aplica con una probabilidad p_m. La probabilidad de que el bit i de H sufra una mutación es p_m y por tanto de sobrevivir a la mutación ($S_m(H_i)$) es:

$$S_m(H_i) = 1 - p_m$$

si el esquema está compuesto de $o(H)$ bits, la probabilidad $S_m(H)$ de que el esquema sobreviva es:

$$S_m(H) = \prod_{i=1} o(H) \cdot S_m(H_i) = (1 - p_m)^{o(H)} \tag{3.7}$$

Si incluimos los efectos del cruce (ecuación 3.6) y la mutación (ecuación 3.7) en la ecuación 3.4, obtenemos lo que se conoce como el *Teorema de los Esquemas*, ecuación 3.8, y que muestra matemáticamente cómo varía un esquema de una generación a otra y por lo tanto cómo funciona un algoritmo genético.

$$E(m(h, t+1)) \geq m(H, t)\frac{\hat{u}(H, t)}{\overline{f}(t)} \cdot 1 - p_m)^{o(H)} \cdot (1 - p_c \cdot \frac{d(H)}{l}) \tag{3.8}$$

Este teorema es la base fundamental pero, como ya se ha mencionado existe un gran campo de investigación en el área teórica. También indica que al evaluar una población de n individuos, el AG está evaluando implíctiamente los valores medios de coste de todos los esquemas presentes en la población y aumentando o disminuyendo su presencia de acuerdo a esta ecuación. La evaluación simultánea e implícita

de un largo número de esquemas en una población se conoce como *paralelismo implícito* de los algoritmos genéticos, lo que los hace especialmente indicados para la paralelización [16].

Capítulo 4

Sistemas Multi-FPGA

La lógica digital de un sistema electrónico se puede implementar de muchas maneras. Desde la utilización de circuitos electrónicos discretos hasta los circuitos integrados. Dentro de los circuitos integrados existen a su vez distintas formas de implementación. Son lo que se denominan estilos de diseño [161].

Hay 2 estilos de diseño fundamentales: *full-custom* y *semi-custom* [148, 182]. En el primero, el diseñador realiza la implementación de todas las máscaras. El estilo full-custom tiene una gran complejidad y por eso se utiliza sólo cuando las ligaduras de área, calidad y rendimiento son muy fuertes. Es necesario que el tiempo disponible para la puesta en el mercado del diseño sea moderadamente largo, pero permite una reducción del área y una alta optimización de los retardos.

Cuando estas restricciones son menores y se busca un diseño más rápido se utilizan los estilos semi-custom, en el que ciertas partes del sistema ya están definidas o prefabricadas. Permite una cierta libertad a la hora de diseñar el circuito, pero se deben tener en cuenta algunas restricciones que varían en función de la tecnología de implementación. Las formas más generales son las celdas estándar, la matriz de puertas y las arquitecturas lógicas programables y circuitos reconfigurables. Podemos encontrar distintos tipos de circuitos reconfigurables [24] entre los que cabe destacar los SPLDs (Simple Programmable Logic Devices), los CPLDs (Complex Programmable Logic Devices) o las FPGAs (Field Programmable Gate Arrays)[163, 23].

El nacimiento de los dispositivos lógicos programables es bastante reciente. Los primeros no aparecieron hasta el año 1975. El motivo del desarrollo de estos circuitos se debe a la proliferación de los sistemas empotrados. Al popularizarse los sistemas digitales gobernados por microprocesadores, tanto los diseñadores como los científicos se dieron cuenta de las carencias que presentaban los circuitos implementados mediante tecnología VLSI [167, 142]. Los circuitos reconfigurables permiten un diseño muy sencillo y una reducción de costes con respecto a otros estilos semicustom, pero la ventaja que se ha ido imponiendo es la reprogramabilidad, que permite la modificación de prototipos y la corrección de fallos en tiempos mínimos.

En 1975 aparece un dispositivo fabricado por la firma Signetics, el 82s100 FPLA, que como sus siglas indican, era un array lógico programable. Estaba formado por 48 términos productos, con posibilidad para 16 entradas y ocho salidas. Dos años más tarde aparece la primera FPLA de la compañía Monolithic Memories, que mejoraba el consumo de potencia al cablear las OR de los arrays lógicos.

Sin embargo, el cambio más importante se produce cuando en 1985 Xilinx presenta su serie 2000. La aportación fundamental consiste en que los bloques lógicos que componen el circuito integrado, además de poderse programar, están compuestos por algo más que puertas AND y OR, lo que hace que el abanico de aplicaciones se abra espectacularmente. Paralelamente, Actel lanza al mercado las FPGA programables por antifusible.

A partir de ese momento el interés y el desarrollo de las FPGA ha evolucionado muy rápidamente y en el año 1987 Xilinx ya comercializa la serie 3000, que completa con la aparición en 1991 con la fabricación de la serie 4000. En el año 1998 presentó la serie Virtex, en 1999 la Virtex-E y en el 2000 la Virtex-EM.

Este breve recorrido histórico nos permite comprobar la relativa novedad de estos circuitos. Como consecuencia de la misma, las herramientas para ayudar al diseñador en el proceso de implementación de hardware mediante FPGAs no están desarrolladas hasta el punto deseado. Debido al avance de la tecnología y de las arquitecturas, se hacen necesarias herramientas automáticas capaces de manejar

especificaciones más complejas. Estos dos puntos son una motivación más del desarrollo de este trabajo.

4.1 ¿Qué es una FPGA?

La arquitectura de una FPGA consiste en una matriz o array de bloques lógicos que se pueden programar. Las FPGAs tienen tres componentes principales:

- bloques lógicos configurables
- bloques de entrada - salida
- bloques de conexión [154].

Los bloques lógicos configurables (CLBs) son los encargados de implementar la lógica del diseño. Están distribuidos en forma de Matriz en el circuito y serán nuestra principal referencia a la hora de hacer el proceso de partición. Por otro lado están los bloques de entrada y salida (IOBs), que son los encargados de conectar la FPGA con el mundo exterior. Este "mundo" exterior puede ser directamente la aplicación para la que esté diseñada o, como en nuestro caso en el que son necesarias varias FPGAs para implementar un circuito, el resto de las FPGAs. Por último están los bloques (switchboxes) y líneas de interconexión que son los elementos de los que dispone el diseñador para hacer el rutado del circuito. En ciertos casos en los que la ocupación de los CLBs no es total, estos se pueden utilizar también para llevar a cabo esta tarea. En la figura 4.1 se muestra la estructura interna de una FPGA. En ella están señalados los bloques lógicos, los IOBs y las matrices de interconexión.

La metodología de implementación de una FPGA es básicamente la misma que para un Gate Array. La entrada puede ser mediante un esquemático o usando un lenguaje de descripción de hardware. El fabricante suministra un software que convierte la descripción del diseño en el programa de la FPGA. El código resultante se puede cargar inmediatamente en el dispositivo y probar el diseño, lo que proporciona una manera muy sencilla y rápida de corregir fallos.

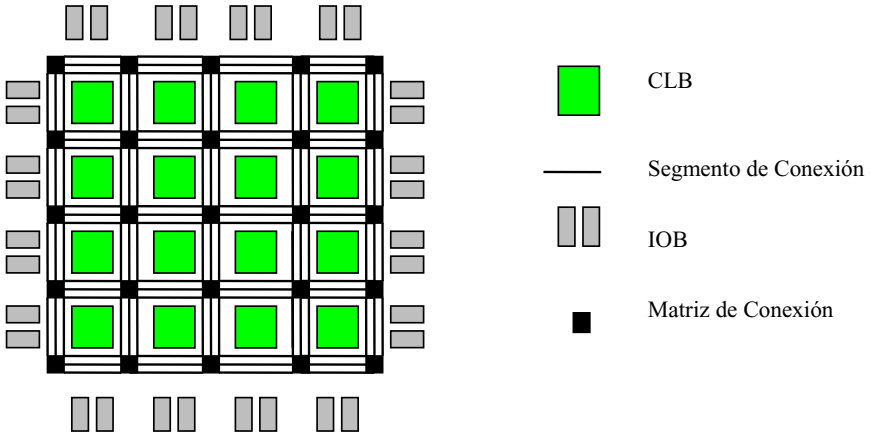

Figura 4.1: *Estructura general de una FPGA*

Las FPGAs se diferencian por la estructura y composición de los bloques lógicos, por sus estructuras de rutado y por la tecnología de programación de sus conexiones [138]. Las arquitecturas de rutado de una FPGA puede ser tan simple como una línea de conexión directa entre dos bloques o tan compleja como un multiprocesador (perfect suffle) [107, 160, 159].

4.1.1 Tecnologías FPGA

Para programar las FPGAs se utilizan diversas tecnologías [151, 97] pero las más importantes son:

- SRAM
- Antifusible
- Puerta flotante

Las FPGAs que se programan mediante SRAM, utilizan celdas RAM estáticas para controlar la puerta de paso o los multiplexores. Está tecnología la utilizan los circuitos fabricados por Xilinx, Plessey, Algotronic, Concurrent Logic y Toshiba. Como la SRAM es volátil, la FPGA se debe cargar y configurar en el momento de

encender el chip, lo que hace necesaria la existencia de una memoria externa que suministre los bits programados por el usuario. Actualmente, con las herramientas de desarrollo existentes, esto no representa un problema insalvable, pues el software permite realizar esta operación en poco tiempo en la mayoría de los casos. La principal desventaja de la SRAM es que necesita una gran cantidad de área para su implementación, pero presenta dos ventajas fundamentales como son que es muy rápida de reprogramar y que sólo requiere tecnología estándar de circuitos integrados para su desarrollo.

Un antifusible es un dispositivo de 2 terminales que cuando no está programado presenta una alta resistencia entre sus terminales. Cuando se le aplica un voltaje de entre 11 y 20 voltios, el antifusible se funde y crea una unión permanente de baja resistencia. Este tipo de tecnologías lo utilizan las casas Actel, Quicklogic y Crosspoint. Su mayor ventaja es el poco espacio que ocupan y la baja resistencia que se produce al realizar la programación. Además la capacidad parásita de un antifusible amorfo sin programar es significativamente más pequeña que en otras tecnologías de programación. Sin embargo tienen el gran inconveniente de no ser reprogramables

Las Tecnologías de puerta flotante están basadas en EPROMs (Erasable Programmable ROM) borrables por rayos ultravioleta. Esta tecnología la utilizan las casas Altera y Plus Logic. La forma de programarlas es decrementar el voltaje umbral de los transistores que de lo contrario quedan permanentemente apagados. La mayor ventaja de está tecnología es que se pueden borrar y volver a programar muy fácilmente. Además no es necesaria la utilización de una memoria externa para alimentarlas. Presenta algunos inconvenientes que las hacen poco adecuadas para nuestros propósitos. En primer lugar el flujo de diseño es más largo que en las otras tecnologías descritas. Otras dos desventajas son su alta resistencia de encendido y el alto consumo estático, debido principalmente a la resistencia de los transistores utilizados para suministrar el uno lógico o bloque de pull-up.

4.1.2 Arquitecturas de los bloques lógicos

Los bloques lógicos de las FPGA comercialmente más utilizadas son:

- Multiplexores
- Look Up Tables (LUTs)
- Estructuras AND/OR de alto fan-in.

La clasificación de los bloques lógicos se realiza por su granuralidad [61]. La granuralidad se puede definir en varias direcciones, el número de funciones booleanas que se pueden implementar con un bloque lógico por ejemplo, o el número de puertas NAND de dos entradas al que es equivalente. Otras formas de clasificación lo hacen por el número total de transistores necesarios, el área total necesaria para implementar el bloque o el número de entradas y salidas. Una forma más homogénea de hacer la clasificación es la que proponen Rose et al [151]., que consiste en agruparlos por bloques lógicos de granuralidad fina y gruesa.

Los bloques lógicos de granuralidad fina más importantes son los que se citan a continuación, precedidos por el nombre de la casa comercial que fabrica el dispositivo:

- Crosspoint: utiliza pares de transistores como módulo básico
- Plessey y Toshiba: NAND de 2 entradas
- Concurrent Logic: Una AND de 2 entradas y una OR de 2 entradas
- Algotronix: conjunto configurable de MUX

La mayor ventaja que ofrecen los bloques lógicos de grano fino es que se utilizan la mayoría de los bloques disponibles. La principal desventaja es que requieren un alto número de segmentos de línea y de interruptores programables. Como resultado las FPGAs que utilizan este tipo de bloques lógicos son en general más lentas y menos densas que las que utilizan bloques de granuralidad gruesa.

En general los bloques lógicos de granuralidad gruesa están formados por multiplexores y LUTs (Look Up Tables). Los LUTs son bloques que pueden implementar cualquier función lógica de tantas entradas como su orden indique. Por ejemplo un LUT de orden 6 puede implementar cualquier función lógica de 6 literales. Es decir

puede tomar 2^6 valores distintos. Los bloques lógicos de granuralidad gruesa más utilizados son los que implementan Actel, Quiclogic y Xilinx:

- *Actel:* Utiliza multiplexores y varía dependiendo de la familia de dispositivos. Por ejemplo, la ACT-1 incorpora un bloque lógico compuesto por 3 multiplexores y una puerta lógica. Tiene un total de 8 entradas y una salida, lo que hace que pueda implementar 702 funciones lógicas distintas. En la ACT-2 Actel fabrica 3 multiplexores y una puerta AND, lo que hace que pueda implementar 766 funciones lógicas distintas.

- *Quicklogic:* utiliza un multiplexor de 4 a 1. Los sistemas basados en multiplexores suministran un alto grado de funcionalidad utilizando un número relativamente pequeño de transistores. Aunque también es cierto que necesitan un mayor número de entradas.

- *Xilinx:* La base de los bloques lógicos de Xilinx es una SRAM funcionando como LUT. La tabla de verdad de una función lógica de K entradas queda almacenada en una SRAM de 2K x 1. La ventaja de las LUTs es que tienen una alta funcionalidad y versatilidad, aunque dependiendo del diseño pueden quedar bastantes sin utilizar. Como contrapartida, estos bloques que han quedado libres se pueden utilizar para hacer el rutado del circuito.

En la tabla 4.1 podemos ver un resumen de las principales características de los dispositivos disponibles más importantes. La tabla muestra junto al nombre distintivo de la familia de FPGAs, el dispositivo de mayor capacidad (Mayor), el bloque lógico (Bloque), el número de bloques lógicos disponibles para el dispositivo mayor (Capacidad), el número de pines de entrada-salida (PE/S) y la tecnología de programación de los bloques lógicos (Tecnología). Los bloques lógicos pueden ser para las familias presentadas en esta tabla o bien el CLB (FPGAs de Xilinx), o el LE (Logic element de Altera). Tanto los CLBs cómo los LEs están compuestos de LUTs y biestables (FFs), pero la composición varía de unas familias a otras.

En la tabla 4.1 también se puede comprobar que los circuitos disponen de un número muy reducido de pines de entrada salida, en comparación con el número de

Familia	Mayor	Bloque	Capacidad	PE/S	Tecnología
Xilinx 4000	XC40250XV	CLB	8464 CLBs	448	SRAM 0.25
Xilinx Virtex	XVC3200E	CLB	16000 CLBs	804	SRAM 0.18
Altera FLEX800	81500	LE	1296 LEs	208	SRAM 0.5
Altera FLEX 10k	EPF10K250	LE	12160 LEs	470	SRAM 0.25
Altera Apex 20k	EP20K1500K	LE	51840LE LEs	808	SRAM 0.18
Lucent ORCA	OR3L225B	LUT	11552 LUTs	612	SRAM 0.25

Tabla 4.1: *Características principales de distintas Familias de FPGAs*

bloques lógicos existentes dentro de la FPGA. Esto será por tanto una restricción importantísima a la hora de realizar un diseño sobre FPGAs.

4.1.3 Ventajas e inconvenientes de las FPGAs

Algunas de las ventajas de las FPGAs ya se han citado anteriormente. A continuación exponemos algunas de las más importantes [24, 23]:

- El tiempo de programación y puesta en el mercado se reduce considerablemente
- Son programables por el usuario. Esto aparte de dar mayor libertad al diseñador y permite una reducción de productos en stock, ya que se pueden utilizar para diferentes aplicaciones
- Algunos tipos son reprogramables, lo que las hace especialmente indicadas para procesos de prototipado en muchos diseños y permite la corrección de errores
- El proceso de diseño es muy simple y asequible.
- Existe una amplia gama de dispositivos que cubren las necesidades de usuarios de todo tipo.

- No necesita procesos de fabricación con máscaras

Los inconvenientes de las FPGAs se deben principalmente a su flexibilidad, lo que las hace que en ocasiones sean inapropiadas:

- Es un dispositivo más lento que otros sistemas de propósito específico, debido principalmente a los transistores y matrices de interconexión que utiliza. Para hacernos una idea aproximada los mecanismos de interconexión de una FPGA introducen aproximadamente entre el 30 y el 50 % del retardo total del circuito.

- En ocasiones se desaprovecha parte de la lógica para poder realizar el rutado completo del sistema.

Como acabamos de ver, los dispositivos FPGAs (Field Programmable Gate Arrays) presentan una gran número de ventajas entre las que podemos destacar la rapidez de implementación, la flexibilidad y su alta capacidad de reprogramación. Su principal inconveniente es su baja capacidad lógica, es decir, el número de puertas equivalentes que puede implementar, y el reducido número de pines de entrada-salida de los que dispone el diseñador para realizar el rutado del circuito, una vez dispuesta la lógica sobre las FPGAs.

Por ello, cuando el tamaño de los circuitos es considerable, es necesaria la utilización de SMFPGAs, que incluyen varios dispositivos de estas características.

4.2 Sistemas Multi-FPGA, (SMFPGAs)

En la actualidad los SMFPGAs se utilizan para multitud de aplicaciones, debido fundamentalmente a sus posibilidades para realizar prototipos y para hacer correcciones sobre los diseños implementados con un bajo costo [Hauck 95]. Entre estas aplicaciones podemos citar las siguientes:

- La emulación lógica de circuitos. Es decir, la verificación de circuitos integrados ya diseñados como paso previo a su fabricación [131, 169, 180].

- Sistemas reconfigurables capaces de llevar a cabo la misma tarea que un procesador pero con la ventaja de poder ser utilizados para distintas aplicaciones [13, 70, 76, 157, 80, 74].
- Implementaciones específicas para calculo númerico y problemas de optimización[39, 19, 81, 111].
- Realización de prototipos [92, 14, 123, 127].
- Sistemas reconfigurables dinámicamente en los que el hardware se va adaptando para distintas funciones durante la ejecución o utilización de un sistema [26, 68, 134, 133, 139, 181].
- Procesamiento digital de señales e imágenes [47, 69].

Este tipo de sistemas no sólo incluyen los circuitos integrados, si no que además suelen incorporan memorias y circuitería de conexión. La diferencias fundamentales que hay entre unos circuitos y otros son dos:

- La disposición de las FPGAs
- Los recursos de rutado

El flujo de diseño de los sistema para implementar circuitos en los SMFPGA se compone de tres tareas principales: partición, ubicación y rutado. Durante la partición se divide el circuito inicial en partes para poder implementarlas sobre una FPGA. Una vez obtenidas se determina su posición sobre la tarjeta o sobre el sistema de implementación, es la fase de ubicación. Finalmente, el rutado realiza las conexiones entre las distintas partes del ciruito, determinando el camino que deben tomar las distintas redes que componen el ciruito.

Una de las tareas más difíciles en el diseño de SMFPGA es la partición del sistema. Los principales problemas se encuentran en la dificultad de utilizar parte de la lógica de la FPGA debido a la limitación en el número de pines de entrada-salida (PE/S) disponibles. En muchas ocasiones dos de estas tareas se realizan de una manera conjunta, entre otras razones porque a la hora de realizar una de ellas hay que tener en consideración las otras. Por ejemplo, al realizar la partición debemos

tener en cuenta la rutabilidad del circuito así como la estructura y topología de las FPGAs.

La partición se puede realizar sin pensar en ninguna topología determinada, es decir, sin tener en cuenta ninguna restricción referente a una tarjeta. En este caso estaremos hablando de una *topología libre*. En realidad, una vez realizado este proceso habría que aplicar algoritmos de ubicación y rutado sobre la topología elegida y realizar un proceso de optimización con las particiones ya obtenidas. Para esta tarea existen otras herramientas como las presentadas en [51, 171, 172, 173, 8, 3, 106]

Lo más normal es realizar el proceso de partición atendiendo a las características propias de una tarjeta ya fabricada o de *topología fija*. Existe una amplia gama de SMFPGAs que van desde los más simples, con 2 FPGAs (Xilinx FPGA Demo Board) [99], hasta sistemas con módulos interconectables que se pueden ampliar tanto como sea necesario (Virtual Wires Emulation System) [9]. Podemos encontrar también SMFPGAs de propósito específico [67, 71].

Aunque la mayoría de los SMFPGAs de topología fija no obedecen a un patrón estricto en cuanto a su topología, sí que se pueden clasificar en dos grandes grupos:

- topología malla
- topología crossbar o grafo bipartito.

Las FPGAs en una topología malla están dispuestas en forma de matriz y están conectadas directamente con sus vecinas más próximas en las cuatro direcciones, salvo las de la periferia que estarán conectadas con los IOBs del sistema. Tiene dos ventajas fundamentales: su simplicidad y su facilidad de ampliación. Como desventajas podemos citar la rigidez de sus recursos de rutado y el desperdicio que se hace en ocasiones de la lógica debido a la falta de pines de entrada-salida. Por este motivo, es necesario realizar un proceso de partición que tenga en cuenta la ubicacicón para poder realizar una utilización mejor de estos recursos. En la figura 4.2 podemos ver una estructura de malla simple.

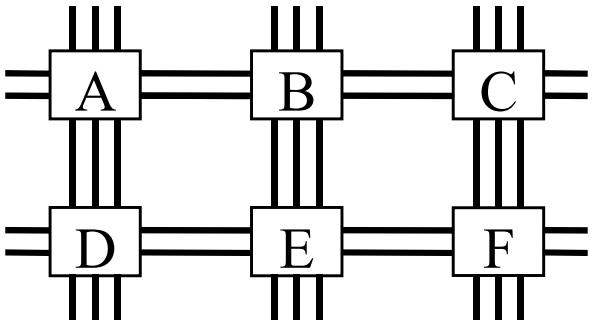

Figura 4.2: *SMFPGA de topología malla*

En las estructuras de grafo bipartito o Crossbar hay una diferenciación entre dos tipos de dispositivos, los que incluyen la lógica del sistema y los que soportan los recursos de rutado. Los circuitos de rutado están conectados únicamente con los circuitos de lógica y viceversa. Por lo tanto tenemos una separación clara entre dos zonas y de ahí su nombre. La idea es que si necesitamos conectar dos circuitos solamente tendremos que pasar por un único circuito intermedio, mientras que en otro tipo de topologías, como la de malla, tendríamos en algunos casos que atravesar varias FPGAs para llegar a la que queremos conectar. Al estar dispuestos los circuitos de una manera simétrica, es indiferente el dispositivo que escogemos para realizar la conexión pues la distancia será la misma. Sin embargo esta topología presenta dos inconvenientes. Al tener que estar interconectados todos los circuitos de lógica solamente con los de rutado, no es posible conectarlos con otros sistemas similares y por lo tanto no se puede expandir. Por otro lado, es evidente que se produce un mal aprovechamiento de los recursos de las FPGAs, al tener la mitad de nuestros circuitos destinados únicamente a las conexiones. La figura 4.3 es un ejemplo de este tipo de estructuras.

4.2.1 Algunos ejemplos de Sistemas Multi-FPGA

Como se ha podido comprobar a lo largo de este capítulo, son muchas las aplicaciones para las que se utilizan los SMFPGAs. Por este motivo también hay un gran número

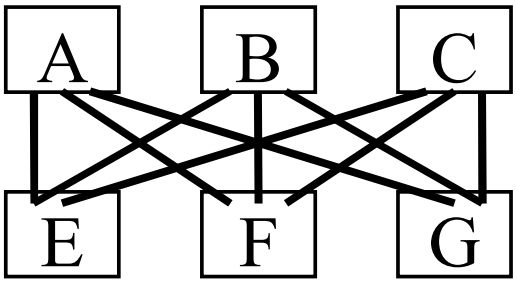

Figura 4.3: *Estructura o topología de grafo bipartito para sistemas Multi-FPGA*

de tarjetas basadas en FPGAs. A continuación veremos algunos ejemplos como muestra significativa.

4.2.1.1 Xilinx FPGA Demo Board

Es un ejemplo de SMFPGA sencillo. La placa incluye una XC4003 y una XC3020, además de una serie de interruptores y un regulador de tensión. Para programar este sistema se utiliza un interface hardware de conexión mediante el cual se le transmiten los datos desde de la computadora donde se este ejecutando el software de programación y prueba (XACT, M1, etc...).

Sus ventajas principales son su bajo costo y portabilidad, pues puede funcionar sobre diferentes plataformas. El inconveniente principal es su poca capacidad por lo que el tipo de diseños que puede implementar está muy limitado.

4.2.1.2 Sistema MP3 de Aptix

Es un sistema que permite la inclusión de hasta 6 circuitos programables. Esta compuesto por dos tipos de dispositivos con una estructura de grafo bipartito y permite además la inclusión de otros tipos de circuitos integrados o dispositivos para completar la implementación [135]. Al igual que la tarjeta de Xilinx, incluye un software de programación de los dispositivos y un analizador lógico que conecta ambos. Está especialmente diseñado para emulación de sistemas en tiempo real y

permite el desarrollo conjunto de funciones hardware y software. Como principales inconvenientes están primero su capacidad, pues realmente sólo disponemos de 3 circuitos programables para implementar la lógica de nuestro circuito. Segundo, presenta todos los inconvenientes anteriormente citados para las topologías de grafo bipartito.

4.2.1.3 Sistema Springbok

El sistema Springbok ha sido desarrollado en la Universidad de Washington [158]. Es un sistema de prototipado rápido para diseños a nivel de placa. Está compuesto por una placa base en la que se conectan los circuitos integrados y los buses de conexión. La ventaja es que estos circuitos integrados además de FPGAs pueden ser memorias, procesadores, elementos de entrada - salida, interruptores o cualquier otro tipo de elemento que sea necesario para la implementación del circuito. La placa base se puede conectar con otras placas iguales mediante unos pines de conexión laterales. A estas placas cuando se conectan varias, se les llaman placas hermanas.

Es una estructura de tipo malla y por lo tanto presenta los inconvenientes de las mismas. Sin embargo, es un sistema que posibilita una gran capacidad de comunicaciones entre los circuitos. Esto se debe a la estructura de su placa base en la que las conexiones son directas.

4.2.1.4 Sistema SPGA

Para finalizar esta revisión de los SMFPGAs vamos a ver un ejemplo de aplicación específica, el sistema SPGA. Se trata de un procesador hardware que implementa un algoritmo genético mediante un sistema de memorias y FPGAs conectadas [66].

El SPGA implementa el algoritmo genético mediante 4 FPGAs y cuatro memorias. Cada uno de los circuitos tiene asociado una memoria y se utiliza para implementar cada uno de los módulos del algoritmo genético. Los módulos implementados son selección, cruce, evaluación y mutación, y un último bloque para las estadísticas. En la figura 4.4 está un esquema de la implementación y funcionamiento del SPGA.

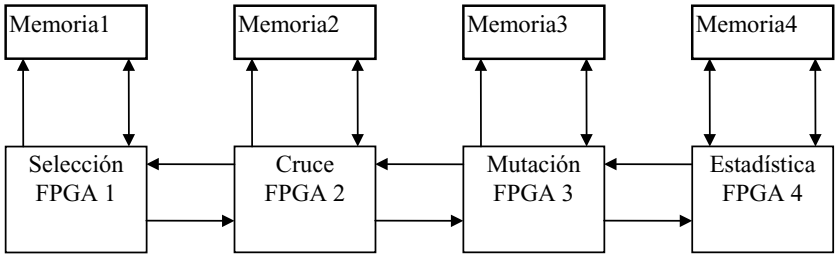

Figura 4.4: *Esquema de conexión y funcionamiento del sistema SPGA*

4.3 Partición y ubicación de Sistemas Multi-FPGA mediante AGs

Una vez seleccionada la topología, habrá que decidir qué parte de la lógica va a cada FPGA, la cantidad de FPGAs y la forma de conectarlas. Este es el problema de la partición en SMFPGAs.

Una primera aproximación al problema es considerar que las particiones del sistema se podrán implementar sobre una topología libre. Es decir, la partición del sistema se realiza sin tener en cuenta las restricciones propias de utilizar una determinada topología. Este problema se puede abordar desde el punto de vista de los algoritmos genéticos con una codificación bastante intuitiva y que ha sido utilizada en otras aproximaciones para realizar la partición de grafos.

Una vez resuelto este problema se puede pasar a estudiar sistemas más complejos que tienen una topología determinada de implementación. En estos sistemas es necesario imponer restricciones relativas al número de conexiones (pines) de entrada-salida disponibles y a la capacidad lógica de los dispositivos FPGA que componen la tarjeta sobre la que se está trabajando. Este tipo de problemas se conoce como partición para una topología fija. Para resolver este problema se ha tenido en cuenta que la estructura del circuito es muy importante para reducir los retardos del mismo una vez implementado. Por ello se ha diseñado un método de partición que conserva la estructura del circuito y que permite una codificación directa del problema.

Los problemas de partición de estas características son NP-completos, lo que hace necesaria la utilización de técnicas heurísticas para su solución [153]. Los algoritmos genéticos han demostrado ser una herramienta muy útil para la resolución de este tipo de problemas de optimización. En los dos próximos capítulos se describen los AGs utilizados para la resolución del problema de la partición, los resultados obtenidos así como las principales conclusiones que se han obtenido de los mismos.

Se han implementado tres tipos de algoritmos genéticos tanto secuenciales cómo paralelos:

- Algoritmos géneticos simples o tradicionales
- Algoritmos géneticos compactos
- Algoritmos géneticos compactos híbridos, que incluyen una mejora local dentro del AG compacto

En primer lugar se hace un estudio de los SMFPGAs de topología libre y a continuación se resuelve el problema para una topología fija de 8 FPGAs conectada en forma de malla.

Capítulo 5

Técnicas de partición para los SMFPGA de topología libre

Definimos un SMFPGA de topología libre cómo aquel en el que las FPGAs se pueden conectar entre sí como desee el diseñador. A pesar de que las topologías libres presentan el inconveniente de necesitar el desarrollo posterior en un sistema específico, hemos realizado un estudio de la partición para este tipo de sistemas como una primera aproximación para abordar topologías más complejas que incluyan un mayor número de restricciones.

5.1 Formulación del problema

La figura 5.1 describe el flujo de diseño e implementación de un SMFPGA de topología libre para FPGAs de Xilinx. En primer lugar se realiza una descripción mediante un esquemático o mediante un lenguaje de descripción de hardware del circuito a implementar. A continuación aplicaremos sobre dicha descripción la herramienta XACT de Xilinx. El resultado será una lista de conexiones o Netlist que incluye el número de CLBs y de IOBs necesarios para la implementación del sistema sobre las FPGAs de Xilinx. Llegado a este punto es necesario determinar la distribución óptima de los CLBs sobre las distintas FPGAs disponibles. Una distribución óptima será aquella que suponga un coste mínimo y asegure la rutabilidad interna de cada una de las FPGAs.

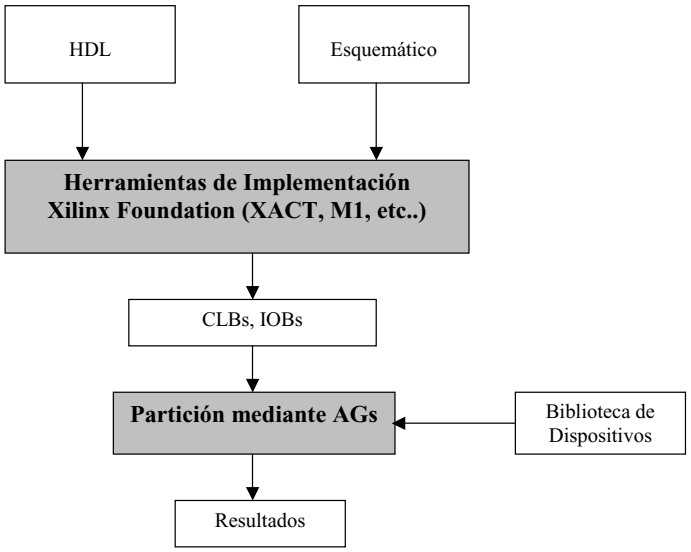

Figura 5.1: *Ciclo de diseño para un SMFPGA de topología libre*

Para obtener tanto el número de CLBs cómo el número de bloques de entrada-salida necesarios para obtener la implementación del circuito, es necesario decidir que tipo de FPGA se va utilizar. Esto se debe a que la estructura de los bloques lógicos no es la misma para todos los dispositivos y unos tienen una mayor capacidad lógica que otros.

La entrada a nuestro algoritmo debe incluir por lo tanto el número de CLBs necesarios para implementar el circuito. Para poder evaluar las distintas soluciones es necesario también disponer de una biblioteca de FPGAs. Esta biblioteca incluye el número de CLBs y el coste de cada FPGA.

La codificación utilizada en el AG es perfectamente adaptable a cualquier tipo de FPGA cómo se explica en el apartado 5.3. La implementación se ha realizado teniendo en cuenta los datos correspondientes a los 3 dispositivos más sencillos de la serie 3000 de Xilinx; XC3020, XC3030 y XC3042. Esta elección se ha realizado para permitir una comparación de resultados con los obtenidos mediante el sistema PROP que se explicó en el capítulo 2 y que vienen recogidos en [114], dado que es

el que ha obtenido los resultados más significativos.

Con todo esto podemos formular el problema de la partición de un circuito para su implementación sobre topologías libres, de una manera más formal, cómo sigue:

Sea un circuito objetivo C, implementable mediante un número de CLBs, $N_{CLB_{tot}}$. Sea un conjunto Set, de tipos de dispositivos FPGA disponibles (F_i), tales que $Set = \{F_1, F_2, ...F_n\}$, un coste $Cost_i$ y una capacidad lógica CL_i asociados a cada tipo de circuito F_i. Sea un límite de ocupación L_i (con $0 \leq L_i \leq 1$) de cada F_i, necesario para asegurar su rutabilidad interna y n el número total de tipos de FPGAs. El objetivo de la herramienta será encontrar el número N_i de FPGAs de cada tipo i con una ocupación lógica N_{CLB_i} que cumplan para n dispositivos:

$$\sum_{i=1}^{n} N_i \cdot Cost_i \quad es \;\; mínimo \tag{5.1}$$

$$\forall Fi, \quad N_{CLB_i} < L_i \cdot CL_i \cdot N_i \tag{5.2}$$

$$\sum_{i=1}^{n} N_{CLB_i} = N_{CLB_{tot}} \tag{5.3}$$

Dicho de otro modo, encontrar las FPGAs que permiten implementar el circuito objetivo minimizando el coste y asegurando la rutabilidad.

5.2 AG secuencial

El algoritmo genético es el que se encarga de realizar la tarea de partición. La primera implementación que se presenta es una versión secuencial del mismo que denominaremos de ahora en adelante *SAG*. También se han implementado tres versiones paralelas con distintas topologías de comunicación. Estos algoritmos paralelos tienen su base en el SAG, dado que utilizan la misma codificación y que representan distintas distribuciones del mismo sobre un conjunto de procesadores que intercambian información cada cierto tiempo. El esquema general del algoritmo génetico simple secuencial viene descrito a continuación en forma de pseudocódigo:

$N_{CLB_{tot}} \Leftarrow num\_bloques$

Leer información de los dispositivos
$g \Leftarrow$ número de generaciones
$poblac \Leftarrow$ población inicial
$ind \Leftarrow$ número de individuos
for $i = 1$ to g **do**
  for $k = 1$ to n **do**
    Obtiene_ocupación($poblac$, N_{CLB_i})
  end for
  for $j = 1$ to ind **do**
    Calcula_coste($poblac$, FC_j)
  end for
  Obtiene_mejor($best$)
  Selecciona($poblac$, $poblac_{sel}$)
  Cruza($poblac_{sel}$, $poblac_{cru}$)
  Muta($poblac_{cru}$, $poblac_{mut}$)
  $poblac \Leftarrow poblac_{mut}$
  Obtiene_peor($worst$)
  $worst \Leftarrow best$
end for
for $i = 1$ to n **do**
  Obtiene_ocupación($poblac$, N_{CLB_i})
end for
for $j = 1$ to ind **do**
  Calcula_coste($poblac$, FC_j)
end for
Mostrar_Resultados

5.3 Codificación

Como se puede comprobar se trata de un algoritmo genético simple con la misma estructura del explicado en el apartado 3.1. El algoritmo implementa el elitismo mediante tres funciones *Obtiene_mejor()*, *Obtiene_peor()* y *Sustituye_Peor()*. La evaluación de los individuos se realiza mediante dos pasos. Primero se calcula la

ocupación de cada tipo de dispositivo N_{CLB_i}, y posteriormente se calcula su valor de la función de coste en la que se incluyen las condiciones de rutabilidad.

Se han tratado dos problemas distintos. En primer lugar para estudiar la convergencia y válidez del algoritmo se ha utilizado el AG para resolver un problema de partición con sólo dos tipos de FPGAs disponibles. Una vez comprobado el correcto funcionamiento del mismo, se ha ampliado a un problema con 3 tipos de FPGAs cómo ejemplo de cómo se puede adaptar a particiones con distintos tipos de FPGAs. Tambíen se explica cómo se realizaría la adaptación a cualquier otro conjunto de dispositivos ya sea hómogeneo o heterógeneo.

Codificación para 2 tipos de FPGAs

Los individuos representan una distribución de los CLBs en los distintos tipos de FPGAs disponibles. Suponemos que podemos utilizar tantas FPGAs de cada tipo como sean necesarias. Para representar esta distribución cada individuo tiene tantos genes como números de CLBs nos haya indicado la herramienta XACT. Las dimensiones del alfabeto dependerán de la cantidad de FPGAs diferentes que podamos utilizar.

Para 2 FPGAs nos bastará utilizar un alfabeto binario. Vamos a suponer que disponemos de 2 tipos de FPGAs de la serie 2000 de Xilinx, la $XC2064$ y la $XC2018$. El valor que pueden tomar los genes (alelos) puede ser 0 ó 1. Cada gen nos indicará sobre que tipo de dispositivo se va a implementar el CLB al que representa. La tabla 5.1 representa la relación entre el valor del gen y el tipo de FPGA y expresa algunas características de los dispositivos.

Alelo	Dispositivo	Coste Normalizado	CLBs
0	XC2064	1.00	64
1	XC2018	1.36	100

Tabla 5.1: *Alfabeto del algoritmo genético y su codificación para dos tipos de FPGA*

Veamos un ejemplo sencillo de la codificación. Supongamos que tenemos un circuito que necesita 10 CLBs para su implementación sobre la serie 2000. Una

Alelo	Dispositivo
0	XC3020
1	XC3030
2	XC3042

Tabla 5.2: *Alfabeto del algoritmo genético y su codificación*

solución puede ser:

0 1 0 1 1 1 1 1 1 1

Esto nos indicaría que los CLBs primero y tercero se asignarían a una FPGA del tipo XC2064 y el resto a una del tipo XC2018. Dado que el número de CLBs asignado a una y a otra no supera el número de CLBs disponibles en cada dispositivo, sólo necesitaríamos un dispositivo de cada clase para la implementación del circuito.

Codificación para 3 tipos de FPGAs

La extensión de la codificación para utilizar un número mayor de dispositvos FPGA es muy sencilla. Lo único que hay que hacer es ampliar el alfabeto para que pueda representar más tipos de FPGAs. Los resultados experimentales en los que se compara la eficacia del algoritmo genético con la herramienta presentada en [114], están obtenidos para una implementación con 3 tipos distintos de circuitos. Los circuitos elegidos son de la serie 3000 de Xilinx, que son los utilizados en el trabajo mencionado anteriormente. Por ello ahora ya no basta una codificación binaria, sino que debemos usar una codificación entera compuesta por 3 valores, de acuerdo a la tabla 5.2.

Para comprender mejor el significado de esta codificación veamos un ejemplo. Supongamos que después de aplicar la herramienta XACT a un circuito sencillo hemos obtenido un circuito compuesto por 6 CLBs. Una posible solución al problema en forma de código genético sería :

0 1 1 2 2 1

Esto nos indicaría que el primer CLB iría implementado sobre una XC3020, el

segundo, el tercero y el sexto sobre una XC3030 y el cuarto y el quinto sobre una FPGA del tipo XC3042.

Codificación para k tipos de FPGAs

Si quisiéramos utilizar cualquier otro conjunto de FPGAs no tendríamos más que hacer corresponder cada valor del gen con un tipo distinto de FPGAs e incluir la información de los dispositivos en la función de coste. Al igual que se ha hecho para el caso de 3 FPGAs estó se puede ampliar para cualquier otro conjunto sin más que extender el alfabeto. Supongamos por ejemplo, que se dispone de 7 FPGAs distintas:

- 2 tipos de FPGAs de la serie 2000 de Xilinx, la $XC2018$ y la $XC2064$.
- 3 tipos de FPGAs de la serie 3000 de Xilinx, la $XC3020$, la $XC3030$ y la $XC3042$.
- 2 tipos de FPGAs de la serie 4000 de Xilinx, la $XC4010$ y la $XC4020$.

en este caso el alfabeto sería también entero y cada valor representaría un tipo de FPGA acorde a la tabla 5.3.

Alelo	Dispositivo
0	XC2018
1	XC2064
2	XC3020
3	XC3030
4	XC3042
5	XC4010
6	XC4020

Tabla 5.3: *Ejemplo de codificación para 7 tipos de FPGAs de Xilinx*

Un ejemplo para un circuito de 15 CLBs podría ser descrito por el cromosoma:

0 1 6 5 4 2 2 3 2 6 6 2 2 3 0

que correspondería a una asignación de CLBs de acuerdo a la tabla 5.4.

Alelo	Dispositivo	CLBs
0	XC2018	1,15
1	XC2064	2
2	XC3020	6,7,9,12,13
3	XC3030	8,14
4	XC3042	5
5	XC4010	4
6	XC4020	3,10,11

Tabla 5.4: *Ejemplo para 7 tipos de FPGAs de Xilinx*

Es posible también adaptar la codificación y todo el método a un sistema heterógeneo. Para ello, sería preciso realizar un estudio con las distintas herramientas de correspondencia con la tecnología que se utilizasen, ya que los bloques lógicos, y por lo tanto lo que representa cada alelo, variaría con cada fabricante. Por supuesto habría que incluir otras consideraciones en la función de coste como las referentes a los pines de cada dispositivo o su capacidad lógica.

5.4 Función de coste

Como se ha explicado, la función de coste puede coincidir con la función objetivo o no. Para la resolución de los problemas de topología libre que se expone en este apartado se han utilizado dos funciones de coste. Una para el problema de dos tipos de FPGA, que coincide con la función objetivo del problema. Mientras que para el caso de 3 FPGAs se ha utilizado una función de coste que mide otros parámetros no recogidos explícitamente en la función objetivo, pero que guían correctamente al algoritmo en la búsqueda de soluciones.

Función de coste para dos tipos de FPGAs

Recordemos que el problema tenía cómo objetivo comprobar la eficacia y el buen funcionamiento del AG. Por ello se ha utilizado una función de coste sencilla que coincide con la función objetivo. El objetivo de la partición es minimizar el coste de la implementación final y por este motivo se ha utilizado la función expresada en la ecuación 5.4.

$$FC_1 = N_{xc2018} \cdot Cost_{xc2018} + N_{xc2064} \cdot Cost_{xc2064} \tag{5.4}$$

donde N_{XC2018} y N_{XC2064} es el número de dispositivos necesarios del tipo $XC2018$ y del $XC2064$, y $Cost_{xc2018}$ y $Cost_{xc2064}$ su coste normalizado respectivamente.

Función de coste para tres tipos de FPGAs

Para el sistema con tres tipos de dispositivos disponibles, se ha incorporado una restricción de rutabilidad al problema anterior. A la hora de realizar la partición del sistema, hay que tener en cuenta varios factores. En primer lugar el número de FPGAs debe ser el menor posible para reducir el coste total. Por otro lado estudios experimentales indican que la ocupación de los CLBs no debe superar en ningún caso el 80 % si queremos que el circuito sea rutable. Estas consideraciones se han de incorporar a la función de coste que guía al algoritmo en la búsqueda de soluciones. Por ello se ha desarrollado la función de coste 5.5 que tiene tres términos (al tratarse de un problema de minimizacíon se utiliza la función inversa como función de coste).

$$FC_2 = \frac{1}{\sum_{i=1}^{3}[(N_i \cdot CL_i - N_{CLB_i}) \cdot k_i + Cost_i + Penalty_i]} \tag{5.5}$$

La notación utilizada es la misma que la que se utilizó para realizar el planteamiento del problema. Es decir, N_{CLB_i} es el número de CLBs asignado a cada tipo de dispositivo, CL_i su capacidad lógica, N_i el número de FPGAs necesarias del tipo i y $Cost_i$ su coste normalizado.

El primer término recoge la minimización del número de FPGAs, minimizando el número de CLBs que quedan libres. Por ello calcula la diferencia entre los CLBs utilizados N_{CLB_i} y los disponibles $(N_i \cdot CL_i)$. Además tendremos una constante distinta, k_i para cada tipo de FPGA que se determina experimentalmente y que indica la influencia sobre el coste de dejar un CLB sin ocupar en cada uno de los tipos. Se han estudiado empíricamente y se han seleccionado aquellos que han dado mejores resultados después de un estudio estadístico. Los valores adoptados son $k_1 = 300$, $k_2 = 500$ y $k_3 = 1000$.

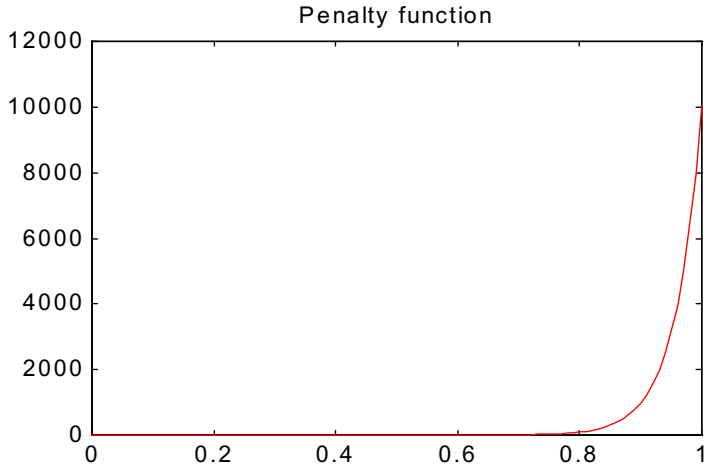

Figura 5.2: *Función de penalización. Representa la penalización en función de la ocupación de CLBs para cada tipo de FPGA*

El segundo término calcula el coste del sistema final, evaluando el número de FPGAs de cada tipo y multiplicándolo por su coste normalizado, para obtener el coste total de la misma forma que se hizo en la ecuación 5.4 pero para los 3 tipos de FPGAs, es decir cómo indica la ecuación 5.6.

$$Cost_i = N_{xc3020} \cdot Cost_{xc3020} + N_{xc3030} \cdot Cost_{xc3030} + N_{xc3042} \cdot Cost_{xc3042} \qquad (5.6)$$

donde N_{XC3020}, N_{XC3030} y N_{XC3042} es el número de dispositivos necesarios del tipo $XC3020$ del $XC3030$ y del $XC3042$, y $Cost_{xc3020}$, $Cost_{xc3030}$ y $Cost_{xc3042}$ su coste normalizado respectivamente.

El tercer término $Penalty_i$ es el encargado de asegurar la rutabilidad. Por ello, penaliza a todos aquellos individuos que representen soluciones con un índice de ocupación OR superior al 0.8. Para ello se utiliza la función representada en la figura 5.2 y expresada en la ecuación 5.7:

$$Penalty_i = \begin{cases} k_a \cdot e^{k_b \cdot OR} & \text{si } OR > 0.8 \\ 0 & \text{en otro caso} \end{cases} \qquad (5.7)$$

donde k_a y k_b son constantes que se han calculado experimentalmente y OR viene dado por la expresión 5.8.

$$OR = \frac{N_{CLB_i}}{CL_i \cdot N_i} \tag{5.8}$$

La función objetivo de éste problema es solamente el segundo término de los explicados anteriormente, minimizar el coste del diseño final.

5.5 Operadores genéticos

Los operadores genéticos que se han utilizado son los mismos para todos los AGs orientados a una topología libre. Se han escogido unos operadores sencillos por su facilidad de implementación y sus buenos resultados:

- *Operador de selección*: La estrategia utilizada, que se encarga de elegir a los padres para el procedimiento de cruce, es la selección por el método de la ruleta, que se explicó con detalle en el capítulo 3.
- *Operador de Cruce*: Se ha utilizado un operador de cruce simple por un punto que nos permite garantizar la transferencia de características de padres a hijos y conservar la aleatoriedad suficiente como para explorar varios espacios de búsqueda. La probabilidad de cruce se ha fijado en 0.8.
- *Operador de Mutación*: Realiza cambios aleatorios en el código de los individuos con una frecuencia dada por la probabilidad de mutación. Si se produce un cambio en un individuo indica que uno de los CLBs pasa de estar en un circuito de un tipo a otro. La probabilidad de mutación utilizada es 0.02.

5.6 AGs Paralelos

Un proceso es una copia de un programa o de una parte de él. Un programa es paralelo si en cualquier momento de su ejecución puede ejecutar más de un proceso. Para crear pogramas paralelos eficientes hay que poder crear, destruir y especificar

procesos, así como la iteración entre ellos. Básicamente existen tres formas de paralelizar un programa [147]:

- *Paralelización de grano fino*: la paralelización del programa se realiza a nivel de instrucción.

- *Paralelización de grano medio*: los programas se paralelizan a nivel de bucle. Esta paralelización se realiza habitualmente de una forma automática en los compiladores.

- *Paralelización de grano grueso*: se basan en la descomposición del dominio de datos entre los procesadores, siendo cada uno de ellos el responsablede realizar los cálculos sobre sus datos locales.

En esta tesis se han implementado únicamente estrategias de grano grueso cuyo mayor atractivo es la portabilidad, ya que se adapta perfectamente tanto a multiprocesadores de memoria distribuida como de memoria compartida. Este tipo de paralelización se puede a su vez realizar siguiendo tres estilos distintos de programación: paralelismo en datos, programación por paso de mensajes y programación por paso de datos.

- *Paralelismo en datos*: El compilador se encarga de la distribución de los datos guiado por un conjunto de directivas que introduce el programador. Estas directivas hacen que cuando se compila el programa los datos se distribuyan entre los procesadores disponibles. Como principal ventaja presenta su facilidad de programación. Sin embargo suelen tener una eficiencia inferior a la que se consigue con el paso de mensajes. Los lenguajes de paralelismo de datos más utilizados son el estándard HPF (High Performance Fortran) y el OpenMP [94, 143].

- *Programación por paso de mensajes*: El método más utilizado para programar sistemas de memoria distribuida es el paso de mensajes o alguna variante del mismo. Básicamente consiste en que los procesos coordinan sus actividades mediante el envio y la recepción de mensajes. Las principales ventajas que presentan son la flexibilidad, la eficiencia, la portabilidad y la controlabilidad del programa. Por contra el tiempo de desarrollo puede ser más elevado que

para un Paralelismo en Datos. Las librerías más utilizadas son por este orden la estándar MPI (Message Passing Interface) [136] y PVM (Parallel Virtual Machine) [62].

- *Programación por paso de datos*: A diferencia del modelo de paso de mensajes, la transferencia de datos entre los procesadores se realiza con primitivas unilaterales tipo *put-get*, lo que evita la necesidad de sincronización entre los procesadores emisor y receptor. Es un modelo de programación de muy bajo nivel pero muy eficiente, aunque en la actualidad son muy pocos los fabricantes que los soportan.

En este trabajo se han utilizado la programación por paso de mensajes mediante MPI por los motivos que se exponen a continuación:

- Los programas paralelos implementados con MPI presentan una alta portabilidad y eficiencia.
- La facilidad de paralelización de los AGs y su estructura bastante regular, hace relativamente sencillo su implementación de AGs paralelos utilizando MPI.
- Las máquinas utilizadas son el Cray T3E y una red de estaciones. El paso de mensajes es el método más eficiente en redes de estaciones y se adapta perfectamente a una máquina de memoria distribuida como el Cray T3E.

Los principales métodos de paralelización de AGs consisten en la división de la población en varias sub-poblaciones *(demes)*. Por ello, el tamaño y distribución de la población entre los distintos procesadores será uno de los factores fundamentales a la hora de paralelizar un algoritmo evolutivo. A continuación veremos una clasificación de los algoritmos genéticos paralelos y el funcionamiento básico de cada uno de ellos.

5.6.1 Clasificación de los AGs paralelos

La computación paralela se ha convertido en una parte fundamental en todas las áreas de cálculo científico, ya que permite la mejora del rendimiento simplemente con la utilización de un mayor número de procesadores, memorias y la inclusión de elementos de comunicación que permitan a los procesadores trabajar conjuntamente

para resolver un determinado problema [137].

Al compartir la carga de trabajo entre N procesadores se puede esperar que el sistema trabaje N veces más rápido que con un solo procesador, lo que permite tratar problemas más grandes y complicados. Las cosas no son tan sencillas, ya que existen varios factores de sobrecarga que hacen disminuir el rendimiento previsible. En ocasiones existen problemas cuya estructura no es lo suficientemente regular cómo para obtener rendimientos similares a los esperados. Otras veces los algoritmos y las técnicas utilizadas no son fáciles de paralelizar. Sin embargo, hay otros, como por ejemplo los algoritmos genéticos, que tienen una estructura que se adapta perfectamente a la paralelización. De hecho la evolución natural es en sí un proceso paralelo ya que evoluciona utilizando varios individuos.

Existen varias formas de paralelizar un algoritmo genético [49]. La primera y más intuitiva es la global que consiste básicamente en paralelizar la evaluación de los individuos manteniendo una única población. Otra forma de paralelización global consiste en realizar una ejecución de distintos AGs secuenciales simultáneamente. El resto de aproximaciones dividen la población en subpoblaciones que evolucionan por separado e intercambian individuos cada cierto número de generaciones. Si las poblaciones son pocas y grandes, tenemos la paralelización de grano grueso [18, 91]. Si el número de poblaciones es grande y con pocos individuos en cada población tenemos la paralelización de grano fino [128]. Por último, existen algoritmos que mezclan propiedades de estos dos últimos y que se denominan mixtos. En la figura 5.3 podemos ver un esquema de la clasificación de los AGPs.

Además de conseguir tiempos de ejecución menores, al paralelizar un AG estamos modificando el comportamiento algorítmico, y esto hace que podamos obtener otras soluciones y experimentar con las distintas posibilidades de implementación y los distintos factores que influyen en ella. Estas poblaciones van evolucionando por separado para detenerse en un momento determinado e intercambiar los mejores individuos entre ellas.

Técnicamente hay 3 características importantes que influyen en la eficiencia de

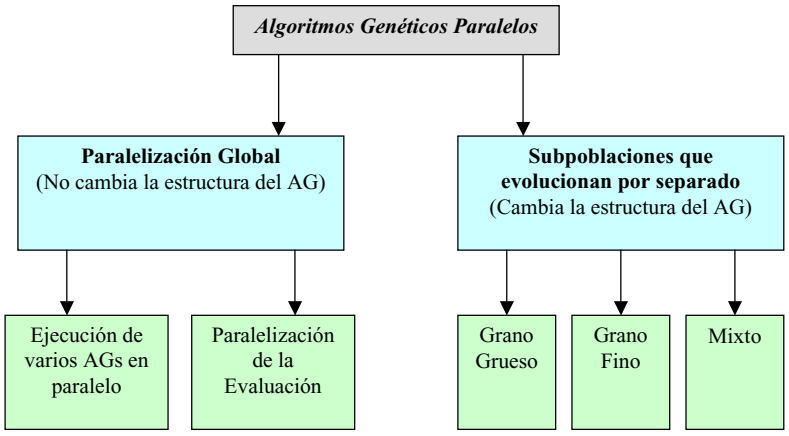

Figura 5.3: *Clasificación de los Algoritmos genéticos paralelos*

un algoritmo genético paralelo y que se analizarán más a fondo en el apartado 5.6.3 [18]:

- La topología que define la comunicación entre subpoblaciones.
- La proporción de intercambio: número de individuos a intercambiar
- Los intervalos de migración: periodicidad con que se intercambian los individuos.

5.6.2 Paralelización global

La paralelización global es la forma más sencilla de implementar un algoritmo genético paralelo. Cómo ya se ha dicho, consiste o en paralelizar la evaluación de los individuos o en realizar una ejecución simultánea de distintos AGs secuenciales. Sin embargo es muy útil, ya que permite obtener mejoras en el rendimiento con respecto al AG secuencial muy fácilmente sin cambiar la estructura principal de éste. En la mayoría de las aplicaciones de los AGs la parte que consume un mayor tiempo de cálculo es la evaluación de la función de coste. Un claro ejemplo de ésto es el problema tratado en el capítulo 6, la partición en SMFPGAs de topología fija. En estos casos se puede ahorrar mucho tiempo de cálculo simplemente encargando la evaluación de una parte de la población a distintos procesadores y de una forma

simultánea.

Normalmente la evaluación de un individuo cuesta exactamente igual para todos ellos y el tiempo de cálculo se puede disminuir aproximadamente en N veces siendo N el número de procesadores. Evidentemente habrá que tener en cuenta el costo de comunicaciones y cuánto más sencilla sea la información a enviar menor será este. También habrá que tener en cuenta el tipo de arquitectura que se esté utilizando y su facilidad para transmitir un tipo de datos u otro. En la figura 5.4 se puede ver una descripción de un algoritmo genético en el que se ha paralelizado la función de coste.

Como se puede ver este método mantiene una población única y la evaluación de los individuos se realiza en paralelo. La aplicación de los operadores puede mantenerse global o realizarse en paralelo aunque generalmente el costo de comunicaciones necesario para paralelizar estas operaciones no compensa el tiempo de computo ahorrado.

```
generación de la población inicial
while no se cumpla la condición de parada do
  for i = 1 to número de procesadores do
    in parallel
    evaluación de los individuos
  end for
  selección
  producción de nuevos individuos
  mutación
end while
```

Figura 5.4: *Esquema de un AG con la evaluación paralelizada..*

Cuando el programa se para y espera el resultado de la evaluación para todos los individuos antes de proceder a crear la siguiente generación, se dice que es una implementación síncrona. Este algoritmo tendrá las mismas características que un algoritmo secuencial. Si el procesador maestro no espera la llegada de todas las evaluaciones tenemos una implementación asíncrona. En este caso la estructura y comportamiento del algoritmo difiere de la versión secuencial ya que la generación de los nuevos individuos no se realiza de la misma forma. No se produce en el mismo

instante de tiempo, por lo que determinados individuos pueden ser seleccionados en una generación anterior o posterior. Puede ser incluso que no sean seleccionados, bien porque su evaluación llega tarde y la calidad de la nueva población ha aumentado o bien por la mera probabilidad intrínseca a los algoritmos evolutivos. La mayoría de las implementaciones de un AGP global suelen ser síncronas por su facilidad de realización.

La otra forma de paralelización global consiste únicamente en enviar varios algoritmos a distintos procesadores y al final del proceso ver la mejor solución. El resultado es el mismo que si ejecutáramos varios AGs secuenciales y escogiéramos la mejor solución de todas las obtenidas. En la Figura 5.5 se puede ver un esquema de este método en forma de pseudocódigo.

```
for i = 1 to número de procesadores do
  in parallel
  generación de la población inicial
  while no se cumpla la condición de parada do
    evaluación de los individuos
    selección
    producción de nuevos individuos
    mutación
  end while
end for
Escoger la mejor solución
```

Figura 5.5: *Esquema de un AG con varias poblaciones que evolucionan en paralelo.*

El modelo de paralelización global no hace ninguna distinción sobre la arquitectura del computador sobre el que se está ejecutando. Se puede implementar tanto en un computador de memoria compartida como de memoria distribuida. En un multiprocesador de memoria compartida, la población se puede guardar en la memoria compartida y cada uno de los procesadores puede leer los individuos que tiene asignados, evaluarlos y devolver los resultados de tal forma que no se produzcan conflictos entre procesadores para acceder a la memoria y no hace falta sincronización en este paso. Los problemas pueden aparecer únicamente como consecuencia de la propia red que pueden hacer disminuir la velocidad de ejecución del algoritmo.

El número de individuos asignado a cada procesador suele ser constante, pero en algunos casos puede ser necesario equilibrar la carga entre procesadores, para lo que se puede utilizar cualquier algoritmo dinámico diseñado para este propósito [147]. El equilibrio de la carga no es nada más que distribuir homogéneamente la cantidad de trabajo que realiza cada procesador de acuerdo a sus características. Todos los problemas tratados en este capítulo producen un equilibrio natural de la carga por lo que no ha sido necesario utilizar ninguna técnica adicional en este sentido.

En un computador de memoria distribuida la población normalmente se almacena en un procesador (maestro) que se encarga de enviar los individuos al resto de procesadores (esclavos), de recoger la información y de aplicar los operadores. En cualquier caso el costo de comunicaciones es similar para un multiprocesador de memoria distribuida y uno de memoria compartida, la diferencia es que en un procesador de memoria distribuida se debe especificar explícitamente. Cuánto mayor es el número de esclavos utilizados mayor es el costo de comunicaciones, pero evidentemente también disminuye el número de operaciones que tiene que realizar cada uno de los procesadores.

Cantú Paz ha realizado un análisis exhaustivo del tiempo de ejecución de un algoritmo genético con paralelización global que vemos a continuación, dado que justifica matematicamente la paralelización [30, 33, 31, 35]. El análisis se centra en el tiempo de ejecución del procesador maestro. El tiempo que el procesador maestro está desocupado (T_{idle}) se puede obtener restando al tiempo que tarda el primer procesador esclavo en terminar la tarea ($T_c + T_{comp}$), el tiempo que tarda el maestro en enviar la población a los esclavos ($S\Delta T_c$). Donde S es el número de procesadores, T_{comp} es el tiempo de computo y T_c el tiempo que se tarda en enviar la información o tiempo de comunicaciones. Con todo ello obtenemos la ecuación 5.9.

$$T_{idle} = T_{comp} - (S-1) \cdot T_c \tag{5.9}$$

Como la población se divide en partes iguales entre los esclavos para evaluarla,

el tiempo que tardan en evaluar viene dado por la ecuación 5.10:

$$T_{comp} = \frac{n \cdot \alpha}{2} \tag{5.10}$$

donde n es el número de individuos de la población y α es el tiempo que se tarda en evaluar un individuo. El tiempo de ejecución del procesador maestro T_{tot} será la suma del tiempo de comunicaciones (envio y recepción de información) y el tiempo de inactividad cuya expresión se encuentra en la ecuación 5.11.

$$T_{tot} = 2 \cdot S \cdot T_c + T_{idle} = \frac{n \cdot \alpha}{S} + (S-1) \cdot T_c \tag{5.11}$$

Como muestra la ecuación, cúanto mayor es el número de esclavos (S), menor es el tiempo de cálculo, pero también aumenta el tiempo de comunicación. Hallemos para que valor de S el tiempo total es mínimo:

$$\frac{\partial T_{tot}}{S} = 0 \tag{5.12}$$

$$S^* = \sqrt{\frac{n \cdot \alpha}{T_c}} \tag{5.13}$$

En la ecuación 5.13 se ha supuesto que T_c es constante. Sin embargo, en muchas ocasiones el costo de intercambio de información entre dos procesadores depende linealmente de la cantidad de información y en nuestro caso ésta depende del número de esclavos. Por lo tanto la ecuación 5.14 es más conveniente para T_c.

$$T_c = A \cdot size + C \tag{5.14}$$

$$size = \frac{n \cdot l}{S} \tag{5.15}$$

donde A y C son constantes que dependen del hardware utilizado. C es un coste de sobrecarga fijo asociado con cualquier comunicación y $size$ es el tamaño que ocupan los individuos enviados a un procesador (n/S) suponiendo que cada individuo utiliza l bytes para su representación. De esta forma y procediendo igual que en la ecuación 5.12 obtenemos el número de procesadores cuando T_c no es constante

(ecuación 5.12):

$$S^* = \sqrt{\frac{(A+\alpha)\cdot n}{C}} \tag{5.16}$$

Una consideración importante a la hora de implementar un algoritmo genético paralelo global es que si se realizan muchas comunicaciones podemos perder cualquier reducción del rendimiento alcanzada mediante la paralelización. El tiempo que un AG simple tarda en realizar una generación es $T_s = \alpha \cdot n$ y para asegurar que la implementación paralela tiene un mejor rendimiento que la secuencial, se debe cumplir la ecuación 5.17:

$$\frac{T_s}{T_{tot}} = \frac{n\cdot\alpha}{\frac{n\cdot\alpha}{S} + (S+1)\cdot T_c} \tag{5.17}$$

Esta ecuación representa el speedup del AGP global. Si despejamos α obtenemos la ecuación 5.18 que es una condición necesaria para obtener una mejora en el rendimiento a la hora de realizar la implementación paralela.

$$\alpha > \frac{S+1}{S-1}\cdot\frac{S}{n}\cdot T_c \approx \frac{S}{n}\cdot T_c \tag{5.18}$$

Está condición tiene la ventaja de que se puede comprobar sin necesidad de implementar el AGP. Lo único que necesitamos es evaluar α y T_c. Como es muy probable que el tiempo de comunicación dependa linealmente del tamaño de la información que se va a transmitir podemos sustituir la ecuación 5.13 en la 5.18 y obtener la condición cuando el tiempo de comunicaciones no es constante 5.19:

$$\alpha > A\cdot l + \frac{S}{n}\cdot C \tag{5.19}$$

Esta condición indica que para problemas simples con tiempos de ejecución cortos, los AGP globales no son una buena opción para mejorar el rendimiento, pero para problemas con tiempos de ejecución elevados consiguen una mejora sustancial.

5.6.3 Algoritmos Geneticos Paralelos de Grano Grueso

Las características fundamentales de un AGP de grano grueso son la utilización de varias subpoblaciones relativamente grandes y la migración. Se conoce como migración al intercambio de individuos entre distintas subpoblaciones [17, 36]. Este

tipo de paralelización es el más utilizado. Las principales razones de su popularidad son:

- La forma de implementar un AGP de grano grueso a partir de una versión secuencial es muy sencilla. Únicamente hay que tomar un conjunto de AG secuenciales ponerlos en distintos procesadores y cada cierto número de generaciones intercambiar individuos. La mayoría del código de la versión secuencial queda exactamente igual después de paralelizar.

- La presencia de los computadores paralelos de grano grueso es habitual en la mayoría de los centros de investigación y allí donde no están disponibles se pueden crear mediante software como MPI (Message Passing Interface) ó PVM (Parallel Virtual Machine).

Los AGP de grano grueso también aparecen en la literatura como AGs distribuidos, debido a que se suelen implementar sobre máquinas de memoria distribuida. En ocasiones se les llama AGP de "isla"(island model) Ŝaenz01 ya que existe un modelo de poblaciones en el que las subpoblaciones están relativamente aisladas. Puede parecer, que dado que utilizamos subpoblaciones de menor tamaño que en un AG secuencial (o en serie), un AGP de grano grueso debería converger en un número menor de generaciones. Aunque es cierto que con una población menor un AG converge más rápido, también es cierto que la calidad de la solución no tiene porque ser la misma si el problema que se está tratando es complejo. En la figura 5.6 se puede ver un diagrama de un AGP de grano grueso en forma de grafo. En él se representan las poblaciones por nodos y las aristas representan las líneas de intercambio de individuos al realizar la migración.

Los AGP de grano grueso simulan el aislamiento geográfico de las diferentes civilizaciones y el intercambio esporádico de características que se realiza con la emigración. En la figura 5.7 podemos ver un esquema en pseudocódigo de un AGP de grano grueso, en el que la frecuencia es el número de generaciones que pasa entre dos intercambios de individuos.

Hay tres parámetros importantes que influyen en la eficiencia de un AGP:

- La topología que define la comunicación entre subpoblaciones

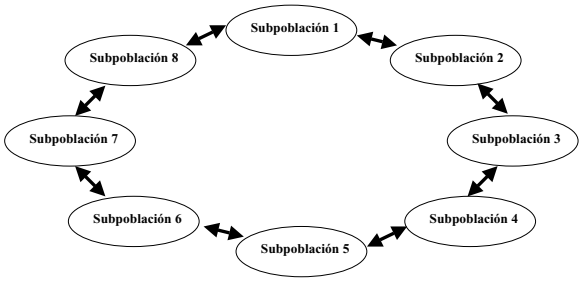

Figura 5.6: *Un esquema general de un AGP de grano grueso*

```
 Inicializar P subpoblaciones con N individuos cada una
 Numero de generacion = 1
 while ( no se cumpla la condicion de fin) do
   for cada subpoblacion) do in parallel
     evaluar y seleccionar individuos por su funcion de coste
if (Numero de generacion mod frecuencia) = 0 then
  enviar K<N mejores individuos a poblacion vecina
  recibir K mejores individuos de poblacion vecina
  reemplazar K individuos de la poblacion
end if
     producir nuevos individuos
     aplicar operador de mutacion
   end parallel do
  Numero de generacion ++
end while
```

Figura 5.7: *Pseudocódigo de un AGP de grano grueso*

- La proporción de intercambio: número de individuos a intercambiar
- Los intervalos de migración o frecuencia: periodicidad con que se intercambian los individuos.

Veamos un poco más extensamente cómo influyen en el funcionamiento de los AGPs de grano grueso y las distintas alternativas que se presentan.

5.6.3.1 Topologías de comunicación

La topología es un factor fundamental en el rendimiento de un AGP ya que determina la velocidad con que una buena solución se propaga de una subpoblación al

resto. Si la topología tiene muchas conexiones entre las subpoblaciones las buenas soluciones se transmiten rápidamente de una población a otra. Es evidente que las malas soluciones también lo harían, pero precisamente el proceso de selección gobernado por la función de coste controla este esparcimiento y lo limita a la pura probabilidad de selección que puede tener una mala solución. Por el contrario si las poblaciones tienen poca comunicación entre ellas las soluciones se extienden más lentamente, permitiendo la aparición de varias soluciones y una evolución más aisalda de cada grupo. Estas distintas soluciones se pueden utilizar posteriormente para generar individuos que superen a los obtenidos con una convergencia mas homogénea.

Otro factor en el que interviene la topología es el coste de comunicaciones. Aunque normalmente no se realizan tantos intercambios cómo para ralentizar el AGP, es necesario buscar un compromiso entre la topología elegida y el costo de comunicaciones, para no perder las ventajas obtenidas sobre el rendimiento con la paralelización. La norma general es utilizar una topología estática que se mantiene constante a lo largo de toda la ejecución del algoritmo. Muchas de estas topologías estáticas aprovechan la propia topología de la red de comunicaciones entre procesadores. Una segunda opción es la implementación de una topología dinámica. En ellas el intercambio entre subpoblaciones se realiza entre procesadores distintos cada vez en función de un criterio que suponga una mejora para el algoritmo. Uno de los factores que más se tienen en cuenta para este tipo de AGP es la diversidad de la población, es decir se trata de favorecer el intercambio con aquellas subpoblaciones con un mayor número de individuos iguales. Existen trabajos recientes en los que se han realizado implementaciones que no atienden a este patrón y que han dado buenos resultados [59]

En este trabajo se han utilizado tres de las topologías más comunes en la implementación del modelo de islas: la topología en anillo, la topología maestro esclavo y la de comunicación todos con todos. En la primera, las poblaciones están distribuidas en un anillo y solo hay intercambio entre vecinos. En la segunda todos los procesos esclavos intercambian sus mejores individuos con el maestro. En la última todos los procesadores intercambian información con cada uno de los otros. En las

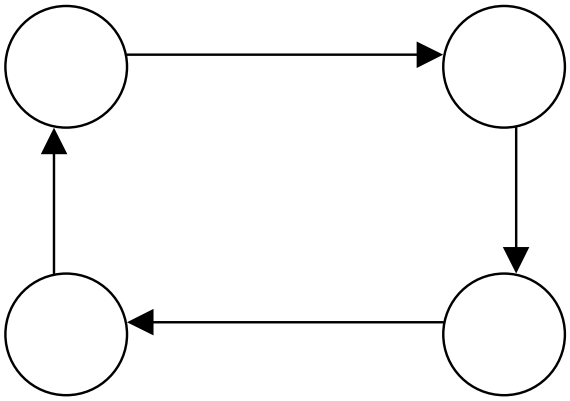

Figura 5.8: *Modelo en anillo del AGP, las líneas indican las comunicaciones entre subpoblaciones o lo que es lo mismo entre procesadores*

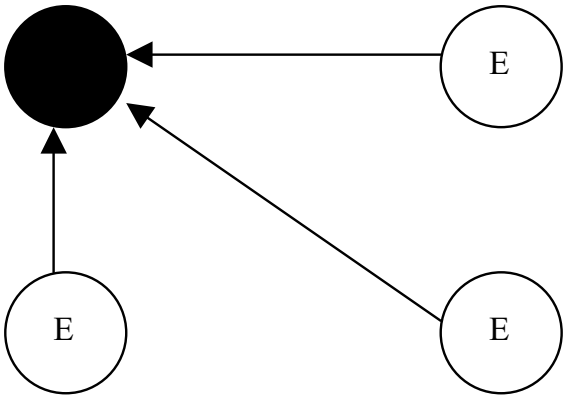

Figura 5.9: *Modelo maestro-esclavo del AGP, las líneas indican las comunicaciones entre subpoblaciones*

figuras 5.8, 5.9 y 5.10 se puede ver un esquema de estas tres topologías.

5.6.3.2 Proporción y frecuencia de intercambio

Tanto la frecuencia cómo la proporción de intercambio (cuántos individuos y cada cuánto tiempo se intercambian)son muy importantes para la convergencia del algoritmo y para la calidad de las poblaciones. Aunque en principio se puede suponer que cuanto mayor sea el intercambio mejor se propagan las buenas soluciones, esto no es cierto totalmente, ya que puede suceder que el excesivo intercambio de individuos entre las poblaciones convierta el AGP en una búsqueda prácticamente aleatoria al no permitir que el AG se desarrolle con normalidad.

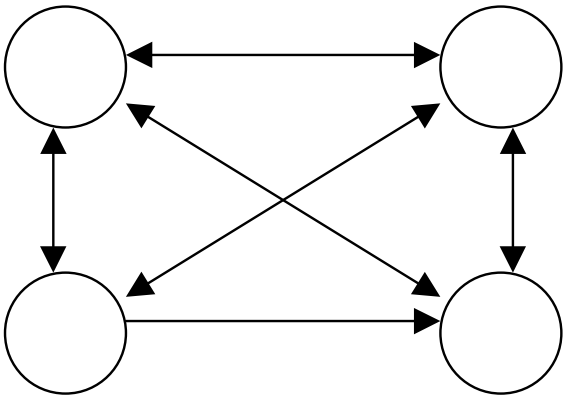

Figura 5.10: *Modelo de comunicación todos con todos, las líneas indican las comunicaciones entre subpoblaciones*

Algunas implementaciones realizan el proceso de migración o intercambio únicamente cuando las poblaciones han convergido totalmente [41, 124]. Con ello se trata de introducir una diversidad en las poblaciones y aliviar así posibles problemas de convergencia prematura. Otras soluciones utilizan distintas políticas de intercambio y lo realizan después de un determinado número de generaciones o bien con una periodicidad fijada de antemano y que se mantiene constante a lo largo de toda la ejecución del programa. Si la migración se produce muy pronto, puede suceder que las propiedades de los individuos intercambiados sean débiles e influyan muy levemente en el proceso de búsqueda.

Por todo lo citado anteriormente es conveniente realizar un estudio de distintas políticas de migración, teniendo en cuenta a la vez la topología de las poblaciones. Existen trabajos en los que se han estudiado estos problemas aunque todavía son un campo emergente y es necesario un estudio particular para cada problema, cada sistema y cada implementación [34, 59].

5.6.4 Algoritmos Genéticos Paralelos de Grano Fino

Los algoritmos genéticos de grano fino, se conocen también como AGP *grid o en parrilla* debido a la disposición de las poblaciones sobre los procesadores. Los indi-

viduos se disponen en una parrilla de dos dimensiones con un individuo en cada una de las posiciones de la rejilla [164, 165]. La evaluación se realiza simultáneamente para todos los individuos y la selección, reproducción y cruce se realizan de forma local con un reducido número de vecinos. Con el tiempo se van formando grupos de individuos que son homogéneos genéticamente como resultado de la lenta difusión de individuos. A este fenómeno se le llama aislamiento por distancia y es debido a que la probabilidad de interacción entre dos individuos disminuye con la distancia.

Este tipo de implementación simula las relaciones personales entre individuos de una misma localidad. Es decir, normalmente dos individuos que vivan cerca tienen más probabilidad de relacionarse que dos que vivan más separados. Los vecinos que se consideran son generalmente los cuatro u ocho más próximos. La forma de seleccionar al individuo de la vecindad con el que se interactúa se puede hacer de varias formas, la más utilizada es por torneo, es decir compiten entre ellos mediante el valor de su función de coste. De igual forma, la sustitución de los antiguos individuos por los nuevos a partir de los descendientes se puede hacer eligiendo el de mejor función de coste o mediante un proceso aleatorio. Todos estos procesos se pueden llevar a cabo de una manera dinámica o bien se pueden realizar con individuos que viajen u otras variantes que se desee implementar. Los modelos de grano fino se adaptan mejor a las maquinas del tipo SIMD, ya que las operaciones necesarias para las comunicaciones locales son muy eficaces implementadas sobre hardware. Cuando se combinan los AGP de grano fino con los de grano grueso se produce lo que se conoce como algoritmo genético híbrido. Alguno de estos algoritmos híbridos añaden un nuevo grado de complejidad al entorno de los AGP, pero otros utilizan la misma complejidad que uno de sus componentes.

5.6.5 Implementación

Para la implementación paralela de los AGs para SMFPGAs de topología libre se han utilizado los algoritmos genéticos de grano grueso (ver capítulo 3. Se han utilizado 3 topologías distintas: Anillo, Maestro-Esclavo y Todos-con-Todos.

Anillo

Como se explicó en el capítulo 3 los AGs paralelos dividen la población entre los distintos procesadores. En la topología en anillo, las poblaciones están distribuidas en un anillo y solo hay intercambio entre vecinos. El número de individuos que se seleccionan en cada población para ser intercambiados (proporción de intercambio) se puede variar en tiempo de ejecución, aunque en las medidas que se presentan solo se seleccionó el mejor individuo. Cada población envía su mejor individuo y los dos que recibe (de sus dos vecinos) sustituyen a sus dos peores.

MPI proporciona un conjunto de funciones para enviar y recibir mensajes y controlar el intercambio de información. La estructura en forma de pseudocódigo del AGP en anillo (AGPA) es:

/* Inicializa MPI */
$MPI\_Init(\&argc, \&argv)$;
/* Obtiene rango del proceso y no. de procesos */
$MPI\_Comm\_rank(MPI\_COMM\_WORLD, \&miRango)$;
$MPI\_Comm\_size(MPI\_COMM\_WORLD, \&numProc)$;
$N_{CLB_{tot}} \Leftarrow num\_bloques$
Leer información de los dispositivos
$g \Leftarrow$ número de generaciones
$poblac \Leftarrow$ población inicial
$ind \Leftarrow$ número de individuos
for $i = 1$ to g **do**
  for $k = 1$ to n **do**
    Obtiene_ocupación($poblac$, N_{CLB_i})
  end for
  for $j = 1$ to ind **do**
    Calcula_coste($poblac$, FC_j)
  end for
  Obtiene_Mejor($best$)
  Copia_Mejor(poblacion,indice1,cod_mejor);
  $EnviaMejor(cod\_mejor, miRango, numProc)$;

```
    Selecciona(poblac, poblac_sel)
    Cruza(poblac_sel, poblac_cru)
    Muta(poblac_cru, poblac_mut)
    poblac ⇐ poblac_mut
    Calcula_Peores (c3, indice_peores);
    RecibeMejores(miRango, numProc, mejores);
    Sustituye_peores(poblacion, indice_peores[l], mejores[l]);
  end for
  for i = 1 to n do
    Obtiene_ocupación(poblac, N_CLB_i)
  end for
  for j = 1 to ind do
    Calcula_coste(poblac, FC_j)
  end for
  if miRango == 0 then
    Mostrar_Resultados
  end if
  /* Finaliza MPI */
  MPI_Finalize();
```

El programa comienza con la inicialización de MPI, mediante la función $MPI\_Init()$ y la asignación de un número de proceso y cantidad de procesos que se van a ejecutar. Para esto último se utilizan otras dos funciones de MPI necesarias para que se produzca la comunicación; $MPI\_Comm\_rank()$ y $MPI\_Comm\_size()$. MPI proporciona una función para enviar mensajes ($MPI\_Send()$) y otra para recibirlos ($MPI\_Recv()$). Estas funciones se utilizan en $EnviaMejor()$ y en $RecibeMejores()$ respectivamente para realizar y controlar el intercambio de individuos entre los vecinos del anillo. Una vez seleccionados los peores individuos y recibidos los mejores, se realiza la sustitución y se continúa con la próxima generación. Cuando finaliza el algoritmo genético se imprimen los resultados y se finaliza MPI mediante la función $MPI\_Finalize()$, que es necesario incluir para realizar correctamente la compilación.

Maestro-Esclavo

En la topología Maestro-Esclavo o master-slave todos los procesos esclavos intercambian sus mejores individuos con el Maestro, pero no existe comunicación entre las poblaciones de los esclavos. Al igual que en el AGP en anillo sólo se selecciona el mejor individuo. En la versión master-slave (para n procesadores), el nodo master envía su mejor individuo a todos los esclavos, y sustituye sus $n-1$ peores con los individuos que recibe de estos. La migración también se produce en cada generación con lo que el algoritmo es similar a la versión secuencial pero con una población mayor. El pseudocódigo del AGP maestro-esclavo (AGPMS) es el siguiente:

/* Inicializa MPI */
$MPI\_Init(\&argc, \&argv)$;
/* Obtiene rango del proceso y no. de procesos */
$MPI\_Comm\_rank(MPI\_COMM\_WORLD, \&miRango)$;
$MPI\_Comm\_size(MPI\_COMM\_WORLD, \&numProc)$;
$N_{CLB_{tot}} \Leftarrow num\_bloques$
Leer información de los dispositivos
$g \Leftarrow$ número de generaciones
$poblac \Leftarrow$ población inicial
$ind \Leftarrow$ número de individuos
for $i = 1$ to g **do**
  for $k = 1$ to n **do**
    Obtiene_ocupación($poblac$, N_{CLB_i})
  end for
  for $j = 1$ to ind **do**
    Calcula_coste($poblac$, FC_j)
  end for
  Obtiene_Mejor($best$)
  Copia_Mejor(poblacion,indice1,cod_mejor);
  $IntercambiaMejores(cod\_mejor, mejores)$;
  Selecciona($poblac$, $poblac_{sel}$)
  Cruza($poblac_{sel}$, $poblac_{cru}$)
  Muta($poblac_{cru}$, $poblac_{mut}$)

```
    poblac ⇐ poblac_mut
    Calcula_Peores (c3, indice_peores);
    if miRango==0/* Si es el proceso master */ then
      for l = 1 to l do
        Sustituye_peores(poblacion, indice_peores[l], mejores[l]);
      end for
    else
      /* Si es un proceso slave */
      Sustituye_peor(poblacion, indice_peores[0], mejores[0]);
    end if
  end for
  for i = 1 to n do
    Obtiene_ocupación(poblac, N_CLB_i)
  end for
  for j = 1 to ind do
    Calcula_coste(poblac, FC_j)
  end for
  if miRango == 0/* Si es el proceso master */ then
    Mostrar_Resultados
  end if
  /* Finaliza MPI */
  MPI_Finalize();
```

La estructura básica es la misma que el anillo y se utilizan las mismas funciones MPI que en el anterior. Sin embargo, ahora el proceso de recepción y sustitución no es igual para todos. El proceso Maestro mantiene una lista con $n-1$ peores (para n procesadores) y los sustituye por los que recibe del resto, mientras que los Esclavos sólo sustituyen su peor individuo.

Todos con Todos

La última implementación que se ha realizado es un AGP en el que todos los procesadores intercambian su mejor individuo con el resto y reciben por lo tanto, $n-1$ individuos que se sustituyen por sus peores. A éste tipo de topología se le conoce

con el nombre de Todos con Todos y nosostros le denominamos AGPT.

/* Inicializa MPI */
$MPI\_Init(\&argc, \&argv)$;
/* Obtiene rango del proceso y no. de procesos */
$MPI\_Comm\_rank(MPI\_COMM\_WORLD, \&miRango)$;
$MPI\_Comm\_size(MPI\_COMM\_WORLD, \&numProc)$;
$N_{CLB_{tot}} \Leftarrow num\_bloques$
Leer información de los dispositivos
$g \Leftarrow$ número de generaciones
$poblac \Leftarrow$ población inicial
$ind \Leftarrow$ número de individuos
for $i = 1$ to g **do**
  for $k = 1$ to n **do**
    Obtiene_ocupación($poblac$, N_{CLB_i})
  end for
  for $j = 1$ to ind **do**
    Calcula_coste($poblac$, FC_j)
  end for
  Obtiene_Mejor($best$)
  Copia_Mejor(poblacion,indice1,cod_mejor);
  $IntercambiaMejores(cod\_mejor, mejores)$;
  Selecciona($poblac$, $poblac_{sel}$)
  Cruza($poblac_{sel}$, $poblac_{cru}$)
  Muta($poblac_{cru}$, $poblac_{mut}$)
  $poblac \Leftarrow poblac_{mut}$
  Calcula_Peores (c3, indice_peores);
  for $l = 1$ to l **do**
    for $l = miRango + 1$ to $numProc$ **do**
      Sustituye_peores(poblacion, indice_peores[l-1], mejores[l]);
    end for
  end for
end for

for $i = 1$ to n **do**
  Obtiene_ocupación($poblac$, N_{CLB_i})
end for
for $j = 1$ to ind **do**
  Calcula_coste($poblac$, FC_j)
end for
if miRango == 0/* Si es el proceso master */ **then**
  Mostrar_Resultados
end if
/* Finaliza MPI */
$MPI\_Finalize()$;

En este caso también se realiza el intercambio en cada generación. Las funciones MPI usadas son las mismas que las otras dos topologías y ahora todos los procesadores realizan el mismo intercambio, pero en este caso todos mantienen una lista con los $n-1$ peores individuos, para sustituirlos con los $n-1$ mejores que recibe de los otros procesadores.

5.7 Resultados Experimentales

En este apartado presentamos los resultados experimentales obtenidos al aplicar los AGs descritos a lo largo del capítulo. En primer lugar se muestran las pruebas del algoritmo genético simple para sistemas con 2 tipos de FPGAs, en los que se prueba la convergencia del algoritmo. A continuación se describen los resultados obtenidos para 3 tipos de FPGAs y se comparan los distintos AGs entre sí y con otras propuestas. Todos los algoritmos se han implementado en C [84, 90, 89].

5.7.1 Resultados Experimentales para 2 tipos de FPGAs

Las pruebas se han realizado con cinco circuitos de prueba básicos y el objetivo es minimizar el coste del diseño final sin imponer ningun tipo de restricción salvo la capacidad lógica de las FPGAs. El coste normalizado es 1.00 $ para la $XC2064$ y 1.36 para la $XC2018$. Los resultados obtenidos no son comparables con otras

herramientas pero sí demuestran el correcto funcionamiento del algoritmo. En la tabla 5.5 están expuestos estos resultados e incluyen las características de estos circuitos. Los circuitos son simples y van desde los 12 a los 164 CLBs. Cómo se puede comprobar en la tabla el algoritmo alcanza el óptimo en todos los casos.

Circuito	CLBs	Coste óptimo	Coste AG
prueba3	75	1.36	1.36
prueba4	100	1.36	1.36
prueba5	164	2.36	2.36

Tabla 5.5: *Resultados obtenidos para 2 tipos de FPGAs*

5.7.2 Resultados Experimentales para 3 tipos de FPGAs

Los circuitos utilizados para realizar los experimentos han sido los Partitioning Benchmark 93 del MCNC [29]. Esto nos permite comparar resultados con otras herramientas. Las características de estos circuitos después de la utilización de la herramienta XACT están recogidos en la tabla 5.6. En ella se expresan el número de CLBs y de IOBs necesarios para su implementación sobre la serie 3000 de Xilinx.

Circuito	Num CLBs	IOBs
C3540	283	72
C5315	377	301
C7552	833	64
C6288	489	313
S5378	381	86
S9234	454	43
S13207	915	154
S15850	842	101
S38584	2904	292

Tabla 5.6: *Características de los circuitos de prueba*

Los resultados se miden en función del coste en dolares del sistema final, es decir las suma de costes de todas las FPGAs necesarias. El coste de los dispositivos está normalizado al de la XC3020 que se toma como un dólar y se recogen en la tabla 5.7.

Dispositivo	Vcc	N§ CLBs	Coste Normalizado
XC 3020	64	64	1.00
XC 3030	100	80	1.36
XC 3042	144	96	1.84
XC 3064	224	110	3.15
XC 3090	320	144	4.83

Tabla 5.7: *Características principales de la serie 3000 de Xilinx*

Se han realizado cuatro implementaciones del AG, una secuencial y tres paralelas que denominaremos:

- SAG: Es una implementación secuencial que utiliza poblaciones de 480 individuos.

- AGPA: AG paralelo con poblaciones de 60 individuos y una topología de comunicación en anillo.

- AGPMS: AG paralelo con poblaciones de 60 individuos y una topología de comunicación maestro-esclavo.

- AGPT: AG paralelo con poblaciones de 60 individuos en la que todas las poblaciones se comunican entre sí.

SAG se ha utilizado para comparar con los resultados de PROP y de Lp-Solve. PROP se ha descrito en el apartado 2.3.1 de esta memoria y Lp-Solve es una herramienta de programación lineal de dominio público [15], que no introduce ningún tipo de restricción a la hora de hacer la partición. Sin embargo, si permite hacernos una idea de la calidad de las soluciones obtenidas. La tabla 5.8 muestra los resultados obtenidos con SAG y su comparación con Lp-Solve. El significado de cada columna es el siguiente:

- Circuito: Nombre identificativo del benchmark

- Dist1: Número de FPGAs de cada tipo.
  Ejemplo: 2,1,1 quiere decir que son necesarias dos $XC3020$, una $XC3030$ y una $XC3042$.

- Dist2: Número de CLBs en cada tipo de FPGA.
  Ejemplo: 102,80,101 indica que 102 CLBs van sobre las $XC3020$, 80 sobre la $XC3030$ y 101 se implementan en la$XC3042$.

- Ocup.: Porcentaje de ocupación de cada tipo de FPGA.
  Ejemplo: 0.79,0.80,0.70, indica que la ocupación de las $XC3020$ es el 79%, de la $XC3030$ el 80% y un 70% el de la $XC3042$.

- CostAG: Coste final obtenido con el SAG.

- CostLP: Coste final obtenido por Lp-Solve.

- CPU: Tiempo de ejecución utilizado por SAG para obtener la solución (en segundos).

Circuito	Dist1	Dist2	Ocup.	CostAG	CostLP	CPU
C3540	2,1,1	102,80,101	0.79,0.80,0.70	5.2	4.51	114.1
C5315	2,2,1	103,160,114	0.80,0.80,0.79	6.56	5.92	129
C7552	3,2,2	129,130,230	0.67,0.65,0.79	9.40	7.36	1688
C6288	5,3,3	257,231,345	0.80,0.77,0.79	14.6	12.77	3201
S5378	3,2,1	129,137,115	0.67,0.68,0.79	7.56	5.52	141
S9234	3,2,2	131,136,187	0.68,0.68,0.64	9.40	6.83	1572
S13207	5,4,3	257,317,341	0.80,0.79,0.79	15.96	13.40	3506
S15850	5,4,2	257,350,235	0.80,0.87,0.81	14.12	12.35	3273
S38584	16,10,8	986,905,1013	0.96,0.90,0.87	44.32	45.50	1061

Tabla 5.8: *Comparación entre el coste normalizado obtenido por la herramienta Lp-Solve y el SAG*

En la tabla 5.8 y en la figura 5.11 se puede ver que, a pesar de que la herramienta Lp-Solve no impone ninguna restricción en cuanto a la ocupación, el coste obtenido por el algoritmo SAG es similar al obtenido por ésta. Para poder sacar conclusiones más reales, se ha hecho una comparación con la herramienta PROP. Los resultados se encuentran recogidos en dos tablas. Por un lado la tabla 5.9 compara el coste final del circuito obtenido por la herramienta PROP y el obtenido por el SAG. Estos resultados se pueden ver graficamente sobre la figura 5.12. Por otro lado, la tabla 5.10 muestra la ocupación media de los dispositivos obtenida por PROP y la obtenida por SAG (figura 5.13).

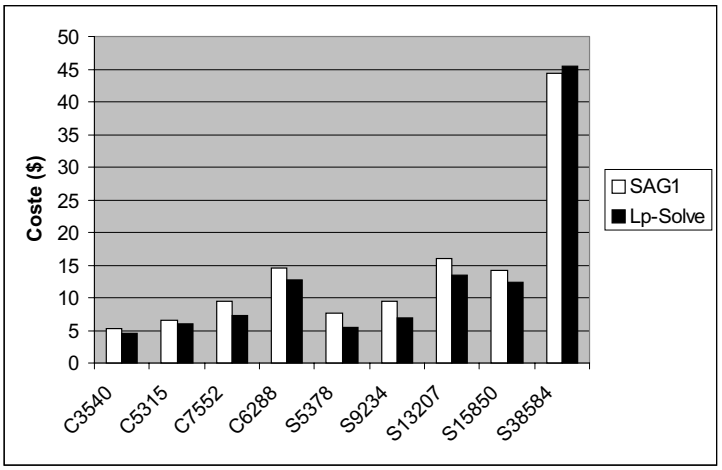

Figura 5.11: *Comparación del coste final obtenido por SAG y Lp-Solve*

Circuito	Coste PROP	Coste SAG
C3540	5.20	4.99
C5315	6.56	7.76
C7552	14.6	13.66
C6288	9.40	7.88
S5378	7.56	6.19
S9234	9.40	7.98
S13207	15.96	16.81
S15850	14.12	14.97

Tabla 5.9: *Comparación del coste obtenido por SAG y PROP*

Como se puede comprobar de la inspección de las tablas 5.8 y 5.10, SAG mejora en todos los casos o bien los resultados de coste o bien los resultados de ocupación obtenidos por PROP. Siguiendo el criterio de que una ocupación de las FPGAs inferior al 80% aseguraría su rutabilidad interna, SAG obtiene circuitos rutables en un 88% de los casos (figura 5.14). En un 45% de los casos el coste obtenido por SAG es inferior al obtenido por PROP.

Los resultados obtenidos por SAG son satisfactorios tanto si los comparamos con la herramienta Lp-Solve como si los comparamos con la herramienta PROP. Realmente, la comparación más importante es ésta última, pues como ya se ha dicho se pretende asegurar la rutabilidad del circuito. Aunque no se mejora el coste en todos los casos la rutabilidad del sistema queda asegurada para casi todos los circuitos.

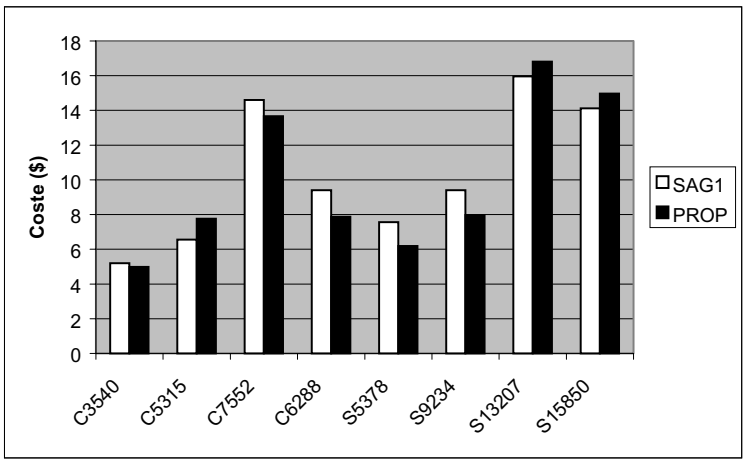

Figura 5.12: *Comparación del coste final obtenido por SAG y PROP*

Circuito	SAG	PROP
C3540	0.76	0.77
C5315	0.79	0.52
C7552	0.79	0.83
C6288	0.70	0.85
S5378	0.71	0.94
S9234	0.66	0.85
S13207	0.79	0.81
S15850	0.83	0.80

Tabla 5.10: *Comparación entre el porcentaje de ocupación de las FPGAs (OR) obtenido por la herramienta PROP y el SAG*

El principal problema del SAG es su elevado tiempo de ejecución. Por ello se han realizado las implementación paralelas AGPA, AGPM y AGPT, como se ha explicado. Los AGP se han ejecutado sobre 8 procesadores de un CRAYT3E.

Los resultados obtenidos para los distintos cicuitos se han recogido en la tabla 5.11. En ella se comparan los tiempos de ejecución de los algoritmos genéticos SAG, AGPA y AGPMS. En la segunda columna aparece el tiempo necesitado por el AG secuencial para obtener una solución comparable a los de las tablas 5.9 y 5.10, y entre paréntesis aparece el número de generaciones necesarias. En la otras 2 columnas aparece el tiempo utilizado por cada uno de los AGP para obtener la misma solución que el SAG. El AGP en anillo necesita en el peor de los casos 225 generaciones y el

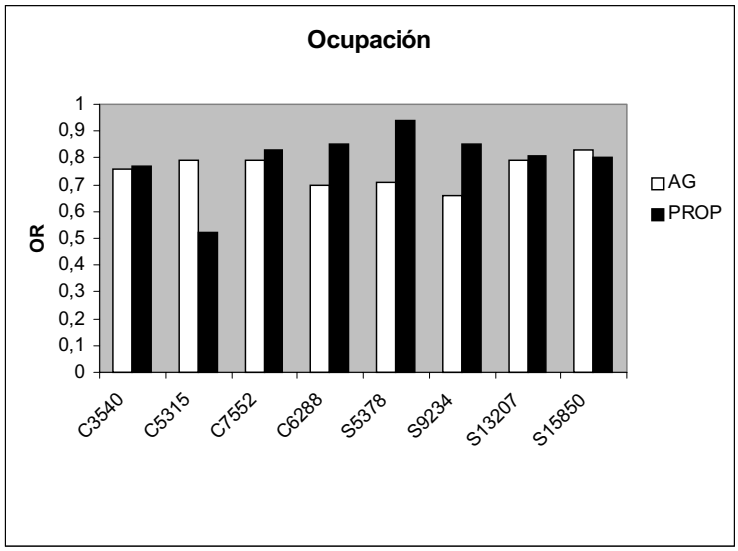

Figura 5.13: *Comparación del Ocupación obtenida por SAG y PROP*

AGPMS 300.

Cabe destacar, que el objetivo de la reducción de los tiempos de ejecución se ha conseguido. Esto se debe a que se realiza un reemplazamiento de la población conservando el mejor individuo, y además tenemos varias poblaciones trabajando en distintas zonas del espacio de búsqueda.

Pero se ha obtenido un resultado más interesante si atendemos a la naturaleza de la topología. El AGPA obtiene mejores resultados que el AGPMS, debido fundamentalmente a problemas de convergencia prematura en el nodo maestro. Esto se debe a que este nodo sustituye en cada generación a una proporción importante de individuos, degradándose así las características del algoritmo genético. Los resultados se agravan aún más en el caso del algoritmo AGPT que no obtiene soluciones comparables a los anteriores y por eso no es posible comparar tiempos de ejecución. Evidentemente, la causa es la misma que la del AGPMS pero sucede lo mismo con todos los nodos.

Hay que mencionar que el concepto de paralelización en los algoritmos genéticos es un poco diferente a la paralelización tradicional. A parte de buscar una reducción

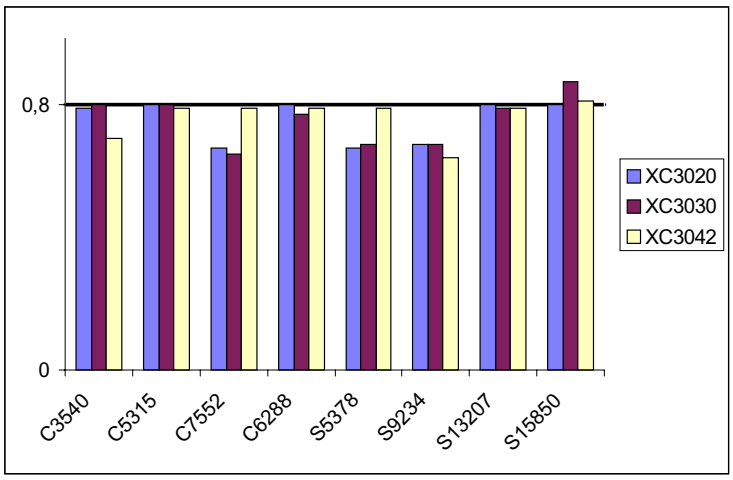

Figura 5.14: *Rutabilidad de los resultados obtenidos por SAG*

Circuito	SAG	AGPA	AGPMS
C3540	19.78 (500)	3.88	3.97
C5315	- (> 2000)	10.30	10.55
C7552	85.31 (725)	11.46	12.27
C6288	61.78 (900)	6.68	8.52
S5378	38.75(725)	6.50	15.97
S9234	57.34(900)	18.57	18.99
S13207	116.627(900)	15.73	28.92
S15850	- (> 2000)	25.98	26.54

Tabla 5.11: *Comparación entre los AGs secuencial y paralelos. Tiempo en segundos. Para el secuencial el número de generaciones aparece entre parentesis. Todos los algoritmos utilizan poblaciones de 60 individuos*

de tiempos de ejecución se desea explorar las diferentes evoluciones de la población en función de la iteración con poblaciones de otros procesadores. Por este motivo las medidas de número de generaciones que se han presentado son bastantes indicativas para un análisis de la convergencia y por ello las figuras se basan en este parámetro.

5.8 Conclusiones

En este capítulo se ha implementado varios algoritmos genéticos, tanto secuenciales cómo paralelos, para la resolución del problema de la partición de circuitos lógicos orientada a SMFPGAs de topología libre. El algoritmo secuencial se ha denominado SAG y los paralelos AGPA (topología de comunicación en anillo), AGPMS (maestro-

esclavo) y AGPT (todos con todos). Del estudio de los resultados podemos sacar las siguientes conclusiones:

- La distribución de bloques lógicos que nos da el SAG respeta las condiciones de rutabilidad del sistema en un 88% de los casos.
- El coste en dólares del circuito resultante se ha reducido también en un 45 % de los experimentos en relación a los resultados de [114].
- La función de coste se puede adaptar para que incluya otras restricciones del sistema, tales como las referentes a utilización del número de bloques de entrada salida o del número de pines.
- La versión secuencial SAG necesita en alguna de las ocasiones más de 2000 generaciones para obtener una solución aceptable, mientras que el AGPA sólo necesita 225 y el AGPMS 300 en el peor de los casos. Este resultado se debe tanto a la búsqueda simultánea realizada por las 8 subpoblaciones como al reemplazamiento sin solapamiento que utiliza el AGP.
- El AGPA obtiene mejores resultados que el AGPMS, debido fundamentalmente a problemas de convergencia prematura en el nodo maestro, ya que éste sustituye en cada generación una proporción importante de individuos, degradándose las características del algoritmo genético.
- El AGPT no funciona debido a un fenómeno de convergencia prematura en todos sus nodos.

En resumen, el SAG y sus versiones consiguen resolver el problema de la partición para topologías libres. Sin embargo para tratar una topología fija o sistemas con un mayor número de restricciones, debemos tratar de conservar la estructura del circuito y realizar una ubicación que nos permita evaluar el consumo de pines de entrada-salida cuando se trasladan las particiones a una tarjeta específica. Para ello se ha desarrollado un método de partición y ubicación, y se ha utilizado una codificación que permita conservar la estructura del circuito y evaluar el número de pines mediante la ubicación. Estos algoritmos y los resultados se explican en el siguiente capítulo.

Capítulo 6

Técnicas de partición para los SMFPGA de topología fija

A diferencia de los SMFPGAs de topología libre, en los que se desconoce la disposición final de las FPGAs, los de topología fija están formados por varios dispositivos FPGA conectados entre sí en una tarjeta ya diseñada. Además suelen incorporar otros recursos de implementación como memorias RAM u otros circuitos ASIC. En la actualidad los SMFPGAs se utilizan en un amplio espectro de aplicaciones (ver capítulo 4).

Hay una gran variedad de tarjetas y topologías Multi-FPGA disponibles en la actualidad para la implementación de circuitos, aunque la mayoría tienen una difusión únicamente académica. Las topologías mas utilizadas son la de malla y la de crossbar o grafo bipartito. En el capítulo 4 vimos que en una malla, las FPGAs están conectadas con sus vecinas más próximas y no existe en principio ninguna distinción entre ellas. Este tipo de tarjetas es fácil de rutar y presenta una buena disposición para expandir el sistema con otros similares. En la figura 6.1(a) se puede ver un esquema general de una topología tipo malla.

Por su parte las topologías crossbar tienen dos tipos de FPGAs claramente diferenciados en cuanto a su funcionalidad, FPGAs de lógica y FPGAs de rutado. Los primeros implementan la lógica del circuito, mientras que los de rutado implementan las líneas de conexión. Los circuitos de lógica están conectados únicamente

con los de rutado y viceversa. Sin embargo los circuitos de un tipo y de otro no tienen porque ser diferentes en lo relativo a su estructura interna, simplemente unos se utilizan para una función y los otros para otra. Este tipo de topologías puede adaptarse bien a ciertos problemas específicos, sin embargo no son fáciles de ampliar y suelen desaprovechar recursos del sistema tanto de lógica como de rutado. En la figura 6.1(b) se puede ver un esquema de este tipo de topologías.

La partición de circuitos orientada a SMFPGAs de topología fija consiste en dividir un determinado circuito en varias partes de tal forma que se pueda implementar sobre una tarjeta objetivo y conectar de una determinada manera impuesta por la topología. Cuando se utiliza una tarjeta se deben tener en cuenta un conjunto de restricciones relativas a la misma entre las que destacan el número de pines de entrada-salida (PE/S) disponibles y la capacidad lógica de los circuitos que componen la tarjeta. Los SMFPGAs ofrecen un número muy reducido de PE/S en comparación con la lógica que pueden implementar. En ciertas ocasiones, hay que comunicar partes del circuito que están en FPGAs no adyacentes por lo que el problema de la escasez de PE/S se agrava aún más. Además a la hora de hacer el rutado del circuito necesitamos dejar libres parte de los bloques lógicos para poder realizar esta tarea. Estos aspectos hacen de la partición una tarea ardua que no es fácil de llevar a buen puerto si no es con un correcto planteamiento del problema que tenga en cuenta éstas restricciones.

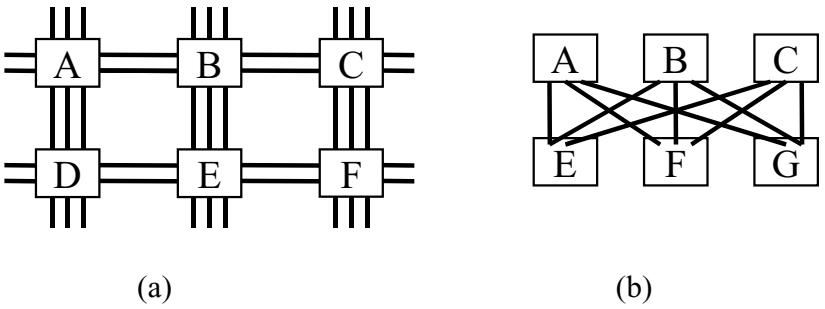

Figura 6.1: *Un esquema de las dos topologías más utilizadas en sistemas Multi-FPGA, malla (a) y crossbar (b)*

Para las topologías fijas no sirve la codificación que se ha utilizado en el capítulo

anterior. La razones fundamentales son:

- No conserva la estructura del circuito. Por ello el consumo de pines entre las FPGAs es dificil de controlar, y las soluciones serían prácticamente imposibles de llevar a la tarjeta.
- No se realiza un proceso de ubicación sobre una tarjeta, dado que no se conoce a priori sobre qué tarjeta se va a implementar.
- La función de coste no tiene en cuenta el número de pines de entrada-salida utilizados.
- No se tienen en cuenta las conexiones que se realizan a través de FPGAs no adyacentes.

En este capítulo presentamos una serie de algoritmos genéticos para partición de grafos para la resolución del problema en SMFPGAs que incorpora un proceso de ubicación para poder evaluar el consumo de PE/S. Para ello, se presenta una técnica de ubicación simultánea a la partición en la que las partes con un mayor número de conexiones entre sí, tienen preferencia para ser ubicadas en FPGAs adyacentes y disminuir así el consumo de PE/S. Previamente al proceso de partición, el algoritmo detecta aquellas partes del circuito que son independientes entre sí y las ubica en FPGAs aisladas, utilizando el resto para realizar la partición sobre las FPGAs que han quedado libres. Ésta es otra forma de reducir el consumo PE/S. Esta opción es útil cuando no se conoce si una descripción corresponde a un único circuito, ya que lo que nos interesa es realizar particiones de circuitos conexos. Dicho de otra forma, un circuito con varias partes inconexas son en realidad varios circuitos.

El método se basa en una representación del circuito en forma de grafo y utiliza los algoritmos genéticos (simple, compacto y compacto híbrido) como herramienta de optimización. El algoritmo se ha aplicado a los circuitos de prueba recogidos en Partitioning 93 benchmarks. Como formato de entrada hemos utilizado el XNF (Xilinx Netlist Format) un formato específico de Xilinx que permite una descripción clara y sencilla de los elementos del circuito. Además, hoy en día las FPGAs de Xilinx son las más utilizadas tanto en la industria como en el ámbito académico.

En la implementación del algoritmo genético simple presentamos también una técnica para evitar la convergencia prematura cuando la diversidad de la población va disminuyendo. Este método consiste en aplicar una mutación a un porcentaje de la población cuando el algoritmo se estanca durante varias generaciones en la búsqueda de soluciones. Los resultados demuestran su efectividad.

Este procedimiento es aplicable tanto a topologías malla como a topologías crossbar. Una topología malla es más complicada de tratar a la hora de de hacer la ubicacíon, por ello los algoritmos se han particularizado para éste tipo de tarjetas. Sin embargo, se explica tambien como se pueden tratar topologías de tipo crossbar sin más que utilizar un algoritmo de ubicación y de evaluación del consumo de pines más sencillo que el de topologías malla.

El resto del capítulo está organizado como sigue. En primer lugar se da una visión general de los conceptos fundamentales en la teoría de grafos, ya que son la base del algoritmo de partición. A continuación se expone el método de partición, incluyendo los objetivos de nuestra herramienta y el flujo de diseño del proceso. El apartado 6.3 presenta las funciones de coste desarrolladas. A continuación se explica la codificación e implementación de los algoritmos genéticos junto con los resultados experimentales y las conclusiones de cada uno. El capítulo incluye también una explicación de la implementación paralela de los AGs compacto y compacto híbrido.

6.1 Resumen de la teoría de grafos

La elección de la representación del circuito es normalmente una consecuencia del objetivo del algoritmo o de la aproximación hecha para la resolución del problema. Nosotros hemos escogido la representación en forma de grafo para describir el circuito. Esta elección viene motivada porque va a facilitar la codificación de los individuos (soluciones) con los que trabajan los AGs y por la facilidad que presenta el formato XNF para transformarse a una descripción de este tipo. Los circuitos se suelen representar mediante grafos [31][32].

La partición de grafos tiene multitud de aplicaciones en distintos campos del diseño VLSI, pero desafortunadamente es un problema de optimización de los conocidos como NP-duro, y debemos buscar algoritmos heurísticos que nos guíen en busca de soluciones aceptables a nuestro problema. Antes de describir la implementación, veremos algunas definiciones de la teoría de grafos, para que la explicación de la codificación del problema y de los algoritmos resulte más clara.

- *Definición*: Un *grafo* se puede definir de una manera informal como un conjunto de nodos conectados por líneas. Pero más formalmente, un grafo es un par $G(V;E)$, donde V es un conjunto de vértices o nodos y E es el conjunto de aristas que los conectan. Por ello una arista queda identificada por los dos nodos que une.

- *Definición*: Un vértice o nodo u es *adyacente* a otro nodo v, si (u,v) es una arista o en otras palabras si (u,v) pertenece a E.

- *Definición*: Un *camino* de un grafo es una secuencia

$$P = (v_0, e_1, v_1, ..., v_{k-1}, e_k, v_k)$$

  de vértices v_i y aristas e_j, tales que $e_i = (v_{i-1}, v_i)$, $1 \leq i \leq k$.

- *Definición*: Un *camino simple* es un camino en el cual todos los vértices que lo componen son distintos, excepto quizás el primero y el último.

- *Definición*: Un *tour* es un camino en el cual todas las aristas son distintas

- *Definición*: Un *lazo* o *bucle* es un camino simple en el que el comienzo y el final son el mismo vértice

- *Definición*: Dos vértices u y v están *conectados* si (u,v) es un camino simple.

- *Definición*: Un grafo es *conexo*, si todos sus pares de vértices están conectados.

- *Definición*: Un *subgrafo* de un grafo $G = (V,E)$ es un grafo $G' = (V',E')$ que cumple que $V' \subseteq V$, y $E' \subseteq E$.

- *Definición*: Una *componente conexa* de un grafo es un sub-grafo conexo de G.

```
Inicializar(árbol)
for i = 1 to num_vertices do
  Incluir (vertice_i,árbol)
end for
while Número de aristas del árbol ¡ (num_vertices-1) do
  Seleccionar arista del grafo
  Eliminar arista del grafo
  if arista no crea un ciclo then
    Incluir (arista,árbol)
  end if
end while
```

Figura 6.2: *Pseudocódigo del algoritmo de Kruskal*

- *Definición*: Una arista se dice *de corte* si al suprimirla del grafo obtenemos una componente conexa del grafo original.

- *Definición*: Un *árbol* es un grafo conexo que no contiene ciclos. En otras palabras un árbol es un grafo conexo en el que todas sus aristas son de corte. Los árboles tienen varias propiedades importantes para la resolución de nuestro problema y que nosotros utilizaremos para obtener las particiones:

  1. Propiedad: Un árbol con n vértices contiene $n-1$ aristas.
  2. Propiedad: Si a un árbol se le añade una arista se obtiene un grafo con un ciclo
  3. Propiedad: Si se elimina una arista de un árbol se obtiene un grafo con dos componentes conexas.

- *Definición*: un *árbol de expansión* de un grafo es un árbol que se ha obtenido mediante la selección de aristas del grafo.

Para obtener el árbol de expansión de un grafo existen varios métodos de entre los que hemos seleccionado el algoritmo de Kruskal [21]. En la figura 6.2 se puede ver una representación en pseudocódigo del algoritmo de Kruskal y en la figura 6.3 un ejemplo de un árbol de expansión obtenido a partir de un grafo.

El algoritmo de Kruskal funciona básicamente seleccionando aristas del grafo que no creen ciclos hasta que se hayan seleccionado las $n-1$ aristas necesarias para

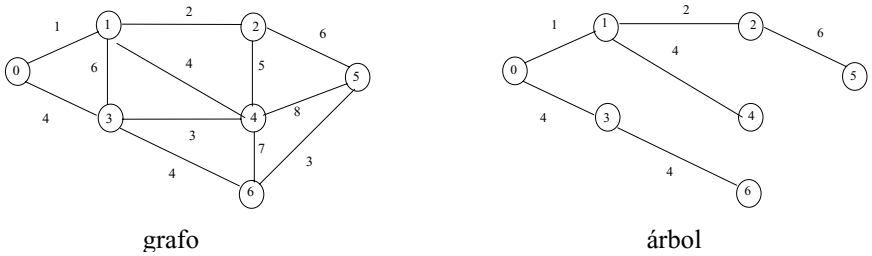

Figura 6.3: *Ejemplo de un árbol obtenido a partir de un grafo (árbol de expansión).*

completar el árbol. Éste se aplica para obtener el árbol de expansión mínimo cuando las aristas tienen asignado un determinado peso. Esto es, se trata de buscar el árbol de expansión cuya suma de pesos de las aristas sea mínima. Sin embargo en nuestro problema no tiene mucha utilidad, puesto que aunque halláramos el árbol mínimo a la hora de hacer la partición estaríamos dividiendo por esas aristas y no aseguraría ninguna mejora del algoritmo y sí aumentaría el tiempo de cálculo. Además limitaría al algoritmo puesto que las soluciones se buscarían sólo a partir de un árbol (el de expansión mínimo), y en nuestra aproximación este árbol se genera de manera aleatoria, proporcionando una mayor libertad en la búsqueda de soluciones.

6.2 Descripción del Método de Partición

En esta sección se describe la solución propuesta para resolver el problema. En primer lugar explicaremos cuáles son los objetivos de la herramienta de partición, a continuación se describe el flujo de diseño y finalmente se describe la implementación completa.

Comenzamos con la definición del problema de la partición. Alpert, en la recopilación de técnicas de partición anteriormente citada [5], define el problema de la siguiente forma: "Dado un conjunto de n módulos de un circuito $V = v_1, v_2, ..., v_n$ el propósito de la partición es asignar los módulos a un número especificado de grupos satisfaciendo un conjunto de propiedades impuestas". Aunque la definición es bastante general es muy clara y da una idea exacta de lo que se pretende al realizar

la partición del circuito.

6.2.1 Objetivos de la partición para sistemas SMFPGA de topología fija

En el caso específico de la partición para SMFPGAs, el objetivo principal es llevar a cabo la partición del circuito de tal forma que se obtengan un conjunto de FPGAs lógicas que sean implementables sobre la tarjeta Multi-FPGA. Llamamos FPGA lógica a una parte del circuito que se puede implementar sobre una FPGA del sistema. El proceso de partición debe respetar las restricciones impuestas por la topología de la tarjeta. Como se explicó en el primer punto de este capítulo hay dos aspectos fundamentales a la hora de realizar la partición: el número de PE/S y los bloques lógicos utilizados. Además las FPGAs no se deben ocupar totalmente ya que es necesario reservar parte de los bloques lógicos para realizar el rutado del circuito. Existen trabajos en los que se demuestra que si la utilización de los CLBs es mayor que el 80 % la FPGA no se puede rutar [115].

Por si esto fuera poco, una vez realizada la partición puede suceder que dos particiones tengan muchas conexiones entre sí y luego se ubiquen en FPGAs no adyacentes. Esto implica un consumo extra de PE/S, ya que dichas conexiones deben atravesar otras FPGAs para alcanzar su objetivo por lo que consumen PE/S de las FPGAs intermedias tanto a la entrada como a la salida de las mismas. Para respetar estas condiciones es necesario realizar un proceso de ubicación junto con la partición. Si no se aborda el problema de esta forma, no es posible cuantificar ni evaluar si se han respetado las condiciones.

En nuestro caso se ha realizado una implementación orientada a un SMFPGA de topología malla de 4 caminos. Una topología malla de 4 caminos no es más que una malla de FPGAs dispuestas en forma de matriz en el que cada una de las FPGAs está conectada con sus vecinas más próximas en las 4 direcciones principales. En la figura 6.4, se puede ver la estructura general de una topología malla de 4 caminos (4-way) para un sistema con 8 FPGAs. A la hora de realizar la partición debemos imponer las restricciones propias de utilizar esta topología. Para ello no sólo re-

alizamos la partición, sino que además se efectúa una ubicación de las particiones sobre la tarjeta.

El método de partición propuesto en este trabajo, basado en la eliminación de aristas, se aplica utilizando varios algoritmos genéticos para buscar la partición que cumpla las restricciones propias de un SMFPGA con una máxima eficacia y tiene las siguientes características:

- Es adaptable a otros tipos de tarjeta sin más que cambiar la parte de evaluación del número de PE/S utilizados.
- Es paralelizable: el método es intrínsicamente paralelo ya que utiliza un algoritmo genético.
- El algoritmo obtiene soluciones que conservan la estructura general del circuito y tiende a agrupar en la misma partición los elementos del circuito que están conectados entre sí. Esto se debe a que se trabaja con el árbol de expansión que se obtiene del grafo representativo del circuito. Este aspecto es muy importante ya que permite reducir el consumo de PE/S debidos al paso de las señales a través de las distintas FPGAs.
- El algoritmo realiza una ubicación para reducir el consumo de pines. Para alcanzar este objetivo se vale de una técnica utilizada en partición borrosa y que se explica junto con la función de coste.

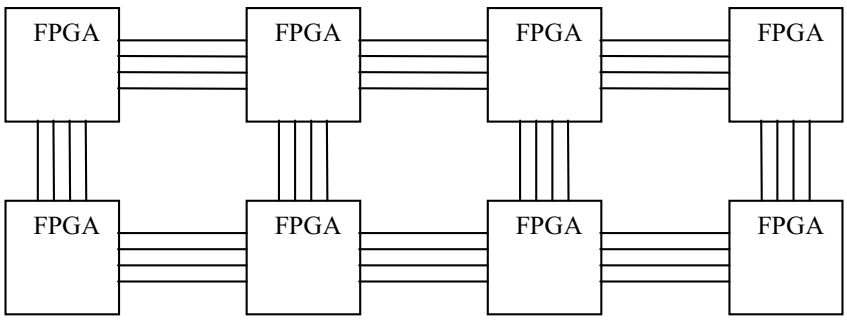

Figura 6.4: *Estructura general de una topología malla de 4 caminos para un sistema con 8 FPGAs*

6.2.2 Entrada al sistema

El algoritmo acepta como entradas una descripción del circuito en formato XNF (Xilinx Netlist Format). XNF es un lenguaje de descripción de circuitos específico de las FPGAs de Xilinx muy sencillo de comprender y que permite la inclusión de información adicional del circuito, tal como la relativa a posiciones de los bloques lógicos, cierta información de retardo, etc...

En la figura 6.5 se puede observar un ejemplo de la descripción de un bloque lógico como un esquema de CLB, en formato XNF y el correspondiente vértice del grafo asociado a ese CLB. Como se puede ver en el ejemplo del formato XNF tanto la descripción del bloque lógico como de los PE/S de entrada salida es muy clara y sucede lo mismo para cualquier tipo de puerta lógica que incluya el circuito. Cada componente que luego delimitará un nodo en el grafo, está separada del resto por una línea SYM (primera línea de la figura 6.5b) al principio de su descripción y por otra línea END al final de la misma. Por ello se puede obtener el grafo representativo del circuito mediante el algoritmo cuyo pseudocódigo se puede ver en la figura 6.6.

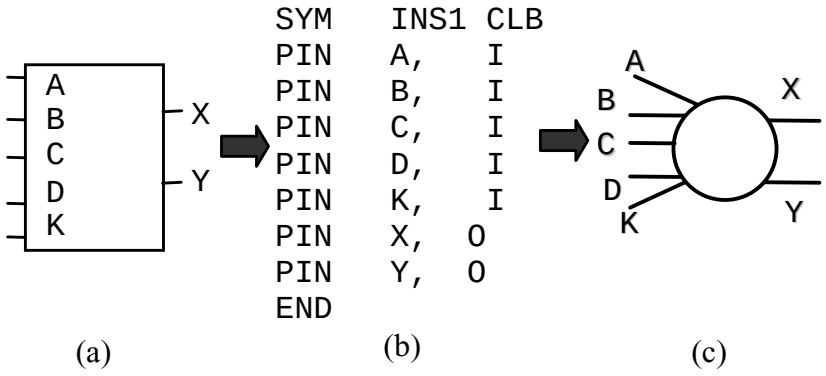

Figura 6.5: *Un ejemplo de la descripción de un bloque lógico como un esquema de CLB (a), en formato XNF (b) y el correspondiente vértice del grafo asociado a ese CLB (c).*

Una vez que se ha obtenido el grafo el circuito se codifica mediante un formato que representa los nodos y las aristas del grafo. Este formato permite describir el grafo e incorporar información relativa al coste de comunicaciones.

1. Lee Netlist en formato XNF
   - 1.1 Lee Bloque SYM
   - 1.2 Genera Vértice
   - 1.3 Rellena Información de aristas
   - 1.4 Rellena información de pines
   - 1.5 Lee Bloque END

2. Enlaza Grafo
   - 2.1 Busca pines comunes
   - 2.2 Genera Aristas
   - 2.3 Enlaza Grafo

3. Genera archivo descriptivo del grafo

Figura 6.6: *Algoritmo para obtener el grafo a partir del formato XNF*

El archivo que genera el programa contiene en primer lugar una línea que indica el número de vértices del grafo. A continuación una línea por cada uno de los vértices, que incluye el coste asociado a cada bloque y dos cifras por cada conexión del vértice con otros vértices del grafo. La primera indica el número del vértice al que se conecta y la segunda el coste asociado a dicha conexión. Finalmente para indicar que no hay mas conexiones la línea concluye con un carácter especial (-1). Una vez que se han expresado todas las conexiones con sus respectivos costes se incluye una línea (.e), que indica el final del archivo. En la figura 6.7b se puede la descripción según este formato del grafo dibujado en la figura 6.7a.

6.2.3 Proceso de partición

Vamos a utilizar las propiedades de los árboles de expansión para obtener las particiones de nuestro circuito. El método funciona cómo sigue. En primer lugar se transforma el netlist del circuito en una descripción en forma de grafo, a continuación se obtiene un árbol de expansión del grafo. Seguidamente, se utilizan las propiedades de los árboles para obtener tantas particiones como sean necesarias y esto se consigue eliminando aristas. Es decir, si necesitamos p particiones debemos seleccionar p-1 aristas del árbol.

En la figura 6.8 presentamos un esquema gráfico del proceso de partición [85].

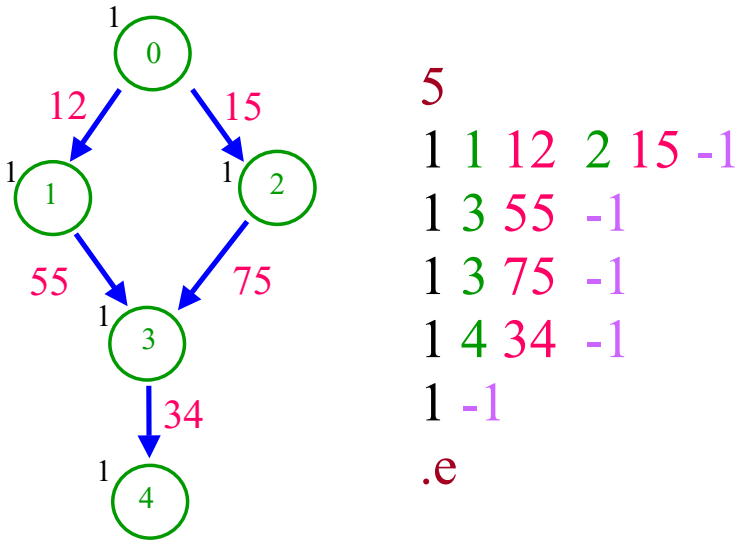

Figura 6.7: *Ejemplo de un grafo y su descripción en el formato de entrada al algoritmo. En el ejemplo todos los bloques tienen asociado un coste 1.*

Comenzando con el grafo obtenido del netlist (a), se obtiene el árbol de expansión mediante el algoritmo de Kruskal (b). A continuación hacemos uso de la propiedad 3 de los árboles explicada en el apartado 6.1. Es decir si queremos obtener n particiones debemos eliminar $n-1$ aristas. El ejemplo de la figura es para 4 particiones. Por lo tanto debemos seleccionar únicamente 3 aristas (c). Finalmente una vez eliminadas las aristas se obtienen las particiones (d).

En la figura 6.8 se puede ver claramente una de las propiedades más importantes del algoritmo propuesto en este trabajo. Al trabajar con el árbol y eliminar las aristas los bloques que están conectados en el circuito original tenderán a estar en la misma FPGA, salvo obviamente los bloques que intervienen en las aristas seleccionadas para obtener las particiones.

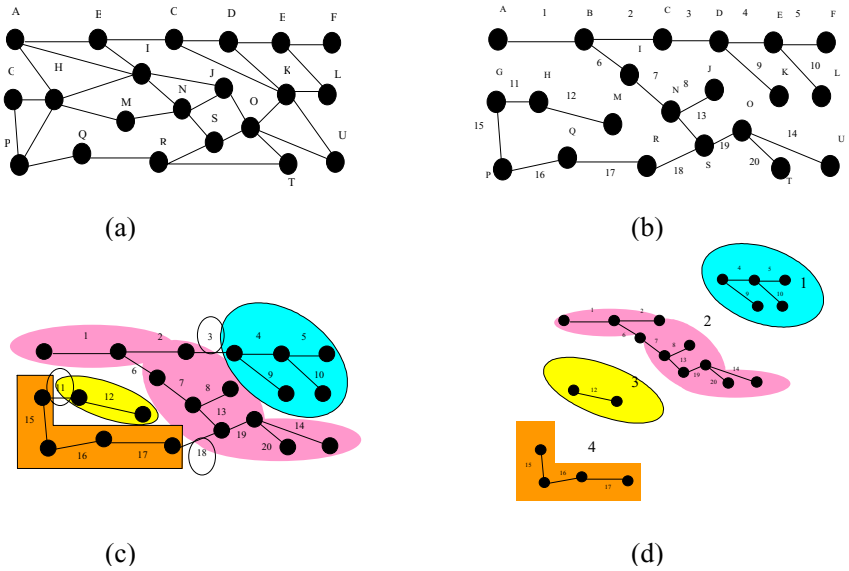

Figura 6.8: *Un ejemplo del proceso de partición para 4 FPGAs*

6.2.4 Salida del algoritmo

Una vez que se han obtenido las particiones y se ha aplicado el AG, la representación del grafo se transforma en archivos XNF que se pueden implementar sobre las FPGAs de la tarjeta. Para ello bastará con indicar en el proceso de conversión a XNF qué aristas han pasado a ser pines de E/S.

Continuando con el ejemplo presentado en la figura 6.8, podemos observar en la figura 6.9 como las particiones obtenidas mediante la eliminación de las aristas, representan la implementación sobre las FPGAs. Es importante destacar que, pese a que el proceso de partición se realiza trabajando con el árbol de expansión, a la hora de realizar la transformación necesaria para comprobar las conexiones entre FPGAs debemos trabajar con el grafo completo. Esto se debe a que la información relativa a las conexiones está incluida en el grafo completo y el árbol de expansión sólo trabaja con alguna de ellas.

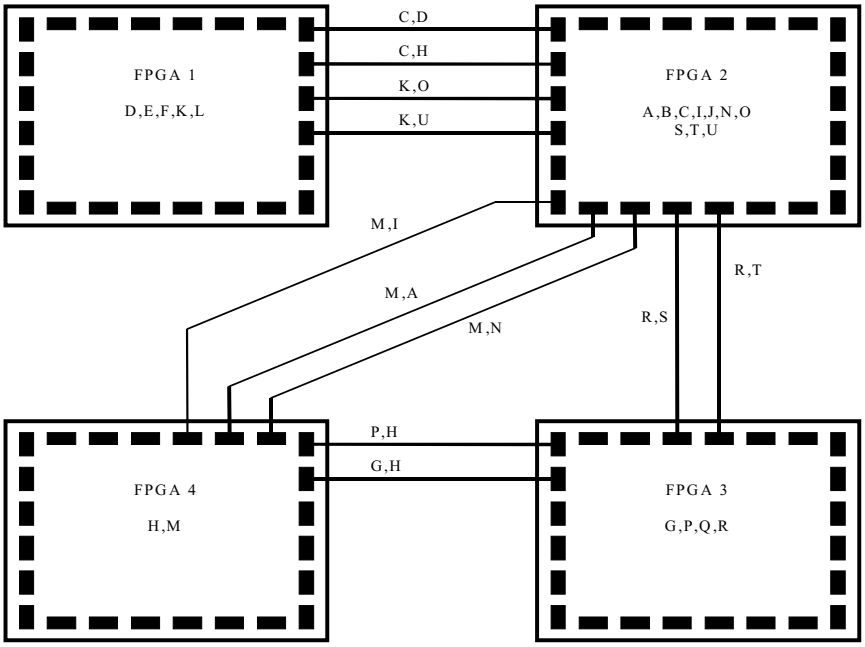

Figura 6.9: *Representación final de las particiones sobre las FPGAs. Las conexiones representan las aristas del grafo y están nombradas por los vértices que representan*

6.3 Función de coste para topologías malla

Una vez obtenida la partición, cada individuo, debe ser evaluado. La función de coste que se describe en este apartado es la encargada de llevar a cabo esta misión. La función de coste debe dirigir al algoritmo para que cumpla las restricciones y busque los objetivos para los que ha sido diseñado. El objetivo de nuestro algoritmo es obtener particiones que cumplan las restricciones propias de la tarjeta sobre la que se va a implementar el circuito. Las principales restricciones son las relativas al número de PE/S y de bloques lógicos [87, 42].

Por lo tanto el algoritmo debe minimizar el número de PE/S utilizados, o lo que es lo mismo el número de aristas inter-FPGA y además realizar una distribución homogénea de los bloques en las FPGAs. Entendemos por aristas inter-FPGA a aquellas que unen bloques de distintas partes. Suponemos también que utilizaremos todas las FPGAs del sistema. Como vemos tenemos que resolver un problema multi-objetivo [40].

Formalmente podemos definir un problema multi-objetivo de la siguiente forma:

"Sea $n > 0$, $k > 0$, $D = \chi_1 \times ... \times \chi_n \subseteq \mathbf{R^n}$, y $\Re = Y_1 \times ... \times Y_n \subseteq \mathbf{R^k}$. Sea $f_i : D \rightarrow Y_i$ para $1 \leq i \leq k$ y finalmente, $F : D \rightarrow \Re$ tal que $F(x) = (f_1(x), f_2(x), ..., f_k(x))$.El propósito es optimizar la función F(x) bajo ciertas condiciones. $G_1(x, p_1) \leq 0, ..., G_u(x, p_u) \leq 0$ donde $p = (p_1, ..., p_u)$ son parámetros reales con un determinado valor".

Este tipo de problema es bastante conocido y se han implementado un buen número de algoritmos, genéticos y no genéticos para resolverlos [44, 145]. Una de las técnicas que se utiliza más habitualmente es la aplicación de funciones objetivo aditivas para guiar al algoritmo. Esto es, funciones objetivos que incluyen varios términos con pesos. Cada uno de los términos se encarga de cumplir un objetivo y el peso que se le asigna a cada término va en consonancia con su influencia sobre la solución. Nosotros hemos utilizado esta técnica y para ello hemos estudiado cada una de las funciones de coste parciales.

Sin embargo, antes de estudiar estas funciones de coste, hicimos una comprobación de la convergencia del algoritmo. Para ello estudiamos su funcionamiento para minimizar el número de aristas inter-FPGA (N_{CE}) del sistema mediante la función de coste FF_1 (ecuación 6.1):

$$FF_1 = \frac{1}{N_{CE}} \tag{6.1}$$

En las figuras 6.10 y 6.11 podemos comprobar que el algoritmo funciona correctamente. En ellas está representado el número de aristas inter-FPGA del circuito final y logra en todos los casos obtener el mínimo, que se produce cuando sólo hay una arista que conecta cada FPGA con una principal en la que se encuentra la mayor parte de la lógica. Dado que estamos obligando al algoritmo a utilizar todas las FPGAs de la tarjeta nunca llegará a hacerse cero, lo que hace la función es asignar la mayor parte de los bloques lógicos a una FPGA y luego un número muy pequeño en cada una de las otras. De esta forma minimizamos las conexiones entre FPGAs, es decir el número de aristas inter-FPGA mínimo será 7. En las figuras 6.12 y 6.13 aparecen unos gráficos descriptivos de las distribuciones obtenidas. Es importante destacar que estás distribuciones se han obtenido permitiendo que puedan ir tantos bloques como se quiera en una sola partición y que no se ha puesto ninguna re-

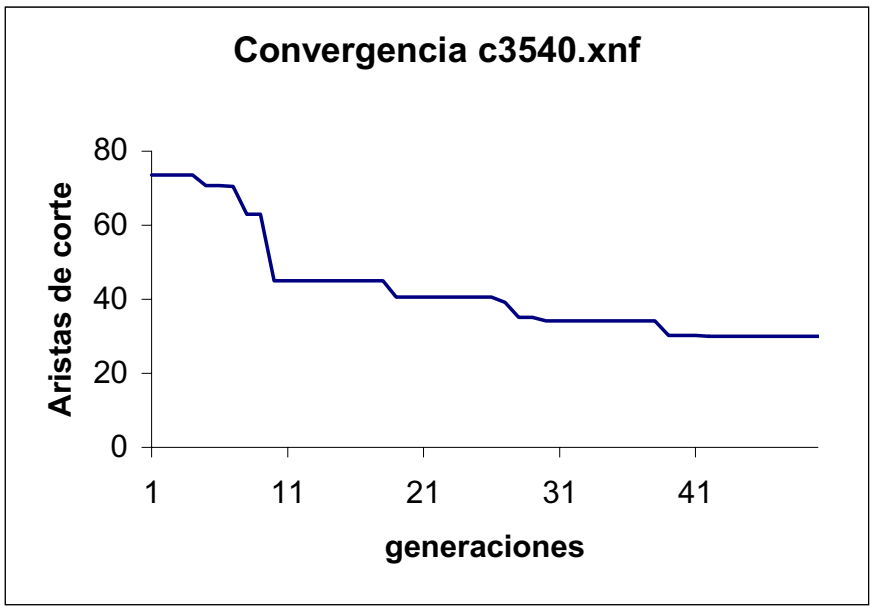

Figura 6.10: *Resultados del AG para el circuito c3540.xnf con FF1*

stricción. El objetivo de estas pruebas es simplemente la comprobación del correcto funcionamiento del algoritmo.

6.3.1 Distribución homogénea de los bloques lógicos

Despues de estudiar diferentes funciones de coste, decidimos que la mejor opción para lograr los objetivos era utilizar una función con términos penalizadores. Introducimos un término que penalice aquellas distribuciones de vértices que no sean ideales. Pero utlizamos una penalización proporcional, es decir, los peores resultados deben tener una mayor penalización.

Vamos a trabajar sobre 8 particiones, y la penalización va a permitir que haya una desviación de un 10 % respecto de lo que sería la partición ideal. Esto significa que cada partición puede tener entre el 11.25% y el 13.75% de los vértices. A esta función de coste la denominamos FF_h.

$$FF_h = \frac{1}{N_{CE} + K \cdot P} \tag{6.2}$$

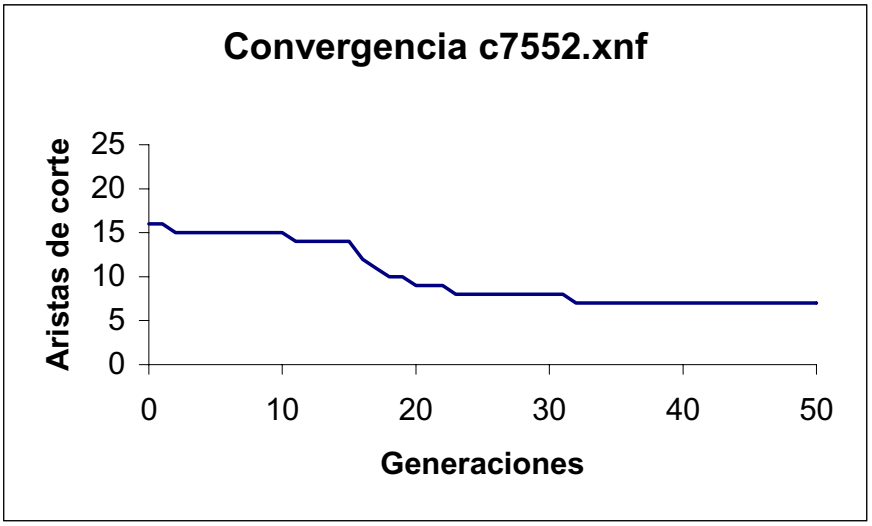

Figura 6.11: *Resultados del AG para el circuito c7552.xnf con FF1*

con

$$P = \sqrt{\frac{\sum_{i=1}^{N_F} P_i}{N_F}} \tag{6.3}$$

$$P_i = \begin{cases} (NB_i - ID)^2 & \text{si} \quad NB_i > 1.10 \cdot ID \\ (ID - NB_i)^2 & \text{si} \quad NB_i < 0.90 \cdot ID \\ 0 & en \quad otro \quad caso \end{cases} \tag{6.4}$$

Donde NCE es el número de aristas inter-FPGA de la solución, NBi el número de vértices de la FPGA i, ID es la distribución ideal, K es una constante que se determina experimentalmente y es el peso en la función multiobjetivo de la penalización, y P es la penalización a aplicar. NB es el número de vértices del circuito con el que estamos trabajando, NF es el número de particiones que queremos obtener, es decir el número de FPGAs que componen el circuito.

El término P es el encargado de cuantificar la penalización. Como la penalización debe ser proporcional al error cometido hemos tomado el error cuadrático medio, ya que eso nos dará una medida de lo que se aproxima la solución a la distribución ideal.

Como hemos dicho K se puede determinar experimentalmente, pero para poder obtener este valor de una manera automática y así conseguir una optimización más

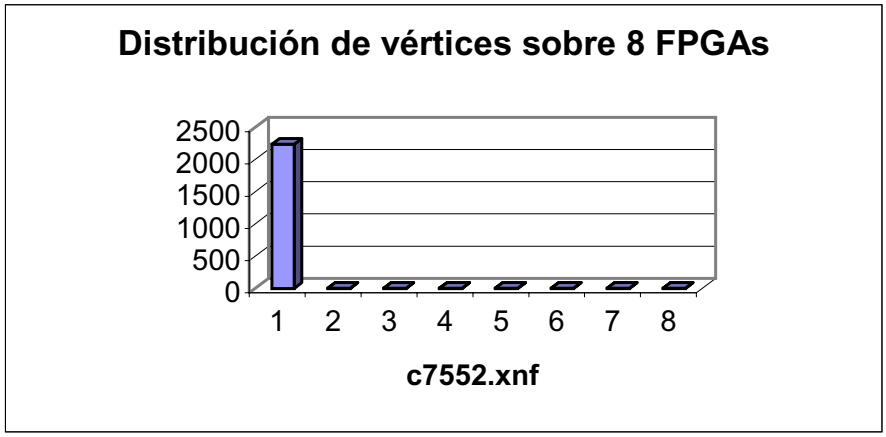

Figura 6.12: *Resultados del AG para el circuito c7552.xnf con FF1*

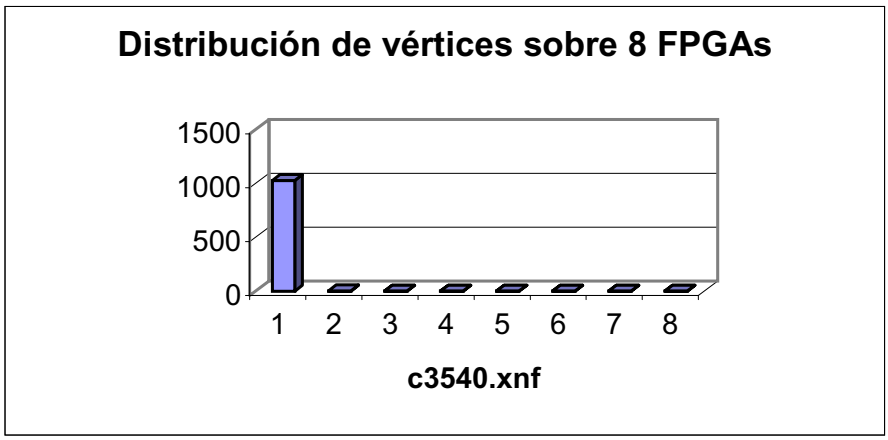

Figura 6.13: *Resultados del AG para el circuito c3540.xnf con FF1*

fina, vamos a tratar de expresarlo en función de las características del problema que estemos tratando. Para ello se hicieron varios experimentos y de éstos se puede deducir lo siguiente:

- El parámetro K depende del tamaño del circuito que se desea implementar (es decir del número de vértices y de aristas que contiene).
- K depende del número de FPGAs del sistema.
- Los mejores resultados se obtienen para aquellos valores de K que cumplen la siguiente expresión:

$$100 \cdot N_{CE} = K \cdot P \tag{6.5}$$

por lo tanto buscaremos una expresión para la constante de la forma $K = f(N_B, N_E, N_F)$ donde N_E es el número de aristas. De (6.5) podemos sacar

$$K \approx \frac{100 \cdot N_{CE}}{P} \tag{6.6}$$

Para hallar el valor estimado de K, debemos obtener los valores estimados de N_{CE} y de P, en función de N_B, N_E, y N_F. Es decir $P = f(N_B, N_E, N_F)$ y $N_{CE} = f(N_B, N_E, N_F)$. Tras hacer una serie de experimentos podemos obtener una relación entre N_{CE}, N_F y N_E. Esta conclusión es bastante lógica, puesto que cuantas más aristas tenga un circuito, más posibilidades habrá de que estén conectados dos vértices de distintas FPGAs. Lo mismo sucede con N_F, cuánto mayor sea el número de particiones más conexiones entre ellas habrá. En la tabla 6.1 están recogidos los resultados de estos experimentos. En ellos se busca una relación de la forma

$$N_{CE} = K_2 \cdot N_F \cdot N_E \tag{6.7}$$

Como se puede ver todos los valores de K_2 son muy próximos a 10^{-2}. Por ello fijamos K_2 al valor medio resultante de analizar todos los experimentos realizados, que es 0.025.

Con estos resultados ya tenemos N_{CE} Busquemos ahora una estimación para P. El término de penalización actúa en los siguientes casos:

- Cuando la FPGA i tiene asignado un número de bloques $NB_i > 1.10 \cdot ID$, y
- Cuando $NB_i < 0.90 \cdot ID$.

Teniendo en cuenta lo anterior, vamos a estimar el valor de la ecuación 6.3. Una buena aproximación es suponer que la mitad de las FPGAs de la tarjeta ($N_F/2$) tiene un exceso de ocupación y la otra mitad un defecto. Como aproximación heurística asumimos que la primera mitad tiene una ocupación de $1.5 \cdot ID$ y la otra mitad

NB	NE	NCE	NF	K2
493	800	164	8	0.025625
493	800	164	8	0.025625
493	800	158	8	0.024687
574	1127	426	8	0.047249
574	1127	425	8	0.047138
574	1127	415	8	0.046029
650	1127	167	8	0.018522
1038	2115	705	8	0.027777
1038	2115	712	12	0.042080
1038	2115	633	8	0.037411
1038	2115	565	8	0.033392
1038	2115	540	8	0.031914
1038	2115	624	8	0.036879
1038	2115	737	8	0.029038
1038	2115	766	12	0.030181
1038	2115	716	12	0.028211
2856	5208	561	12	0.013464
2856	5208	575	8	0.013800
2856	5208	582	8	0.013968
2856	5208	569	8	0.013656
2856	5208	592	8	0.014208
3223	4997	485	12	0.008120
3223	4997	560	12	0.009376
3223	4997	574	12	0.009610
3223	4997	585	12	0.009795

Tabla 6.1: *Obtención experimental de K2*

$0.5 \cdot ID$. Sustituyendo esta información en el sumatorio de la ecuación 6.3 se obtiene:

$$\sum_{i=1}^{N_F} (NB_i - ID)^2 =$$

$$\frac{N_F}{2} \cdot (1.5 \cdot \frac{N_B}{N_F} - \frac{N_B}{N_F})^2 + \frac{N_F}{2}(\frac{N_B}{N_F} - 0.5 \cdot \frac{N_B}{N_F})^2 \tag{6.8}$$

$$P = \sqrt{\frac{(0.5^2 \cdot \frac{N_B}{N_F})^2 + 0.5^2 \cdot \frac{N_B}{N_F})^2) \cdot \frac{N_F}{2}}{N_F}} = \frac{N_B}{2 \cdot N_F} \tag{6.9}$$

Ahora solamente necesitamos sustituir 6.7 y 6.9 en la ecuación 6.6 para obtener

un valor estimado de K antes de la aplicación del algoritmo (ecuación 6.11).

$$K \approx \frac{100 \cdot N_{CE}}{P} = \frac{100 \cdot 0.025 \cdot N_F \cdot N_E}{\frac{N_B}{2 \cdot N_F}} \tag{6.10}$$

$$K = \frac{5 \cdot N_F^2 \cdot N_E}{N_B} \tag{6.11}$$

6.3.2 Estudio de la ubicación para la minimización del número de pines de entrada-salida utilizados

El otro objetivo de nuestro algoritmo es cumplir las restricciones de PE/S. Pero si queremos evaluar el consumo de PE/S que se produce en una partición, debemos suponer algunas condiciones de ubicación. Por este motivo hemos implementado una evaluación de dos pasos. Primero se realiza un proceso de ubicación y con estas condiciones se evalúa en un segundo paso el consumo de PE/S. Para el proceso de ubicación suponemos una malla de FPGAs compuesta por 2 x 4 FPGAs conectadas con sus vecinas más próximas en horizontal y vertical. Un esquema de la topología supuesta se puede ver en la figura 6.4. El algoritmo se puede cambiar fácilmente para una tarjeta con otras características.

Para el proceso de ubicación vamos a considerar las conexiones entre las FPGAs lógicas resultantes. Aquellas FPGAs lógicas que tengan un mayor número de conexiones entre ellas deben ubicarse lo más próximas entre sí que sea posible. Sin embargo, algunas veces dos partes del circuito, que no están directamente conectadas, tienen una fuerte dependencia la una de la otra. Una forma de medir este tipo de conexiones indirectas es la matriz de relación inexacta (MRI) [103]. Esta es una matriz normalizada que se obtiene de la matriz de relación y que se ha utilizado en procesos de clasificación con información imprecisa [79]. En nuestro caso la matriz de relación representa las conexiones entre las distintas FPGAs después del proceso de partición. A partir de ella se calcula la matriz de relación inexacta, pero antes veamos exactamente qué es y cómo se calcula.

Cuando tenemos un grafo en el que hay unas relaciones de dependencias entre los vértices, podemos definir relaciones entre pares de nodos atendiendo a un

determinado criterio y ordenarlos de mayor a menor relación. Utilizamos el concepto de relación de similaridad borrosa, definido previamente en la teoría borrosa o fuzzy. Decimos que la matriz $S = [s_{ij}]$, representa una relación de similaridad borrosa [186] sobre $X = x_1, x_2, ..., x_n$ si los elementos s_{ij}, de S satisfacen las siguientes propiedades:

1. $\forall i, j \in \{1, ..., n\}, s_{ij} \in [0, 1]$
2. $\forall i, \in \{1, ..., n\}, s_{ii} = 1$
3. $\forall i, j \in \{1, ..., n\}, s_{ij} = s_{ji}$
4. $\forall i, j, k \in \{1, ..., n\}, s_{ij} \geq (max\ (min(s_{ij}, s_{kj})), k \in \{1, ..., n\} - \{i, j\}$

Los elementos de S pueden tomar cualquier valor en el intervalo $[0, 1]$. Cuanto mayor sea el valor de s_{ij}, mayor relación habrá entre x_i y x_j. La matriz S es simétrica y todos los elementos de la diagonal son 1, ya que indican la relación que tiene un vértice con él mismo.

La propiedad 4 tiene una importancia fundamental. Esta propiedad muestra el fenómeno de transitividad que existe en las relaciones borrosas. Indica que la relación directa entre dos vértices debe ser mayor o igual que la relación de éstos a través de cualquier otro vértice intermedio. Si esta relación se aplica de una forma iterativa, podemos concluir que la relación directa entre dos vértices es mayor o igual que la que se puede obtener a través de cualquier secuencia intermedia de vértices. A la matriz S se le denomina matriz de relación inexacta.

Para calcularla, una vez que tenemos una matriz que modele la relación entre los distintos vértices, obtenemos la matriz normalizada N con todos los elementos de la diagonal iguales a 1. El objetivo es ahora transformarla en una matriz que tenga en cuenta no sólo las relaciones directas entre vértices, si no también las relaciones indirectas a través de caminos alternativos y se calcula a partir de la matriz N mediante la siguiente relación:

$$s_{ij} = \sum_{\forall k_1,k_2,...,k_l} (n_{i,k_1} * n_{k_1,k_2} * ... * n_{k_l,j}) \tag{6.12}$$

En esta expresión la notación utilizada es la usual en la teoría borrosa, y los símbolos + y * no tienen el significado matemático clásico, si no que representan el máximo y el mínimo respectivamente. En realidad lo que se hace es calcular todos los caminos posibles entre los dos vértices que indican i y j. De cada uno de estos caminos, se selecciona su arista de menor peso y de todas éstas la mayor; éste será el valor de s_{ij}. Por ejemplo supongamos el grafo de la figura 6.14, en él ya están normalizados los valores de las aristas. Si queremos calcular el término s_{12} (que es el mismo que el s_{21}), lo primero que hago es ver todos los caminos posibles:

Camino a): $1 - - \mathit{0.9} - - 2$
Camino b): $1 - - \mathit{0.8} - - 5 - - \mathit{0.7} - - 3 - - \mathit{0.5} - - 2$
Camino c): $1 - - \mathit{0.8} - - 5 - - \mathit{0.7} - - 3 - - \mathit{0.8} - - 4 - - \mathit{0.2} - - 2$

Entre los números de los vértices se encuentra el valor de la arista que los une. Pues bien, de cada uno de estos caminos escogemos la arista de menor peso:

Camino a): 0.9
Camino b): 0.5
Camino c): 0.2

Finalmente para saber el valor de s_{12} de estos valores se selecciona el mayor que es 0.9. En este ejemplo no ha cambiado el valor de s_{12} con respecto al valor de n_{12}. Veamos un caso donde si cambia y en el que se observa claramente la utilidad de la matriz de relación inexacta. Calculemos s_{45} , si nos fijamos en el grafo n_{45}=0 lo que indicaría que no hay relación entre los vértices 4 y 5. De igual forma calculamos todos los caminos para conectar 4 y 5:

Camino a): $4 - - \mathit{0.8} - - 3 - - \mathit{0.5} - - 2 - - \mathit{0.9} - - 1 - - \mathit{0.8} - - 5$
Camino b): $4 - - \mathit{0.8} - - 3 - - \mathit{0.7} - - 5$
Camino c): $4 - - \mathit{0.2} - - 2 - - \mathit{0.9} - - 1 - - \mathit{0.8} - - 5$

Las aristas de menor peso son:

Camino a): 0.5
Camino b): 0.7
Camino c): 0.2

Y la relación entre 4 y 5 será por tanto distinta de cero, $s_{45} = 0.7$ que es el máximo valor de las aristas de menor peso. Como muestra este valor aunque entre los vértices 4 y 5 no existe relación directa, sí que existe una dependencia a través de los otros vértices del grafo.

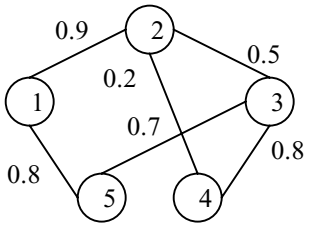

Figura 6.14: *grafo de ejemplo para el calcular la MRI.*

La matriz de relación inexacta se puede calcular a partir de la matriz de relación normalizada mediante un algoritmo de complejidad n^3 que se describe en la figura 6.15 en forma de pseudocódigo (recordemos la notación especial en la que "+" indica el máximo y "*" el mínimo).

Veamos ahora como se puede utilizar esta matriz particularizada para nuestro caso. Supongamos que después de realizar la partición hemos obtenido una solución en la que hay unas conexiones entre las distintas FPGAs cómo la representada en

```
for k = 1 to p
        for i = 1 to p
                if n_ik <> 0 then
                        for j = 1 to p
                                n_ij = n_ij + n_ik * n_kj
                        endfor
                endif
        endfor
endfor
```

Figura 6.15: *Algoritmo para calcular la matriz de relación inexacta.*

la tabla 6.2. A partir de ella construimos la matriz de relación sin normalizar R.

$$R = \begin{bmatrix} 0 & 21 & 14 & 15 & 0 & 1 & 0 & 3 \\ 21 & 0 & 0 & 12 & 0 & 7 & 1 & 0 \\ 14 & 0 & 0 & 10 & 9 & 0 & 0 & 0 \\ 15 & 12 & 10 & 0 & 1 & 2 & 3 & 4 \\ 0 & 0 & 9 & 1 & 0 & 0 & 15 & 7 \\ 1 & 7 & 0 & 2 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 3 & 15 & 0 & 0 & 0 \\ 3 & 0 & 0 & 4 & 7 & 1 & 0 & 0 \end{bmatrix}$$

La única diferencia con respecto a la definida anteriormente es que los elementos de la diagonal los hemos definido como 0, pues no los vamos a utilizar en el proceso de ubicación. Para normalizar dividimos por el valor máximo de los elementos de la matriz, en el ejemplo este valor es igual a $R_{11} = 21$. Con ello obtenemos la matriz N.

FPGA Inicial	FPGA Final	Conexiones
1	2	21
1	3	14
1	4	15
1	5	0
1	6	1
1	7	0
1	8	3
2	3	0
2	4	12
2	5	0
2	6	7
2	7	1
2	8	0
3	4	10
3	5	9
3	6	0
3	7	0
3	8	0
4	5	1
4	6	2
4	7	3
4	8	4
5	6	0
5	7	15
5	8	7
6	7	0
6	8	1
7	8	0

Tabla 6.2: *Ejemplo de conexiones entre las FPGAs después de la partición*

$$N = \begin{bmatrix} 0 & 1 & 0.66 & 0.71 & 0 & 0.047 & 0 & 0.14 \\ 1 & 0 & 0 & 0.57 & 0 & 0.33 & 0.047 & 0 \\ 0.66 & 0 & 0 & 0.48 & 0.43 & 0 & 0 & 0 \\ 0.71 & 0.57 & 0.48 & 0 & 0.047 & 0.095 & 0.14 & 0.19 \\ 0 & 0 & 0.43 & 0.047 & 0 & 0 & 0.71 & 0.33 \\ 0.047 & 0.33 & 0. & 0.095 & 0 & 0 & 0 & 0.047 \\ 0 & 0.47 & 0 & 0.14 & 0.71 & 0 & 0 & 0 \\ 0.14 & 0 & 0 & 0.19 & 0.33 & 0.047 & 0 & 0 \end{bmatrix}$$

A continuación, utilizando el algoritmo de la figura 6.15, obtenemos la matriz de relación inexacta.

$$S = \begin{bmatrix} 0 & 1 & 0.66 & 0.71 & 0.43 & 0.33 & 0.43 & 0.33 \\ 1 & 0 & 0.66 & 0.71 & 0.43 & 0.33 & 0.43 & 0.33 \\ 0.66 & 0.66 & 0 & 0.66 & 0.43 & 0.33 & 0.43 & 0.33 \\ 0.71 & 0.71 & 0.66 & 0 & 0.43 & 0.33 & 0.43 & 0.33 \\ 0.43 & 0.43 & 0.43 & 0.43 & 0 & 0.33 & 0.71 & 0.33 \\ 0.33 & 0.33 & 0.33 & 0.33 & 0.33 & 0 & 0.33 & 0.33 \\ 0.43 & 0.43 & 0.43 & 0.71 & 0.71 & 0.33 & 0 & 0.33 \\ 0.33 & 0.33 & 0.33 & 0.33 & 0.33 & 0.33 & 0.33 & 0 \end{bmatrix}$$

A partir de esta obtenemos la matriz triangular superior tu(S), para eliminar información redundante.

$$tu(S) = \begin{bmatrix} 0 & 1 & 0.66 & 0.71 & 0.43 & 0.33 & 0.43 & 0.33 \\ 0 & 0 & 0.66 & 0.71 & 0.43 & 0.33 & 0.43 & 0.33 \\ 0 & 0 & 0 & 0.66 & 0.43 & 0.33 & 0.43 & 0.33 \\ 0 & 0 & 0 & 0 & 0.43 & 0.33 & 0.43 & 0.33 \\ 0 & 0 & 0 & 0 & 0 & 0.33 & 0.71 & 0.33 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.33 & 0.33 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.33 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Una vez obtenida construimos lo que hemos llamado lista de relación inexacta (LRI) (tabla 6.3). Para ello no tenemos más que colocar los elementos de tu(S) por su valor en orden descendente. Cada elemento de la lista contendrá 3 valores, el elemento de la matriz y los dos índices que indican su posición dentro de la misma. Estos dos índices nos indican las FPGAs lógicas que están relacionadas por ese valor (recordamos que estamos realizando una ubicación de las FPGAs lógicas obtenidas de la partición sobre las físicas que contiene la tarjeta de implementación). En las explicaciones las FPGAs lógicas están numeradas de 1 a 8 y las físicas etiquetadas de A a F.

La tabla 6.3 muestra la LRI del ejemplo. Con ella ya tenemos un orden de preferencia para la ubicación sobre la tarjeta. En la figura 6.16 (a) podemos ver un esquema de la tarjeta. El primer elemento de LRI indica las 2 primeras FPGAs lógicas a colocar. Éstas irán en las posiciones B y C de la tarjeta. Puesto que son

FPGA Inicial	FPGA Final	Conexiones
1	2	1
1	4	0.71
2	4	0.71
5	7	0.71
1	3	0.66
2	3	0.66
3	4	0.66
1	5	0.43
1	7	0.43
2	5	0.43
2	7	0.43
3	5	0.43
3	7	0.43
4	5	0.43
4	7	0.43
1	6	0.33
1	8	0.33
2	6	0.33
2	8	0.33
3	6	0.33
3	8	0.33
4	6	0.33
4	8	0.33
5	6	0.33
5	8	0.33
6	7	0.33
6	8	0.33
7	8	0.33

Tabla 6.3: *LRI correspondiente a la matriz S del ejemplo*

las que tienen un mayor número de conexiones, tendrán una mayor probabilidad de conectarse con el resto y al ponerlas en estas FPGAs quedan un mayor número de posiciones cercanas libres para situar otras FPGAs conectadas a ellas. Para seguir la ubicación descendemos en la LRI y seleccionamos con el siguiente criterio:

- Si ninguna de las 2 FPGAs lógicas ha sido ubicada, se colocarán en las posiciones que haya libres con el siguiente orden: F, G, A, E, D y H. La razón es la misma que la explicada para ubicar las dos primeras.

- Si sólo se ha ubicado una de las 2 FPGAs lógicas indicadas por la LRI, la que no se ha ubicado se tratará de poner tan próxima a la ubicada como sea posible. El orden para buscar una posición libre será el siguiente: centro, izquierda y

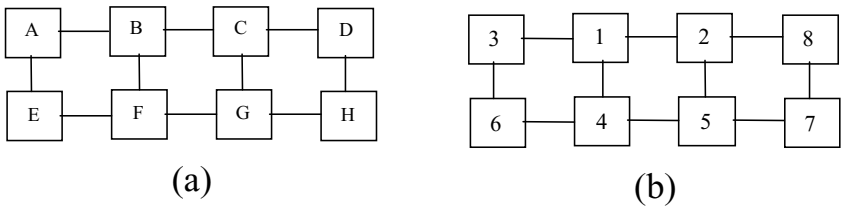

Figura 6.16: *Esquema de la tarjeta Multi-FPGA. En (a) las letras indican las FPGAs físicas y en (b) los números indican cómo quedarán ubicadas las FPGAs lógicas para nuestro ejemplo.*

derecha. Por ejemplo si tenemos que ubicar una FPGA conectada con otra que esta en la posición B, primero intentaremos la posición F, a continuación la posición A, después la C y así sucesivamente hasta que se encuentre una posición libre. Este orden de ubicación es simplemente una decisión heurística, cualquier otro sería perfectamente válido.

- Si ambas están ubicadas se ignora ese elemento y se pasa al siguiente en la lista.

Este proceso de ubicación finalizará cuando todas las FPGAs lógicas se hayan ubicado (independientemente de si se ha llegado al final de la LRI o no). Una vez hecho esto se puede calcular el consumo de PE/S que sería necesario para implementar dicha solución. Lo único que hay que hacer es contar las conexiones, teniendo en cuenta que para conectar 2 FPGAs separadas necesitaremos atravesar otras y se producirá también un consumo de PE/S en éstas. Las FPGAs tienen PE/S libres para las conexiones en 3 laterales que denominamos izquierda, derecha y centro. Los PE/S de entrada salida exteriores (4^o lateral) quedan libres para las señales de entrada. Con esto calculamos el número de PE/S que faltarían para implementar la solución (N_{IOL}). En caso de que este valor sea positivo habrá que penalizar la solución que se está estudiando.

Resumiendo, la función de coste total está expresada en la ecuación 6.13, en ella se recogen las dos restricciones impuestas por la tarjeta, por una lado los PE/S de entrada-salida (N_{IOL}) y por otro la distribución de los bloques lógicos sobre la tarjeta mediante el término ($K \cdot P_1 + N_{CE}$)

$$FF = \frac{1}{N_{IOL} + K \cdot P_1 + N_{CE}} \tag{6.13}$$

6.4 Función de coste para topologías Crossbar

La función de coste necesaria para tratar una topología Crossbar es la misma que para una topología malla. Es decir la ecuación 6.13. Sin embargo la forma de calcular (N_{IOL}) es mucho más sencilla, ya que no es necesario tener en cuenta las conexiones entre la FPGAs para hacer la ubicación.

Supongamos una topología crossbar de 12 FPGAs, como la de la figura 6.17. En este sistema tenemos 6 FPGAs (Li) para implementar la lógica y otras 6 (Ri) para implementar las conexiones. En este caso a la hora de contabilizar los PE/S utilizados no es necesario contabilizar conexiones a través de otras FPGAS porque todas las conexiones se realizan a través de las FPGAs de rutado.

La ubicación es irrelevante y por simplificación se asignan las FPGAs lógicas a las físicas directamente. A continuación se van evaluando los pines libres que quedan al realizar todas las conexiones entre las FPGAs físicas. En la figura 6.17 si vamos a conectar las FPGAs $L1$ y $L2$, lo podemos hacer a través de $R1$ o $R2$. FP_X indica el número de pines libres de la FPGA X. $Conexiones_{XJ}$ es el número de conexiones después de la partición entre la FPGA X y la J. El proceso para contabilizar los pines sería el siguiente:

- Comprobar cual tiene más pines sin utilizar de $R1$ y $R2$
- Se elige el que tenga más pines libres, supongamos $R1$.
- El número de pines libres de $R1$ pasa a ser $FP_{R1} = FP_{R1} - Conexiones_{L1,L2}$
- $FP_{L1} = FP_{L1} - Conexiones_{L1,L2}$
- $FP_{L2} = FP_{L2} - Conexiones_{L1,L2}$

Una vez que se han contabilizado todas las conexiones se obtiene (N_{IOL}) y se utiliza la ecuación 6.13 para evaluar la partición.

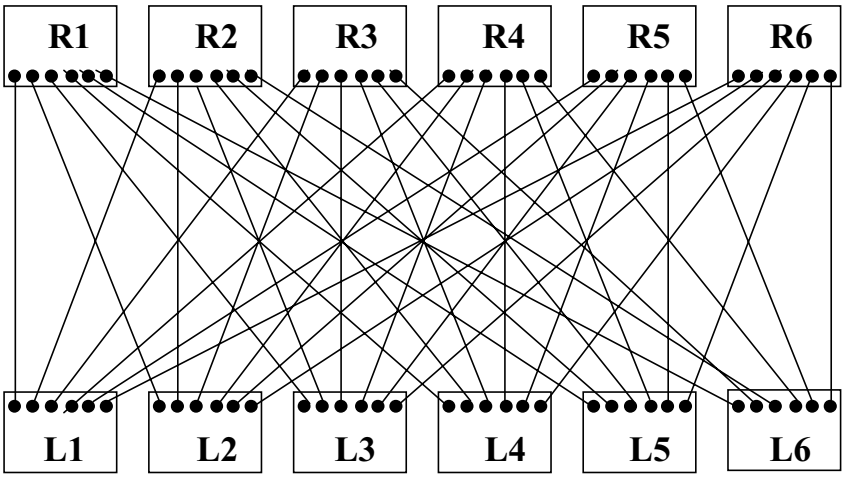

Figura 6.17: *Ejemplo de topología crossbar para 16 FPGAS*

6.5 Codificación genética del problema

Una solución es un conjunto de FPGAs lógicas que nosotros representamos mediante un árbol de expansión del que se han eliminado un determinado número de aristas. La codificación elegida para representar una solución es el conjunto de aristas eliminadas del árbol de expansión para generar las FPGAs lógicas. Por ello debemos identificar las aristas del árbol y para ello lo que hacemos es numerarlas. Puesto que para k particiones debemos eliminar $k-1$ aristas, si queremos hacer k particiones tendremos individuos cuyos cromosomas tendrán $k-1$ genes. El valor de estos genes puede ser cualquiera de los valores identificativos de las aristas del árbol. El valor del gen indica la arista que se ha de eliminar. Por lo tanto el alfabeto de nuestro algoritmo es:

$$\Omega = \{\ 0\ ,\ 1\ ,\ 2\ ,\ \ldots\ ,\ n-1\ \}$$

siendo n el número de vértices del grafo que representa al circuito que se quiere implementar.

En la figura 6.18 vemos un ejemplo de la codificación: supongamos que tenemos un árbol como el de la figura 6.18a. Para llevar a cabo la partición eliminamos las

aristas 3, 4 y 6. Con esto obtendríamos las 4 particiones representadas en la figura 6.18b. Pues bien, el cromosoma que representa esta solución será precisamente el (3 4 6).

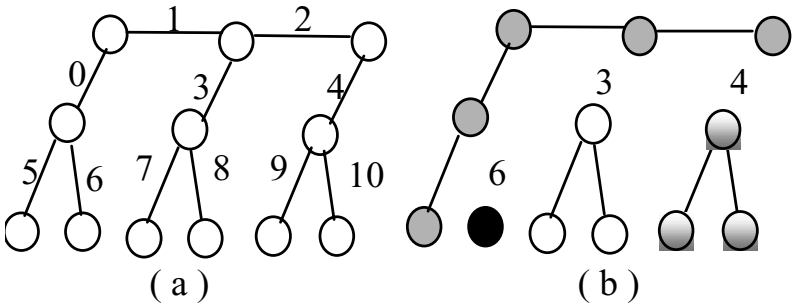

Figura 6.18: *Ejemplo de la codificación utilizada. La figura muestra el significado del cromosoma (3 4 6)*

6.6 AG simple

El problema de la partición lo hemos solucionado mediante AG simples, compactos y compactos híbridos y la representación de las soluciones es la misma para todos ellos, sin embargo hay una diferencia a la hora de trabajar, el algoritmo genético simple utiliza operadores genéticos mientras que los compactos simulan la existencia de la población mediante un vector de probabilidades.

El algoritmo genético tiene la misma estructura que el presentado en el apartado 3.1. Los cambios realizados son los necesarios para poder tratar nuestro problema. En primer lugar se obtienen los datos del circuito ya transformado en un grafo. A continuación se crea el árbol de expansión y se calcula la ocupación ideal ID. Como en cualquier AG simple se genera la población inicial y se aplica el algoritmo. La particularidad aparece en la evaluación. En primer lugar se decodifican los individuos y se obtienen las particiones que representan. Se evalua el número de bloques por cada partición y se realiza la ubicación y la cuenta de pines. Con todo esto se calcula el valor de la función de coste para cada individuo y se sigue ejecutando el

algoritmo genético como se realiza habitualmente, es decir aplicando los operadores de selección, cruce y mutación. El pseudocódigo es el siguiente:

leer_archivo(archivo,&datos);
sacar_aristas(datos,&aristas);
crear_arbol(aristas,&arbol);
ocupa_ideal=1.0*nbloques/fpgas;
$g \Leftarrow$ número de generaciones
$poblac \Leftarrow$ población inicial
$ind \Leftarrow$ número de individuos
for $i = 1$ to g **do**
  for $k = 1$ to n **do**
    decodificacion(poblacion,arbol,particion);
    crea_distribucion(particion,&distribucion);
    bloques_distribucion(distribucion,n_clbs2,situacion);
    cuenta_aristas(situacion,aristas,&aristascorte,&fpines);
    fitness_function(aristascorte,fpines,&coste[0],n_clbs2);
  end for
  Obtiene_mejor($best$)
  calculo de la probabilidad de seleccion
  Selecciona($poblac$, $poblac_{sel}$)
  Cruza($poblac_{sel}$, $poblac_{cru}$)
  Muta($poblac_{cru}$, $poblac_{mut}$)
  $poblac \Leftarrow poblac_{mut}$
  Obtiene_peor($worst$)
  $worst \Leftarrow best$
end for
for $j = 1$ to ind **do**
  decodificacion(poblacion,arbol,particion);
  crea_distribucion(particion,&distribucion);
  bloques_distribucion(distribucion,n_clbs2,situacion);
  cuenta_aristas(situacion,aristas,&aristascorte,&fpines);
  fitness_function(aristascorte,fpines,&coste[0],n_clbs2);

Calcula_coste(*poblac*, FC_j)
end for
Mostrar_Resultados

6.6.1 Operadores genéticos

Los operadores genéticos utilizados son los mismos que en el caso de las topologías libres.

Operador de selección

Hay varias formas de realizar la selección de los individuos. En nuestro algoritmo se ha utilizado la más sencilla, el método de la ruleta. Recordemos el proceso a seguir:

- Primero se calcula la probabilidad de selección (Ps). Para ello, debemos evaluar la función objetivo para cada individuo y obtener la suma de todas ellas.
- Con esto la probabilidad de selección será $P_{s_i} = FF_i / \sum_{i=1}^{n} FF_i$
- A continuación se calcula la probabilidad de selección acumulada para cada individuo.
- Se generan tantos números aleatorios como individuos tenga la población
- Cada número aleatorio se compara con las probabilidades acumuladas y se escoge el individuo que tenga asociada una probabilidad inmediatamente menor al número aleatorio.

Existen diversas variaciones de este proceso de selección simple. Entre ellas podemos destacar el modelo elitista que está tambień incorporado en nuestro algoritmo. El modelo elitista consiste en guardar siempre el mejor individuo de la población para la siguiente generación, normalmente sustituyéndolo por el peor. Nosotros hemos incorporado el elitismo una vez realizado el proceso de cruce.

Operador de Cruce

El operador de cruce realiza una mezcla de dos individuos que se han seleccionado inicialmente. También se lleva a cabo atendiendo a una probabilidad, en este caso probabilidad de cruce (Pc) fija e igual para todos los individuos. El proceso es simple. Se generan dos números aleatorios, uno para cada individuo. Si estos números son menores que la probabilidad de cruce se realiza el cruce. Mediante un proceso aleatorio se elige el punto de cruce, que será un número menor que k, siendo k el número de genes de un cromosoma, y a partir de ese gen se intercambian todos entre los dos individuos. Este es el cruce por un punto.

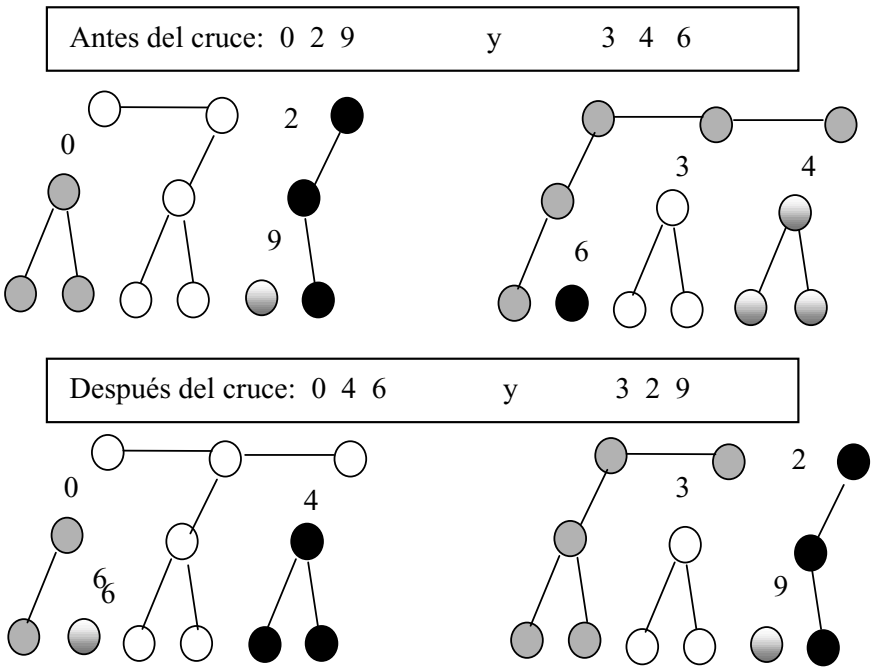

Figura 6.19: *Ejemplo del efecto del operador cruce sobre dos individuos*

Existen otras variaciones en el que se seleccionan varios puntos de cruce y se aplica el mismo mecanismo de intercambio de genes. Los algoritmos utilizados en este trabajo utilizan el cruce por un punto debido a que tenemos un número reducido de genes y si intercambiamos por varios puntos afectaría a la convergencia del algoritmo [32]. En la figura 6.19 podemos ver un ejemplo de cómo actúa el

operador de cruce y su resultado sobre la partición. Supongamos que tenemos los dos individuos siguientes:

$$Individuo\ \ A = (0\ \ 2\ \ 9)$$

$$Individuo\ \ B = (3\ \ 4\ \ 6)$$

Supongamos además que los números aleatorios generados indican que se produzca el cruce por el primer gen. En este caso obtendremos los dos siguientes individuos después del cruce:

$$Individuo\ \ A = (0\ \ 4\ \ 6)$$

$$Individuo\ \ B = (3\ \ 2\ \ 9)$$

Operador de mutación

El operador de mutación en los algoritmos genéticos tiene como misión permitir que el algoritmo escape de óptimos locales en los que puede caer durante la búsqueda de soluciones. Su objetivo es introducir cambios aleatorios en los individuos con una probabilidad muy baja, de forma que se abran nuevos caminos de búsqueda en el espacio de soluciones. Al ser la probabilidad de mutación baja, no provoca que la búsqueda se convierta en aleatoria. Nosotros hemos implementado un operador de mutación clásico.

El valor de la probabilidad de mutación es 0.002, es decir el 0.2%. El operador de mutación actúa de la siguiente forma: se genera un número aleatorio y si es menor que la probabilidad de mutación el gen se sustituye por otro, es decir se cambia la arista que indica el gen por otra para realizar la partición. La figura 6.20 muestra como afectaría a una partición una mutación para el mismo ejemplo utilizado anteriormente. Para el individuo (3 4 6) se produce una mutación en el segundo gen y el nuevo cromosoma sería (3 5 6).

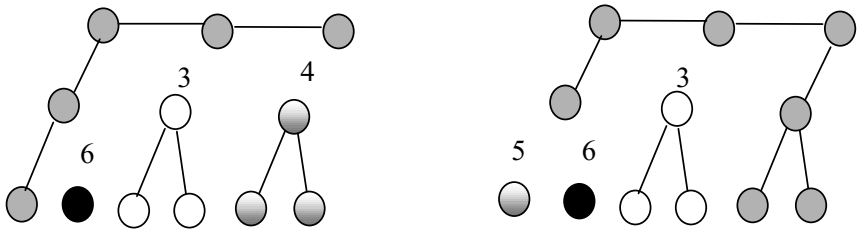

Figura 6.20: *Ejemplo del efecto del operador mutación sobre una solución*

6.6.2 Operador de Regeneración

La codificación elegida es muy directa y por supuesto esto es una ventaja a la hora de implementar el algoritmo. Sin embargo cuando el número de particiones que se quiere obtener es reducido el número de genes con el que estamos trabajando también lo es y pueden aparecer problemas de convergencia prematura. Esto es, el algoritmo evoluciona bien en las primeras generaciones pero la diversidad de la población se reduce rápidamente por lo que converge en pocas generaciones a una solución no deseada.

Para subsanar este problema se ha implementado una nueva técnica basada en la mutación. Es lo que hemos denominado *operador de regeneración*. El operador controla el valor del mejor individuo durante sucesivas generaciones. Cuando detecta que el mejor valor se repite durante 10 generaciones, se aplica una mutación con probabilidad 1 al 25% de la población en todos sus genes, en realidad lo que hacemos es reemplazar el 25% de los individuos por unos nuevos de la misma forma que se generó la población inicial. La única precaución que hay que tener es la de no eliminar el mejor individuo, que se va guardando de generación en generación.

En las gráficas siguientes se puede ver cómo el algoritmo genético sin el operador de regeneración converge muy rápidamente mientras que, aplicando esta mutación especial el algoritmo consigue salir de esa convergencia y mejorar el resultado de la búsqueda considerablemente. En las figuras 6.21 a 6.23 se muestran estos resultados para 3 circuitos de los utilizados en la fase experimental y con la función de coste explicada en el apartado 6.3

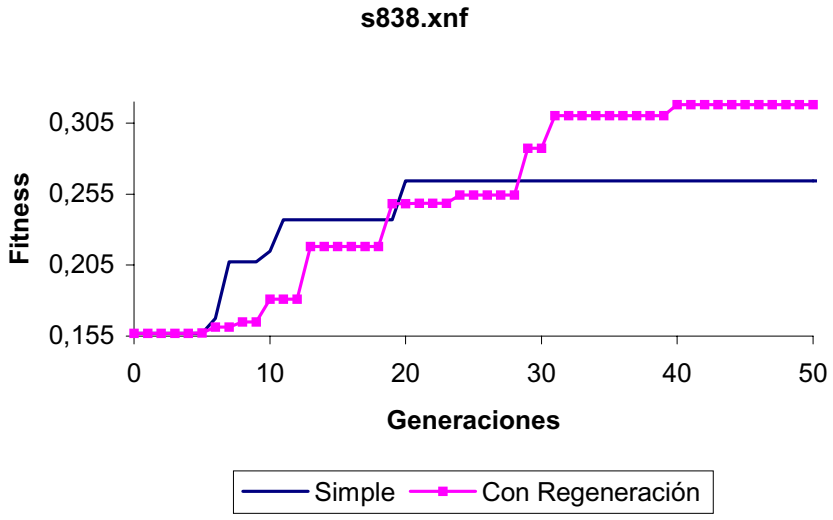

Figura 6.21: *Efecto del operador de Regeneración sobre la convergencia del algoritmo para el circuito s838.xnf*

Analizando por ejemplo la figura 6.21, comprobamos que el algoritmo cuando el operador de regeneración no actúa sufre una rápida convergencia antes de las 50 primeras generaciones. Sin embargo para el algoritmo que utiliza el operador, obtenemos un mejor valor para la función de coste y además el algoritmo tarda más en estabilizar dicho valor.

Los resultados para el resto de circuitos son muy similares, por lo que se demuestra que el operador propuesto es una técnica eficiente para salvar los problemas derivados de la convergencia prematura.

6.6.3 Resultados experimentales

El algoritmo se ha implementado en C y se ha ejecutado sobre un procesador $Pentium\ II$ a 450 MHz. Para realizar las pruebas se han utilizado los circuitos de prueba MCNC partitioning93 benchmarks en formato XNF. Como el número y características de los CLBs (Configurable Logic Blocks) dependen de la FPGA utilizada, hemos supuesto que cada bloque del circuito utilice un CLB independientemente de la puerta lógica que represente, y asignamos un vértice a cada CLB [82].

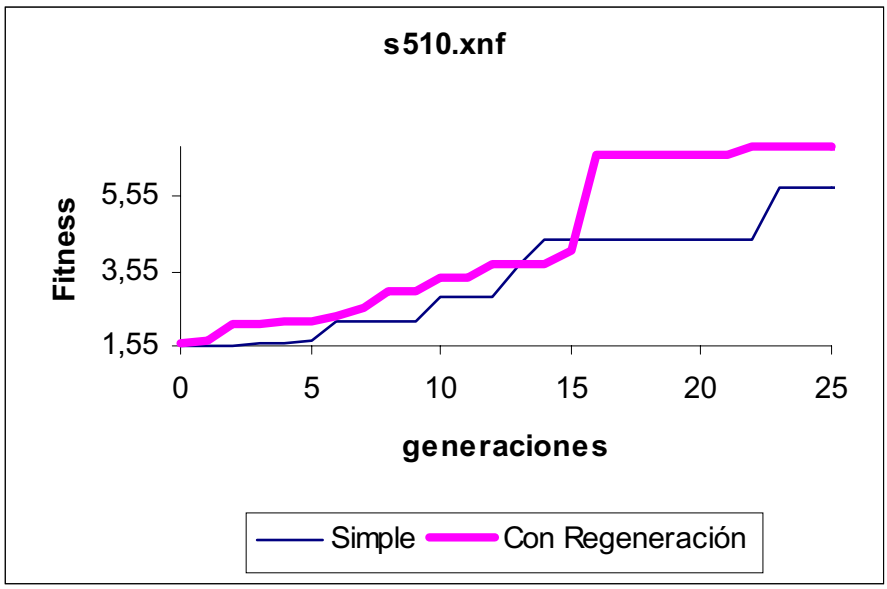

Figura 6.22: *Efecto del operador de Regeneración sobre la convergencia del algoritmo para el circuito s510.xnf*

Se han utilizado las características de la serie 4000 de Xilinx, que aparecen resumidas en la tabla 6.4.

FPGA	Max CLB	Max IOB
XC4003	100	30
XC4005	196	42
XC4008	324	54
XC4020	784	84
XC4025	1024	96
XC4036	1296	108

Tabla 6.4: *Características de las FPGAS de la serie Xilinx 4000*

La tabla 6.5 contiene los resultados experimentales. La tabla tiene 7 columnas:

- el nombre del circuito (Circuito),
- el número de vértices del grafo que representa al circuito (CLB)
- el número de aristas del mismo grafo (Aristas)
- el número de PE/S que faltan para implementar las solución (N_{IOL})

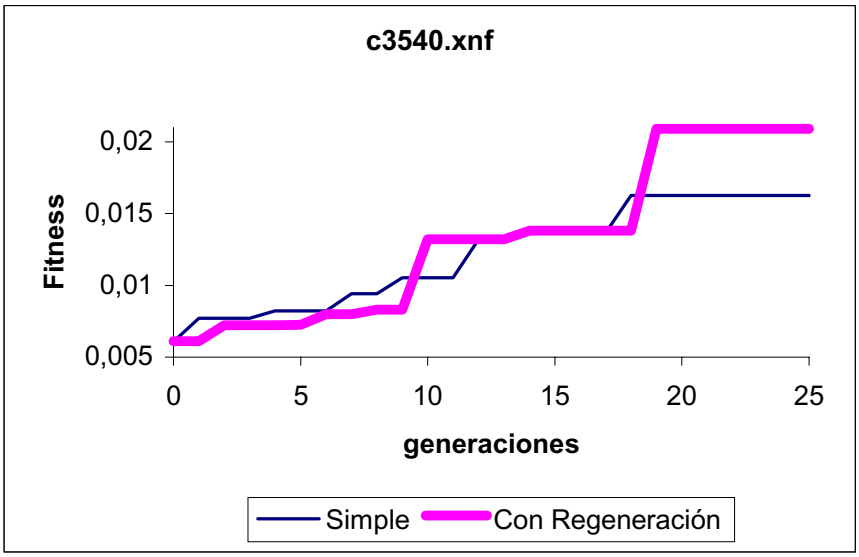

Figura 6.23: *Efecto del operador de Regeneración sobre la convergencia del algoritmo para el circuito c3540.xnf*

- el tipo de FPGA de las serie 4000 que se usará para la implementación (FPGA)
- la distribución de los CLBs sobre las FPGAs despúes de la partición (Distribución)
- el tiempo de CPU necesario para obtener la solución con 100 generaciones y para una población de 500 individuos (T (sec)).

Los resultados importantes son los relativos al número de PE/S y a la distribución. Para los circuitos más pequeños obtenemos una distribución homogénea y circuitos rutables, puesto que bajo las suposiciones de ubicación realizadas no faltarían PE/S para realizar todas las conexiones entre FPGAs. Para los circuitos de mayor tamaño, se obtienen resultados distintos a los anteriores. Hay algunos circuitos (c499, c2670, c7552) que están compuestos por más de un grafo aislado. En estos casos el algoritmo lo detecta y coloca cada uno de los subgrafos sobre una única FPGA distinta, a excepción del subgrafo más grande que es el que distribuye a lo largo de las otras FPGAs que han quedado libres, el algoritmo encuentra soluciones con un compromiso entre la distribución y la rutabilidad. En todos los casos se obtienen circuitos rutables.

Circuito	CLB	Aristas	N_{IOL}	FPGA	Distribución	T(sec)
S208	127	200	0	4003	16,14,20,20,12,13,24,8	34.82
S298	158	306	0	4003	4,23,22,10,15,15,23,26	48.81
S400	217	412	0	4005	50,53,26,27,15,11,27,8	79.74
S432	217	365	0	4005	43,44,39,37,19,11,14,10	78.21
S444	234	442	0	4005	44,30,43,45,16,22,32,2	88.53
S420	251	402	0	4005	46,40,42,34,36,17,27,9	102.53
S510	251	455	0	4005	42,46,26,32,27,17,34,27	106.88
C499	325	486	0	4008	92,62,107,58,1,1,2,2	456.21
S832	336	808	2	4008	68,75,25,42,33,31,27,35	163.06
S820	338	796	0	4008	119,34,72,21,19,50,21,2	168.00
S953	494	882	12	4008	111,43,87,61,58,75,49,10	343.44
S838	495	800	0	4008	100,118,54,38,68,29,49,37	343.63
S1238	574	1127	1	4008	65,135,68,84,56,40,26,100	442.16
C1423	829	1465	0	4025	500,65,49,45,69,5,42,54	970.32
C2670	924	1515	0	4020	14,6,8,138,219,184,208,147	1302.6
C3540	1038	2115	0	4020	635,204,91,23,9,59,13,4	1570.2
C5315	1778	3393	0	4036	1274,106,177,182,8,7,8,16	5680.7
C7552	2247	4031	0	4025	14,838,286,322,577,184,7,19	8517.2

Tabla 6.5: *Características e los circuitos utilizados y esultados experimentales*

El resto de los circuitos se traducen en grafos con una única componente conexa y se obtienen distribuciones de bloques lógicos bastante homogéneas incluso respetando el número de PE/S disponibles para casi todos los circuitos. En la figura 6.24 se pueden ver algunas de estas distribuciones representadas sobre un gráfico de barras. Solamente se han obtenido 3 circuitos para los que haría falta un mayor número de PE/S que los disponibles (s832, s953 y s1238). Sin embargo dado que faltan pocos PE/S (2, 12 y 1 respectivamente), es preferible buscar manualmente un rutado alternativo utilizando los posibles PE/S que hayan quedado sin ocupar.

En los resultados también podemos observar que hay algunas distribuciones que no son homogéneas. Esto es un resultado lógico y hasta cierto punto necesario. Ya que hay señales que tienen que atravesar varias FPGAs para conectar dos circuitos separados, es conveniente que algunas contengan un menor número de bloques lógicos para que las señales puedan pasar a través de ellas. La función de coste ha sido estudiada para que realice ambas funciones y por lo tanto el funcionamiento del algoritmo genético queda demostrado. La comparación con otras técnicas, como

la presentada en [112], no es posible debido a que prácticamente todos los trabajos previos están orientados a topologías libres y sólo optimizan el número de FPGAs utilizadas. Otras utilizan además circuitos antiguos de la serie 3000 y 2000.

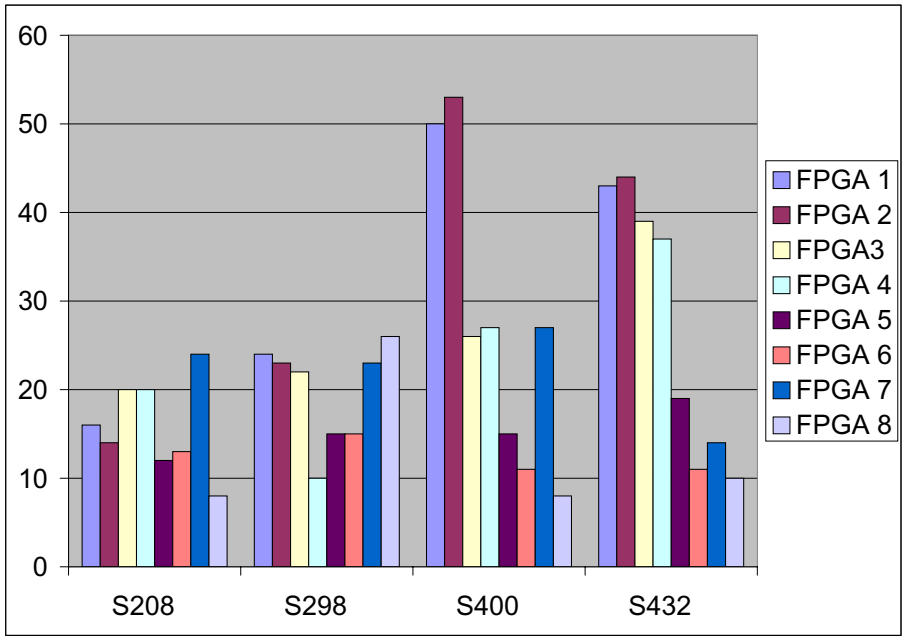

Figura 6.24: *Distribución final de los bloques lógicos sobre las FPGAs para distintos circuitos*

6.6.4 Conclusiones

En este apartado se ha presentado un nuevo método de partición y ubicación para SMFPGAs. Las conclusiones atañen tanto al aspecto algorítmico como a la partición.

En el aspecto algorítmico:

- Se ha presentado un nuevo algoritmo para la partición de grafos cuyo punto fuerte reside en la codificación del problema, lo que permite utilizar un algoritmo genético clásico para su resolución.

- La codificación presentada es perfectamente aplicable a cualquier problema de partición de grafos independientemente de cuál sea el objetivo del mismo, que vendrá particularizado por la función de coste que guía el algoritmo.

- Se ha resuelto el problema de convergencia prematura, que puede aparecer al tener un número de genes reducido, mediante el operador de regeneración. Este operador se activa automáticamente cuando se detecta que el mejor individuo se repite un número determinado de generaciones.

En lo referente a la partición de SMFPGAs:

- Se ha presentado un método de partición y ubicación de circuitos orientado a SMFPGAs de topología fija.
- El método respeta las restricciones de número de PE/S.
- La forma de evaluar tiene en cuenta el consumo de PE/S necesario para conectar dos elementos del circuito que estén en FPGAs no adyacentes y que por lo tanto consume PE/S al atravesar otras FPGAs para realizar la conexión.
- El método respeta la estructura inicial del circuito realizando particiones y ubicaciones que tienden a poner en la misma partición los elementos del circuito que están conectados entre sí.
- La ubicación no se realiza de forma aleatoria ya que tiene en cuenta la relación entre las distintas particiones. Mediante la utilización de la matriz de relación inexacta no solamente se tienen en cuenta las relaciones directas si no también las relaciones a través de otras FPGAs.
- La solución propuesta busca las partes del circuito que son independientes y las ubica en FPGAs independientes, disminuyendo de esta forma el consumo de pines.

Aunque el método de partición es perfectamente válido, el AG presenta unos tiempos de ejecución excesivos. El problema reside en la evaluación de los individuos como veremos en los apartados siguientes. Para obtener los valores de la funciones de coste hay que decodificar las soluciones, lo que conlleva un gran tiempo de cálculo. Para subsanar estos problemas, se deben probar otras técnicas aprovechando las formas de codificación y ubicación utilizadas. Por ello en el siguiente apartado investigamos que mejoras puede introducir un algoritmo genético compacto.

6.7 Algoritmo Genético Compacto

En el capítulo anterior hemos visto que el tiempo de cómputo necesario y la cantidad de memoria utilizada para representar la población y las estructuras de datos son dos aspectos importantes para la resolución de problemas de partición mediante el AG implementado. Los problemas de memoria se pueden solucionar parcialmente aprovechando las características de los algoritmos genéticos compactos. El tiempo se puede mediante la paralelización.

Además en problemas como el que estamos tratando, en los que la mayor parte del tiempo de cálculo se utiliza para la evaluación de la función de coste, el AGc también puede ayudar a reducirlo. De hecho para una población de 500 individuos, si se utiliza un AG simple se deben evaluar todos en cada generación. Sin embargo, si se utiliza un AGc solo es necesario calcular el valor de la función de coste para los individuos generados en cada iteración. Si la presión de selección es 10, el AGc sólo debe evaluar 10 individuos. Dado que el AGc representa la población de una forma compacta mediante un array, permite tratar problemas que de otra manera requeriría una mayor cantidad de memoria para la población, y por lo tanto permite solucionar problemas de partición para SMFPGAs más complejos.

6.7.1 Algoritmo Genético Compacto Simple

En [72, 73] se propuso un algoritmo genético compacto (AGc). En lugar de trabajar con la población (como un algoritmo genético simple), el AGc unicamente simula su existencia. Para ello, representa la población mediante un vector de probabilidades de valores $p_i \in [0,1], \forall i = 1, \ldots, l$, donde l es el número de elementos del alfabeto necesarios para representar las soluciones. Cada valor p_i indica la proporción de individuos en la poblacion simulada que tienen un cero (uno) en la posición i de su representación. Si estos valores se toman como probabilidades, se pueden generar nuevos individuos y actualizar el vector, favoreciendo a los mejores individuos para realizar este proceso.

Los valores de las probabilidades p_i, se fijan inicialmente a 0.5 para representar

una población generada aleatoriamente en el que el valor de cada alelo tiene la misma probabilidad de pertenecer a una solución. En cada iteración el AGc genera dos individuos, basándose en la representación elegida y compara los valores de sus funciones de coste. Llamamos W a la representación del individuo con mejor coste y L a la del peor. Las representaciones de los competidores se usan para actualizar el vector de probabilidad de la iteración k a la $k+1$ segun la ecuación 6.14:

$$p_i^{k+1} = \begin{cases} p_i^k + 1/n & \text{si } W_i = 1 \wedge L_i = 0 \\ p_i^k - 1/n & \text{si } W_i = 0 \wedge L_i = 1 \\ p_i^k & \text{si } W_i = L_i \end{cases} \tag{6.14}$$

siendo n la dimensión de la población simulada, y W_i (L_i) el valor del alelo *iesimo* de W (L). El AGc finaliza cuando todos los valores del vector son iguales a 0 ó a 1. En este momento el propio vector p, representa la solución final. Para representar n individuos, el AGc actualiza el vector de probabilidad mediante un valor constante igual a $1/n$. De esta forma, sólo hacen falta $\log_2 n$ bits para almacenar cada valor de p_i. Por lo tanto el AGc solo necesita $\log_2 n * l$ bits con respecto a los $n * l$ bits necesarios de un AG convencional. De esta forma se pueden explorar poblaciones de una mayor dimensión sin un aumento significativo de la memoria utilizada, aunque se ralentiza la convergencia del algoritmo. Por ello los AGcs pueden ser muy útiles en problemas en los que la dimensión de la población es una de las restricciones.

Veamos un ejemplo sencillo para 6 genes y una población de 10 individuos. Los valores iniciales del vector de probabilidad que representa la población serán:

$$P^0 = [0.5 \;\; 0.5 \;\; 0.5 \;\; 0.5 \;\; 0.5 \;\; 0.5]$$

Supongamos que generamos los dos siguientes individuos:

$$Individuo\ 1 = 0\ 1\ 0\ 1\ 0\ 0$$

$$Individuo\ 2 = 0\ 1\ 0\ 0\ 1\ 0$$

```
Program TSP_Cga
  begin
    Initialize(P,method);
    F_best := INT_MAX;
    repeat
       S[1] := Generate(P);
       F[1] := Tour_Length(S[1]);
       idx_best := 1;
       for k := 2 to s do
          S[k] := Generate(P);
          F[k] := Tour_Length(S[k]);
          if (F[k] < F[idx_best]) then idx_best := k;
       end for
       for k := 1 to s do
          if (F[idx_best] < F[k]) then Update(P,S[idx_best],S[i]);
       end for
       if (F[idx_best] < F_best) then
          count := 0;
          F_best := F[idx_best];
          S_best := S[idx_best];
       else
          Update(P,S_best,S[idx_best]);
          count := count + 1;
       end if
    until (Convergence(P) OR count > CONV_LIMIT)
    Output(S_best,F_best);
  end
```

Figura 6.25: *Pseudo-codigo del AG compacto para el TSP.*

y que sus valores de la función de coste son $F_1 = 1.7$ y $F_2 = 2.3$. Por lo que:

$$W = Individuo\ 2$$

$$L = Individuo\ 1$$

o lo que es lo mismo

$$L = 0\ 1\ 0\ 0\ 1\ 0$$

$$W = 0\ 1\ 0\ 1\ 0\ 0$$

Calculemos ahora los nuevos valores del vector P^1:

$p_1^1 = p_1^0$ porque $w_1 = l_1$.

Por el mismo motivo

$p_2^1 = p_2^0$, $p_3^1 = p_3^0$, y $p_6^1 = p_6^0$.

Sin embargo,

$p_4^1 = p_4^0 + 1/10 = 0.6$, ya que $w_4 = 1$ y $l_4 = 0$ y

$p_5^1 = p_5^0 - 1/10 = 0.4$, porque $w_4 = 0$ y $l_4 = 1$.

Finalmente el vector para la siguiente iteración será:

$P^1 = [0.5 \;\; 0.5 \;\; 0.5 \;\; 0.6 \;\; 0.4 \;\; 0.5]$.

EL algoritmo continuará trabajando hasta que todos los valores del vector sean iguales a 0 o a 1. Si después de 20 generaciones el algoritmo converge y los valores de las probabilidades son:

$$P^{20} = [0 \; 1 \; 1 \; 1 \; 0 \; 0]$$

quiere decir que la solución a nuestro problema es

$$0 \; 1 \; 1 \; 1 \; 0 \; 0$$

Para resolver problemas de mayor complejidad los AGs necesitan aumentar tanto las proporciones de selección cómo el tamaño de la población. La presión de selección del AGc se puede aumentar modificando el algoritmo de la siguiente forma:

- En cada iteración generar s individuos en lugar de dos.
- Elegir entre los s individuos el mejor y seleccionar su representación como W.
- Comparar W con los otros $s - 1$ individuos y actualizar las probabilidades.

Al diseñar un AGc para resolver la partición en SMFPGAs, hemos adoptado la representación basada en las aristas que se ha explicado anteriormente, es decir, un individuo viene representado por las aristas que se eliminan para obtener la partición (ver apartado 6.5).

Consideramos la frecuencia con que cada arista aparece en las soluciones para representar el vector de probabilidades. Para almacenar estos valores se usa un vector de dimensiones iguales al número de nodos menos uno (número de aristas que componen un árbol de expansión). Cada elemento v_i del vector v, representa la proporción de individuos cuya partición utiliza la arista e_i. Los elementos del vector se inicializan todos a un valor de 0.5 para simular una poblacion en la que todas las aristas tengan la misma probabilidad de aparecer.

Después de la fase de inicialización se generan los otros s individuos de acuerdo con la presión de selección establecida. Se evalúan y se obtiene el mejor (W). Este último se compara con el resto (L_i) para actualizar el vector de probabilidades. Además, W se compara con el mejor individuo encontrado hasta ese momento y se actualiza de nuevo los valores de P.

El AGc propuesto en [73] finaliza cuando todos los valores del vector de probabilidades son iguales a cero o a uno. Dado que esto no sucede en nuestro problema, se ha implementado una condición de finalización adicional, que consiste en limitar el numero de iteraciones sin que se produzca una mejora en la función de coste. Cuando se alcanza este límite la ejecución del programa finaliza y se obtiene el mejor individuo. El esquema del algoritmo está en la figura 6.26

```
leer_archivo(archivo,&datos);
sacar_aristas(datos,&aristas);
crear_arbol(aristas,&arbol);
ocupa_ideal=1.0*nbloques/fpgas;
n ⇐ presión de selección
genera_matriz(matriz)
while continuar do
  poblac ⇐ población
  for i = 1 to n do
    decodificacion(poblacion,arbol,particion);
    crea_distribucion(particion,&distribucion);
    bloques_distribucion(distribucion,n_clbs2,situacion);
    cuenta_aristas(situacion,aristas,&aristascorte,&fpines);
    fitness_function(aristascorte,fpines,&coste[0],n_clbs2);
  end for
  Obtiene_mejor(best)
  for i = 1 to n do
    Actualiza(matriz, mejor, poblac[n])
  end for
end while
Mostrar_Resultados
```

Figura 6.26: *Esquema del algoritmo genético compacto para SMFPGAs*

Para comprender mejor el funcionamiento del algoritmo veamos un ejemplo utilizando el mismo árbol de la codificación del AG simple (figura 6.18). Este árbol

tiene 12 nodos y 11 aristas, por lo tanto el vector de probabilidades P tendrá 11 elementos p_i:

$$P_0 = [0.50 \;\; 0.50 \;\; 0.50 \;\; 0.50 \;\; 0.50 \;\; 0.50 \;\; 0.50 \;\; 0.50 \;\; 0.50 \;\; 0.50 \;\; 0.50] \tag{6.15}$$

Los otros parámetros que hay que definir son:

- La presión de selección s
- El número de particiones o lo que es lo mismo de FPGAs, N_F
- El número de individuos de la población simulada, n

Para el ejemplo supongamos que $s = 3$ (por lo tanto se generan 3 individuos aleatoriamente en cada iteración), $N_F = 5$ (los cromosomas tienen 4 genes) y n=100 (con cada actualización del vector, los valores pueden variar 0.01 ó quedarse igual). Una iteración actuaría de la siguiente manera. Se generan aleatoriamente 3 individuos:

$$I_1 = 1 \;\; 2 \;\; 3 \;\; 6$$

$$I_2 = 4 \;\; 5 \;\; 6 \;\; 7$$

$$I_3 = 3 \;\; 4 \;\; 6 \;\; 9$$

si los valores de la función de coste asociados a cada uno son $FC_1 = 27$, $FC_2 = 13$ y $FC_3 = 18$ y es un problema de minimimización quiere decir que el mejor individuo W es I_2 y los otros dos, $I_1 = L_1$ e $I_3 = L_2$, se comparan con éste para actualizar el vector P. Recordando lo visto en el apartado 6.7 y modificando la ecuación 6.14 para una presión de selección distinta de uno, obtenemos la ecuación 6.16 que está adaptada a nuestro problema (k es el número de iteración):

$$p_i^{k+1} = \begin{cases} p_i^k + 0.01 & \text{si } W \supset i \wedge L_j \not\supset i \\ p_i^k - 0.01 & \text{si } L_j \supset i \wedge W \not\supset i \\ p_i^k & \text{en otro caso} \end{cases} \tag{6.16}$$

Calculemos los nuevos valores de P al comparar W y L_1:

- $p_1^1 = p_1^0 - 0.01 = 0.49$ porque la arista 1 aparece en L_1 y no en W
- $p_2^1 = 0.49$ y $p_3^1 = 0.49$ por el mismo motivo
- las aristas 4, 5 y 7 no aparecen representadas en el individuo L_1 pero sí en W, por lo tanto $p_i^1 = p_i^0 + 0.01$ para $i \in \{4,\ 5,\ 7\}$.
- $p_6^1 = p_6^0$ porque la arista 6 aparece tanto en W como en L_1.
- El resto de los valores no varían puesto que las aristas 0, 8, 9 y 10 no aparecen en ninguno de los dos individuos comparados.

Después de la primera iteración el vector P quedaría como indica la ecuación 6.17:

$$P_0 = [0.50\ \ 0.49\ \ 0.49\ \ 0.49\ \ 0.51\ \ 0.51\ \ 0.50\ \ 0.51\ \ 0.50\ \ 0.50\ \ 0.50] \qquad (6.17)$$

Con el individuo L_2 y W se obtendría una nueva actualización del vector P, siguiendo el mismo proceso que el utilizado con L_1.

Como se ha dicho, para nuestro problema el AGc no converge nunca con todos los valores a 0 o a 1. El mejor individuo obtenido se va guardando en cada iteración. Si después de 500 iteraciones el algoritmo no ha obtenido una solución que mejore a éste, el programa se detiene. Para obtener la solución indicada por el vector se escogen los $N_F - 1$ mayores valores de p_i. Siguiendo con nuestro ejemplo si el proceso termina con el vector de probabilidades:

$$P_0 = [0.21\ \ 0.02\ \ 0.59\ \ 0.86\ \ 0.95\ \ 0.35\ \ 0.78\ \ 0.98\ \ 0.44\ \ 0.24\ \ 0.45]$$

la solución sería:

$$sol = (3\ \ 4\ \ 6\ \ 7)$$

siempre y cuando el valor de la función de coste de la partición que representa sea inferior al del mejor individuo (recordemos que es un proceso de minimización).

6.7.2 Resultados experimentales

Para evaluar el AGc se han realizado varias pruebas con circuitos pertencientes al Partitioning 93 benchmarks utilizando un Pentium II a 450 MHz sobre Linux . Los circuitos de entrada están descritos en formato XNF. Recordamos que, dado que las características de los CLBs dependen del dispositivo utilizado para la implementación, supondremos que cada bloque del circuito utiliza un CLB. Para establecer las condiciones de ubicación se utilizan de nuevo las características de las FPGA de Xilinx pertenecientes a la serie 4000 [83, 88].

Las tablas 6.6 y 6.7 muestran los resultados experimentales del AGc. Estás tablas contienen en la columna *Tiempo* los tiempos de ejecución al ejecutar nuestro AGc y el AG simple propuesto en el apartado anterior [86]. Las tablas tienen 5 columnas, la primera contiene los nombres de los circuitos (Circuito), la segunda recoge los tiempos de ejecución mencionados (Tiempo). Las otras tres muestran los resultados de la partición. La columna Distribución indica el número de CLBs que se implementarán en cada una de las FPGAs. N_{IOL} expresa el número de pines de entrada-salida que faltan para poder realizar la implementación y FPGA el tipo de dispositivo de la serie 4000 de Xilinx. Utilizamos estos dos parámetros para expresar la calidad de una solución. Se buscan distribuciones lo más homogeneas posible de los bloques de tal forma que no falten pines, es decir que sean circuitos rutables.

De los resultados se puede ver que el AGc necesita casi en todas las ocasiones, menos tiempo que el AG para obtener soluciones de una calidad equivalente a la obtenida por éste. El AGc permite el estudio de circuitos más complejos, cómo ejemplo se muestran los resultados obtenidos con el circuito c6288 que con el AG simple no se pudo resolver. Se han realizado también pruebas con otros circuitos de tamaño superior y con resultados equivalente similares.

En las figuras 6.27 y 6.28 se comparan gráficamente los tiempos de ejecución de ambos algoritmos. Las figuras comparan los tiempos uno a uno para cada circuito de prueba y dan una visión general de cómo se ha reducido el tiempo.

Aunque los resultados obtenidos por el algoritmo genético compacto dan dis-

Circuito	Tiempo	Distribución	N_{IOL}	FPGA
S208				
sAG	34.82	16,14,20,20,12,13,24,8	0	4003
cAG	19.66	16,15,18,20,12,10,31,5	0	4003
S298				
sAG	48.81	24,23,22,10,15,15,23,26	0	4003
cAG	68.54	34,29,29,25,6,24,5,6	0	4003
S400				
sAG	79.74	50,53,26,27,15,11,27,8	0	4005
cAG	45.73	50,32,26,39,21,10,38,1	0	4005
S432				
sAG	78.21	43,44,39,37,19,11,14,10	0	4005
cAG	71.77	48,39,51,26,24,2,1,26	0	4003
S444				
sAG	88.53	44,30,43,45,16,22,32,2	0	4005
cAG	67.44	21,73,55,3,32,32,14,4	0	4005
S420				
sAG	102.53	46,40,42,34,36,17,27,9	0	4005
cAG	91.555	61,75,15,24,35,7,18,16	0	4005
S510				
sAG	106.88	42,46,26,32,27,17,34,27	0	4005
cAG	93.32	48,31,48,34,46,31,5,8	0	4003
S832				
sAG	163.06	68,75,25,42,33,31,27,35	0	4008
cAG	104.26	104,51,58,6,68,43,5,1	0	4008

Tabla 6.6: *Resultados experimentales comparados para el AG y el AGC (I)*

tribuciones de una calidad similar al AG y en ocasiones mejora la rutabilidad, si se analizan las distribuciones se observa que las obtenidas por el AG simple ofrecen una disposición de los bloques más homógenea sobre las FPGAs. Dada la función de coste utilizada esto quiere decir que las soluciones son de una calidad inferior, pues recordemos que teníamos dos objetivos (una distribución homogenea y utilización de pines). Esto se puede solucionar usando un mecanismo de búsqueda local para mejorar las soluciones como se ha demostrado en [11] para otros problemas. La razón fundamental es que el AGc realiza una búsqueda menos exhaustiva que el AG simple, puesto que en cada iteración trabaja con un número mucho menor de individuos que el AG simple.

Circuito	Tiempo	Distribución	N_{IOL}	FPGA
S820				
sAG	168.00	119,34,72,21,19,50,21,2	0	4008
cAG	145.80	244,25,24,26,10,6,2,1	0	4008
S953				
sAG	343.44	111,43,87,61,58,75,49,10	0	4008
cAG	300.20	104,20,85,126,52,80,26,1	0	4008
S838				
sAG	343.44	100,118,54,38,68,29,49,37	0	4008
cAG	289.35	222,20,21,32,5,7,28,1	0	4008
S1238				
sAG	442.16	65,135,68,84,56,40,26,100	1	4008
cAG	493.34	110,343,74,19,8,9,2,9	0	4008
C1423				
sAG	970.32	500,65,49,45,69,5,42,54	0	4025
cAG	853.26	615,1,49,20,99,1,13,31	0	4020
C2670				
sAG	1302.6	14,6,8,138,219,184,208,147	0	4020
cAG	855.78	14,6,8,124,360,189,155,68	0	4020
C3540				
sAG	1570.2	635,204,91,23,59,9,13,4	0	4020
cAG	1334.3	842,17,4,45,56,71,2,1	0	4020
C7552				
sAG	8517.2	14,838,286,322,577,184,7,19	0	4025
cAG	7440.1	14,769,4,684,750,7,19,13	0	4025
C6288				
sAG	——	————————	0	—-
cAG	21680	1183,1424,173,49,2,13,11,1	0	4036

Tabla 6.7: *Resultados experimentales comparados para el AG y el AGC (II)*

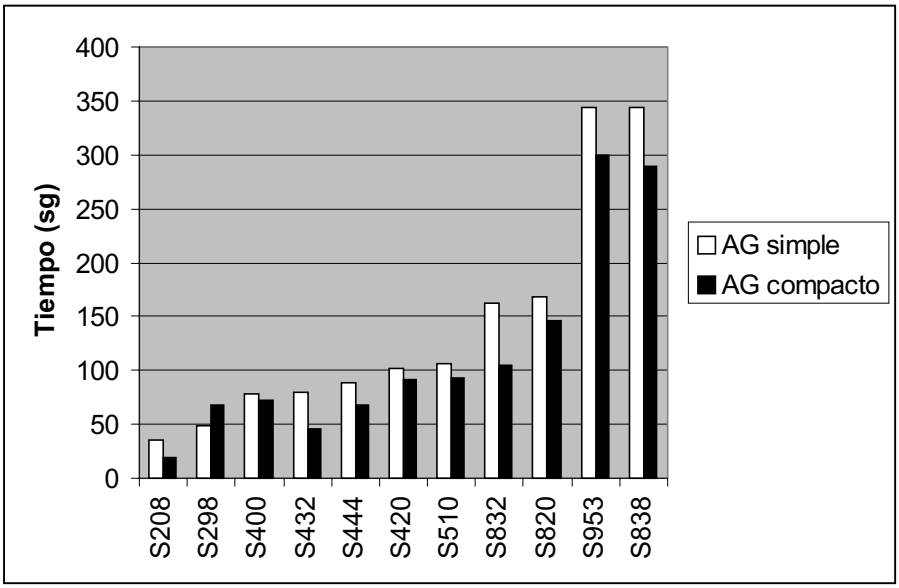

Figura 6.27: *Comparación de los tiempos de ejecución del AG y del AGc*

6.7.3 Conclusiones

En este apartado se ha propuesto un algoritmo genético compacto para resolver el problema de la partición en SMFPGAs de topología fija. Las principales conclusiones que se han obtenido son las siguientes.

- Dado que el AGc no trabaja con un población, si no que solamente simula su existencia, se produce un considerable ahorro de memoria para representarla.
- En problemas como el de la partición de SMFPGAs, en los que la complejidad de la función de coste hacen de la evaluación la parte mas costosa en tiempo de cálculo, el AGc permite reducir también los tiempos de ejecución.

En definitiva, los resultados obtenidos muestran que los algoritmos genéticos compactos son capaces de resolver satisfactoriamente el problema de la partición en SMFPGAs. Pese a que los resultados en lo referente a los tiempos de ejecución son satisfactorios, la calidad de las soluciones es ligeramente inferior que las obtenidas con el AG simple. El AGc se puede mejorar con técnicas de búsqueda local como se demuestra el siguiente apartado.

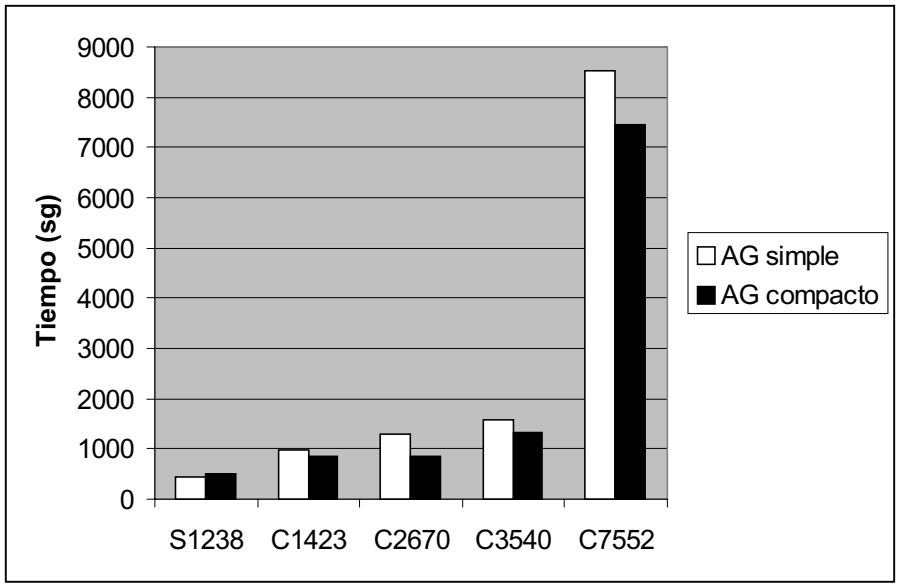

Figura 6.28: *Comparación de los tiempos de ejecución del AG y del AGc*

6.8 Algoritmo Genético Compacto Híbrido

En ocasiones no basta con aumentar la presión de selección para obtener buenos resultados con los algoritmos genéticos compactos. Para subsanar estos problemas se pueden combinar con técnicas de búsqueda local [100]. A este tipo de algoritmos se conocen con el nombre de híbridos.

En [11, 10] presentamos un AGc híbrido que utiliza una búsqueda local con el algoritmo de Lin-Kenighan (LK) [125, 126]. LK es un algoritmo de búsqueda iterativa muy similar en cuanto a su funcionamiento al algoritmo KL presentado en el capítulo 2 y desarrollado por los mismos autores. El algoritmo ha demostrado su efectividad para el problema del TSP (o del viajero) [12, 22, 43]. El AGc se utiliza para explorar las partes del espacio de búsqueda más interesantes, de tal forma que se alcancen buenas soluciones iniciales. Estas soluciones se van refinando mediante el algoritmo LK.

En la figura 6.29 se presenta el esquema general de este algoritmo en forma de pseudocódigo. Cómo se puede ver el funcionamiento es el mismo que un AGc con una determinada presión de selección (que determinará el tamaño de la población),

pero se incluye un paso de optimización con LK mediante la función *Aplicar_LK*(). Para el problema de la partición utilizaremos una modificación del mismo, aunque no utilizaremos el algoritmo LK como herramienta de búsqueda local por su alto coste computacional y la dificultad para adaptarlo a este problema.

```
Program TSP_CGa+LK
  begin
    Inicializa(P);
    Genera(initial_population);
    Aplicar_LK(initial_population,optimized_population);
    Actualizar(P,optimized_population)
    F_best := INT_MAX;
    repeat
       S[1] := Generate(P);
       F[1] := Tour_Length(S[1]);
       Aplicar_LK(S[1],S[LK_Opt];
       if (F[LK_Opt] < F[1]) then
          idx_best := LK_Opt;
       else
          idx_best := 1;
       end if
       if (F[idx_best] < F[1]) then Update(P,S[idx_best],S[1]);
       if (F[idx_best] < F_best) then
          count := 0;
          F_best := F[idx_best];
          S_best := S[idx_best];
       else
          Update(P,S_best,S[idx_best]);
          count := count + 1;
       end if
    until (Convergence(P) OR count > CONV_LIMIT)
    Output(S_best,F_best);
  end
```

Figura 6.29: *Pseudocódigo del* cGAKL *para el TSP.*

Para mejorar los resultados obtenidos con el algoritmo genético compacto en SMFPGAs, se ha incorporado una búsqueda local al mismo. La mayoría de las técnicas que utilizan una búsqueda local para el refinamiento de soluciones, hacen una exploración lo más exhaustiva posible en todas las soluciones vecinas a la mejor encontrada. Sin embargo puede que esto no sea posible debido al excesivo costo computacional que supondría y se toman otras alternativas más económicas en términos de tiempo de ejecución.

```
leer_archivo(archivo,&datos);
sacar_aristas(datos,&aristas);
crear_arbol(aristas,&arbol);
ocupa_ideal=1.0*nbloques/fpgas;
n ⇐ presión de selección
f ⇐ frecuencia de búsqueda local
genera_matriz(matriz)
cont ⇐ 0
while continuar do
  poblac ⇐ población
  for i = 1 to n do
    decodificacion(poblacion,arbol,particion);
    crea_distribucion(particion,&distribucion);
    bloques_distribucion(distribucion,n_clbs2,situacion);
    cuenta_aristas(situacion,aristas,&aristascorte,&fpines);
    fitness_function(aristascorte,fpines,&coste[0],n_clbs2);
  end for
  Obtiene_mejor(best)
  for i = 1 to n do
    Actualiza(matriz, mejor, poblac[n])
  end for
  if cont MOD f = 1 then
    Busqueda_local(mejor, arbol, new)
  end if
  cont++
end while
Mostrar_Resultados
```

Figura 6.30: *Esquema del algoritmo genético compacto híbrido para SMFGAs*

En nuestro caso, lo ideal sería examinar todas las posibilidades cada cierto número de generaciones utilizando el mejor individuo encontrado hasta ese momento. Dado que la decodificación de las soluciones y su evaluación son muy costosas computacionalmente hablando, se ha buscado una alternativa que no sea exhaustiva. La figura 6.30 recoge el esquema del algoritmo genético compacto con búsqueda local para el problema de la partición y ubicación de circuitos en SMFPGA.

El algoritmo realiza una mejora local cada cierto número de iteraciones del algoritmo genético compacto. Al igual que sucede con los algoritmos géneticos paralelos se debe buscar una frecuencia de mejora local que no distorsione el funcionamiento intrínseco del AGc. En nuestro problema se ha comprobado experimentalmente que esta mejora no se debe llevar a cabo con una frecuencia superior a 20 iteraciones para obtener buenos resultados.

La búsqueda local realiza una sustitución de las aristas de la mejor solución por otras de una manera aleatoria. Cuando se realiza la mejora local se determina en primer lugar la mejor solución obtenida hasta ese momento (recordamos que esta solución contiene las aristas eliminadas del árbol de expansión para obtener una partición, en nuestro caso se seleccionan 7 aristas). Para obtener el mejor individuo se elije entre la mejor opción encontrada hasta esa iteración y la solución que se puede obtener del vector de probabilidades siguiendo el proceso explicado en el apartado anterior (sección 6.7).

A continuación se sustituye la primera arista (primer gen) por otra del árbol que no esté contenida en la solución a mejorar. Se evalua la nueva solución y si su valor de la función de coste es mejor se acepta el cambio, en caso contrario se rechaza. Este proceso se realiza con cada una de las aristas que pertenecen a la solución. De esta forma se utilizan las mejores soluciones obtenidas por el AGc para llegar a particiones de mayor calidad sin incrementar dramáticamente el tiempo de ejecución.

Veamos un ejemplo con el mismo grafo que se ha utilizado anteriormente para este propósito. En el problema de 5 FPGAs de la sección 6.2.3 habíamos llegado a una solución representada por:

$$sol = (3 \quad 4 \quad 6 \quad 7)$$

Esta solución elimina las aristas 3, 4, 6 y 7 del árbol para obtener las partes y por lo tanto (es un árbol con 11 aristas) para realizar la búsqueda local sobre este individuo disponemos del siguiente conjunto de aristas A_{BL}:

$$A_{BL} = \{0, \quad 1, \quad 2, \quad 5, \quad 8, \quad 9, \quad 10\}$$

De este grupo se selecciona aleatoriamente un elemento y se sustituye por el primer gen de *sol*:

$$sol2 = (0 \quad 4 \quad 6 \quad 7)$$

suponiendo que el valor de la función de coste para sol2 es mejor que el de sol:

$$sol = sol2$$

Con el nuevo individuo se realiza el mismo proceso para el resto de genes que componen el cromosoma. Ahora tomaríamos el segundo gen (4) y los sustituiríamos por otro de A_{BL} del que hemos eliminado el 0 e incorporado el 3.

6.8.1 Resultados experimentales

Para comprobar la efectividad del AGc con mejora local se han realizado experimentos con los mismos circuitos utilizados para evaluar el AGs y el AGc simple. Los resultados se encuentran en las tablas 6.8 y 6.9. En ellas se ha introducido un nuevo parámetro, ECM, que permite evaluar la distribución de los CLBs sobre las FPGAs. Este parámetro es el error cuadrático medio de las distribuciones obtenidas por cada algoritmo con respecto a una distribución con el mismo número de bloques lógicos en cada FPGA.

Como se puede ver tanto en las citadas tablas como en las figuras 6.31 y 6.32 las distribuciones obtenidas por el AGc con mejora local obtiene distribuciones con un menor error cuadrático que el AGs y que el AGc y por lo tanto particiones más equilibradas. Dichas soluciones respetan las restricciones de la tarjeta.

Aunque en un principio se podría pensar que el AGc híbrido representaría un mayor coste computacional, se puede ver también en la tabla 6.9, en la tabla 6.8, en la figura 6.33 y en la figura 6.34, que el nuevo algoritmo tarda menos tiempo en obtener una solución de una calidad igual o mejor que los explicados en los dos apartados anteriores. Esto se debe a que la búqueda local acelera la convergencia del algoritmo.

Circuito	Tiempo	Distribución	ECM	N_{IOL}	FPGA
S208					
sAG	34.82	16,14,20,20,12,13,24,8	5.84	0	4003
cAG	19.66	16,15,18,20,12,10,31,5	7.78	0	4003
hAG	20.7	16,15,17,21,11,11,19,17	4.06	0	4003
S298					
sAG	48.81	24,23,22,10,15,15,23,26	6.62	0	4003
cAG	68.54	34,29,29,25,6,24,5,6	10.10	0	4003
hAG	45.92	20,23,19,20,20,20,25,11	7.25	0	4003
S400					
sAG	79.74	50,53,26,27,15,11,27,8	14.77	0	4005
cAG	45.73	50,32,26,39,21,10,38,1	15.33	0	4005
hAG	52.96	28,47,37,23,16,20,33,13	10.76	0	4005
S432					
sAG	78.21	43,44,39,37,19,11,14,10	78.21	0	4005
cAG	71.77	48,39,51,26,24,2,1,26	71.77	0	4003
hAG	69.51	36,42,25,13,39,23,28,11	69.51	0	4003
S444					
sAG	88.53	44,30,43,45,16,22,32,2	13.92	0	4005
cAG	67.44	21,73,55,3,32,32,14,4	17.30	0	4005
hAG	52.99	27,38,36,29,41,37,16,27	16.08	0	4005
S420					
sAG	102.53	46,40,42,34,36,17,27,9	14.05	0	4005
cAG	91.555	61,75,15,24,35,7,18,16	24.97	0	4005
hAG	24.25	34,41,38,42,32,19,24,21	14.27	0	4005
S510					
sAG	106.88	42,46,26,32,27,17,34,27	16.10	0	4005
cAG	93.32	48,31,48,34,46,31,5,8	24.19	0	4003
hAG	96.74	34,41,38,42,32,19,24,2	12.72	0	4005

Tabla 6.8: *Resultados experimentales AGc con búsqueda local, comparado con el AG y el AGc (I)*

Inst.	Tiempo	Dist	ECM	N_{IOL}	Dev
S832					
sAG	163.06	68,75,25,42,33,31,27,35	11.18	0	4008
cAG	104.26	104,51,58,6,68,43,5,1	21.14	0	4008
hAG	96.74	230,11,25,14,17,18,16,5	12.72	0	4008
S820					
sAG	168.00	119,34,72,21,19,50,21,2	19.92	0	4008
cAG	145.80	244,25,24,26,10,6,2,1	39.44	0	4008
hAG	138.87	237,17,24,14,21,8,7,10	70.39	0	4008
S953					
sAG	343.44	111,43,87,61,58,75,49,10	34.51	0	4008
cAG	300.20	104,20,85,126,52,80,26,1	75.00	0	4008
hAG	194.69	168,60,82,31,101,289,15	72.78	0	4008
S838					
sAG	343.44	100,118,54,38,68,29,49,37	32.82	0	4008
cAG	289.35	222,20,21,32,5,7,28,1	43.19	0	4008
hAG	293.42	100,92,37,77,67,60,43,17	58.22	0	4008
S1238					
sAG	442.16	65,135,68,84,56,40,26,100	35.64	1	4008
cAG	493.34	110,343,74,19,8,9,2,9	65.98	0	4008
hAG	320.15	91,11,293,56,50,25,17,31	33.76	0	4008
C1423					
sAG	970.32	500,65,49,45,69,5,42,54	34.82	0	4025
cAG	853.26	615,1,49,20,99,1,13,31	104.09	0	4020
hAG	273.68	525,52,114,14,37,51,28,8	86.68	0	4020
C3540					
sAG	1570.2	635,204,91,23,59,9,13,4	195.72	0	4020
cAG	1334.3	842,17,4,45,56,71,2,1	268.82	0	4020
hAG	844.13	614,135,88,89,56,26,28,2	182.81	0	4020

Tabla 6.9: *Resultados experimentales AGc con búsqueda local, comparado con el AG y el AGc (II)*

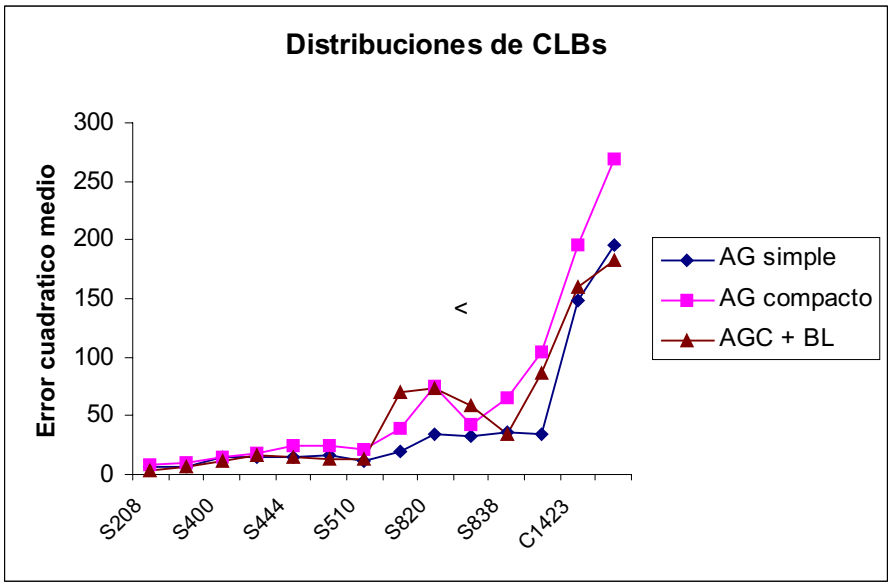

Figura 6.31: *Comparación del ECM de las soluciones obtenidas por el AGc, el AGc con búsqueda local y el AG simple (I)*

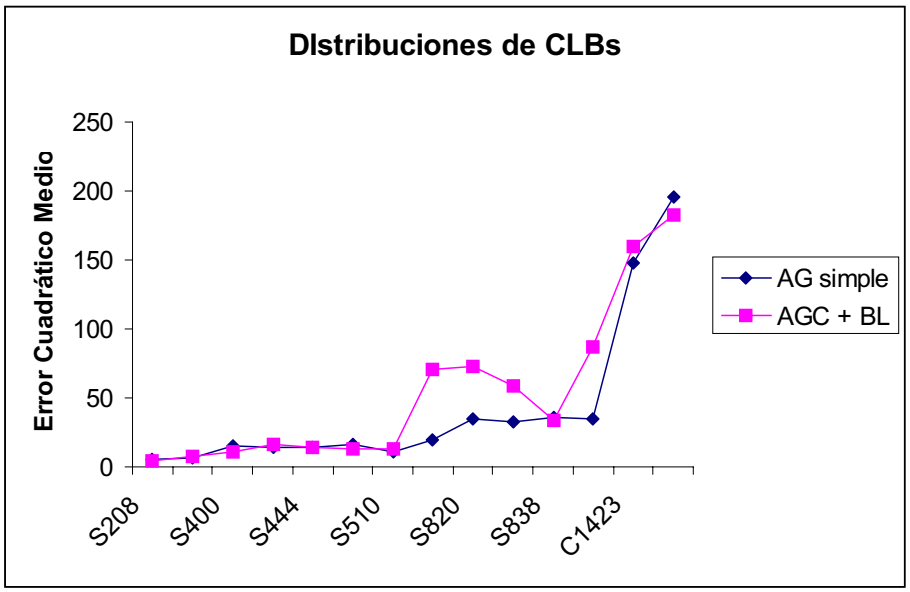

Figura 6.32: *Comparación del ECM de las soluciones obtenidas por el AGc con búsqueda local y el AG simple (II)*

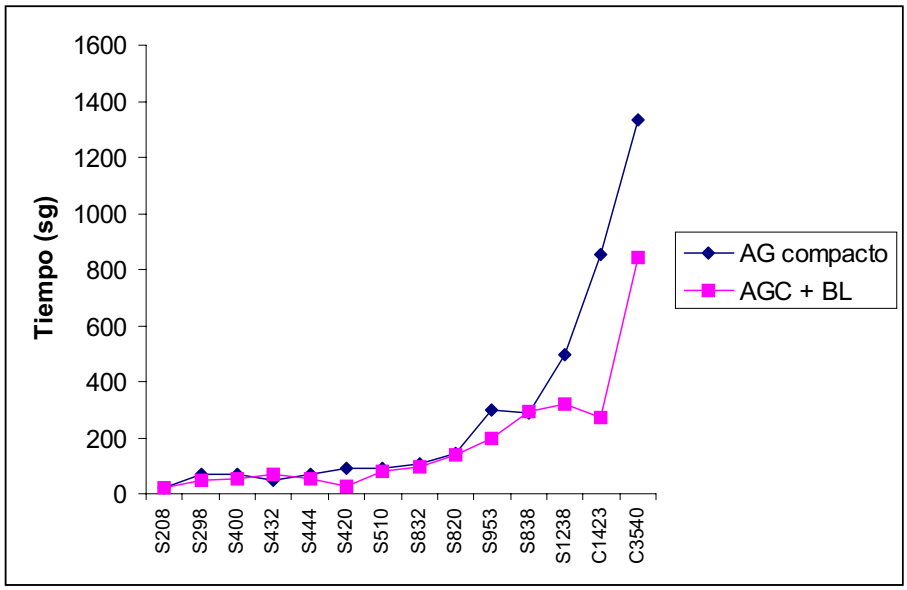

Figura 6.33: *Comparación de los tiempos de ejecución entre el AG y el AGc con mejora local para distintos benchmarks*

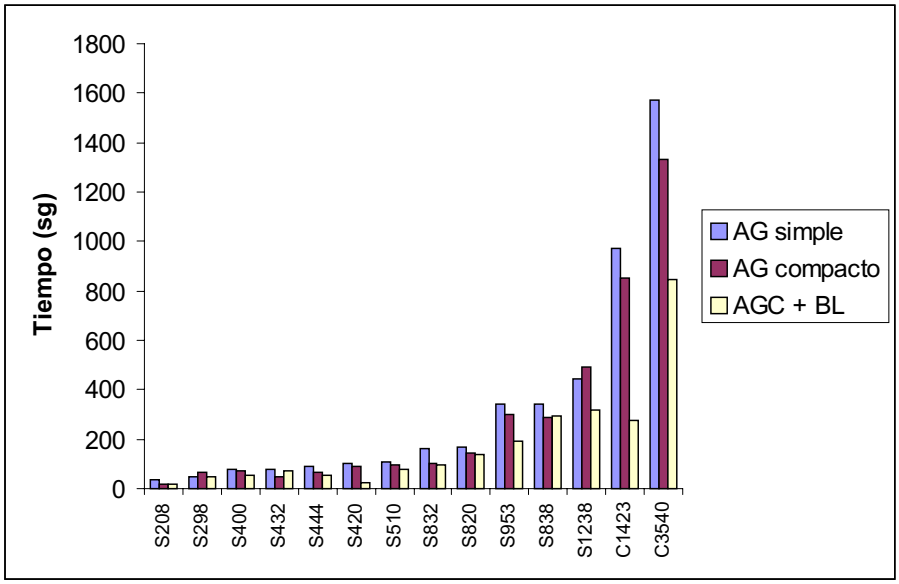

Figura 6.34: *Comparación de los tiempos de ejecución entre el AG, el AGc y el AGc con mejora local para distintos benchmarks*

6.8.2 Conclusiones

En este apartado se ha presentado un algoritmo genético híbrido. Mediante la implementación de una búsqueda local adicional al AGc se han conseguido mejoras tanto en los tiempos de ejecución como en la calidad de las particiones obtenidas, respetando en todo momento las restricciones impuestas por un SMFPGA de topología fija, compuesto por una malla de 8 FPGAs.

6.9 Implementación paralela

Con el objetivo de reducir los tiempos de ejecución de los algoritmos descritos en este capítulo se ha realizado una implementación paralela de los mismos. Para la implementación de los AGs paralelos se ha usado el modelo de paralelización global utilizando la libreria de paso de mensajes MPI.

Recordemos que existen varias formas de paralelizar un algoritmo genético. La primera y más intuitiva es la global que consiste en paralelizar la evaluación de los individuos manteniendo una población. Otra forma de paralelización global consiste en realizar una ejecución de distintos AGs secuenciales simultáneamente y es lo que hemos denominado paralelización de todo el programa. El resto de aproximaciones dividen la población en subpoblaciones que evolucionan por separado e intercambian individuos cada cierto número de generaciones. Se han realizado implementaciones de los dos tipos de paralelización global descritos.

6.9.1 Paralelización de todo el programa

Dado que cada árbol de expansión genera una búsqueda en un espacio diferente, es necesario ejecutar el algoritmo varias veces para obtener una buena solución. Esto sucede en la mayoría de las técnicas heurísticas. Este problema se puede resolver mediante la paralelización. En la figura 6.35 se ve de forma gráfica la forma de trabajo de éste modelo.

La estructura sencilla de los AGs secuenciales permiten una fácil utilización del paralelismo mediante un modelo *task farm* que se ha implementado adoptando un paradigma de programación $SPMD$. De acuerdo a este paradigma todos los procesadores ejecutan el mismo programa y uno de los procesos actúa como recolector (*farmer*) de los resultados del resto de procesadores y proporciona la solución final.

Al diseñar un programa paralelo hay que tener en cuenta dos aspectos fundamentales: la posibilidad de obtener siempre la misma solución utilizando reproducción de las ejecuciones en paralelo y el equilibrio de la carga de trabajo entre los proce-

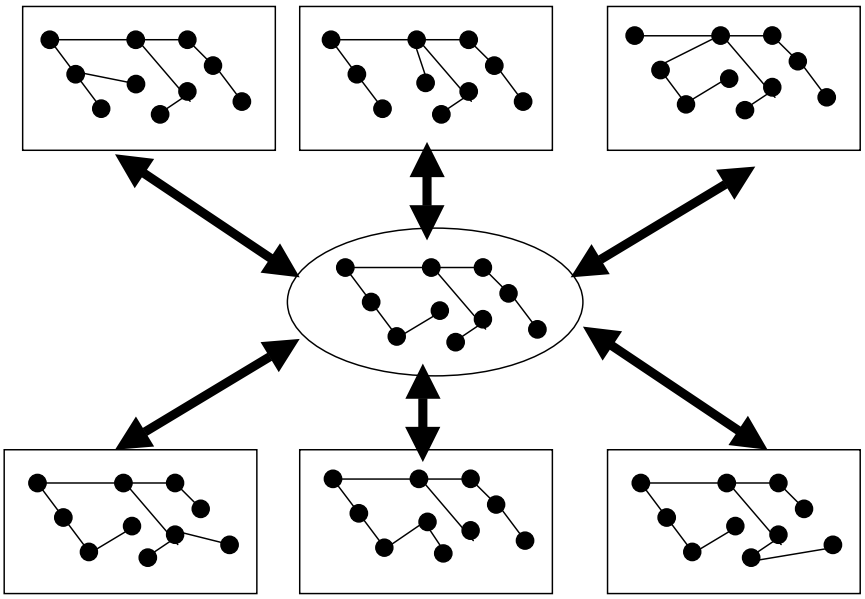

Figura 6.35: *Esquema de la paralelización. Cada procesador trabaja con un árbol de expansión diferente*

sadores. Para garantizar la primera, después de la fase de inicialización MPI cada proceso establece un número pseudo-aleatorio añadiendo el número de proceso a la semilla generada inicialmente. Al igual que sucedía con la versión secuencial, cada proceso elabora un árbol de expansión utilizando el número pseudoaleatorio generado y trabaja con él en busca de soluciones. Por lo que se refiere al equilibrio de carga, al trabajar con un mismo circuito el cálculo de cada árbol de expansión requiere el mismo tiempo de cálculo. Por lo tanto, este tipo de cálculo tiene la ventaja de que la carga de trabajo queda equilibrada de una forma natural. El algoritmo que se ha paralelizado es el AGc con búsqueda local, ya que es el que mejores resultados ha obtenido (figura 6.36).

Las medidas más comunes del rendimiento de las implementaciones paralelas son la acelearación (speed-up) y la eficiencia [130]. El speed-up alcanzado por un algoritmo paralelo ejecutado en P procesadores se define como el cociente entre el tiempo de ejecución del algoritmo paralelo en un único procesador y el tiempo de ejecución del mismo algoritmo paralelo ejecutado sobre P procesadores.La eficiencia se define como la aceleración dividida por el número de procesadores P. Estas medi-

```
leer_archivo(archivo,&datos);
sacar_aristas(datos,&aristas);
for j = 1 to procesos do
  crear_arbol(aristas,&arbol);
  ocupa_ideal=1.0*nbloques/fpgas;
  n ⇐ presión de selección
  f ⇐ frecuencia de búsqueda local
  genera_matriz(matriz)
  cont ⇐ 0
  while continuar do
    poblac ⇐ población
    for i = 1 to n do
      decodificacion(poblacion,arbol,particion);
      crea_distribucion(particion,&distribucion);
      bloques_distribucion(distribucion,n_clbs2,situacion);
      cuenta_aristas(situacion,aristas,&aristascorte,&fpines);
      fitness_function(aristascorte,fpines,&coste[0],n_clbs2);
    end for
    Obtiene_mejor(best)
    for i = 1 to n do
      Actualiza(matriz, mejor, poblac[n])
    end for
    if cont MOD f = 1 then
      Busqueda_local(mejor, arbol, new)
    end if
    cont++
  end while
end for
Mostrar_Resultados
```

Figura 6.36: *Esquema del algoritmo genético compacto híbrido*

das sólo dan información de como ha ido la paralelización pero no de la calidad de las soluciones.

La figura 6.37 muestra los valores medios del speed-up calculados para los distintos circuitos. Cómo era de esperar, se observa que los valores del speed-up son siempre muy próximos al ideal (relación lineal), ya que a este nivel de paralelización el problema tratado es totalmente paralelo.

6.9.2 Paralelización de la función de coste

En los algoritmos genéticos se puede disminuir notablemente el tiempo de ejecución paralelizando alguna de las partes del programa. Intituivamente podemos pensar que lo mejor es paralelizar el cálculo de la función de coste, ya que suele ser la parte

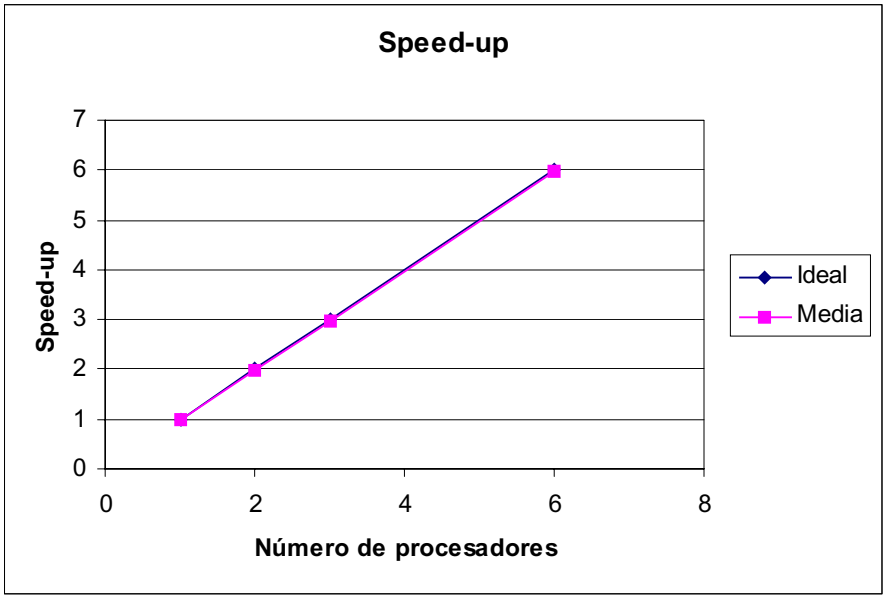

Figura 6.37: *Valores medios del speed-up para los distintos circuitos de prueba*

que más tiempo consume. Para comprobarlo se ha utilizado la herramienta Developer Magic de Silicon Graphics [98]. Esta herramienta permite estudiar el tiempo de ejecución de un programa, indicando que proporción de tiempo se utiliza en cada función. En nuestro caso la función de coste consume el 83% del tiempo total del programa. Resulta evidente por lo tanto lo ventajoso de paralelizarla.

A la hora de hacer la implementación se han tomado las siguientes decisiones:

- Existe un procesador maestro que recoge la información del resto y se encarga de actualizar la matriz que simula la población, así como de realizar las búsquedas locales. La forma de paralelizar es encargar la evaluación de un individuo a cada procesador.

- El algoritmo genético compacto trabaja con un número reducido de individuos y fija el número de individuos por generación con la presión de selección s. Para nuestro problema es suficiente un valor de s entre 5 y 8. Por este motivo se ha escogido una red de estaciones de 8 procesadores cuyas características están en la tabla 6.10. La comunicación se realiza mediante una red *fast ethernet* con un *switch fast ethernet*.

Nombre Nodo	Tipo Nodo	Procesador	Configuración CPU	Tamaño Memoria (MB)	Disco (GB)
castilla	Servidor	PII 450	Dual	256	40,8,4,4
pila 0	Computo	PII 866	Dual	512	40
pila 1	Computo	PII 866	Dual	512	40
pila 2	Computo	PII 866	Dual	512	40
pila 3	Computo	PII 866	Dual	512	40

Tabla 6.10: *Características de la red de estaciones utilizada*

- El tiempo necesario para evaluar un individuo es mucho mayor que el necesario para actualizar la matriz y la búsqueda local se realiza sólo cada cierto número de generaciones, lo que puede provocar la inactividad del maestro. Para estudiar este problema se han realizado dos implementaciones que se han denominado *Paralelización 1* y *Paralelización 2*.

  - En la *Paralelización 1* el procesador maestro no evalúa ningún individuo, por lo que será más eficiente cuando la búsqueda local se realice más frecuentemente.
  - En la *Paralelización 2* el procesador maestro evalúa los mismos individuos que el resto de procesadores, por lo que será más eficiente cuando la búsqueda local se realice con menor frecuencia.

Para comprobar la efectividad de la paralelización y comprobar las suposiciones realizadas en la descripción de *Paralelización 1* y *Paralelización 2* se han realizado una serie de estudios que se explican a continuación. Todas las gráficas presentan la eficiencia de la paralelización, para distintos circuitos y distintas presiones de selección, definida según la ecuación 6.18.

$$Eficiencia = \frac{Tiempo\_Secuencial}{Tiempo\_Paralelo \cdot Numero\_de\_Procesadores} \tag{6.18}$$

Para obtener todas las medidas se han realizado 12 ejecuciones, eliminando la mejor y la peor y obteniendo la media. El problema consiste en obtener una solución para 8 FPGAs.

En primer lugar, para una mejor claridad de las gráficas, se han comprobado que la eficiencia es aproximadamente igual para todos los circuitos. Los resultados

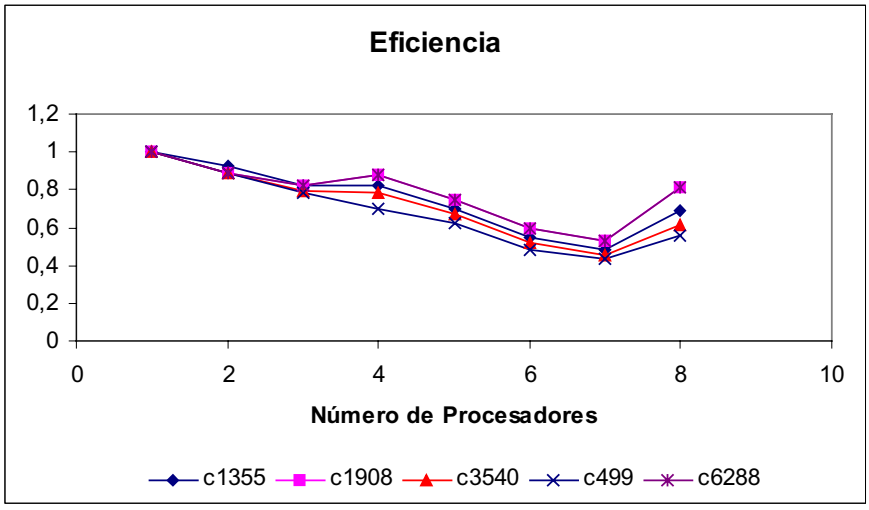

Figura 6.38: *Eficiencia de Paralelización 2 para distintos circuitos*

posteriores se centran únicamente en el circuito c1355 pero se pueden generalizar para todos los problemas que se han tratado con el AGc con búsqueda local y cuyos resultados de partición y ubicación se presentaron en el apartado 6.8. En la figura 6.38 se presenta la eficiencia para distintos circuitos utilizando la *Paralelización 2* y fijando los parámetros del algoritmo a los siguientes valores:

- Número de generaciones: 1000
- Presión de selección, s : 8
- Búsqueda local cada 100 generaciones

En ella se puede comprobar que el comportamiento es similar para los distintos circuitos, es decir que la eficiencia no varía sustancialmente con el tamaño del problema. Evidentemente si el problema es muy pequeño y la presión de selección alta, la eficiencia baja al aumentar el número de procesadores, ya que el tiempo de evaluación disminuye y los procesadores esclavos están más tiempo inactivos que en un problema mayor. Sin embargo, estos problemas no son interesantes para nuestro estudio pues no tienen tiempos de ejecución elevados, que es el motivo principal para realizar la paralelización. Para *Paralelización 1* los resultados son similares. Se podría pensar en otro tipo de paralelización en la que el individuo que ha terminado de evaluar antes comience la evaluación del siguiente cuando el reparto de individuos

entre procesadores está desequilibrado. Sin embargo no parece lo más indicado por los motivos siguientes:

- El tiempo de evaluación para todos los individuos es aproximadamente el mismo. Por ello, a pesar de que comience la evaluación del último individuo antes, no compensaría el tiempo de comunicaciones necesario para informar de cuál es el procesador inactivo

- La evaluación de los individuos no se puede compartir entre dos procesadores de una manera eficiente, por lo que siempre habrá procesadores que evalúen más que otros.

A continuación estudiaremos la eficiencia de los dos modelos de paralelización propuestos para ver cuál es el más interesante para nuestro problema particular. Para realizar el estudio debemos fijar un valor de la presión de selección de tal manera que tanto *Paralelización 1* como *Paralelización 2* tengan máxima eficiciencia con 8 procesadores. Esto sucederá cuando cada procesador evalúe un único individuo.

Para el análisis de *Paralelización 1* se ha fijado s a 7. Puesto que disponemos de 8 procesadores y en este modelo no existe evaluación por parte del maestro, de esta forma cada procesador estará encargado de evaluar un individuo y la carga de trabajo estará compensada para ellos. Por el contrario para analizar *Paralelización 2* cuando cada procesador evalúa un único individuo el valor de la presión de selección debe ser 8, pués en este caso todos los procesadores están encargados de realizar la evaluación.

Paralelización 1

Antes de ver los valores de la eficiencia para los distintos casos de estudio veamos la tabla 6.11 que muestra cuantos individuos evalúa cada uno de los procesadores dependiendo de la cantidad de ellos utilizados cuando se ejecuta *Paralelización 1* con s=7. En ella se han identificado los procesadores con letras. El procesador A es el maestro y el resto son los esclavos que evalúan.

Procesador	A	B	C	D	E	F	G	H
Nº Procesadores								
1	7	–	–	–	–	–	–	–
2	0	7	–	–	–	–	–	–
3	0	4	3	–	–	–	–	–
4	0	3	2	2	–	–	–	–
5	0	2	2	2	1	–	–	–
6	0	2	2	1	1	1	–	–
7	0	2	1	1	1	1	1	–
8	0	1	1	1	1	1	1	1

Tabla 6.11: *Número de Individuos evaluados por cada procesador con Paralelización 1 y s=7*

Como podemos ver, unicamente en el caso en el que se utilizan 8 procesadores todos lo individuos evaluadores (B a H) tienen la misma carga de trabajo y por lo tanto se produce la máxima eficiencia. El procesador A se encarga de realizar la búsqueda local y resto del AGc. Cuando este procesador está trabajando, la mayor parte del tiempo la emplea en la evaluación de los individuos que se prueban en las tentativas de mejora local. Para N particiones, cada vez que se realiza una búsqueda local se evalúan $N-1$ individuos. Obviamente, un parámetro fundamental para el estudio de la implementación paralela es el periodo de búsqueda local.

En primer lugar se debe comprobar que con la paralelización conseguimos una reducción notable en los tiempos de ejecución, pues este es nuestro objetivo fundamental al implementarla. La figura 6.39 muestra la reducción en los tiempos de ejecución con *Paralelización 1* para el circuito c1355, ejecutando 1000 generaciones con una presión de selección de 7 individuos y distintos periodos de búsqueda local. Como se puede ver para 8 procesadores el tiempo de ejecución es aproximadamente una quinta parte que el necesario para un procesador y por lo tanto la paralelización es conveniente. Cuando la búsqueda local se realiza cada iteración lo que sucede es que el que marca el tiempo final es el procesador maestro y mientras que el resto de procesadores evalúa un individuo en cada generación, él debe evaluar 7 individuos, pues el problema consiste en obtener 8 particiones.

Veamos a continuación cómo varía la eficiencia. En la figura 6.40 se representa

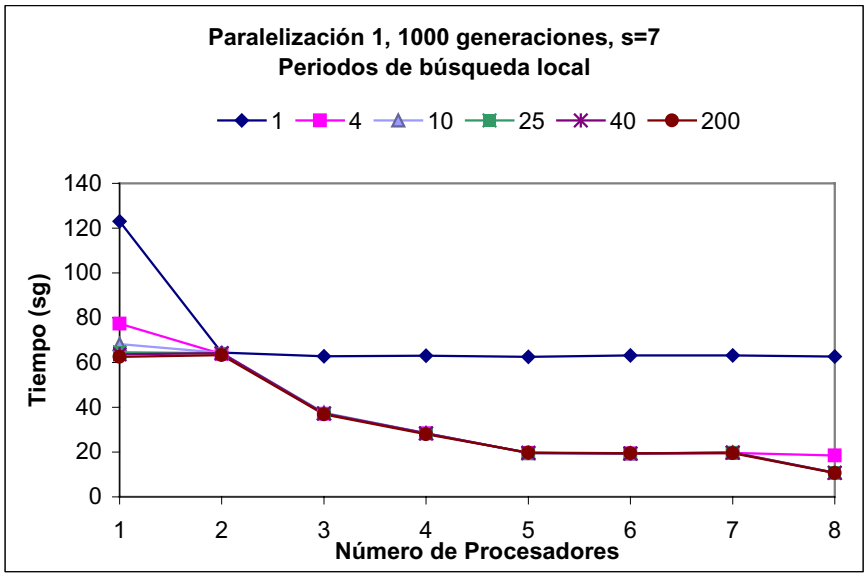

Figura 6.39: *Tiempos de ejecución (sg) para 1000 generaciones y el circuito c1355, con* Paralelización 1 *y distintos periodos de búsqueda local*

la eficiencia de *Paralelización 1* para s=7, 1000 generaciones y distintos periodos de búsqueda local. En ella podemos comprobar que, salvo para frecuecias muy altas, obtiene una mejor eficiencia cuando se producen más búsquedas locales, pues es cuando más se aprovecha el tiempo que le sobra al procesador maestro después de actualizar la matriz de probabilidades y realizar el resto de pasos del AGc. Para intercambios cada 1 y cada 4 generaciones esta eficiencia baja debido a que se somete al procesador maestro a un trabajo excesivo con lo que tarda más en realizar las exploraciones locales que el resto en la evaluación de los individuos. En realidad, no se debe nunca hacer búsquedas locales con una frecuencia tan alta, ya que el funcionamiento del algoritmo se degenera y no se obtienen buenos resultados hablando en términos de calidad de las soluciones.

Paralelización 2

El análisis para *Paralelización 2* es análogo al realizado para *Paralelización 1*. La tabla 6.12 muestra cuantos individuos evalúa cada uno de los procesadores cuando la presión de selección es 8. En ella el procesador A vuelve a ser maestro y el resto son

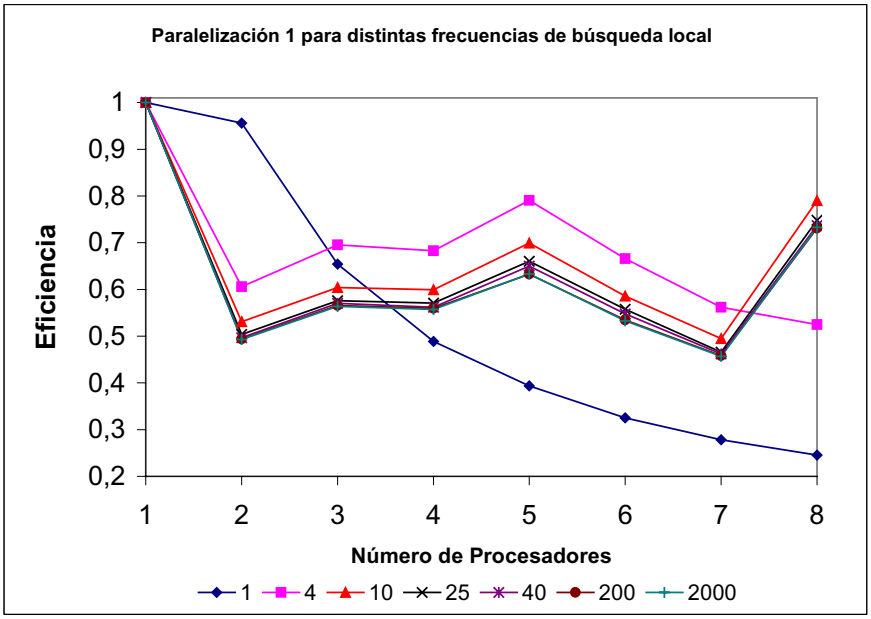

Figura 6.40: *Eficiencia para Paralelización 1, s=7, 1000 generaciones y distintos valores del periodo de búsqueda local*

los esclavos. En este caso el maestro también evalúa y además actualiza la matriz y realiza las búsquedas locales. Por lo tanto el periodo con que se realizan éstas vuelve a ser un parámetro determinante en nuestro estudio.

En la tabla 6.12 podemos ver que hay dos casos en los que se distribuyen las evaluaciones hómogeneamente; para 4 y para 8 procesadores. El tiempo de ejecución viene determinado siempre por el maestro, pues es el que más carga de trabajo realiza y por eso con 5, 6 y 7 procesadores el tiempo de ejecución es prácticamente igual que con 4 procesadores. Esto se puede observar claramente en la figura 6.41. Cuanto menor sea el periodo de búsqueda local, mayor será el tiempo final para *Paralelización 2*.

La figura 6.42 muestra la eficiencia de *Paralelización 2* para s=8, 1000 generaciones y distintos periodos de búsqueda local. En ella podemos comprobar que, para frecuecias bajas, obtiene una mejor eficiencia, ya que tiempo que tarda el procesador maestro en actualizar la matriz de probabilidades y realizar el resto de pasos del AGc es muy pequeño en comparación con el que se tarda en evaluar a los individuos. Se

Procesador	A	B	C	D	E	F	G	H
Nº Procesadoress								
1	8	–	–	–	–	–	–	–
2	4	4	–	–	–	–	–	–
3	3	3	2	–	–	–	–	–
4	2	2	2	2	–	–	–	–
5	2	2	2	1	1	–	–	–
6	2	2	1	1	1	1	–	–
7	2	1	1	1	1	1	1	–
8	1	1	1	1	1	1	1	1

Tabla 6.12: *Número de Individuos evaluados por cada procesador con Paralelización 2 y s=8*

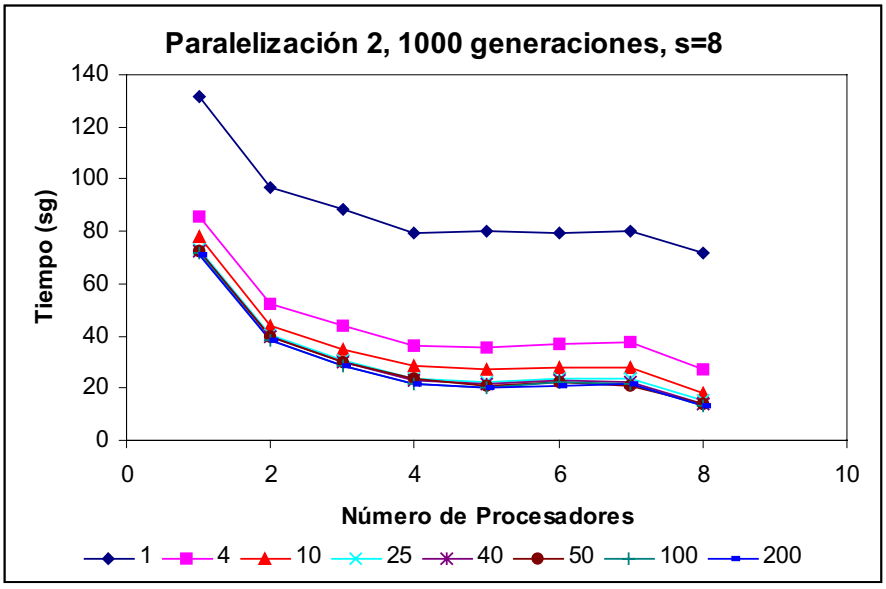

Figura 6.41: *Tiempos de ejecución (sg) para 1000 generaciones y el circuito c1355, con* Paralelización 2

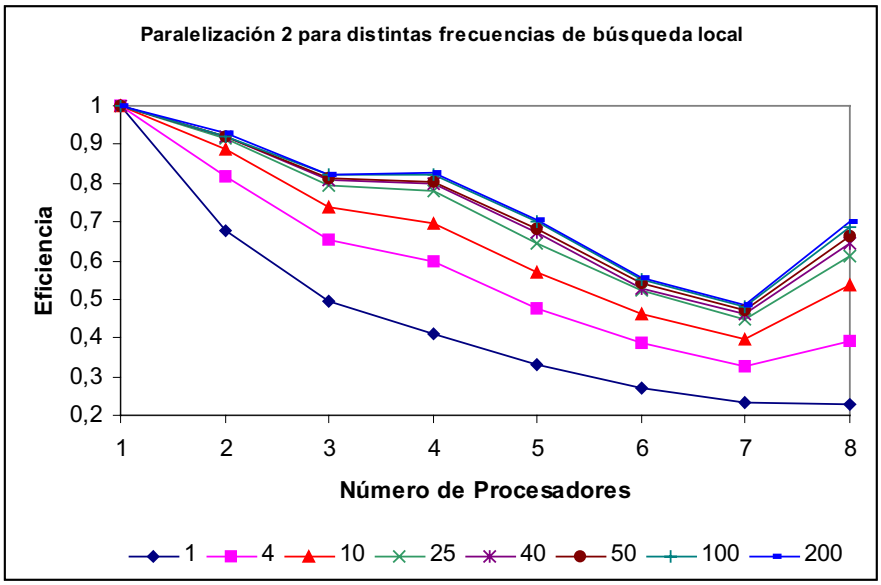

Figura 6.42: *Eficiencia para Paralelización 2, s=8, 1000 generaciones y distintos valores del periodo de búsqueda local*

observa también que al contrario de lo que se supuso la mayor eficiencia se produce cuando se utilizan 4 procesadores. Esto se debe a que la diferencia entre la evaluación y el resto de tareas es mayor. No obstante, como el objetivo es reducir el tiempo de ejecución siempre será más conveniente utilizar 8 procesadores que 4 pues el tiempo de ejecución se reduce aproximadamente en un 30%.

Eficiencia comparada

Para ver cuál de las dos propuestas es más conveniente para nuestro problema, analizamos la eficiencia de ambas cuando el tiempo de ejecución es menor, es decir para 8 procesadores. La figura 6.43 representa la eficiencia de *Paralelización 1* con s=7 y la de *Paralelización 2* con s=8 para distintos valores de la frecuencia de la búsqueda local. En ella se puede observar que la *Paralelización 1* es más eficiente en todos los casos.

La figura 6.44 muestra una comparación de los tiempos de ejecución en las mismas condiciones que las de la figura 6.43. Puesto que el tiempo de ejecución también

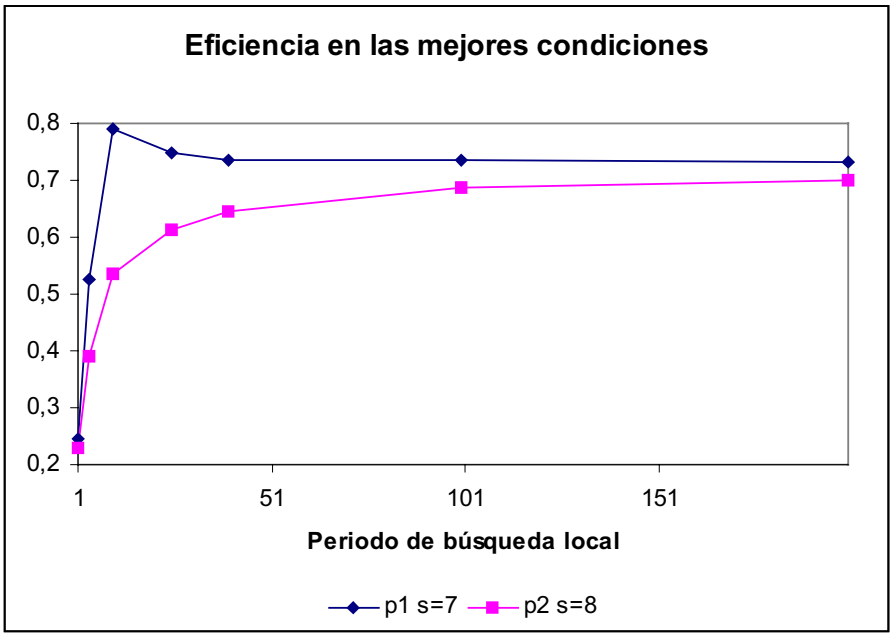

Figura 6.43: *Eficiencia para Paralelización 2 con s=8 y Paralelización 1 con s=7 1000 generaciones, 8 procesadores y distintos valores del periodo de búsqueda local*

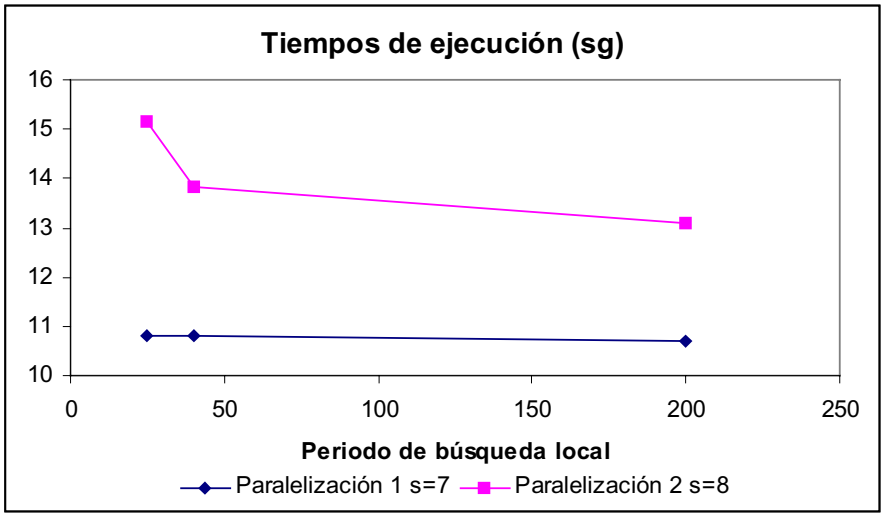

Figura 6.44: *Tiempos de ejecución para Paralelización 2 con s=8 y Paralelización 1 con s=7 1000 generaciones, 8 procesadores y distintos valores del periodo de búsqueda local*

es menor para los periodos de búsqueda local con los que se obtienen buenos resultados (25 y 40) y que no hay diferencias en la calidad de las soluciones con s=7 y s=8, podemos concluir que para el problema de la partición para SMFPGA con 8 particiones es mas conveniente el modelo implementado en *Paralelización 1*.

6.10 Conclusiones

Despúes de cada apartado se han recogido una serie de conclusiones pero podríamos resumir los resultados obtenidos en este capítulo en los siguientes puntos:

- Se ha diseñado un método de partición y ubicación para SMFPGAs basado en AGs, que conserva la estructura del circuito inicial y respeta las restricciones de la tarjeta
- Al explotar las características del cálculo paralelo, del algoritmo genético compacto y técnicas de mejora local conjuntamente, se pueden tratar y resolver problemas más complejos de los que se habían tratado con un algoritmo genético simple.
- Se han realizado dos implementaciones de la paralelización de la función de coste que se han denominado *Paralelización 1* y *Paralelización 2*.
  - En la *Paralelización 1* el procesador maestro no evalúa ningún individuo y es más eficiente cuando la búsqueda local se realice más frecuentemente.
  - En la *Paralelización 2* el procesador maestro evalúa los mismos individuos que el resto de procesadores, por lo que es más eficiente cuando la búsqueda local se realice con menor frecuencia.
- para el problema de la partición para SMFPGA con 8 particiones es mas conveniente el modelo implementado en *Paralelización 1*.
- Se ha implementado una paralelización global de todo el programa, en el que se ejecutan varios AGs simultáneamente con un speed lineal, prácticamente igual al ideal.

Capítulo 7

Conclusiones y Trabajo futuro

En el primer capítulo definimos los objetivos generales de esta tesis como:

- Obtener una técnica de partición que conserve la estructura inicial del grafo.
- Integrar esta técnica de partición en un AG mediante una codificación sencilla que represente soluciones reales sin necesidad de aplicar ope-radores reparadores o cualquier otra técnica de doble codificación que aumente la complejidad del algoritmo.
- Estudiar los algoritmos genéticos compactos como herramienta de partición de grafos e incorporar técnicas de búsqueda local para mejorar las soluciones.
- Estudiar, diseñar y paralelizar estos algoritmos de partición de grafos basados en algoritmos genéticos para mejorar su rendimiento.
- Aplicar estos algoritmos a un problema de diseño como el de la partición de circuitos para su implementación sobre SMFPGAs, abarcando las distintas topologías y teniendo en cuenta las restricciones propias de estos sistemas (consumo de pines y capacidad lógica).
- Incorporar una técnica de ubicación para SMFPGAs de topología malla que permita reducir el consumo de PE/S, asegurando la rutabilidad del sistema final. Esta ubicación debe dar prioridad para situar más próximas aquellas particiones que tienen una mayor comunicación entre sí. Debe tener en cuenta

que en ocasiones estas comunicaciones no son directas, sino a través de particiones no adyacentes.

Todos estos objetivos se han alcanzado y tendremos que distinguir las aportaciones y conclusiones obtenidas en este trabajo cómo de dos tipos:

- las derivadas del aspecto algorítmico
- las propias de la partición para la implementación de circuitos en SMFPGAs.

7.1 Principales aportaciones del trabajo

Aspecto Algorítmico

- Se ha implementado en primer lugar un algoritmo genético simple (SAG) que permite realizar la partición utilizando una codificación sencilla. Este algoritmo no usa ningún mecanismo para conservar la estructura del grafo y por lo tanto sus aplicaciones pueden verse limitadas.
- El S AG necesita en alguna de las ocasiones más de 2000 generaciones para obtener una solución aceptable, mientras que el AGP en anillo sólo necesita 225 y el AGPMS 300 en el peor de los casos. Este resultado se debe tanto a la búsqueda simultánea realizada por las 8 subpoblaciones como al reemplazamiento sin solapamiento implícito del AGP.
- Se han realizado tres implementaciones paralelas del SAG. Para ello se ha utilizado la paralelización de grano grueso y tres topologías de comunicación diferentes; Topología en anillo (AGPA), topología maestro esclavo (AGPMS) y topología Todos con Todos (AGPT).
- Dado que la frecuencia de intercambio entre las subpoblaciones es muy alta, las topologías maestro esclavo y todos con todos sufren un problema de convergencia prematura que no sufre la topología en anillo. Este problema podría solucionarse mediante un estudio de dicha frecuencia. Puesto que el objetivo de la implementación paralela era reducir los tiempos de ejecución

del algoritmo secuencial y esto se consigue con la topología en anillo, no se ha considerado oportuno realizar este estudio para el caso tratado. Además la comunicación en la topologías AGPMS y AGPT, necesitan más memoria para almacenar y mantener el conjunto de individuos que se van a reemplazar.

- El AGPA en anillo obtiene mejores resultados que el AGPMS, debido fundamentalmente a problemas de convergencia prematura en el nodo master, ya que éste sustituye en cada generación una proporción importante de individuos, degradándose las características del algoritmo genético.

- Para solucionar el problema de la conservación de la estructura del grafo se ha desarrollado una técnica de partición basada en el árbol de expansión del grafo que no destruye la forma inicial. Este algoritmo se ha incorporado a 3 tipos de AGs distintos; un algoritmo genético simple, un algoritmo genético compacto y un algoritmo genético compacto con búsqueda local.

- Se ha presentado un nuevo algoritmo para la partición de grafos cuya originalidad reside en la codificación del problema, basada en las aristas y no en los vértices.

- La codificación presentada es perfectamente aplicable a cualquier problema de partición de grafos independientemente de cuál sea el objetivo del mismo, que vendrá particularizado por la función de coste que guía el algoritmo.

- Se ha resuelto el problema de convergencia prematura, que puede aparecer al tener un número de genes reducido, mediante el operador de Regeneración. Este operador se activa automáticamente cuando se detecta que el mejor individuo se repite un número determinado de generaciones.

- Se ha demostrado que el tiempo de cómputo necesario y la cantidad de memoria necesaria para representar la población y las estructuras de datos suponen serios problemas para la resolución de problemas de partición mediante AG.

- Se han implementado tanto una versión secuencial como una paralela del algoritmo genético compacto y se han ejecutado sobre un cluster de PCs.

- Dado que el AGc no trabaja con un población, si no que solamente simula su existencia, se produce un considerable ahorro de memoria para representarla.

- En problemas en los que la complejidad de la función de coste hacen de la evaluación la parte mas costosa en tiempo de cálculo, el AGc permite reducir los tiempos de ejecución.
- El AGc puede en ocasiones dar soluciones de una calidad ligeramente inferior al AG simple que se solucionan mediante una búsqueda local incorporada en el AGc.
- El AGc con búsqueda local mejora los resultados y los tiempos de ejecución obtenidos por el AG simple y por el AGc.
- La paralelización del AGc permite tratar y resolver problemas más complejos de los que se habían tratado con un algoritmo genético simple.
- Para la implementación del AGc paralelo se ha utilizado el modelo de paralelización global mediante la libreria de paso de mensajes MPI. Dado que cada árbol de expansión genera un búsqueda en un espacio diferente, es necesario ejecutar el algoritmo varias veces para obtener una buena solución. La estructura sencilla del AGc secuencial permite una fácil utilización del paralelismo. En este tipo de cálculos la carga de trabajo queda equilibrada de una forma natural.

Partición y ubicación de sistemas Multi-FPGA

Para la partición de SMFPGAs de Topología libre:

- Los resultados obtenidos por SAG son satisfactorios tanto si los comparamos con la herramienta Lp-Solve como si los comparamos con la herramienta PROP. Realmente, la comparación más importante es ésta última pues se pretende asegurar la rutabilidad del circuito.
- Aunque no se mejora el coste en todos los casos la distribución de bloques lógicos que nos da el AG se asegura la rutabilidad del sistema en un 88% de los casos.
- El coste del circuito resultante se ha reducido también en un 45 % de los experimentos en relación a los resultados de PROP.

- La función de coste utilizada es fácilmente adaptable para que incluya otras restricciones del sistema, tales como las referentes a utilización del número de bloques de entrada salida o del número de pines.

Para la partición de SMFPGAs de Topología fija:

- Se ha presentado un método de partición y ubicación de circuitos orientado a sistemas de topología fija.
- El método respeta la estructura inicial del circuito realizando particiones y ubicaciones que tienden a poner en la misma partición los elementos del circuito que están conectados entre sí.
- El método respeta las restricciones de número de PE/S. La forma de evaluar tiene en cuenta el consumo de PE/S necesario para conectar dos elementos del circuito que estén en FPGAs no adyacentes y que por lo tanto consume PE/S al atravesar otras FPGAs para realizar la conexión.
- La ubicación no se realiza de forma aleatoria si no que tiene en cuenta la relación entre las distintas particiones. Mediante la utilización de la matriz de relación inexacta no solamente se tienen en cuenta las relaciones directas si no también las relaciones a través de otras FPGAs.
- La solución propuesta busca las partes del circuito que son independientes y las ubica en FPGAs independientes, disminuyendo de esta forma el consumo de pines.
- los resultados obtenidos muestran que los algoritmos genéticos compactos son capaces de resolver satisfactoriamente el problema de la partición en SMFPGAs.
- El AGc se puede complementar con una búsqueda local para obtener mejores particiones.

7.2 Futuras líneas de investigación

Las futuras líneas de investigación se pueden orientar en varios caminos. En primer lugar los algoritmos paralelos tanto de grano grueso cómo de paralelización global han demostrado su eficiencia, pero es posible hacer un estudio más detallado de las proporciones de intercambio para optimizar su funcionamiento. Otro aspecto a estudiar son los distintos parámetros del operador de regeneración. Esto permitiría su aprovechamiento para otros algoritmos genéticos con los mismos problemas de convergencia prematura.

La influencia de los operadores de selección, cruce y mutación en el funcionamiento del AG puede ser decisiva para el correcto funcionamiento del mismo. Por ello, proponemos estudiar otros métodos de selección cómo el basado en el ranking. Entre los operadores de mutación proponemos estudiar las técnicas presentadas en [1] y [2] que pueden hacer que el operador se vaya adaptando al problema a medida que éste avanza.

El algoritmo de partición se puede aplicar a problemas de partición de grafos. Dentro de éstos proponemos problemas de mínimo corte que puede tener aplicaciones en cualquier área donde se utilicen técnicas de grafos. Por otro lado la técnica de partición desarrollada se puede aplicar a otro tipo de problemas de diseño. En estos momentos estamos trabajando en su aplicación para problemas de partición temporal en sistema de hardware reconfigurable dinámicamente.

El AGc con búsqueda local ha dado unos resultados muy buenos tanto para el problema del TSP cómo para la partición y ubicación. Su aplicación a otros problemas de optimización podría suponer un ahorro considerable en tiempos de ejecución.

En la actualidad estamos también trabajando en la integración de la partición y técnicas de programación genética para ubicación y rutado como las presentadas en [57], [58] y [170].

7.3 Publicaciones

Los trabajos realizados durante el desarrollo de esta tesis han sido recogidos en distintas publicaciones científicas. El algoritmo genético aplicado a la partición en codiseño, que se explica en el apéndice Aque es la base del utilizado para SMFPGA de topología libre, se presentó en el artículo:

- J.I. Hidalgo, J. Lanchares. Functional Partitioning for Hardware Software Codesign using Genetic Algorithm. Proceedings of the 23rd EUROMICRO conference, Budapest (Hungary), 1-4 de Septiembre de 1997. IEEE Press. Páginas 631-638 (ISBN 0-8186-8129-2)

Los resultados referentes a la partición de SMFPGAs de topología libre han quedado recogidos en:

- J.I. Hidalgo, M. Prieto, J. Lanchares, F.Tirado. A Parallel Genetic Algorithm for solving the Partitioning Problem in Multi-FPGA systems. Proceedings of 3rd international Meeting on vector and parallel processing. Oporto (Portugal), 21-23 de Junio de 1998. Páginas 717-722.

- J.I. Hidalgo, M. Prieto, J. Lanchares, R. Hermida. Un Algoritmo genético paralelo para la resolución del problema de la partición en sistemas Multi-FPGA. Actas de las IX Jornadas de Paralelismo. San Sebastián(España), 2-4 Septiembre de 1998, páginas 349-355.

La implementación de los AG paralelos anteriores fueron utilizados para la resolución de otros problemas de optimización que se recogen en:

- J.I. Hidalgo, M. Prieto, J. Lanchares, F.Tirado, B.de Andrés, S. Esteban, D. Rivera. A Method for Model Parameter Identification using Parallel Genetic Algorithms. Proceedings of Euro PVM-MPI 99 Conference, 1999, páginas 291-298.

- B.de Andrés, S. Esteban , D. Rivera, J.I. Hidalgo, M. Prieto, J. Lanchares, F.Tirado, Parallel Genetic Algorithms an application for Model Parameter Identification in Process Control, Proceedings of the 2000 Genetic and Evolutionary Computation Conference Late Breaking Papers, Las Vegas(USA), 2000, páginas 65-69.

El Algoritmo genético compacto híbrido ha sido desarrollado junto con el Parallel Processing Group del CNUCE (Pisa). Antes de aplicarlo al problema de la partición y ubicación en sistemas Multi-FPGA de topología fija se probó con exito para resolver el problema del TSP o del viajante:

- R. Baraglia, J.I. Hidalgo, R. Perego. A Hybrid Heuristic for the Traveling Salesman Problem. IEEE Transactions on Evolutionary Computation, En prensa.

- R. Baraglia, J.I. Hidalgo, R. Perego. A Parallel Hybrid Heuristic for the TSP. Proceedings of EvoCOP2001, the First European Workshop on Evolutionary Computation in Combinatorial Optimization, Lake Como (Milan), April 18-19, 2001, En prensa.

Finalmente, las siguientes publicaciones recogen los resultados para topologías fijas explicados en el capítulo 6

- J.I. Hidalgo, J. Lanchares, R. Hermida. Graph Partitioning methods for Multi-FPGA systems and Reconfigurable Hardware based on Genetic algorithms. Proceedings of the 1999 Genetic and Evolutionary Computation Conference Workshop Program, Orlando (USA), 1999, páginas 357-358.

- J.I. Hidalgo, J. Lanchares, R. Hermida. Partitioning and Placement for Multi-FPGA systems using Genetic algorithms. Proceedings of the 26th Euromicro conference, Maastrich, (Holanda), 5-7 Septiembre 2000. IEEE Press, páginas 204-211 (ISBN 0-7695-0780-8)

- J.I. Hidalgo. Evolutionary Algorithms for Solving the Partitioning and Placement Problems in Multi-FPGA systems. Proceedings of the 2000 Genetic and Evolutionary Computation Conference Workshop Program, Las Vegas (USA), 2000 páginas 281-284.

- J.I. Hidalgo, R. Baraglia, R. Perego, J. Lanchares, F. Tirado, A Parallel Compact Genetic Algorithm for multi-FPGA Partitioning, Proceedings of the 9th Euromicro Workshop on Parallel and Distributed Processing, Mantova, Italia 7-9 de Febrero 2001. IEEE Press, páginas 113-120 (ISBN 0-7695-0987-8)

- J.I. Hidalgo, M. Prieto, J. Lanchares, R. Hermida. Un Algoritmo genético para la resolución del problema de la partición en sistemas Multi-FPGA. Actas de las IX Jornadas de Paralelismo. San Sebastián(España), 2-4 Septiembre de 1998, Páginas 349-355.

- J.I. Hidalgo, J. Lanchares, R. Hermida, F. Tirado. Un Algoritmo genético compacto paralelo para la resolución de problemas de partición y ubicación en sistemas Multi-FPGA. Actas de las XI Jornadas de Paralelismo. Granada (España), Septiembre de 2000, páginas 253-258(ISBN 84-699-3003-6).

- J.I. Hidalgo, J. Lanchares, R. Hermida, A. Ibarra. Un método de ubicación y evaluación de pines para sistemas Multi-FPGA. Jornadas sobre Computación reconfigurable, Alicante, Septiembre de 2001.

- J.M. Colmenar, J.I. Hidalgo, J. Lanchares. Estimación de la ocupación lógica en FPGAs para hardware reconfigurable. Jornadas sobre Computación reconfigurable, Alicante, Septiembre de 2001.

7.4 Proyectos de investigación y ayudas recibidas

Esta tesis ha sido posible gracias a distintos proyectos de investigación:

- Proyecto PR181/96-6776 de la Universidad Complutense de Madrid, "Desarrollo de un sistema automático de diseño de circuitos integrados orientado FPGAs basado en programación evolutiva". Investigador principal: Juan Lanchares Dávila.

- Proyecto 07 T/0019/ 1197 de la Comunidad Autónoma de Madrid, "Sistema de desarrollo para aplicaciones de hardware reconfigurable". Investigador principal: José Jaime Ruz Ortiz.

- Proyecto TIC 99/0474 de la Comisión Interministerial de Ciencia y Tecnología, "Computación paralela sobre plataformas de bajo coste. Nuevas Metodologías para la verificación y el diseño de alto nivel". Investigador principal: Francisco Tirado Férnandez.

- Proyecto CHRX-CT94-0459 de la Unión Europea, "Behavioral Deign Methodologies for Digital Systems". Investigador principal: Francisco Tirado Férnandez.

- Proyecto PQE2000 del Gobierno Italiano. "Progetto nazionale italiano per la realizzazione hardware e software di un computer parallelo ad altissime prestazioni". Investigador principal: Raffaele Perego.

Se han recibido las siguientes ayudas:

- Beca de viaje de la American Asociation for Artificial Inteligence (AAAI) para la asistencia al Congreso *Genetic and Evolutionary Computation 1999*, Orlando, USA.

- Beca de viaje de la American Asociation for Artificial Inteligence (AAAI) para la asistencia al Congreso *Genetic and Evolutionary Computation 2000*, Las Vegas, USA.

- Assegno de Ricerca del Consorzio Pisa-Ricerca.

Bibliografía

[1] H. Aguirre, K. Tonaka, and T. Sugimura. Cooperative crossover and mutation operators in genetic algorithms. In *Proceedings of the 1999 Genetic and Evolutionary Computation Conference*, page 772. Morgan Kaufmann, 1999.

[2] H. Aguirre, K. Tonaka, T. Sugimura, and S. Oshita. Cooperative-competitive model for genetic operators: Contributions of extinctive selection and paralell genetic operators. In *Proceedings of the 2000 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 6–14, 2000.

[3] M.J. Alexander and G. Robins. New performance driven FPGA routing algorithms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Sytems*, 15(12):1505–1517, December 1996.

[4] C.J. Alpert. *Multi-way Graph and Hypergraph Partitioning.* PhD thesis, University of California, Los Angeles, 1996.

[5] C.J. Alpert and A.B. Kahng. Recent directions in netlist partitioning: A survey. Technical Report CA 90024-1596, UCLA Computer Science Department, 1997.

[6] C.J. Alpert, L.W.Hagen, and A.B. Kahng. A hybrid multilevel/genetic approach for circuit partitioning. Technical Report 96002, UCLA Computer Science Department, 1996.

[7] B. de Andrés, S. Esteban, D. Rivera, J.I. Hidalgo, and M. Prieto. Parallel genetic algorithms an application for model parameter identification in process control. In *Proceedings of the 2000 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 65–69, 2000.

[8] K.J. Antreich, F.M. Johannes, and F.H. Kirsch. A new approach for solving the placement problem using force models. In *Proceedings of ISCAS*, pages 481–486, 1982.

[9] J. Babb, R. Tessier, M. Dahl, S. Hanono, , and M. Hoki A. Agarwal. Logic emulation with virtual wires. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Sytems*, 16(6):609–626, June 1997.

[10] R. Baraglia, J.I. Hidalgo, and R. Perego. A hybrid heuristic for the traveling salesman problem. IEEE Transactions on Evolutionary Computation. en prensa.

[11] R. Baraglia, J.I. Hidalgo, and R. Perego. A hybrid approach for the TSP combining genetics and the lin-kerninghan local search. Technical Report CNUCE-B4-2000-007, CNUCE - Institute of the Italian National Research Council, 2000.

[12] R. Baraglia, J.I. Hidalgo, and R. Perego. A parallel hybrid heuristic for the tsp. In *Proceedings of EvoCOP2001, the First European Workshop on Evolutionary Computation in Combinatorial Optimization*, pages 193–202, Berlin, April 2001. Springer Verlag.

[13] M. Baxter. Icarus: A dynamically reconfigurable computer architecture. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 278–279, 1999.

[14] T. Benner, R. Ernst, I. Koenenkamp, and U. Holtmann. FPGA based prototyping for verification and evaluation in hardware-software cosynthesis. In *Lecture Notes in Computer Science No. 849*. Springer-Verlag, 1994.

[15] M. Berkelaar. LPSOLVE source code. ftp://ftp.es.ele.tue.nl/pub/lp_solve/lp_solve.tar.gz, 1996.

[16] A. Bertoni and M. Dorigo. Implicit paralellism in genetic algorithms. *Artificial Intelligence*, 61(2):307–314, Februrary 1993.

[17] R. Bianchini and C. M. Brown. Parallel genetic algorithms on distributed-memory architectures. In *Transputer Research and Applications 6*, pages 67–82, Amsterdam, 1993. S. Atkins and A. S. Wagner.

[18] R. Bianchini and C.M. Brown. Parallel genetic algorithm on distributed-memory architectures. Technical Report TR 436, Computer Sciences Department University of Rochester, 1993.

[19] I.M. Bland and G.M. Megson. The systolic array genetic algorithm, an example of systolic arrays as a reconfigurable design methodology. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 260–261, 1998.

[20] D.E. van den Bout and T.K. Miller III. Graph partitioning using annealed neural networks. *IEEE Transactions on Neural Networks*, 1(2):192–203, 1990.

[21] G. Brassard and P. Bratley. *Fundamentals of Algorithmics.* Prentice-Hall, 1996.

[22] H. Braun. On solving traveling salesman problems by genetic algorithms. In *Parallel Problem Solving from Nature - Proceedings of 1st Workshop PPSN , volume 496 of Lecture Notes in Computer Science*, pages 129–133. Springer-Verlag, 1991.

[23] S.D. Brown. Recent advances in FPGAs. Technical Report 8-Sept, Toronto University, 1995.

[24] S.D. Brown and J. Rose. FPGA architectural research: A survey. *IEEE Design and Test of Computer*, pages 9–15, Winter 1996.

[25] T.N. Bui and B.R. Moon. Genetic algorithm and graph partitioning. *IEEE Transactions on Computers*, 45(7):841–855, Julio 1996.

[26] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. de Wit. A dynamic reconfiguration run-time system. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 66–75, 1997.

[27] C.A.Coello. A comprehensive survey of evolutionary-based multiobjective optimization techniques. *Knowledge and Information Systems*, 1(3):269–308, 1999.

[28] C.A.Coello and A.D. Christiansen. Moses : A multiobjective optimization tool for engineering design. *Engineering Optimization*, 31(3):337–368, 1999.

[29] CAD benchmark labaoratory web site. http://zodiac.cbl.ncsu.edu/CBL_Docs/Bench.html, 1993.

[30] E. Cantú-Paz. Designing efficient master-slave parallel genetic algorithms. In *Genetic Programming: Proceedings of the Third Annual Conference*, page 455, San Francisco, CA, 1998. Morgan Kaufmann.

[31] E. Cantú-Paz. Migration policies and takeover times in parallel genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-99*, page 775, San Francisco, CA, July 1999. Morgan Kaufmann.

[32] E. Cantú-Paz. Migration policies and takeover times in parallel genetic algorithms. Technical Report Illigal Report 99008, University of Illinois at Urbana Champaign, 1999.

[33] E. Cantú-Paz. Topologies, migration rates, and multi-population parallel genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-99*, pages 91–98, San Francisco, CA, July 1999. Morgan Kaufmann.

[34] E. Cantú-Paz. *Efficient and Accurate Parallel Genetic Algorithms*. Kluwer Academic Publishers, Boston, MA, 2000.

[35] E. Cantú-Paz and D.E. Goldberg. Efficient parallel genetic algorithms: theory and practice. *Computer Methods in Applied Mechanics and Engineering*, -(186):221–238, 2000.

[36] E. Cantu-Paz. A summary of research on parallel genetic algoritms. Technical Report 95007, University of Illinois at Urbana-Champaign, Genetic Algoritms Lab. (IlliGAL), http://gal4.ge.uiuc.edu/illigal.home.html, July 1995.

[37] P.K. Chan, M.D.F. Schlag, and J.Y. Zien. Spectral based multi-way FPGA partitioning. Technical Report ucsc-crl-94-44, University of California, Santa Cruz, 1994.

[38] N.C. Chou, L.T. Liu, C.K. Cheng, W.J. Dai, and R.Lindelof. Circuit partitioning for huge logic emulation systems. In *Proceedings of the ACM/IEEE 31st Design Automation Conference*, pages 244–249, 1994.

[39] H.A. Chow, A. Elnaggar, and H.M. Alnuweiri. Higly parallel signal processing on the virtual computer. In *Field Programmable Gate Arrays for Fast Board Development and Reconfigurable Computing (SPIE 2607 TheInternational Society for Optical Engineering)*, pages 42–53, 1995.

[40] C. Coello. *An Empirical Study of Evolutionary Techniques for Multiobjective Optimization in Engineering Design*. PhD thesis, Tulane University of New Orleans, Louisiana. E.E.U.U., 1996.

[41] S. Cohoon, J. Hedge, S. Martin, and D. Richards. Punctuated equilibria: a parallel genetic algorithm. *IEEE Transaction on CAD*, 10(4):483–491, April 1991.

[42] J.M. Colmenar, J.I. Hidalgo, and J. Lanchares. Estimación de la ocupación lógica en fpgas para hardware reconfigurable. In *Actas de las Jornadas sobre Computación reconfigurable*, page en prensa, Alicante, Sept. 2001.

[43] G. A. Croes. A method for solving the traveling salesman problem. *Operations Research*, 6:791–812, 1958.

[44] D. Cvetkovic and I.C. Parmee. Use of preferences for GA-based multi-objective optimization. In *Proceedings of the 1999 Genetic and Evolutionary Computation Conference*, pages 1504–1509. Morgan Kaufmann, 1999.

[45] F. Darema, S. Kirkpatrick, and V.A. Norton. Parallel algorithms fo chip placement by simulated annelaing. *IBM J.Res.Develop*, 31:391–402, May 1987.

[46] A. Dasdan and C. Aykanat. Improved multiple-way circuit partitioning algorithms. In *Proceedings ACM/SIGDA International Workshop on FPGAs*, 1994.

[47] J.P. David and J.D.Legat. A multi-FPGA based coprocessor for digital signal processing. Technical report, Universite Catholique de Louvain, Belgium, 1996.

[48] L. Davis. *Handbook of Genetic Algorithm.* Van Nostrand Reinhold, 1991.

[49] M. Dorigo and V. Maniezzo. Parallel genetic algorithms: Introduction and overview of current research. In *Parallel Genetic Algorithms*, pages 5–42. IOS Press, 1993.

[50] S. Dutt and H. Theny. Partitioning around roadblocks tackling constraints with intermediate relaxations. In *Proceedings of the International Conference on Computer Aided Design*, pages 350–355, 1997.

[51] C. Ebeling, L. McMurchie, S. Hauck, D. Song, and E.A. Walkup. Routing tools for the triptych FPGA. *IEEE Transactions on VLSI Systems*, 3(4):473–482, December 1995.

[52] M.D. Edwards. *Automatic Logic Synthesis Techniques for digital systems.* MacMillan New Electronics, London, 1992.

[53] M. Enos. Replication for logic partitioning. Master's thesis, Northwestern University, Dept. of E.C.E., September 1996.

[54] M. Enos, S. Hauck, and M. Sarrafzadeh. Replication for logic partitioning. In *Proceedings of International Conference on Computer Aided Design*, pages 342–349, 1997.

[55] M. Enos, S. Hauck, and M. Sarrafzadeh. Evaluation and optimization of replication algorithms for logic bipartitioning. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 18(9):1237–1248, September 1998.

[56] W.J. Fang and A. Wu. A hierachical functional structuring and partitioning approach for multiple-FPGA implementations. In *Proccedings of the International Conference on Computer Design*, pages 638–643, 1996.

[57] F. Fernández. *Modelos de Programación Genética distribuida con aplicaciones a síntesis lógica en FPGAs.* PhD thesis, Universidad de Extremadura, 2000.

[58] F. Fernández and M. Tomassini. Genetic programming and reconfigurable hardware: A proposal for solving the problem of placement and routing. In

A.S. Wu University of Central Florida, editor, *Proceedings of the 2000 Genetic and Evolutionary Computation Conference Workshop Program*, pages 281–284, 2000.

[59] F. Fernández, M. Tomassini, W. Punch, and J.M. Sánchez. Experimental study of multipopulation parallel genetic programming. In *Genetic Programming, proceedings of EuroGP2000, Lecture Notes in Computer Science,Vol. 1802*, pages 1–15. Springer-Verlag, 2000.

[60] C.M. Fiduccia and R.M Matheyses. A linear time heuristic for improving network partition. In *Proceedings of ACM/IEEE Design Automation Conference*, pages 175–181, 1982.

[61] D.D. Gajski, F. Vahid, S.Narayau, and J.Gong. *Specification and Design of Embedded Systems*. Prentice-Hall, 1994.

[62] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam. *PVM: Parallel Virtual Machine. A users guide and tutorial for network parallel Computers*. MIT Press Books, Massachussets, 1994.

[63] D. E. Goldberg, K. Deb, and B. Korb. Don't worry, be messy. In *Proc. of the Fourth International Conference on Genetic Algorithms*, pages 24–30, San Diego, CA, 1991.

[64] D.E. Goldberg. Optimal initial population size for binary-coded genetic algorithms. Technical Report TCGA 85001, University of Alabama, 1985.

[65] D.E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison Wesley, 1989.

[66] P. Graham and B.Ñelson. A hardware genetic algorithm for the traveling salesman problem on splash 2. In *Proceedings of the 5th International Workshop on Field Programmable Logic and Applications*, pages 352–361, England, August 1995. Oxford.

[67] P. Graham and B.Ñelson. Genetic algorithms in software and in hardware - a performance analysis of workstation and custom computing machine implementations. In *Proceedings of the IEEE Symposium on FPGAs for Custom*

Computing Machines, pages 216–225, Los Alamitos, CA, April 1996. IEEE Press.

[68] G. Mc Gregor and P. Lysaght. Self controlling dynamic reconfiguration. In *Field Programmable Logic and Applications*, pages 144–154, Berlin, August/September 1999. Springer-Verlag.

[69] G. Gu. Accelerating photoshop applications with reconfigurable hardware. Master's thesis, Nortwestern University, 1999.

[70] J.D. Hadley. The performance enhancementy of a run-time reconfigurable FPGA sytem through partial reconfiguration. Master's thesis, Brigham Young University, November 1995.

[71] J.O. Haenni. Renco: A reconfigurable network computer. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 288–289, 1998.

[72] G. Harik, F. Lobo, and D. Goldberg. The compact genetic algorithm. Technical Report 97006, University of Illinois at Urbana-Champaign, Urbana, IL, 1997.

[73] G. Harik, F. Lobo, and D. Goldberg. The compact genetic algorithm. *IEEE Transactions on Evolutionary Computation*, 3(4):287–297, 1999.

[74] R.W. Hartenstein, R. Kress, and H. Reinig. A reconfigurable data-driven alu for xputers. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 139–146. IEEE Press, April 1994.

[75] S. Hauck. *Multi-FPGA systems*. PhD thesis, University of Washington, 1995.

[76] S. Hauck. Configuration prefetch for single context reconfigurable processors. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 65–74, 1998.

[77] S. Hauck. The roles of FPGAs in re-programmable systems. *Proceedings of the IEEE*, 86(4):615–638, April 1998.

[78] S. Hauck and G. Borriello. An evaluation of bipartitioning techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):849–866, August 1997.

[79] R. Hermida, M. Fernández, F. Tirado, V. Sanchez, and P. Rupérez. An approach to module binding by fuzzy partitioning. In *Proceedings of EuroDAC Conference, IEEE Press*, pages 58–63, 1993.

[80] H.J. Herpel, N. Wehn, M. Gasteier, and M. Glesner. A reconfigurable computer for embedded control applications. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 111–120. IEEE Press, April 1993.

[81] M.I. Heywood and A.N. Zincir-Heywood. Register based genetic programming on FPGA computing platforms. In *Proceedings of EuroGP 2000*, pages 44–59. Springer-Verlag, April 2000.

[82] J.I. Hidalgo. Evolutionary algorithms for solving the paritioning and placementproblems in Multi-FPGA systems. In A.S. Wu University of Central Florida, editor, *Proceedings of the 2000 Genetic and Evolutionary Computation Conference Workshop Program*, pages 281–284, 2000.

[83] J.I. Hidalgo, R. Baraglia, R. Perego, J. Lanchares, and F. Tirado. A parallel compact genetic algorithm for multi-fpga partitioning. In *Proceedings of the 9th Euromicro Workshop on Parallel and Distributed Processing*, pages 113–120. IEEE Press, 2001.

[84] J.I. Hidalgo and J. Lanchares. Fuctional partitioning for hardware-software codesign using genetics algorithms. In *Proceedings of the 23th Euromicro Conference IEEE Press*, pages 631–638. IEEE Press, 1997.

[85] J.I. Hidalgo, J. Lanchares, and R. Hermida. Graph partitioning methods for multi-FPGA systems and reconfigurable hardware based on genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference Workshop Program*, pages 357–358, 1999.

[86] J.I. Hidalgo, J. Lanchares, and R. Hermida. Partitioning and placement for multi-FPGA systems using genetic algorithms. In *Proceedings of the 26th Euromicro COnference, IEEE Press*, pages 204–211, 2000.

[87] J.I. Hidalgo, J. Lanchares, R. Hermida, and A. Ibarra. Un método de ubicación y evaluación de pines para sistemas multi-fpga. In *Actas de las Jornadas sobre Computación reconfigurable*, page en prensa, Alicante, Sept. 2001.

[88] J.I. Hidalgo, J. Lanchares, R. Hermida, and F. Tirado. Un algoritmo genético compacto paralelo para la resolución de problemas de partición y ubicación en sistemas multi-fpga. In *Actas de las XI Jornadas de Paralelismo. Granada (España)*, pages 253–258, Sept. 2000.

[89] J.I. Hidalgo, M. Prieto, J. Lanchares, and R. Hermida. Un algoritmo genético para la resolución del problema de la partición en sistemas multi-fpga. In *Actas de las IX Jornadas de Paralelismo. San Sebastián(España)*, pages 349–355. Universidad del País Vasco, Sept. 1998.

[90] J.I. Hidalgo, M. Prieto, J. Lanchares, and F. Tirado. A parallel genetic algorithm for solving the partitioning problem in multi-FPGA systems. In *Proceedings of the 3rd International Meeting on vector and parallel processing (VECPAR)*, pages 717–722, 1998.

[91] J.I. Hidalgo, M. Prieto, J. Lanchares, F. Tirado, B. de Andr´es, and D. Rivera. A method for model parameter identification using parallel genetic algorithms. In *Proceedings of Euro PVM-MPI Conference LNCS 1697*, pages 291–298. Springer-Verlag, 1999.

[92] H.Kropp, C. Reuter, M. Wiege, T.T. Do, and P. Pirsch. An FPGA-based prototyping system for real-time verification of video processing schemes. In *Field Programmable Logic and Applications*, pages 333–338, Berlin, August/September 1999. Springer-Verlag.

[93] J. Holland. *Adaptation in Natural and Artificial Systems.* Univ. of Mitchigan Press, 1975.

[94] High Performance Fortran. http://www.crpc.rice.edu/HPFF/.

[95] X. Hue. Genetic algorithms for optimization background and applications. Parallel Computing Center, Edimburgh, February 1997. Disponible en: http://www.epcc.ed.ac.uk/epcc-tec/documents.

[96] M. Hulin. Circuit partitioning with genetic algorithms using a coding scheme to preserve the structure of a circuit. In *Lecture Notes in Computer Science, 496*, pages 75–79. Springer-Verlag, 1989.

[97] T. Hwang, R.M. Owens, M.J. Irwin, and K.H. Wang. Logic synthesis for FPGAs. *IEEE Transactions on CAD of ICs and Systems*, 13(10):1280–1287, October 1994.

[98] Silicon Graphics Inc. Developer magic: Debugger user's guide, 1999. Document Number:007-2579-005.

[99] Xilinx Inc. Xnf: Xilinx netlist format. www.xilinx.com.

[100] D. S. Johnson and L. A. McGeoch. *Local Search in Combinatorial Otimization*. John Wiley and Sons, New York, 1996.

[101] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation. In *Workshop on statistical physics in engineering and biology*. Yorktown Heights, 1986.

[102] J. Juliany and M.D. Vose. The genetic algorithm fractal. *Evolutionary Computation*, 2(2):165–180, 1994.

[103] A. Kandel and L. Yelowitz. Fuzzy chains. *IEEE Transactions on Systems Man and Cybernetics*, SMC-4(5):472–475, September 1994.

[104] G. Karypis and V. Kumar. hMetis, a hypergraph partitioning package, 1998.

[105] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Tech. Journal*, 2(49):291–307, 1970.

[106] M.A.S. Khalid. *Routing Architecture and Layout Synthesis for Multi-FPGA Systems*. PhD thesis, University of Toronto, 1999.

[107] M.A.S. Khalid and J. Rose. A hybrid complete-graph partial crossbar routing architecture for multi-FPGA systems. In *Proc. of 1998 Sixth ACM International Symposium on Field-Programmable Gate Arrays (FPGA'98)*, pages 45–54, 1998.

[108] Y.H. Kim and B.R. Moon. A hybrid genetic search for graph partitioning based on local gain. In *Proccedings of The Genetic and Evolutionary Computation Conference 2000*, pages 167–172, 2000.

[109] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[110] J.R. Koza. *Genetic Programming: On the Programming of Computers by means of Natural Selection*. MIT Press, Massachussets, 1992.

[111] J.R. Koza, F.H. Bennett III, J.L. Hutchings, S.L. Bade, M.A. Keane, and D. Andre. Evolving computer programs using rapidly reconfigurable FPGAs and genetic programming. In *Proceedings of the Sixth International Symposium on Field Programmable Gate Arrays*, pages 209–219. ACM Press, February 1998.

[112] H. Krupnova and G. Saucier. Iterative improvement based multi-way netlist partitioning for FPGAs. In *Proceedings of DATE 99 Conference*, pages 587–594. IEEE Press, 1999.

[113] H. Krupnova and G. Saucier. FPGA technology snapshot: Current devices and deign tools. In *Proceedings of 11th IEEE International Workshop in Rapid System Prototyping*, pages 200–205, Paris, June 2000. IEEE Press.

[114] R. Kuznar and F. Brglez. PROP: A recursive paradigm for area-efficient and performance oriented partitioning of large FPGA netlists. In *International Conference on Computer Aided Design*, 1995.

[115] R. Kuznar, F. Brglez, and K. Kozminski. Cost minimization of partitions into multiple devices. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 315–320, 1993.

[116] R. Kuznar, F. Brglez, and B. Zajc. Multi-way netlist partitioning into heterogeneous FPGAs and minimization of total device cost and interconnect. In *Proceedings of the ACM/IEEE 31st Design Automation Conference*, pages 238–243, 1994.

[117] R. Kuznar, F. Brglez, and B. Zajc. A unified cost model for min-cut partitioning with replication applied to optimization of large heterogeneous FPGA partitions. In *European Design Automation Conference*, pages 271–276, 1994.

[118] J. Lanchares. *Desarrollo de Metodologías para las síntesis y optimización de circuitos lógicos multinivel*. PhD thesis, Universidad Complutense de Madrid, 1995.

[119] J. Lanchares, J.I. Hidalgo, and J.M. Sánchez. Boolean network decomposition using genetic algorithms. *Microelectronics Journal. Elsevier Science*, 28(2):143–150, February 1997.

[120] J. Lanchares, J.I. Hidalgo, and J.M. Sánchez. A method for multiple-level logic synthesis based on the simulated annealing algorithm. *Microelectronics Journal. Elsevier Science*, 28(5):551–560, Jun. 1997.

[121] G.von Laszewski. A collection of graph partitioning algorithms. Technical Report 17 May, Science and Technology Center of Syracuse, 1993.

[122] G.von Laszewski and H. Muhlenbein. A parallel genetic algorithm for the k-way graph partitioning problem. In *1st inter. Workshop on Parallel Problem Solving from Nature*, November 1990.

[123] D.M. Lewis, D.R. Galloway, M.V. Ierssel, J. Rose, and P. Chow. The Transmogrifier 2: A million gate rapid prototyping system. *IEEE Transactions on VLSI Systems*, 5(2):188–198, June 1998.

[124] J. Liening. A parallel genetic algorithm for performance-driven VLSI routing. *IEEE Transactions on Evolutionary Computation*, 1(1):29–39, April 1997.

[125] S. Lin. Computer solution of the traveling salesman problem. *Bell Syst. Tech. J.*, 44:2245–2269, 1965.

[126] S. Lin and B. Kernighan. An effective heuristic for the traveling salesman problem. *Oper. Res.*, 21:498–516, 1973.

[127] R. Macketanz and W. Karl. JVX - a rapid prototyping system based on Java and FPGAs. In *Field Programmable Logic: From FPGAs to Computing Paradigm*, pages 99–108, Berlin, August/September 1998. Springer-Verlag.

[128] B. Manderick and P. Spiessens. Fine-grained parallel genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 428–433. Morgan Kaufmann Publishers, 1989.

[129] O. Martin and S.W. Otto. Combining simulated annealing with local search heuristic, 1993. unpublished.

[130] P. Martínez-Ortigosa. *Métodos estocásticos de optimización global. Procesamiento paralelo*. PhD thesis, Universidad de Málaga, 1999.

[131] M. Meerwein, C. Baumgartner, T. Wieja, and W. Glauert. Embedded systems verification with FPGA-enhanced in-circuit emulator. In *International Symposium on System Synthesis*, pages 143–148, Madrid, September 2000. ACM/IEEE.

[132] Z. Michalewicz. *Genetic Algorithms + data structures = Evolution Programs*. Springer-Verlag, 1996.

[133] M. Migliardi, J. Dongarra, A. Gueist, and V. Sunderam. Dynamic reconfiguration and virtual machine management in the harness metacomputing system. *Lecture Notes in Computer Science*, 1505:127–134, 1998.

[134] J.M. Moreno, J. Madrenas, J. Faura, and E. Canto. Feasible evolutionary and self-repairing hardware by means of the dynamic reconfiguration capabilities of the FIPSOC devices. In *Lecture Notes in Computer Science 1478*, pages 345–352, Berlin, 1998. Spinger-Verlag.

[135] MP3 databook. www.aptix.com, 1997.

[136] MPI: A message passing interface standard. http://www.mpi-forum.org.

[137] H. Muhlenbein. Parallel genetic algorithms, population genetic and combinatorial optimization. In *Parallel Problem Solving from Nature - Proceedings of 1st Workshop PPSN*, volume 496, pages 407–417. Lecture Notes in Computer Science, 1991.

[138] R. Murgai, R.K. Brayton, and A. Sangiovanni-Vicentelli. *Logic Synthesis for FPGAS*. Kluwer, New York, 1995.

[139] J.Ñoguera and R. Badia. Run-time HW/SW codesign for discret event systems using dynamically reconfigurable architectures. In *International Symposium on System Synthesis*, Madrid, September 2000. ACM/IEEE.

[140] G. Ochoa, I. Harvey, and H. Buxton. On recombination and optimal mutation rates. In *Proceedings of the 1999 Genetic and Evolutionary Computation Conference*, pages 488–495. Morgan Kaufmann, 1999.

[141] G. Ochoa, I. Harvey, and H. Buxton. Optimal mutation rates and selection pressure in genetic algorithms. In *Proceedings of the 2000 Genetic And Evolutionary Computation Conference*, pages 315–322. Morgan Kaufmann, 2000.

[142] J.V. Oldfield and R.C. Dorf. *Field Programmable Gate Arrays. Reconfigurable Logic for Rapid Prototyping.* John Wiley, Chinchester, UK, 1995.

[143] The OpenMP application program interface. http://www.openmp.org.

[144] I.C. Parmee, D. Cvetkovic, A.H. Watson, and C. R. Bonham. Multiobjective satisfaction within an interactive evolutionary design environment. *Evolutionary Computation*, 8(2):197–222, 2000.

[145] I.C. Parmee and A.H. Watson. Preliminary airframe design using co-evolutionary multi-objective genetic algorithms. In *Proceedings of the 1999 Genetic And Evolutionary Computation Conference*, pages 1657–1665. Morgan Kaufmann, 1999.

[146] M.D.F. Schlag P.K. Chan and J.Y. Zien. Spectral k-way ratio-cut partitioning and clustering. In *Proceedings of the ACM/IEEE 30st Design Automation Conference*, pages 749–754, 1993.

[147] M. Prieto. *Paralelizacion de Metodos Multimalla Robustos.* PhD thesis, Universidad Complutense de Madrid, 2000.

[148] J.M. Rabaey. *Digital Integrated Circuits and Systems. A Design Perspective.* Prentice-Hall, New Jersey, 1997.

[149] G. Rawlins. *Foundations of Genetic Algorithms.* Morgan Kaufmann, 1991.

[150] B.M. Riess, K. Doll, and F.M. Johannes. Partitioning very large circuits using anlytical placement techniques. In *Proceedings of the ACM/IEEE 31st Design Automation Conference*, pages 646–651, 1994.

[151] J. Rose, A. El Gamal, and A.Sangiovanni-Vicentelli. Architectures of FPGAs. *Proceedings of the IEEE*, 81(7):1013–1029, July 1993.

[152] G. Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE Transactions on Neural Networks*, 5(1):96–101, Enero 1994.

[153] L. Sanchis. Multiple-way network partitioning. *IEEE Transactions on Computers*, 38(1):62–81, April 1989.

[154] A. Sangiovanni-Vicentelli, A. El Gamal, and J. Rose. Syhntesis methods for FPGAs. *Proceedings of the IEEE*, 81(7):1057–1083, July 1993.

[155] V. Sankarsubramanian and D. Bhatia. Multiway partitioner for high performance FPGA based board architecture. In *Proccedings of the International Conference on Computer Design*, pages 579–585, 1996.

[156] H.P. Schwefel. *Evolutionsstrategie und numerische Optimierung*. PhD thesis, Uechnische Universitat Berlin, 1975.

[157] S.Hauck. The future of reconfigurable systems. In *5th Canadian Conference on Field Programmable Devices*, page Keynote Addres, 1998.

[158] S.Hauck, G. Borriello, S. Burns, and C. Ebeling. Springbok: A rapid prototyping system for board level design. In *ACM/SIGDA International Workshop on Field Programmable Gate Arrays*, 1994.

[159] S.Hauck, G. Borriello, and C. Ebeling. Mesh routing topologies for FPGAs arrays. In *ACM/SIGDA International Workshop on Field Programmable Gate Arrays*, 1994.

[160] S.Hauck, G. Borriello, and C. Ebeling. Mesh routing topologies for multi-FPGA systems. In *International Conference on Computer Design*, pages 170–177, 1994.

[161] N.A. Sherwani. *Algorithms for VLSI Physical Design Automation*. Kluwer Academic Publishers, 1993.

[162] S.J. and B.R. Moon. A hybrid genetic approach for multi-way graph partitioning. In *Proccedings of The Genetic and Evolutionary Computation Conference 2000*, pages 159–164, 2000.

[163] S.Trimberger. A reprogrammable gate array and applications. *Proceedings of the IEEE*, 81(7):1030–1040, July 1993.

[164] R. Tanese. Parallel genetic algorithms for a hypercube. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 177–183. L. Erlbaum Associates, 1987.

[165] R. Tanese. Distributed genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 434–440. M. Kaufmann, 1989.

[166] M. Tommassini. Parallel and distributed evolutionary algorithms: A review. In *Evolutionary Algorithms in Engineering and Computer Science*, pages 113–133. J. Wiley and Sons, 1999.

[167] R.L. Ukeiley. *FPGA´s . The 3000 series*. Prentice Hall, 1993.

[168] P.J.M. van Laarhoven and E.H.L. Aarts. *Simulated Annealing: Theory and Applications*. D.Reidel Publishing Company, 1987.

[169] J. Varghese, M. butts, and J. Batcheller. An efficient logic emulation system. *IEEE Transactions on Very Large Scale Integration Systems*, 1(2):171–174, June 1993.

[170] J. de Vicente. *Optimización del Diseño Físico de Circuitos Digitales orientado aDispositivos Reconfigurables*. PhD thesis, Universidad Complutense de Madrid, 2001.

[171] J. de Vicente and J. Lanchares. FPGA probabilitic placement avoiding routing congestion by evolution programs. In *International ICSC Symposium on Engineering of Intelligent Systems*, February 1998.

[172] J. de Vicente, J. Lanchares, and R. Hermida. RSR: A new rectilinear steiner minimum tree approximation for FPGA placement and global routing. In *Euromicro Conference*, pages 192–195, Vasteras, Sweden, September 1998. IEEE Press.

[173] J. de Vicente, J. Lanchares, and R. Hermida. Adaptative FPGA placement by natural optimization. In *Proceedings of the IEEE International Workshop on Rapid System Prototyping*, pages 188–193. IEEE, June 2000.

[174] M. D. Vose and G. E. Liepins. Schema disruption. In *Proc. of the Fourth International Conference on Genetic Algorithms*, pages 237–242, San Diego, CA, 1991.

[175] M.D. Vose. Modeling simple genetic algorithms. *Evolutionary Computation*, 3(4):453–472, 1995.

[176] M.D. Vose. *The Simple Genetic Algorithm. Foundations and Theory.* MIT Press, London, 1999.

[177] M.D. Vose and A.H. Wright. Simple genetic algorithms with linear fitness. *Evolutionary Computation*, 2(4):347–368, 1994.

[178] M.D. Vose and A.H. Wright. The simple genetic algorithm and the walsh transform: Part i, theory. *Evolutionary Computation*, 6(3):253–273, 1998.

[179] M.D. Vose and A.H. Wright. The simple genetic algorithm and the walsh transform: Part ii, the inverse. *Evolutionary Computation*, 6(3):275–289, 1998.

[180] K. Weiss, T. Steckstor, G. Koch, and W.Rosentiel. Exploiting FPGAs features during the emulation of a fast reactive embedded system. In *International Symposium on Field Programmable Gate Arrays*, pages 235–242. ACM Press, February 1999.

[181] M. Wermelinger. A hierachic architecture model for dynamic reconfiguration. In *Proceedings of the Secnd International Workshop on Software Engineering for Paralell and DIstributed Systems*, pages 243–254. IEEE Press, 1997.

[182] N.H.E. Weste and K. Eshragian. *Principles of CMOS VLSI Design. A Systems Perspective.* Addison-Wesley, Massachusetts, 1993.

[183] L.D. Whitley. *Foundations of Genetic Algorithms 2*. Morgan Kaufmann, 1993.

[184] L.D. Whitley and M.D. Vose. *Foundations of Genetic Algorithms 3*. Morgan Kaufmann, 1995.

[185] D.F. Wong, H.W. Leong, and C.L. Lin. *Simulated Annealing for VLSI Design*. Kluwer Academic, 1992.

[186] L. Zadeh. Similarity relations and fuzzy orderings. *Information Science*, 3:177–200, 1971.