DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSIDAD POLITÉCNICA DE VALENCIA

MÁSTER EN COMPUTACIÓN PARALELA Y DISTRIBUIDA

Verificación de Covering Arrays utilizando Supercomputación y Computación Grid

TESIS DE MÁSTER

Himer Avila George
Junio 2010, Valencia.

Directores:

Dr. Vicente Hernández García
Dr. José Torres Jiménez

Resumen

Verificación de Covering Arrays utilizando Supercomputación y computación Grid

Himer A. George

Máster en Computación Paralela y Distribuida
junio, 2010

Un covering array (CA) es una matriz C de tamaño $N \times k$, donde cada elemento $c_{i,j}$ toma como valor un símbolo del conjunto $S = \{1, 2, ..., v\}$, tal que cada $N \times t$ subarreglo contiene todas las posibles combinaciones de los v^t símbolos al menos una vez. Los covering arrays (CAs) han sido aplicados principalmente en el diseño de pruebas de software y hardware [89, 103, 110, 107].

En el proceso de construir CAs se requiere verificar que cada uno de los $N \times t$ subarreglos contienen al menos una vez las v^t combinaciones de símbolos. En esta tesina de máster se presenta un algoritmo para validar CAs, y sus implementaciones en Secuencial (AS), paralelo (AP) y Grid (AG). El algoritmo ha sido probado usando un benchmark formado por 25 CAs de fuerza variable. La conclusión principal de este trabajo es que el proceso de verificación de CAs es naturalmente paralelizable, de tal manera que puede beneficiarse no solo de la computación paralela a nivel de un clúster de computadoras; sino que además, puede descomponerse en una serie de trabajos totalmente independientes que pueden ejecutarse en un entorno Grid.

Índice general

Resumen I

Siglas XI

1 Introducción 1
- 1.1 Diseño de experimentos . . . 1
  - 1.1.1 Conceptos básicos . . . 2
  - 1.1.2 Diseños clásicos . . . 2
  - 1.1.3 Diseños combinatorios . . . 4
- 1.2 Motivación de la Tesis de Máster . . . 8
  - 1.2.1 Diseño de pruebas de software . . . 8
  - 1.2.2 Estrategias para construir pruebas combinatorias . . . 9
- 1.3 Planteamiento del problema . . . 11
  - 1.3.1 Descripción del problema . . . 11
  - 1.3.2 Complejidad del problema . . . 12
  - 1.3.3 Trabajo a realizar . . . 12
- 1.4 Objetivos generales y específicos . . . 13
  - 1.4.1 Objetivo general . . . 13
  - 1.4.2 Objetivos específicos . . . 13
  - 1.4.3 Metodología a seguir . . . 13
- 1.5 Estructura del documento . . . 13

2 Covering Arrays: estado del arte 15
- 2.1 Definiciones y ejemplos . . . 15
- 2.2 Construcción de Covering Arrays óptimos en tiempo polinomial . . . 20
  - 2.2.1 Caso: CA(N; k, 2, 2) . . . 20
  - 2.2.2 Caso: $CA(v^t; v+1, v, t)$ . . . 21
- 2.3 Covering Arrays Cíclicos . . . 21
- 2.4 Transformaciones algebraicas . . . 23
  - 2.4.1 TConfig . . . 23
  - 2.4.2 Combinatorial Test Services . . . 23
- 2.5 Estrategias deterministas . . . 23
  - 2.5.1 Automatic Efficient Test Generator . . . 23
  - 2.5.2 In-Parameter-Order (IPO) . . . 24
  - 2.5.3 Test Case Generation . . . 24
  - 2.5.4 Deterministic Density Algorithm . . . 25
- 2.6 Estrategias no deterministas . . . 25
  - 2.6.1 Recocido Simulado . . . 25
  - 2.6.2 Algoritmo del Diluvio Universal . . . 25

2.6.3 Búsqueda Tabú 26
2.6.4 Algoritmos Genéticos 26
2.6.5 Colonia de Hormigas 27
2.7 Resumen del capítulo 27

3 Supercomputación y Computación GRID 29
3.1 Conceptos generales 29
3.2 Computación Paralela 30
3.2.1 Clasificación de las computadoras paralelas 31
3.2.2 Principales Arquitecturas Paralelas 32
3.2.3 Modelos de programación 34
3.2.4 Diseño de algoritmos paralelos 36
3.2.5 Aceleración y eficiencia 38
3.3 Computación Grid 42
3.3.1 Antecedentes y perspectivas de la computación Grid 42
3.3.2 Tecnología Grid 43
3.3.3 Arquitecturas Grid 44
3.3.4 Open Grid Services Architecture 45
3.3.5 Open Grid Service Infraestructure 46
3.3.6 Web Services Resource Framework 46
3.3.7 Diferencias entre OGSI y WSRF 46
3.3.8 Uniendo las partes 47
3.4 Resumen del capítulo 47

4 Metodología propuesta 49
4.1 Proceso para validar CAs 49
4.2 Validando un $MCA(6; 4, 2^3\ 3^1, 2)$ 50
4.3 Validando un CCA(7;7,2,2) 52
4.3.1 Automorfismo aditivo 52
4.3.2 Automorfismo multiplicativo 53
4.4 Algoritmos propuestos 54
4.4.1 Algoritmo secuencial (AS) 54
4.4.2 Algoritmo paralelo (AP) 56
4.4.3 Algoritmo Grid (AG) 58
4.5 Adaptaciones para CCA 59
4.6 Resumen del capítulo 59

5 Experimentación y Resultados 61
5.1 Experimentación 61
5.2 Resultados 61
5.2.1 Coverings Arrays Generales 61
5.2.2 Coverings Arrays Cíclicos 64
5.3 Resumen del capítulo 65

6 Conclusiones y trabajos futuros 67
6.1 Conclusiones 67
6.2 Trabajos futuros 67
6.3 Publicaciones 68
6.3.1 Aceptadas 68

6.3.2 En proceso 68

A Algoritmos **71**

B Recursos computacionales utilizados **75**
B.1 Tirant 75
B.2 Odin 76
B.3 Enabling Grids for E-SciencE 77

Referencias **86**

Lista de figuras

1.1 Proceso como caja negra. 1
1.2 Estrategias combinatorias para construir pruebas. 10

2.1 Grafo: colonia de hormigas 27

3.1 Taxonomía de Flynn. 31
3.2 Modelo de memoria compartida. 32
3.3 Modelo de memoria distribuida. 33
3.4 Aceleración en términos del porcentaje de código paralelizable. 39
3.5 Eficiencia η para p procesadores y los distintos porcentajes de código paralelizable. 41
3.6 Globus Toolkit 4 47

5.1 Análisis para $t = 2$ y $k \leq 1024$. 62
5.2 Análisis para $t = 3$ y $k \leq 1024$. 63
5.3 Análisis para $t = 4$ y $k \leq 1024$. 63
5.4 Análisis para $t = 4$ y $k \leq 1024$. 63
5.5 Análisis para $t = 4$ y $k \leq 1024$. 64
5.6 Ejemplo del rendimiento del AG cuando $t = 4$. 65

B.1 Tirant. 75
B.2 Clúster Odin. 76

Lista de tablas

1.1 Diseño factorial completo 2^k. 3
1.2 Diseño factorial fraccionado 2^{k-p}. 4
1.3 Orthogonal Array de fuerza 2. 5
1.4 Ejemplo de un covering array de fuerza 3. 5
1.5 Sistema con cuatro componentes con dos valores cada uno. 6
1.6 Conjunto de pruebas que cubre los pares de componentes de la *tabla 1.5*. 6
1.7 Interpretación del conjunto de pruebas de la tabla 1.6. 6
1.8 $MCA(12;\ 4,\ 4^1\ 3^2\ 2^1,\ 2)$. 7
1.9 Ejemplo de un MCA. 7
1.10 Interpretación del conjunto de pruebas de la tabla 1.9(b). 8

2.1 Latin square de orden 4. 16
2.2 Tres ejemplos de *latin squares*. 16
2.3 Combinación de los *latin squares* 1 y 2 mostrados en la *tabla 2.2*. 16
2.4 Combinación de los *latin squares* 1 y 3 mostrados en la *tabla 2.2*. 16
2.5 Tres *Orthogonal Latin Squares* de orden 4. 17
2.6 $OA_2(4,\ 2,\ 2)$. 17
2.7 Construcción de un $OA(5,\ 4,\ 2)$. 18
2.8 $CA(5; 4, 2, 2)$. 18
2.9 $MCA(9; 5, 2^3\ 3^2, 2)$. 19
2.10 Construcción de un $CA(6;\ 10,\ 2,\ 2)$. 21
2.11 Una CM de tamaño $k \times k$ creada a partir del vector $(x_1,\ x_2,\ \ldots,\ x_k)^T$. 22
2.12 Un $CCA(7; 7, 2, 2)$. 23

3.1 Aceleración en términos del porcentaje de código paralelizable. 39
3.2 Eficiencia η para p procesadores y los distintos porcentajes de código paralelizable. 41

4.1 $MCA(6; 4, 2^3\ 3^1, 2)$. 50
4.2 Inicialización de variables. 50
4.3 Validando un MCA, *t-ada* {0,1}. 51
4.4 Validando un MCA, *t-ada* {0,2}. 51
4.5 Validando un MCA, *t-ada* {0,3}. 51
4.6 Validando un MCA, *t-ada* {1,2}. 51
4.7 Validando un MCA, *t-ada* {1,3}. 51
4.8 Validando un MCA, *t-ada* {2,3}. 52
4.9 Validación del MCA 52
4.10 Un $CCA(7; 7, 2, 2)$. 52
4.11 Conjunto Λ para validar un $CCA(7; 7, 2, 2)$ completamente. 52
4.12 Ejemplo de automorfismo aditivo. 53
4.13 Conjunto Δ para validar el $CCA(7; 7, 2, 2)$ usando automorfismos aditivos. 53

4.14 Residuos cuadráticos para el caso $CCA(7;7,2,2)$. 54
4.15 Ejemplo de automorfismo multiplicativo. 54
4.16 Conjunto $\mathcal{C}$ para validar el $CCA(7;7,2,2)$ usando automorfismos multiplicativos. 54
4.17 Optimizaciones. 55
4.18 Ejemplo $CA(10;\ 7,\ 2,\ 4)$, combinaciones de $\binom{k}{t}$. 57
4.19 Ejemplo $CA(10;7,2,4)$, t-adas iniciales de cada procesador. 57
4.20 Ejemplo $CA(10;7,2,4)$, t-adas asignadas a cada procesador. 58

5.1 CAs utilizados en la experimentación. 62
5.2 Comparación de SA y GA para CA. 64
5.3 Comparación de SA y GA para CCA. 65

Lista de Algoritmos

A.0.1 TestCA, algoritmo secuencial para validar CA. 71

A.0.2 Siguiente_t-ada. 72

A.0.3 Algoritmo secuencial por columnas para validar CA. 72

A.0.4 Algoritmo secuencial por columnas con modificación de expresiones. 73

A.0.5 Calcular la $t - ada$ inicial. 73

A.0.6 Algoritmo optimizado para calcular la $t - ada$ inicial. 74

A.0.7 TestCA_Paralelo. 74

A.0.8 TestCA_Grid 74

Siglas

AC All Combinations. 10
ACA Ant Colony Algorithm. 10, 27
AETG Automatic Efficient Test Generator. 10, 23, 24
AG Algoritmo Grid. 59, 64, 65
AP Algoritmo Paralelo. 57, 64
API Application Programming Interface. 44
AR Anti Random. 10
AS Algoritmo Secuencial. 55, 64, 65

BC Based Choice. 10

CA Covering Arrays. 5, 7, 8, 10–13, 15, 18–26, 28, 48–50, 54, 55, 57, 59, 61, 62, 64, 65, 67
CAN Covering Array Number. 20
CAP Computación de Altas Prestaciones. 30
CATS Combinatorial Algorithms Test Sets. 10
CCA Cyclic Covering Array. 21, 22, 52, 64, 65
CERN Consejo Europeo para la Investigación Nuclear. 43
CM Cyclic Matrix. 21, 52
CP control parallelism. 56
CTS Combinatorial Test Services. 23

DDA Deterministic Density Algorithm. 25
DOE Diseño de Experimentos - Design of Experiments. 1, 4, 11, 23, 67
DP data parallelism. 56, 57, 62

EC Each Choice. 10
EDG European DataGrid. 43
EGEE Enabling Grids for E-SciencE Project. 77
EP Elemento de Proceso. 32–34

GA Genetic Algorithms. 10, 26
GDA Great Deluge Algorithm. 25
GF Galois Fields. 21, 22
GGF Global Grid Forum. 46

HPC High Performance Computing. 56, 57
HTC High Throughput Computing. 56, 62

IDL Interface Definition Language. 45
IPG Information Power Grid. 43

IPO In-Parameter-Order. 10, 24
IST Information Society Technologies. 43

LS Latin Squares. 15–17

MCA Mixed Covering Arrays. 7, 9, 11, 19, 20, 50, 51
MIMD Multiple Instruction Multiple Data. 32
MISD Multiple Instruction Single Data. 32
MOLS Mutually Orthogonal Latin Squares. 17, 21
MPI Message Passing Interface. 35–37

NIST National Institute for Standards and Technology. 8
NSF Nacional Science Foundation. 43

OA Orthogonal Arrays. 4, 5, 7, 10, 11, 15, 17–19, 21, 23
OASIS Organization for the Advancement of Structured Information Standards. 46
OGF Open Grid Forum. 45
OGSA Open Grid Services Architecture. 45–47
OGSI Open Grid Service Infraestructure. 45–47
OLS Orthogonal Latin Squares. 17

PPW Partly Pair-Wise. 10
PVM Parallel Virtual Machine. 36, 37

SA Simulated Annealing. 10, 25
SDK Software Development Kit. 44
SIMD Single Instruction Multiple Data. 32
SISD Single Instruction Single Data. 31
SOAP Simple Object Access Protocol. 45

TCG Test Case Generation. 24
TS Tabu Search. 26

VO Virtual Organization. 43–45

WS Web Services. 45–47
WSDL Web Services Description Language. 45
WSRF Web Services Resource Framework. 45–47

Capítulo 1

Introducción

La experimentación juega un papel fundamental en la mayoría de las investigaciones científicas y de la industria, en muchas de las cuales, los resultados del proceso de interés se ven afectados por la presencia de distintos factores, cuya influencia puede estar oculta por la variabilidad de los resultados muestrales.

El objetivo de la experimentación es obtener información de calidad, la cual permita: desarrollar nuevos productos y procesos, comprender mejor un sistema y tomar decisiones sobre como optimizarlo y mejorar su calidad.

Un proceso puede considerarse como una caja negra *(figura 1.1)* a la cual se ingresan diversos factores que interactúan para producir un resultado. Los factores que ingresan al proceso se denominan *factores de entrada*, y el resultado, *factor de salida*. El nivel del *factor de salida* depende de los niveles que adopten los *factores de entrada* por lo cual es de vital importancia saber qué combinación de *factores de entrada* producen el *factor de salida óptimo*.

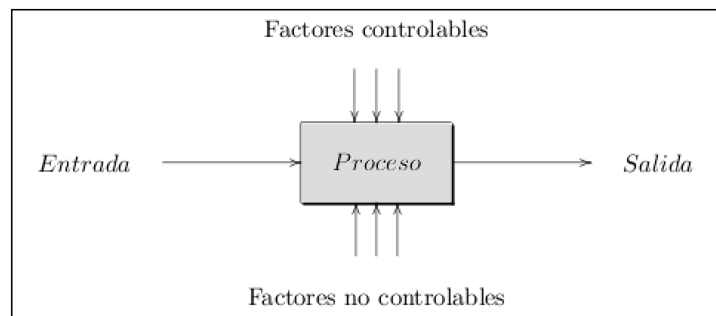


Figura 1.1: Proceso como caja negra.

La búsqueda de combinaciones óptimas de los factores de entrada da lugar al diseño de experimentos.

1.1. Diseño de experimentos

El *Diseño de Experimentos - Design of Experiments (DOE)* está conformado por modelos estadísticos cuyo objetivo es averiguar si unos determinados factores influyen en una variable de interés y, si existe influencia de algún factor, poder cuantificarla.

El diseño de experimentos tiene sus orígenes en los trabajos de Ronald A. Fisher (1890-1962) [32], desarrollados en la Estación Agrícola Experimental de Rothamsted en las cercanías de Londres, Inglaterra. De 1919 a 1925 estudió y analizó experimentos relativos al trigo que se habían realizado desde 1843. Como resultado de sus investigaciones estadísticas de éstos y otros experimentos, Fisher desarrolló el análisis de varianza y unificó sus ideas básicas sobre el diseño de experimentos.

1.1.1. Conceptos básicos

Montgomery [69] define el diseño de experimentos como una prueba o conjunto de pruebas en las que se hacen cambios deliberados en los factores de entrada de un proceso o sistema para observar he identificar las razones de los cambios que pudieran observarse en la respuesta de la salida.

Los *niveles* son el número de alternativas o ajustes para cada factor. Los *factores* son las variables de interés para las cuales se quiere estudiar el impacto que tienen las mismas en la respuesta.

Los *factores de salida* son las variables respuesta del experimento. La respuesta puede ser univariada (una sola salida de interés) o multivariada (múltiples salidas de interés). Estas pueden clasificarse en variables cualitativas y cuantitativas. Se clasifican como cualitativas cuando por ejemplo: se refiere a características, donde la respuesta es un "si" o un "no" (cuando se desea saber si un producto es aceptable o no de acuerdo a características observadas, o cuando se tienen en cuenta las características de una persona para tomar una decisión). Se clasifican como cuantitativas cuando se mide algo numérico como el tamaño, el tiempo, el peso, etc.

Dos factores F_i y F_j con I y J niveles, respectivamente, son *ortogonales* si en cada nivel i de F_i el número de observaciones de los J niveles de F_j están en las mismas proporciones. Esta propiedad permite separar los efectos simples de los factores en estudio.

Por último, los *tratamientos* son el conjunto de circunstancias creadas para el experimento, en respuesta a la hipótesis de investigación.

1.1.2. Diseños clásicos

Fisher [32] observó que el proverbio más repetido respecto a los experimentos de campo era el que decía:

> *"Debemos hacer algunas preguntas a la naturaleza, y de preferencia, una a la vez".*

Fisher hacia alusión al método tradicional de *variar un factor a la vez*, el cual consiste en seleccionar un punto de partida de cada factor, es decir, se definen unas condiciones iniciales para realizar el experimento. Luego, se va modificando el valor de un solo factor, manteniendo los demás fijos. Posteriormente se modifica el valor del siguiente factor y sucesivamente se va modificando el valor de cada factor. El inconveniente de este método es que no se pueden observar las interacciones entre factores. Se dice que existe *interacción* entre dos factores A y B cuando el efecto del factor A es diferente según qué valor tome el factor B, y viceversa. Por lo tanto este método sólo proporciona información limitada.

Volviendo al artículo de Fisher [32] el comentó: *"La naturaleza responderá mejor a un cuestionario concebido lógica y cuidadosamente; en realidad, si le planteamos una sola pregunta, rehusará responder hasta que se analice algún otro tema".*

Fisher entendía que en los sistemas naturales no se sabe si la influencia de un tratamiento es independiente de otra o si su influencia se relaciona con la variación de otros tratamientos. En consecuencia, las condiciones en las que se comparan los tratamientos pueden ser aspectos importantes del diseño, por lo tanto es necesario probar mas de un factor a la vez.

Para probar más de un factor a la vez, se desarrolló un tipo especial de diseño de tratamientos, conocido como *diseño factorial*. Este diseño consiste en realizar todas las combinaciones posibles de los niveles de varios factores, siguiendo la *expresión 1.1*. Con frecuencia, los experimentos con diseños factoriales se conocen como *factoriales* o *experimentos factoriales*.

$$n = \prod_{i=1}^{k} m_i \tag{1.1}$$

Los *diseños factoriales* producen experimentos más eficientes (que el método tradicional de modificar un factor a la vez), pues cada observación proporciona información sobre todos los factores, y es factible ver las respuestas de un factor en diferentes niveles de otro factor en el mismo experimento. La respuesta a cualquier factor observado en diferentes condiciones indica si los factores actúan en las unidades experimentales de manera independiente. La interacción entre factores ocurre cuando su actuación no es independiente.

Los *diseños factoriales* 2^k son diseños en los que se trabaja con k factores, todos ellos con dos niveles (se suelen denotar + y -). Estos diseños son útiles para realizar estudios preliminares con muchos factores para identificar los más importantes y sus interacciones, en la *tabla 1.1* se muestra un ejemplo de este tipo.

Tabla 1.1: Diseño factorial completo 2^k.

Combinación	X_1	X_2	X_3
1	-	-	-
2	+	-	-
3	-	+	-
4	+	+	-
5	-	-	+
6	+	-	+
7	-	+	+
8	+	+	+

Sin embargo, si k es grande, el número de observaciones que necesita el diseño factorial 2^k es muy grande (n = 2^k), una alternativa es utilizar *fracciones factoriales*.

Las *fracciones factoriales* son diseños con k factores a dos niveles, que mantienen la propiedad de ortogonalidad de los factores y donde se suponen nulas las interacciones de orden alto por lo que para su estudio solo se necesitan 2^{k-p} observaciones.

Los *diseños factoriales fraccionarios* usan sólo la mitad, la cuarta parte o incluso una fracción menor de las 2^k combinaciones de tratamientos y se usan por una o varias de las siguientes razones: el número de tratamientos necesario excede a los recursos, sólo se requiere información sobre los efectos principales y las interacciones de bajo orden, se necesitan estudios exploratorios para muchos factores, o se hace la suposición de que sólo unos cuantos efectos son importantes.

En la *tabla 1.2* se puede ver un diseño factorial fraccional 2^{k-p} para $k = 3$ y $p = 1$, donde se observa un diseño con sólo 4 combinaciones.

Tabla 1.2: Diseño factorial fraccionado 2^{k-p}.

Combinación	X_1	X_2	X_3
1	-	-	+
2	+	-	-
3	-	+	-
4	+	+	+

En los últimos años han alcanzado una gran popularidad en la industria los enfoques del DOE desarrollados en Japón por Genichi Taguchi [100]. Para la implementación de sus ideas de Diseño Robusto [81], una filosofía de calidad orientada hacia la consecución de productos y procesos que sean poco sensibles a la presencia de causas de variabilidad, Taguchi propone la utilización generalizada del DOE en las fases de diseño de productos y procesos.

Taguchi [101] diseñó un conjunto de tablas y gráficos para el *diseño de experimentos fraccionado*, pensadas para simplificar la tarea de diseñar experimentos altamente fraccionados sin necesidad de conocer los fundamentos teóricos de los mismos. Dichas tablas son planes factoriales altamente fraccionados a las que se les conoce como Orthogonal Arrays (OA).

Algunas de estos diseños clásicos han sido aplicados al diseño de pruebas de software [30, 5].

1.1.3. Diseños combinatorios

Además de las técnicas de diseño de experimentos clásicas, el DOE se ha visto beneficiado de una amplia parte de la literatura que describe varios objetos matemáticos conocidos como *diseños combinatorios*, los cuales cuentan con las propiedades necesarias para definir óptimamente una serie de experimentos. Por ejemplo, si se quiere probar todas las interacciones en parejas de parámetros o en combinaciones de n parámetros de un sistema, entonces el problema puede ser mapeado a un objeto matemático conocido como OA.

Orthogonal Arrays

La definición de un OA es simple y se puede obtener de forma natural (*véase la página 17* para obtener un definición formal), por ejemplo, en la *tabla 1.3* se muestra un OA de fuerza (grado de interacción) 2. Si se toma cualquier par de columnas, por ejemplo las últimas 2, cada una de las 4 posibles combinaciones {0,0}, {0,1}, {1,0}, {1,1} aparecerán allí el *mismo número de veces* (de hecho aparecen exactamente 3 veces), estás son las características que se deben cumplir para crear OAs.

Para usar los OA en el DOE se pueden remplazar los 0's y 1's de la primer columna por "Blanco" y "Obscuro", los" de la segunda por "con azúcar" y "sin azúcar", los de la tercera por "caliente" y "frío" y así sucesivamente hasta remplazar todos los 0's y 1's de las 11 columnas, dependiendo de la aplicación.

Cuando solo hay 0's y 1's en un OA entonces se le conoce como *OA de nivel 2*. En la *tabla 1.3* hay 11 columnas, lo cual significa que se pueden variar los niveles de hasta 11 *variables diferentes*, hay 12 renglones, lo que significa que hay 12 maneras de preparar un té o 12 diferentes casos de prueba.

Tabla 1.3: Orthogonal Array de fuerza 2.

0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	1	0	1	0	0	0	1
1	0	1	1	0	1	0	0	0	1	1
0	1	1	0	1	0	0	0	1	1	1
1	1	0	1	0	0	0	1	1	1	0
1	0	1	0	0	0	1	1	1	0	1
0	1	0	0	0	1	1	1	0	1	1
1	0	0	0	1	1	1	0	1	1	0
0	0	0	1	1	1	0	1	1	0	1
0	0	1	1	1	0	1	1	0	1	0
0	1	1	1	0	1	1	0	1	0	0
1	1	1	0	1	1	0	1	0	0	0

En resumen, definimos este OA de la siguiente manera: $OA(12;\ 11,\ 2,\ 2)$. Donde el primer 2 significa el número de diferentes valores (nivel) que se le pueden asignar a cada columna, el segundo 2 hace referencia a la "fuerza", es decir el número de columnas donde aparecerán todas las posibles combinaciones el mismo número de veces.

Aunque existen muchas aplicaciones donde los OA [49, 65, 101] pueden ser utilizados, el hecho de que deben estar balanceados (los valores de cada parámetro deben aparecer el mismo número de veces) hace que sean demasiado restrictivos para algunas aplicaciones como el diseño de pruebas de software y hardware. Por lo cual, muchas veces es necesario recurrir a otros objetos matemáticos menos restrictivos conocidos como *Covering Arrays*.

Covering Arrays

Los Covering Arrayss (CAs) son objetos combinatorios que a diferencia de los OA, no necesitan estar balanceados, es decir en un CA que satisface una cobertura de fuerza t, cada posible combinación de valores de variables (t-*ada*) debe estar presente al menos una vez pero no necesariamente el mismo número de veces, en la *tabla 1.4(a)* se muestra un ejemplo de un CA de *fuerza* 3.

Tabla 1.4: Ejemplo de un covering array de fuerza 3.

(a)

0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1

(b)

t-adas	C_8	C_9	C_{10}	**Ocurrencias**
1	0	0	0	2
2	0	0	1	2
3	0	1	0	2
4	0	1	1	1
5	1	0	0	2
6	1	0	1	1
7	1	1	0	1
8	1	1	1	2

Si se toma cualquier terna de columnas, cada una de las 8 combinaciones {0,0,0}, {0,0,1}, {0,1,0}, {0,1,1}, {1,0,0}, {1,0,1}, {1,1,0}, {1,1,1} debe aparecer al menos una vez. Por ejemplo, si se toman las últimas 3 columnas (véase la *tabla 1.4(b)*) se tiene que cada una de las 8 *t-adas* aparece al menos una vez.

Para representar estos objetos combinatorios se usa la siguiente notación:

$$CA(N;\ k,\ v,\ t)$$

N	Es el número de experimentos o pruebas.
k	Es el número de factores.
v	Es el número de símbolos por cada factor, denominado como *alfabeto*.
t	Es el grado de interacción entre los factores, denominado como *fuerza*.

Ejemplo de un CA de alfabeto uniforme y fuerza 2

Considere el sistema de la *tabla 1.5*. Hay dos piezas de hardware, dos sistemas operativos, dos conexiones de red y dos configuraciones de memoria. En caso de existir una interacción entre pares de componentes, entonces los valores para la construcción de un conjunto de pruebas son: k=4, v=2 t=2.

Tabla 1.5: Sistema con cuatro componentes con dos valores cada uno.

	Hardware	Sistema operativo	Conexión de red	Memoria
0	PC	Windows	Dial-up	128 MB
1	Laptop	Linux	Cable	256 MB

Si los posibles valores que pueden tomar los componentes se etiquetan con 0,1 entonces un conjunto de pruebas que cubre todas las combinaciones entre pares de componentes se muestra en la *tabla 1.6*. Este es un ejemplo de un $CA(5; 4, 2, 2)$. Nótese que en cada par de columnas aparecen las combinaciones {0,0}, {0,1}, {1,0} y {1,1} al menos una vez.

Tabla 1.6: Conjunto de pruebas que cubre los pares de componentes de la *tabla 1.5*.

1	1	1	0
0	0	0	0
0	1	0	1
1	0	0	1
0	0	1	1

Al sustituir los números por la configuración correspondiente de cada componente, el conjunto de pruebas quedaría como se indica en la *tabla 1.7*.

Tabla 1.7: Interpretación del conjunto de pruebas de la tabla 1.6.

Prueba	Hardware	SO	Conexión	Memoria
1	Laptop	Linux	Cable	128 MB
2	PC	Windows	Dial-up	128 MB
3	PC	Linux	Dial-up	256 MB
4	Laptop	Windows	Dial-up	256 MB
5	PC	Windows	Cable	256 MB

Estos objetos combinatorios son fundamentales en el diseño de pruebas de interacción cuando todos los factores tienen el mismo número de niveles (*alfabeto uniforme*). Sin embargo, muchos de los problemas en los diversos campos de la ingeniería no tienen todos sus factores con el mismo número de niveles. Por lo tanto, para poder hacer frente a estos problemas se debe hacer uso de los *Covering Arrays de alfabeto Mixto o Heterogéneo*.

Mixed Covering Arrays

Un Mixed Covering Arrays (MCA) a diferencia de los CA de alfabeto uniforme, permite como su nombre indica, tener factores con diferentes niveles. Lo cual permite aplicarlos a muchos campos de la ingeniería donde no es necesario que los niveles de los factores sean iguales, por ejemplo el diseño de pruebas de software y hardware. Para representar estos objetos combinatorios se usa la siguiente notación:

$$MCA(N;\ k,\ \prod_{i=1}^{k} v_i,\ t)$$

En la *tabla 1.8* se muestra un ejemplo de un MCA con 12 experimentos, 4 factores, el primero de ellos de nivel 4, los siguientes 2 de nivel 3 y el último factor de nivel 2, por último la fuerza es de 2. Este *mixed covering array* se representa de la siguiente manera: $MCA(12;\ 4,\ 4^1\ 3^2\ 2^1,\ 2)$.

Tabla 1.8: $MCA(12;\ 4,\ 4^1\ 3^2\ 2^1,\ 2)$.

0	0	0	0
0	1	2	1
0	2	1	0
1	0	2	0
1	1	1	1
1	2	0	1
2	0	2	1
2	1	0	0
2	2	1	1
3	0	1	1
3	1	0	1
3	2	2	0

Ejemplo 3 (Mixed covering array)

En los sistemas del mundo real el número de valores posibles de cada parámetro suele ser distinto. Para explicar esta situación considérese el siguiente ejemplo: supóngase una aplicación con 4 componentes, el primero con tres posibles configuraciones y el resto con dos (tal como se indica en la *tabla 1.9(a)*). Etiquetando las configuraciones del primer componente con 0,1,2 y la de los tres restantes como 0,1; un conjunto de pruebas que cubre todas las combinaciones entre pares de componentes se muestra en la *tabla 1.9(b)*.

Tabla 1.9: a) Ejemplo de un sistema con cuatro componente donde no todos los componentes tienen el mismo número de valores diferentes. b) Conjunto de pruebas que cubre todos los pares de componentes.

(a)

Cliente	Servidor web	Pago	BDD
Firefox	Apache	MasterCard	Oracle
IE	.NET	Visa	MySQL
Netscape	—	—	—

(b)

0	0	0	0
0	1	1	1
1	0	0	1
1	1	1	0
2	0	1	0
2	1	0	1

La tabla 1.9(b) es un ejemplo de un *mixed covering array* $MCA(6;\ 4,\ 3^1\ 2^3,\ 2)$ donde N=6, k=4, t=2 y la cardinalidad del primer componente es 3 y la del resto 2. A diferencia de los *OA* y los *CA*, en los *MCA* cada componente tiene una cardinalidad distinta.

Si se sustituye cada número por la configuración correspondiente, el conjunto de pruebas que se obtendría se muestra en la tabla 1.10.

Tabla 1.10: Interpretación del conjunto de pruebas de la tabla 1.9(b).

Prueba	Cliente	Servidor web	Pago	BDD
1	Firefox	Apache	MasterCard	Oracle
2	Firefox	.NET	Visa	MySQL
3	IE	Apache	MasterCard	MySQL
4	IE	.NET	Visa	Oracle
5	Netscape	Apache	Visa	Oracle
6	Netscape	.NET	MasterCard	MySQL

1.2. Motivación de la Tesis de Máster

Hedayat *et ál.* [49], comentan que los modelos combinatorios son usados principalmente para el diseño de experimentos, siendo ampliamente utilizados en disciplinas como: la medicina, agricultura y manufactura. Meagher y Stevens [67], reportan que estos objetos (CA) son utilizados en el área de las matemáticas en las teorías de búsqueda y las funciones de verdad. En los últimos años estos objetos combinatorios se están aplicando con éxito al diseño de pruebas de hardware y software [8, 59, 114, 60, 82].

1.2.1. Diseño de pruebas de software

La realización de pruebas es una pieza clave en el desarrollo del software, ya que de esta manera se verifica su funcionalidad y, por ende, se identifican fallos para posteriormente corregir los errores de programación y evitar que se originen situaciones desfavorables en el momento de su operación. A pesar de la aportación de las pruebas en la mejora de la calidad del software, evaluar todos los casos posibles es usualmente irrealizable [20], por ejemplo, si un sistema tiene 10 parámetros con 5 valores admisibles, se generarían $5^{10} = 9{,}765{,}625$ de casos diferentes para cubrir todos los escenarios.

Golumbic y Hartman [43] destacan que el proceso de pruebas resulta ser muy costoso, comentan que abarca al menos el 50 % del costo de desarrollo de un nuevo componente de software, Hnich *et ál.* [52] concuerdan con esta opinión al mencionar que, en ocasiones, las pruebas consumen más de la mitad del costo total del desarrollo. Concerniente a este tema, el National Institute for Standards and Technology (NIST) [104] reportó que los defectos de software afectan la economía de Estados Unidos con 60 billones de dólares al año. Ellos estiman que aproximadamente 22 billones de dólares podrían ser reducidos mediante pruebas más efectivas.

Teniendo como base lo que se ha comentado y añadiendo el hecho de que un significativo número de fallos son causados por interacciones de tamaño pequeño (involucrando parejas, ternas, etc.) de parámetros [60], los criterios combinatorios son un enfoque aceptable que influye en el costo de las pruebas y el grado requerido de cobertura [113], incluso este planteamiento ha sido utilizado en la reducción de costos en la industria [59] siendo los CA un modelo muy utilizado. Otro beneficio aportado por las pruebas combinatorias es su alto nivel de cobertura [59], es decir, todas las interacciones de un cierto tamaño son cubiertas con este método, siendo más factible encontrar un alto porcentaje de los errores [60].

Por otra parte, Ronnesseth y Colbourn [87] enfatizan que los CA gozan de una amplia gama

de aplicaciones, entre las cuales se encuentran: interacciones entre las entradas del diseño del hardware, experimentos de resistencia de materiales y primordialmente las pruebas de software.

Considerando que los sistemas de software tienen generalmente parámetros con cardinalidad distinta, es importante construir conjuntos de prueba usando MCA, y dado que el tamaño de la interacción entre parámetros que producen todos los errores es variable (se reporta que puede ir de 2 a 6 [60]), elaborar un algoritmo para la construcción de MCA de fuerza variable resultaría sustancialmente benéfico en lo concerniente a las pruebas de software [92], ya que el conjunto de pruebas generado sería considerablemente menor que el exhaustivo, reflejando esto en la disminución del costo total del software y teniendo además el grado de cobertura requerido.

1.2.2. Estrategias para construir pruebas combinatorias

Las estrategias de combinación son métodos de selección de casos de prueba que construyen los casos de prueba combinando los valores de los diferentes objetos, que se pasan como parámetros mediante alguna estrategia de combinación.

Criterios de cobertura de casos de prueba

Se pueden definir criterios de cobertura para los valores de los parámetros de los casos de prueba en función de su "interés". A continuación se presentan algunos criterios de cobertura para valores.

- ***each-used, o 1-wise:*** Es el criterio más simple. Se satisface cuando cada valor de cada parámetro se incluye, al menos, en un caso de prueba.
- ***pair-wise (o 2-wise):*** Este criterio requiere que cada posible par de valores interesantes de cualesquiera dos parámetros sea incluido en algún caso de prueba.
- ***t-wise:*** Es una extensión del 2-wise, en la que cada combinación posible de valores interesantes de los t parámetros se incluye en algún caso de prueba.
- ***N-wise:*** Es un caso especial de t-wise: siendo N el número de parámetros, N-wise se satisface cuando todas las combinaciones posibles de todos los parámetros se incluyen en casos de prueba. Es decir, en t-wise, t=N.

Grindal *et ál.* [45], elaboraron un análisis de más de 40 artículos publicados entre 1985 y 2004 encontrando 16 diferentes estrategias combinatorias enfocadas en las pruebas de software. La *figura 1.2* muestra una clasificación de dichas estrategias.

Estrategias no deterministas

En este tipo de estrategias, el azar cumple siempre un papel importante en la determinación de qué casos de prueba se generarán. Por lo tanto, no se asegura que dos ejecuciones del mismo algoritmo produzcan siempre los mismos resultados.

Como se muestra en la *figura 1.2* la estrategia combinatoria no determinista más simple es la aleatoriedad pura.

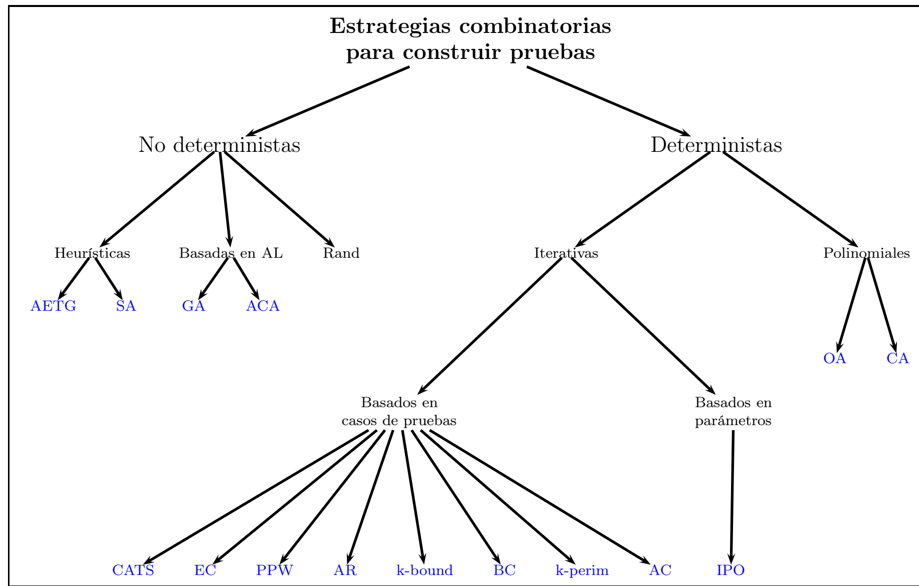


Figura 1.2: Estrategias combinatorias para construir pruebas.

Los *algoritmos genéticos* constituyen un método de resolución de problemas de minimización. Parten de una solución inicial, que se va aproximando a la óptima según una serie de iteraciones. De la solución inicial van construyéndose soluciones mejores de acuerdo con una *función objetivo (fitness)*.

El *algoritmo de colonia de hormigas* [28] representa los casos de prueba en un grafo. El grafo es recorrido por un conjunto de hormigas. Cuando una hormiga alcanza el nodo objetivo, se deposita en cada arco de los que ha visitado una cantidad de feromonas proporcional a la calidad de la solución. Si una hormiga tiene que elegir entre diferentes arcos, se va por aquel que tenga más feromonas.

Como al principio todos los arcos no tienen feromonas, se calcula una heurística para cada arco, que no cambia durante el algoritmo, según la *Fórmula 1.2*, donde: $C_{i,j}$ representa el número de casos de prueba que contienen el valor $V_{i,j}$.

$$h_{i,j} = \frac{C_{i,max} - C_{i,j} + 1}{C_{i,max} - C_{i,min} + 1} \tag{1.2}$$

La idea de la heurística es que los parámetros que aparecen menos veces sean añadidos a los nuevos casos de prueba.

Deterministas

En estas estrategias, dos ejecuciones del mismo algoritmo producen siempre los mismos resultados con el mismo conjunto de entradas. El grupo de estrategias deterministas está dividido en dos subgrupos: instantáneas y polinomiales. Estas estrategias siempre producen el mismo resultado para un modelo específico de parámetros de entrada.

En el subgrupo de estrategias de combinación polinomiales, están los casos resolubles en orden polinomial para los OA y CA.

Las estrategias iterativas construyen los casos de prueba paso a paso. Por ejemplo, la estrategia de combinación basada en parámetros *In Parameter Order* (IPO) comienza creando un conjunto de pruebas para un subconjunto de los parámetros en el modelo de entrada. Un parámetro a la vez es añadido y los casos de prueba en el conjunto son modificados.

El grupo más grande de estrategias es el basado en casos de prueba, éstas comparten la propiedad de que los algoritmos generan un caso a la vez y lo van añadiendo al conjunto de pruebas. A esta categoría pertenecen: Each Choice (EC), Partly Pair-Wise (PPW), Based Choice (BC), All Combinations (AC), Anti-Random (AR), CATS, k-perim y k-bound.

1.3. Planteamiento del problema

En el ámbito de pruebas de software muchas veces no basta con analizar las interacciones entre 2 factores, muchas veces se necesitan interacciones completas, en ocasiones con mas de 6 factores [61]. Sin embargo, la construcción de estos casos de prueba no se puede abordar con una computadora secuencial en un tiempo razonable, cuando el número de parámetros es muy grande.

El problema que se abordará es la creación de conjuntos de pruebas combinatorias óptimas, o cercanas a las óptimas, a través de los objetos matemáticos conocidos como Covering Arrays, para los casos donde: exista una cantidad considerable de variables (cientos), fuerzas mayores que 2 con alfabetos uniformes y mixtos.

1.3.1. Descripción del problema

Si se desea probar un software de manera completa, que tiene k parámetros y cada uno de los parámetros tiene v posibles valores, son necesarias v^k configuraciones, sin embargo esto puede ser realizado desde otro enfoque. La pruebas de software usando métodos DOE con frecuencia son referidas como pruebas combinatorias [59], éstas proporcionan un balance entre el costo de las pruebas y el grado requerido de cobertura [113].

La pregunta principal de las pruebas combinatorias radica en cómo encontrar un conjunto de configuraciones de tamaño mínimo (óptimo) [112]. Si durante la etapa de verificación del software se utilizan las pruebas combinatorias, es importante que se maximice la cobertura y se disminuya la cardinalidad del conjunto de pruebas. Los principales objetos combinatorios para satisfacer el criterio de cobertura para este tipo de problemas son los OA y los CA [20].

Los métodos para la construcción de los CA se han orientado en dos áreas principalmente [20]: algunos investigadores de la comunidad matemática están enfocando sus esfuerzos en construir CA más pequeños con fuerzas más grandes; por otra parte, la comunidad de las pruebas de software está centrándose en la búsqueda de algoritmos para construir CAs en un ambiente más flexible, que se ajuste a las necesidades reales de las pruebas. *Lo ideal sería combinar estas ideas y construir conjuntos de prueba de interacciones más grandes, de tamaño mínimo y generados eficientemente* y tomando en consideración que como en los sistemas del mundo real los parámetros suelen tener cardinalidades diferentes, sería más efectivo el uso de los MCA para la creación del conjunto de pruebas.

1.3.2. Complejidad del problema

Existen una gran diversidad de problemas computacionales que pueden solucionarse en tiempo polinomial, otros sólo pueden resolverse en tiempo exponencial, de ellos se dice que son *intratables*. Los problemas mencionados con anterioridad se clasifican como pertenecientes a la clase P y a la clase NP respectivamente. La clase **P** es el conjunto de problemas de decisión resolubles en tiempo polinomial por una máquina de Turing determinista; la clase **NP** es el conjunto de problemas para los que existe un algoritmo para una máquina de Turing no determinista capaz de dar respuesta en tiempo polinómico.

Lei y Tai [63] demostraron que la construcción de un CA óptimo de fuerza *2* es **NP-completo**. Primero probaron que el problema de cobertura par pertenece a los problemas de la clase NP, posteriormente demostraron que es un problema NP-completo a través de la técnica de reducción, eligiendo para ello el problema NP conocido como cobertura de vértices (vertex cover) [36]. Referente a la complejidad del problema, Khun y Okun [86] comentan que la construcción de un conjunto de pruebas es un problema NP. Adicionalmente, en el trabajo de Colbourn *et ál.* [21], se hace referencia a la NP-completez con interacciones de tamaño 2, reduciendo este caso al problema de Máxima Satisfactibilidad (con cláusulas 2-SAT).

Existen unos pocos casos de las pruebas combinatorias que pueden resolverse en forma polinomial, uno de ellos es cuando se desea construir un CA con parámetros t=v=2. Este problema fue resuelto completamente por Rényi (para N par) y de forma independiente por Katona y posteriormente por Kleitman y Spencer (para toda N) [94]. Otro caso que puede resolverse en forma polinomial es cuando $v = p^\alpha$ (p es primo y α es potencia entero positivo), $k \leqslant (p^\alpha + 1)$, $p^\alpha > t$. Este CA se puede expresar como: $CA(p^{t^\alpha}; p^\alpha + 1, p^\alpha, t)$. La forma de construirlo fue descrita por Bush [10], el cual utiliza los campos finitos de Galois. Salvo estos dos casos, el caso general de construir un conjunto óptimo de pruebas pertenece a la clase NP-completo [89].

Otras deducciones interesantes fueron proporcionadas por Tai y Lei [102], que mostraron los resultados empíricos de herramientas para construir CA con interacción entre dos parámetros, argumentando que si el dominio de los parámetros es grande, el número de casos de prueba puede ser enorme. El ejemplo proporcionado dice que si un sistema tiene v valores y t=*2*, la cantidad de configuraciones requeridas sería $\geqslant v^2$, de esta forma, si cada parámetro tiene 1,000 valores, se obtendrían al menos 1,000,000 de casos de prueba.

Debido a la complejidad del problema, la mayoría de los algoritmos son aproximados, en el sentido de que encuentran una solución en tiempo razonable, pero no necesariamente la solución óptima.

1.3.3. Trabajo a realizar

Para poder construir CA de fuerza mayor que 6, con tamaños de k mayores a 3 cifras y con alfabetos mixtos, es necesario disponer de herramientas que permitan lograrlo en un lapso razonable de tiempo.

Nosotros partimos del siguiente supuesto: "El cálculo de CA es un problema altamente paralelizable".

Una de las herramientas esenciales para poder construir CA es el hecho de verificar si una matriz cumple o no con los requisitos para ser considerada como CA; ser un arreglo de $N \times k$

elementos, donde cada $N \times t$ subarreglo contiene todas las combinaciones de los $\prod_{i=1}^{k} v_i$ símbolos al menos una vez.

En este trabajo se mostrará como verificar CAs haciendo uso de la Supercomputación y la Computación Grid, para ello se diseñaran una serie de algoritmos para validar CA de cualquier tamaño de fuerza, para alfabetos uniformes y mixtos y para cualquier valor de k.

1.4. Objetivos generales y específicos

1.4.1. Objetivo general

Elaborar un algoritmo para la verificación de CA que pueda sacar provecho de las ventajas de la Supercomputación y la Computación GRID.

1.4.2. Objetivos específicos

1. Estudiar diferentes estrategias para la construcción de los modelos combinatorios.
2. Validar los resultados obtenidos y refinar el algoritmo.

1.4.3. Metodología a seguir

Pasos a seguir:

- Se diseñara y programará un algoritmo secuencial para hacer la verificación de CA.
- Se diseñara e implementará un algoritmo paralelo para hacer la verificación de CA.
- Se diseñara un algoritmo que pueda ser ejecutado en un entorno Grid.
- Se correrá un conjunto de pruebas para comparar los algoritmos.
- Se analizarán los resultados.

1.5. Estructura del documento

El resto del presente documento se conforma de la siguiente manera:

En el capítulo 2 se definen los objetos combinatorios mencionados a lo largo del presente trabajo, haciendo especial énfasis en los CA; después se procede a mostrar un estado del arte sobre los diferentes enfoques que se han desarrollado para construirlos.

En el capítulo 3 se presenta una introducción a la supercomputación, se hace una descripción de la computación de altas prestaciones y sus aportaciones a los problemas denominados como de *gran desafío*, se presentan algunos elementos a tener en cuenta al momento de diseñar aplicaciones paralelas. Además, se hace una descripción de la computación Grid y sus campos de aplicación.

En el capítulo 4 se presenta una descripción detallada de la metodología utilizada, que permite la verificación de *covering arrays* de forma secuencial, paralela y Grid.

En el capítulo 5 se presentan los resultados obtenidos, a partir de la aplicación de la supercomputación y la computación GRID, al problema de verificar *covering arrays*.

En el capítulo 6 se presentan las conclusiones a las que se llegó a partir de los resultados obtenidos en este trabajo y se mencionan los trabajos futuros.

Capítulo 2

Covering Arrays: estado del arte

Investigadores y expertos de diversas disciplinas han sido atraídos por la aplicación de objetos combinatorios (LS, OA y CA) aplicándolos a diversas áreas de investigación. Por ejemplo: Cawse [11] los aplicó al diseño de materiales, Hedayat *et ál.* [49] mencionan aplicaciones en agricultura y medicina, Shasha *et ál.* [90] aplicaron estos objetos a la biología, Phadke y Taguchi [81, 100] los aplicaron a procesos industriales, en se registran [89, 103, 110, 107] aplicaciones en el área de pruebas de hardware y, por último, es en el área de pruebas de software donde mas aplicación han tenido recientemente [65, 8, 54, 16, 23, 27, 114].

En el presente capítulo se definen los objetos combinatorios mencionados a lo largo del presente trabajo, haciendo especial énfasis en los CA. A continuación se procede a mostrar el estado del arte de los diferentes enfoques que se han desarrollado para construirlos.

2.1. Definiciones y ejemplos

Los Latin Squares (LS) son un diseño combinatorio muy antiguo y ampliamente estudiado. Se cree que Euler, alrededor de 1782, fue el primero en estudiarlos. A principios del siglo XX Fisher [33] demostró su utilidad para el control de experimentos estadísticos agrícolas. Robert Mandl en 1985 [65] los aplicó al diseño pruebas de software.

Definición 2.1 (Latin Squares) *Sea n un estero positivo. Un latin square de orden n es un arreglo de tamaño $n \times n$ y con n símbolos diferentes, con la propiedad de que cada símbolo aparece exactamente una vez en cada fila y en cada columna.*

Se dice que un Latin Squares (LS) está en su *forma estándar* si la primera fila y la primera columna están en orden natural.

Ejemplo 2.2 *Frecuentemente un LS se escribe como un arreglo de letras latinas (motivo de su nomenclatura), por convenio las letras son A, B, C, ..., Z. En la tabla 2.1 se muestra un ejemplo de un LS estándar de orden $n = 4$. Dicho ejemplo deja patente de que un LS es un arreglo combinatorio de tres clasificaciones: renglones, columnas y letras.*

Tabla 2.1: Latin square de orden 4.

A	B	C	D
B	A	D	C
C	D	A	B
D	C	B	A

Definición 2.3 (Ortogonal Latin Squares) *Sea n un entero positivo y sean $A = [a_{i,j}]$ y $B = [b_{i,j}]$ latin squares de orden n. Considérese un arreglo de $n \times n$ con entradas $[(a_{i,j}, b_{i,j})]$; si en este arreglo, cada una de las n^2 parejas de símbolos ocurre exactamente una vez, entonces los latin squares A y B son ortogonales, y se denota como: $A \perp B$.*

Ejemplo 2.4 *Consideremos los 3 LS de orden $n = 3$ mostrados en la tabla 2.2, resulta que el LS (1) no es ortogonal con el LS (2) debido a que la pareja {A,E} se repite (véase la tabla 2.3).*

Tabla 2.2: Tres ejemplos de *latin squares*.

1			2			3		
A	*B*	*C*	*E*	*F*	*D*	*G*	*H*	*I*
B	*C*	*A*	*F*	*D*	*E*	*I*	*G*	*H*
C	*A*	*B*	*D*	*E*	*F*	*H*	*I*	*G*

Tabla 2.3: Combinación de los *latin squares* 1 y 2 mostrados en la *tabla* 2.2.

A,E	B,F	C,D
B,F	C,D	**A,E**
C,D	A,E	B,F

Ejemplo 2.5 *Por otra parte, al combinar el LS (1) con LS (3) resulta que son ortogonales pues aparecen las n^2 parejas de símbolos sin ninguna repetición (véase la tabla 2.4).*

Tabla 2.4: Combinación de los *latin squares* 1 y 3 mostrados en la *tabla* 2.2.

A,G	B,H	C,I
B,I	C,G	A,H
C,H	A,I	B,G

Definición 2.6 *Un conjunto de latin squares $L_1, L_2, ..., L_m$ es un conjunto de mutually orthogonal latin squares, o conjunto de MOLS, si para cada $1 \leq i \leq j \leq m$, L_i y L_j son ortogonales.*

Tabla 2.5: Tres *Orthogonal Latin Squares* de orden 4.

1

0	2	3	1
3	1	0	2
1	3	2	0
2	0	1	3

2

0	2	3	1
1	3	2	0
2	0	1	3
3	1	0	2

3

0	2	3	1
2	0	1	3
3	1	0	2
1	3	2	0

Ejemplo 2.7 *En la tabla 2.5 se muestran 3 Orthogonal Latin Squares (OLS) de orden 4, los cuales forman un Mutually Orthogonal Latin Squares (MOLS), denotado por MOLS(3,4).*

Definición 2.8 (Orthogonal Arrays) *Sean N, k, v, t enteros positivos con $t \leq k$. Un orthogonal array, $OA_\lambda(N; k, v, t)$, de índice λ, alfabeto v, fuerza t, es un arreglo A de tamaño $N \times k$, donde cada elemento $a_{i,j}$ toma como valor un símbolo del conjunto $S = \{1, 2, ..., v\}$, tal que cada $N \times t$ subarreglo contiene todas las posibles combinaciones de los v^t símbolos exactamente λ veces.*

Un OA tiene la propiedad $\lambda = \frac{N}{v^t}$, razón por la cual se puede dejar fuera la N de la notación, y expresarlo de la siguiente manera: $OA_\lambda(k, v, t)$. Cuando se suprime λ de la notación se asume que el valor del índice es 1.

Ejemplo 2.9 *En la tabla 2.6 se muestra un ejemplo de un OA con $\lambda = 2$, $k = 4$, $v = 2$ y $t = 2$. Si se toman cualquier par de columnas del arreglo se verá que existen todas las parejas {0,0},{0,1},{1,0},{1,1} exactamente 2 veces.*

Tabla 2.6: $OA_2(4,\ 2,\ 2)$.

0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Un $OA(k+2,\ s,\ 2)$ es equivalente a k *orthogonal latin squares* de orden s. Cuando existe un *mutually orthogonal latin squares*(k,s), se pueden transformar los k *orthogonal latin squares* de orden s en un $OA(s^2; k+2, s, 2)$, siguiendo los siguientes pasos:

1. Las combinaciones entre los índices de las columnas y filas de los LS constituirán las primeras dos columnas del OA.

2. Hay que agregar una columna al OA por cada uno de los k LS. Esta columna tendrá el valor correspondiente a la celda representada por los índices de las primeras dos columnas del OA.

Ejemplo 2.10 *En la tabla 2.5 se muestran 3 orthogonal latin squares de orden 4, con los cuales se construirá el OA(5,4,2). En la tabla 2.7(a) se muestran las combinaciones de los índices de filas y columnas de los orthogonal latin squares. En la tabla 2.7(b) se muestra el llenado de las primeras 3 filas del OA. Por último en la tabla 2.7(c) se muestra el* $OA(5,\ 4,\ 2)$ *obtenido a partir de los 3 orthogonal latin squares de orden 4.*

Tabla 2.7: a) Primeras dos columnas. b) Primeras tres filas. c) $OA(5,\ 4,\ 2)$.

(a)

0	0	.	.	.
0	1			
0	2			
0	3			
1	0			
1	1			
1	2			
1	3			
2	0			
2	1			
2	2			
2	3			
3	0			
3	1			
3	2			
3	3			

(b)

0	0	0	0	0
0	1	2	2	2
0	2	3	3	3
0	3			
0	4			
.				
.				
.				
.				
.				
.				
.				
.				
.				
.				
.				

(c)

0	0	0	0	0
0	1	2	2	2
0	2	3	3	3
0	3	1	1	1
1	0	3	1	2
1	1	1	3	0
1	2	0	2	1
1	3	2	0	3
2	0	1	2	3
2	1	3	0	1
2	2	2	1	0
2	3	0	3	2
3	0	2	3	1
3	1	0	1	3
3	2	1	0	2
3	3	3	2	0

Los primeros trabajos realizados, donde se aplicaron los objetos combinatorios al diseño de pruebas, fueron hechos a través de *Orthogonal Arrays* en disciplinas como la medicina, la agricultura y la manufactura[49].

Un problema de este modelo es que puede dar lugar a pruebas excesivamente grandes con $\lambda > 1$. Para los casos de v y k donde exista un OA con $\lambda = 1$, éste sería el conjunto óptimo de pruebas. Sin embargo, hay muchos valores de v y k donde los OA con $\lambda = 1$ no existen, por ello se tiene que recurrir a una estructura menos restrictiva conocida como: Covering Arrays.

Definición 2.11 (Covering Arrays) *Sean* N, k, v, t *enteros positivos con* $t \leq k$*. Un covering array, CA(N;k,v,t), de índice* λ*, alfabeto* v*, fuerza* t*, es un arreglo* C *de tamaño* $N \times k$*, donde cada elemento* $a_{i,j}$ *toma como valor un símbolo del conjunto* $S = \{1, 2, ..., v\}$*, tal que cada* $N \times t$ *subarreglo contiene todas las posibles combinaciones de los* v^t *símbolos al menos* λ *veces.*

Ejemplo 2.12 *En la tabla 2.8 se muestra un ejemplo de un CA(5;4,2,2). Nótese que en cada par de columnas aparecen las combinaciones {0,0}, {0,1}, {1,0} y {1,1} al menos una vez.*

Tabla 2.8: $CA(5; 4, 2, 2)$.

1	1	1	0
0	0	0	0
0	1	0	1
1	0	0	1
0	0	1	1

Definición 2.13 (Covering Array Number) *Es el tamaño mínimo de N para un CA(k,v,t) y se denota por: CAN(k,v,t).*

Muchos *covering array number* son aún desconocidos, sin embargo en la literatura se registran el *lower bound* y el *upper bound* para un determinado CA. El *lower bound* es el tamaño mínimo teórico para un determinado CA, mientras que el *upper bound* es el tamaño mínimo para el cual existe un arreglo conocido.

Los investigadores publican el mejor *upper bound* conocido para un determinado CA. Stevens [97] hace un resumen de los resultados para fuerza 2, Sloane [94] presenta un excelente resumen de los resultados publicados de CA de fuerza 2 y 3. Sherwood [91] mantiene un sitio web de construcciones de OA y CA mediante permutaciones de grupos. Por último, Hartman [47] presenta un survey tanto para CA de alfabeto uniforme como mixto.

El *lower bound* para los CA de alfabeto uniforme es determinado por v^t. Para alfabetos mixtos el *lower bound* es determinado por $\prod_{i=1}^{t} v_i$, donde $v_1 \geq v_2 \geq ... \geq v_k$.

Los CA son fundamentales en el diseño de experimentos y pruebas cuando todos los factores tienen el mismo número de niveles. Sin embargo, típicamente los sistemas no están compuestos por componentes (factores) que tengan exactamente el mismo número de parámetros (niveles). Para eliminar esta limitación de los CA, se puede hacer uso de los MCA.

Definición 2.14 (Mixed Covering Arrays) *Sean N, k, v, t enteros positivos donde $t \leq k$. Un Mixed Covering Array de tipo $\prod_{i=1}^{k} v_i$, de índice λ, con fuerza t y tamaño N, denotado por MCA(N;k,$\prod_{i=1}^{k} v_i$,t), es un arreglo A de tamaño $N \times k$. Sea $\{i_1, ..., i_t\} \subseteq \{1, ..., k\}$, y B un subarreglo de tamaño $N \times t$ obtenido al seleccionar las columnas $i_1, ..., i_t$ del MCA. Hay $\prod_{i=1}^{k} v_i$ t-tuplas distintas que pueden aparecer como filas de B, un MCA requiere que al menos aparezcan λ veces.*

Ejemplo 2.15 *En la tabla 2.9 se muestra un MCA de 9 filas (experimentos), 5 columnas (factores), de alfabeto 2 para los primeros 3 factores (2^3), de alfabeto 3 para los últimos dos factores (3^2) y con una fuerza de interacción entre los factores de 2, denotado por: $MCA(9; 5, 2^3\ 3^2, 2)$.*

Tabla 2.9: $MCA(9; 5, 2^3\ 3^2, 2)$.

0	0	0	0	0
0	0	0	1	1
0	0	1	2	2
0	1	0	0	2
0	1	0	2	1
0	1	1	1	0
1	0	0	2	0
1	1	1	0	1
1	1	1	1	2

Se sabe muy poco acerca de los tamaños mínimos para MCA. En [49] Hedayat y Sloane presentan un estudio sobre los *orthogonal arrays*. Sloane *et ál.* [95] amplían dicho trabajo y

usando programación lineal obtienen mínimos para *mixed orthogonal arrays*. Stardom [96] y Chateauneuf [12] sugieren la necesidad de extender sus trabajos sobre CA de alfabetos uniformes a MCA, sin embargo, la mayoría de trabajos reportados son solo para $t = 2$. Recientemente, Moura *et ál.* [71] reportaron algunas transformaciones algebraicas para construir MCA. Dichas transformaciones están limitadas a fuerza 2, pero obtuvieron el mínimo para los casos donde $k \leq 4$ y para algunos casos donde $k = 5$.

La mayoría de la literatura correspondiente a los CA, aplicados al diseño de pruebas de software, incluyen métodos para construirlos de forma eficiente mientras se obtienen conjuntos de pruebas pequeños.

Básicamente existen dos líneas de construcción de CA, en la literatura matemática se presentan nuevas construcciones algebraicas [14, 17, 94, 98, 99], mientras que en la literatura de ingeniería de software se proponen nuevos algoritmos determinísticos [15, 7, 63, 115] y no determinísticos [20, 78, 93].

2.2. Construcción de Covering Arrays óptimos en tiempo polinomial

La construcción de CA óptimos es en general un problema NP-Completo [89], sin embargo se conocen 2 casos para los cuales es posible construirlos en tiempo polinomial: CA(N;k,2,2) y CA$(v^t; v+1, v, t)$.

2.2.1. Caso: CA(N; k, 2, 2)

En 1971 Rényi [85] determinó el Covering Array Number (CAN) de los CA para el caso $v = t = 2$ cuando N es par. Kleitman y Spencer [58] y Katona [56] determinaron de forma independiente el CAN para todas las N cuando $v = t = 2$. Una vez dados k, v, t, el problema se reduce a determinar el mínimo entero N que satisface la desigualdad 2.1.

$$k \leq \binom{N-1}{\lceil \frac{N}{2} \rceil}. \tag{2.1}$$

El algoritmo sigue los siguientes pasos: llenar la primer fila del arreglo del CA con 0's, calcular el total de lineas faltantes, siguiendo la *Formula 2.1*, se calcula el total de 1's faltantes por columna, $\lceil \frac{N}{2} \rceil$., se calcula el total de 0's faltantes por columna $N - 1 - \lceil \frac{N}{2} \rceil$. y por último se llenan las filas restantes con las combinaciones de 1's y 0's obtenidos en los pasos anteriores.

Ejemplo 2.16 *Considérese la creación de un CA con $k = 10, v = t = 2$. Siguiendo la Formula 2.1 se obtiene que $N = 6$, con lo cual se obtendrá un $CA(6;\ 10,\ 2,\ 2)$. Se procede a llenar la primera fila con 0's (véase la tabla 2.10(a)). La cantidad de 0's requeridos es de: $6 - 1 - \lceil \frac{6}{2} \rceil = 2$, estos se colocan en la primer columna. La cantidad de 1's requeridos por columna es de: $\lceil \frac{6}{2} \rceil = 3$, estos se colocan debajo de los 0's (véase la tabla 2.10(b)). A partir de aquí se procede a combinar los 1's y 0's hasta llenar todas las columnas, la instancia completa se muestra en la tabla 2.10(c).*

Tabla 2.10: Construcción de un $CA(6;\ 10,\ 2,\ 2)$.

(a)

0	0	0	0	0	0	0	0	0	0

(b)

0	0	0	0	0	0	0	0	0	0
0									
0									
1									
1									
1									

(c)

0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	1
0	1	1	1	0	0	0	1	1	1
1	0	1	1	0	1	1	0	0	1
1	1	0	1	1	0	1	0	1	0
1	1	1	0	1	1	0	1	0	0

Para los casos donde k es muy grande, N crece logarítmicamente. El valor mínimo de N [56, 94, 58] satisface lo expresado por la *fórmula 2.2*.

$$N = log_2\ k + \frac{1}{2}\ log_2\ log_2\ k. \tag{2.2}$$

En 1990 Gargano *et ál.* [37] obtuvieron un mínimo probabilístico para los casos donde $t = 2$ y $v > 2$, véase la *expresión 2.3*. Sin embargo, no proveen algún método para construir un arreglo con dicho tamaño [94].

$$N = \frac{v}{2}\ log\ k(1 + o(1)). \tag{2.3}$$

2.2.2. Caso: $CA(v^t; v + 1, v, t)$

Bush [10] propone un teorema para realizar construcciones directas de CA a través de OA y Galois Fields (GF). Este teorema requiere que $v = p^{\propto}$ sea una potencia de un número primo ($v > t$), para este caso CAN(k,v,t)$=v^t$ para todo $k \leq v + 1$. Cuando el valor de $\propto = 1$ la construcción se basa en MOLS, para los casos cuando $\propto > 1$ los CA se construyen usando campos finitos.

2.3. Covering Arrays Cíclicos

Según Meagher y Stevens [68], un Cyclic Covering Array (CCA) es una Cyclic Matrix (CM) $\mathcal{O}$ de tamaño $k \times k$ que es construida mediante $k - 1$ rotaciones de un vector inicial s de tamaño k. En la *tabla 2.11* se muestra un ejemplo de una CM.

El vector inicial s puede ser calculado mediante $GF(v^t)$. La característica principal de un CCA es que por medio de automorfismos se reduce el espacio de búsqueda para verificar que sea un CA. Un automorfismo es un isomorfismo de un objeto matemático en sí mismo. Usualmente el conjunto de automorfismos de un objeto puede recibir una estructura de grupo con la operación

de composición, tal grupo recibe el nombre de *grupo de automorfismos* y es, a grandes rasgos, el grupo de simetría del objeto.

Tabla 2.11: Una CM de tamaño $k \times k$ creada a partir del vector $(x_1, x_2, \ldots, x_k)^T$.

x_1	x_2	$\cdots$	x_k
x_2	x_3	$\cdots$	x_1
x_3	x_4	$\cdots$	x_2
$\vdots$	$\vdots$	$\ddots$	$\vdots$
x_k	x_1	$\cdots$	x_{k-1}

En el caso de los CCA, hay grupos de automorfismos entre las diferentes t-tuplas. Los grupos de automorfismos pueden ser definidos de la siguiente forma: en un $CCA(k; k, v, t)$ dos t-tuplas son isomórficas si una de ellas contiene el conjunto $\{0, 1, ..., v-1\}^t$ implica que la otra también lo contiene.

Dado que el vector inicial s requerido para construir un CCA es obtenido mediante GF, las propiedades de suma y multiplicación de la teoría de campos finitos pueden ser usadas para identificar las t-tuplas isomórficas [22].

Sea Λ el conjunto formado por $\binom{k}{t}$ diferentes t-tuplas. De acuerdo con la propiedad aditiva, dos vectores $v = \{v_0, v_1, ..., v_{t-1}\}$ y $w = \{w_0, w_1, ..., w_{t-1}\}$, $v, w \in \Lambda$, son isomórficos si y solo si hay una constante a tal que $s_i = (w_i + a) \ \mathbf{mod}\ k$, para todo $0 \le i \le t-1$. Entonces, los elementos del grupo de automorfismos de las t-tuplas $v = \{v_0, ..., v_{t-1}\}$ obtenidos mediante la propiedad aditiva son el conjunto definido en la *expresión 2.4*.

$$\{w = \{w_0, ..., w_{t-1}\} | w_i = (v_i + a) \ \mathbf{mod}\ k, \forall\ 0 \le i \le t-1, \forall\ 0 \le a \le k-1\} \tag{2.4}$$

Al aplicar la propiedad aditiva al grupo de automorfismos se crea un conjunto $\Delta \subset \Lambda$ donde cada t-tupla $v = \{v_0, ..., v_{t-1}\}$ inicia con el mismo valor, por ejemplo, $v_0 = 0$. La cardinalidad del conjunto Δ es $\binom{k-1}{t-1}$.

Continuando con la propiedad multiplicativa, dos elementos $v, w \in \Lambda$ son isomórficos si hay una constante c tal que $v_i = (w_i \cdot c) \ \mathbf{mod}\ k$ y c sea un cuadrado perfecto. La *expresión 2.5* describe el grupo de automorfismos t-tupla v obtenido al aplicar esta propiedad.

$$\{w = \{w_0, ..., w_{t-1}\} | w_i = (v_i \cdot c) \ \mathbf{mod}\ k, \forall\ 0 \le i \le t-1, c = j^2, \forall\ 1 \le j \le k-1\} \tag{2.5}$$

Combinar las propiedades aditiva y multiplicativa permite crear un conjunto $\mathcal{C} \subset \Delta$ suficiente para verificar si se trata de un CA. Cada elemento $v \in \mathcal{C}$ es una t-tupla $v = \{v_0, v_1, ..., v_{t-1}\}$ que contiene $v_0 = 0$ y $v_1 = \{1, \alpha\}$. El valor α es el mínimo de los residuos no cuadráticos, caracterizado por la *expresión 4.5*. La cardinalidad de este nuevo conjunto de t-tuplas es $\mathcal{C} = \binom{k-2}{t-2} + \binom{k-\alpha-1}{t-2}$.

$$\alpha = \text{mín}\{a | \nexists_{1 \le b \le k} b^2 \equiv a(\ \mathbf{mod}\ k)\} \tag{2.6}$$

En la *tabla 2.12* se muestra un ejemplo de un $CCA(7; 7, 2, 2)$ obtenido a partir de $s = \{0, 0, 0, 1, 0, 1, 1\}$. En [22, 13, 64] se pueden encontrar mas ejemplos de este tipo de construcciones.

Tabla 2.12: Un $CCA(7;7,2,2)$.

0	0	0	1	0	1	1
0	0	1	0	1	1	0
0	1	0	1	1	0	0
1	0	1	1	0	0	0
0	1	1	0	0	0	1
1	1	0	0	0	1	0
1	0	0	0	1	0	1

2.4. Transformaciones algebraicas

2.4.1. TConfig

TConfig es una herramienta desarrollada por Williams y Probert [109, 111] que hace uso de construcciones algebraicas para generar CA de fuerza 2 y alfabeto uniforme.

Emplean la teoría de Orthogonal Arrays [49] para construir diferentes instancias de CA. El enfoque que se emplea en TConfig para construir CA esta basado en la filosofía de “divide y vencerás”, un ejemplo del funcionamiento de esta estrategia es el siguiente: supóngase que se quiere generar un CA para probar un sistema con 12 parámetros j $(0 \leq j < 12)$ y la cardinalidad o posibles valores que puede tomar cada parámetro j es $|v_j| = 3$. La primer etapa para construir el CA es concatenar horizontalmente 4 OA de 4 parámetros de entrada cada uno, para agregar las combinaciones faltantes se sigue una segunda etapa, en donde debajo de la primera columna de cada OA concatenado en la etapa uno se agrega un OA reducido. Las combinaciones faltantes pueden ser completadas agregando una columna más de 0’s en los primeros OA, y valores desde 1 hasta $|v|-1$ repetidos $|v|$ veces verticalmente junto a la columna derecha de cada OA reducido.

La teoría de esta construcción está basada en el trabajo reportado en [6], donde se requiere que $|v|$ sea una potencia de un número primo.

2.4.2. Combinatorial Test Services

En el 2004, Hartman y Raskin [48] presentan otra herramienta para construir CA de alfabetos uniformes y mixtos a la que llaman Combinatorial Test Services (CTS). Para realizar las construcciones dicha herramienta utiliza los campos finitos, teoría de subgrupos, teoría de códigos y técnicas recursivas, entre otras.

Hartman y Raskin plantearon un conjunto de teoremas y lemas matemáticos que permiten generar en tiempo polinomial diferentes construcciones de CA, muchas de las cuales son óptimas o cercanas al óptimo.

2.5. Estrategias deterministas

2.5.1. Automatic Efficient Test Generator

Automatic Efficient Test Generator (AETG) es una herramienta desarrollada originalmente por la empresa Bellcore (ahora Telecordia). Esta herramienta está basada en las ideas del DOE y

crea un conjunto de pruebas con una cobertura de fuerza 2. Las primeras aplicaciones de AETG tuvieron lugar en los procesos industriales [9]. Cohen *et ál.* [15] fueron los primeros en publicar una descripción de los algoritmos usados en la implementación de AETG.

En este sistema un usuario puede especificar los requerimientos de prueba para las interacciones entre un grupo de parámetros para crear una o más relaciones, así como la interacción entre los parámetros. Para crear una relación, el usuario especifica los parámetros en la relación y un conjunto de valores válidos y no válidos para cada parámetro. El sistema solo aceptará como entradas aquellas construidas por valores válidos y rechazará las demás.

En este algoritmo se construyen los CA utilizando la técnica de *una fila a la vez*. AETG genera un conjunto de vectores candidatos y se selecciona al que cubre la mayor cantidad de combinaciones faltantes, cabe mencionar que AETG no siempre obtiene la mejor solución conocida (*upper bound*) y por lo general los resultados no se obtienen en tiempo logarítmico.

2.5.2. In-Parameter-Order (IPO)

In-Parameter-Order (IPO) fue desarrollado por Lei y Tai [63] para generar CA de fuerza 2 con alfabetos uniformes y mixtos en lapsos cortos de tiempo, pero sin garantizar la mejor solución posible. El algoritmo asume que el sistema $\mathcal{S}$ tiene parámetros $p_1, p_2, ..., p_n$ donde $n \geq 2$.

IPO trabaja de la siguiente manera: suponiendo que se tiene un sistema con dos o más parámetros de entrada, la estrategia IPO genera un conjunto de combinaciones para los primeros dos parámetros del sistema y continua de la misma manera para cada parámetro del sistema. Para agregar los nuevos vectores del CA se siguen los siguientes dos pasos:

1. Crecimiento horizontal, en este paso se agrega uno de los diferentes valores de cada parámetro que cumpla con la combinación de parámetros faltantes.

2. Crecimiento vertical, en este paso se agregan nuevos vectores, si es necesario para cubrir todas las combinaciones necesarias.

Este algoritmo permite el uso de técnicas de optimización local para la generación de pruebas, y el reuso de las pruebas existentes cuando un sistema es extendido con nuevos parámetros o nuevos valores para los mismos.

En [62] se muestra una nueva versión llamada *IPOG* que permite resolver instancias de fuerza $2 \leq t \leq 6$.

2.5.3. Test Case Generation

Test Case Generation (TCG) es un algoritmo desarrollado por Tung y Aldiwan [115]. Se asume que se tiene un sistema $\mathcal{S}$ con k parámetros de entrada $(F_1, F2, ..., F_k)$. Para cada parámetro i, $1 \leq i \leq k$, hay $m(i)$ diferentes valores, $V_{i1}, V_{i2}, ..., V_{im(i)}$. Nótese que $m(i)$ representa el tamaño o cardinalidad del vector V_i el cual se va a representar como $|V_i|$. El algoritmo sigue los siguientes 4 pasos básicos:

1. Ordena de forma descendente los parámetros de entrada i con respecto a su cardinalidad $|V_i|$ tal que $|V_1| \leq |V_i| \leq |V_k|$.

2. Se generan M vectores candidatos tal que contengan nuevas combinaciones faltantes.
3. Se selecciona el candidato que contiene más combinaciones presentes en su configuración.
4. Se repite el procedimiento de los pasos $\{1, 2, 3\}$ hasta que el arreglo de vectores contenga todas las combinaciones de parámetros.

Este algoritmo es capaz de generar casos de prueba con alfabetos uniformes y mixtos de fuerza 2. Su principal ventaja es su rapidez en la generación de CA, garantizando que se tienen todas las combinaciones.

2.5.4. Deterministic Density Algorithm

Deterministic Density Algorithm (DDA) es un algoritmo diseñado por Bryce, Colbourn y Cohen [7] para la creación de CA; este algoritmo emplea la técnica de *una prueba a la vez* mejorando algunos de los algoritmos que emplean la misma técnica [15, 115]. La ventaja de emplear algoritmos llamados *una prueba a la vez* es poder generar CA en tiempo polinomial, pero con el inconveniente de no producir CA óptimos.

Considere la construcción de un conjunto de pruebas con k factores. El número de niveles para el factor i se denota por v_i. Para los factores i y j, la densidad local es $\delta_{i,j} = r_{i,j}/v_i v_j$ donde $r_{i,j}$ es el número de parejas no cubiertas para los niveles i y j. La densidad global es $\delta = \sum_{1 \leq i \leq j \leq k^{\delta i,j}}$. En cada pasada del algoritmo, se busca al menos δ parejas no cubiertas.

La forma de operar de este algoritmo está centrada en la probabilidad de tener la menor cantidad de combinaciones repetidas en una configuración dada.

2.6. Estrategias no deterministas

2.6.1. Recocido Simulado

Cohen M. B. *et ál.* [20, 19] construyen CA mediante un Simulated Annealing (SA). Los resultados que publican mediante este enfoque permite construir CA con mejores resultados que los reportados por los algoritmos *una prueba a la vez*.

Los fundamentos y la teoría sobre el algoritmo SA se pueden encontrar en [1].

2.6.2. Algoritmo del Diluvio Universal

Dueck propuso el Great Deluge Algorithm (GDA) [29], el cual sigue una estrategia similar a SA pero que converge más rápido. GDA en lugar de usar la probabilidad para decidir un movimiento, rechaza una nueva solución que tenga una función de coste inferior a un nivel estipulado. Este principio se relaciona de alguna forma con el enfoque Darwiniano, GDA trabaja eliminando a los más débiles.

Myra B. Cohen en su tesis doctoral [18] propone la utilización de este algoritmo para la construcción de CA.

2.6.3. Búsqueda Tabú

En el año 2004, Nurmela [78] reporta el uso de Tabu Search (TS) [41] para construir CA. TS es una estrategia basada en una memoria que guía el sistema hacia zonas del espacio de soluciones que aún no han sido exploradas. Para evitar que el proceso vuelva a un óptimo local anterior, TS clasifica como "tabú" un determinado número de los movimientos más recientes, los cuales no pueden repetirse durante un horizonte temporal estipulado.

TS toma de los principios generales de la *Inteligencia Artificial* el concepto de *memoria*, con el objetivo de dirigir la exploración atendiendo a las consecuencias de la historia más reciente. En este sentido se puede decir que hay un cierto aprendizaje y que la búsqueda es inteligente. De alguna forma el principio básico consiste en suponer que una mala elección, basada en cierta estrategia, es preferible a una buena solución fruto del azar, ya que ésta última no proporciona información para acciones posteriores.

El procedimiento sigue la estrategia de la búsqueda local, sin embargo el entorno de una solución disminuye al tomar en consideración la memoria de la exploración. La forma de construir una instancia de CA según [78] es la siguiente: para representar un CA emplea una matriz M de $N \times k$ elementos aleatorios, donde a cada fila N_i de la matriz M le corresponde lo que Nurmela llama *palabra* del CA; a la cantidad de combinaciones faltantes le llama el *costo* de la matriz M; para realizar un cambio de un elemento $m_{i,j}$ en la matriz M se selecciona aleatoriamente una combinación que no está presente por la configuración actual de la matriz M; posteriormente se revisa qué filas requieren sólo un simple cambio de uno de sus elementos para que la combinación seleccionada está presente en esa fila; a estos cambios se les llama *movimientos* y son realizados en la vecindad del algoritmo. El costo correspondiente a cada movimiento es calculado y se selecciona el movimiento que reduzca el costo de la matriz M. Si existe más de un movimiento con el mismo costo, se selecciona aleatoriamente uno de los movimientos realizados. Este proceso es repetido hasta que el costo de la matriz M sea cero o el número de intentos sea alcanzado.

Los resultados obtenidos por este trabajo son de buena calidad, pero los inconvenientes principales del enfoque es el tiempo de ejecución, ya que se reporta que fueron necesarios algunos meses de tiempo de cómputo para resolver las diferentes instancias de prueba.

2.6.4. Algoritmos Genéticos

Los *algoritmos genéticos*(Genetic Algorithms (GA)) toman como modelo la supervivencia de los individuos más aptos. Usan normalmente como operadores la cruza y la mutación [31]. Al inicio del algoritmo se seleccionan m casos de prueba candidatos de forma aleatoria. Posteriormente se inicia el ciclo de evaluación. El GA continua hasta que se dé la condición de paro cuando se generan un cierto número de casos candidatos M.

Los GA emplean una estrategia de elitismo en el cual en cada generación sobreviven intactos los p mejores individuos (cromosomas). A los $m - p$ individuos restantes se les aplica selección por torneo, eligiendo aleatoriamente 2 individuos y escogiendo de éstos el que tenga mejor valor, y así sucesivamente hasta completar $m - p$ casos de prueba, para posteriormente aplicarles el proceso de cruza uniforme con 0.5 % de probabilidad, y en el proceso de mutación se reemplaza el valor de una posición elegida aleatoriamente por otro valor. Como la representación de este tipo de algoritmos es solo con 0's y 1's, si en la posición elegida se encuentra un uno se reemplaza por un cero y viceversa. Los mejores p individuos son los seleccionados para formar la siguiente

generación. Si se llega a la condición de paro y no se ha encontrado una solución se muta la población masivamente en cada posición de cada cromosoma.

Este algoritmo recibe un conjunto de casos de prueba candidatos y proporciona como salida el caso de prueba de más alto valor.

2.6.5. Colonia de Hormigas

En 1996, Marco Dorigo *et ál.* [28] propusieron el Ant Colony Algorithm (ACA). El cual está basado en el comportamiento estructurado de una colonia de hormigas, donde los individuos se comunican entre sí por medio de una sustancia química denominada *feromona*, estableciendo el camino más adecuado entre su nido y su fuente de alimentos. El método consiste en simular computacionalmente la comunicación indirecta que realizan las hormigas para establecer el camino más corto entre su ubicación inicial y una fuente de comida.

Shiba *et ál.* [93] proponen el uso de ACA para generar casos de prueba, un caso de prueba a la vez, es decir, agregado al conjunto de casos de prueba hasta completar el número de casos de prueba deseado de acuerdo al criterio de cobertura.

Un caso de prueba es representado como una ruta desde un punto de inicio hasta un punto final objetivo. Cada nodo p_i representa un parámetro, *y* cada arco $V_{j,i}$ representa el *j-ésimo* valor interesante del parámetro *i-ésimo*. Cada camino desde el nodo inicial hasta el nodo objetivo se asocia con un caso de prueba, véase la *figura 2.1*. Así, un posible caso de prueba para el problema mostrado en la *figura 2.1* vendría determinado por $V_{1,2}, V_{2,1}, V_{3,1}, ..., V_{n,2}$.

El grafo es recorrido por un conjunto de hormigas. Cuando una hormiga alcanza el nodo objetivo, se deposita en cada arco de los que ha visitado una cantidad de feromonas proporcional a la calidad de la solución. Si una hormiga tiene que elegir entre diferentes arcos, se va por aquel que tenga más feromonas.

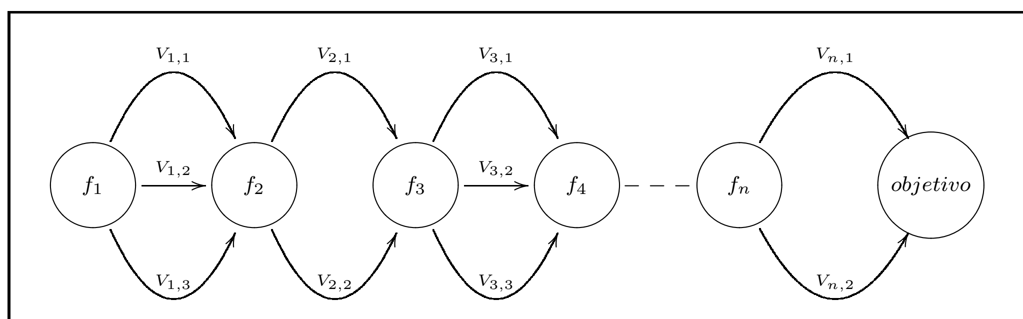


Figura 2.1: Ejemplo de un grafo para el algoritmo de colonia de hormigas.

2.7. Resumen del capítulo

En este capítulo se presentaron una serie de definiciones y ejemplos de los principales objetos combinatorios aplicados al diseño de pruebas de software.

Quedó claro el hecho de que aunque teóricamente se puede calcular el tamaño óptimo de un CA (*lower bound*), la realidad es que muchas veces no se conoce una forma de construirlos o computacionalmente es imposible calcularlos en una computadora secuencial. Por ello se presentó una revisión bibliográfica sobre las diferentes técnicas aplicadas a la construcción de covering arrays cuasi-óptimos (*upper bound*).

En el siguiente capítulo se hace una recopilación sobre los conceptos y arquitecturas de la supercomputación y la computación en GRID, además se muestra lo que estas tecnologías ofrecen a la investigación de áreas como la física, medicina, genética, entre otras. Dichas capacidades de cómputo son las que se pretenden aplicar al cálculo de CA.

Capítulo 3

Supercomputación y Computación GRID

En el capítulo anterior se describió el problema de construir *covering arrays óptimos*, aunque matemáticamente se ha definido como calcular el número óptimo (lower bound) de filas, la realidad es que para la mayoría de los casos no se reporta un algoritmo para construirlos, o para determinados parámetros de entrada el tamaño del arreglo es intratable con una computadora convencional.

En este capítulo se presenta una descripción de la Supercomputación y sus aportaciones a los problemas denominados como de *gran desafío*. Por último se presentan algunos elementos a tener en cuenta al momento de diseñar aplicaciones paralelas y Grid.

3.1. Conceptos generales

El término Supercomputación fue acuñado para hacer referencia a computadoras con capacidades muy superiores a las de otras máquinas disponibles. Sin embargo en la actualidad con tantos avances tecnológicos el término supercomputación sirve para describir el uso de computadoras para la resolución de problemas de índole numérica que requieren una gran capacidad de cálculo. Actualmente existen problemas cuya solución se conoce de forma matemática, pero que, cuando se trata de obtener, se genera un número de cálculos muy elevado [2]. La Teoría de la Complejidad estudia la forma de calcular este número de operaciones para un problema algorítmico dado.

El total de operaciones necesarias para resolver un problema suele estar en función del tamaño del problema. Si suponemos que cada operación aritmética se ejecuta en un tiempo τ, el tiempo para resolver un problema será como mínimo lo establecido por la *fórmula 3.1*, donde n representa el tamaño del problema y P_r representa un polinomio de grado r en la variable n.

$$T = \tau P_r(n) \tag{3.1}$$

Si se dispone de una supercomputadora que permita realizar varias operaciones a la vez y si se es capaz de dividir el trabajo en procesos independientes, entonces es posible conseguir, bajo ciertas circunstancias, reducir el tiempo de ejecución en proporción al número de operaciones

que pueden llevarse a cabo simultáneamente. Si este número de operaciones que se realizan simultáneamente puede hacerse crecer, entonces es posible conseguir una potencia de computación elevada que puede hacer resolubles problemas que de otra forma no se podrían.

Existen numerosas aplicaciones (Genómica, Física de Altas Energías, Biotecnología, Aeronáutica, Astrofísica, Computación Simbólica, Simulación, entre otras.) denominadas *problemas de gran desafío*, que requieren una potencia computacional superior a las disponibles en una computadora tradicional (PC).

Según Grama *et al.* [44] la Computación de Altas Prestaciones (CAP) permite el desarrollo de aplicaciones que aprovechan el uso colaborativo de múltiples procesadores con el objetivo de resolver un problema común.

Continuamente se suele hablar indistintamente de CAP y computación paralela. Sin embargo, el primero es un término más general que abarca la obtención de las máximas prestaciones, a través de los siguientes niveles de un sistema de computación:

- Gestión eficiente de la jerarquía de memorias.
- Optimización del proceso de cálculo secuencial.
- Gestión eficiente del proceso de E/S.
- Aplicación de la computación paralela.

La CAP se basa en una administración eficiente de la jerarquía de memorias. La existencia de diferentes niveles de memoria implica que el tiempo empleado en realizar una operación de acceso a la memoria de un nivel aumenta con la distancia al nivel más rápido, que es el acceso a los registros del procesador. Por lo tanto, una mala administración de la memoria provoca fallos de página, que abusan del sistema de memoria virtual, realizando costosos accesos al disco. Por lo cual, el diseño de aplicaciones eficientes requiere conocer la estructura de la memoria.

El siguiente nivel de la CAP es la optimización del proceso de cálculo secuencial, el cual implica una adecuada selección de las estructuras de datos a emplear para representar la información relativa al problema, atendiendo a criterios de eficiencia computacional.

Dado que el acceso a disco representa tiempos muy superiores a los correspondientes al acceso a memoria RAM, resulta indispensable realizar una administración eficiente del proceso de E/S para que tenga el mínimo impacto sobre el proceso principal.

Finalmente, y por encima del resto de capas, aparece la utilización de computación paralela. En este sentido, una buena estrategia de partición de tareas que garantice una distribución equitativa de la carga entre los procesadores involucrados, así como la minimización del número de comunicaciones entre los mismos, permite reducir los tiempos de ejecución. Además, con la partición y distribución de las principales estructuras de datos de la aplicación, entre los diferentes procesadores, se puede conseguir resolver problemas de mayor dimensión.

3.2. Computación Paralela

La computación paralela no es una idea reciente, la referencia más antigua que se tiene está ligada a la máquina analítica de Babbage [70] donde su diseñador proponía configurar la máquina

de tal manera que proporcionara varios resultados al mismo tiempo. Mas tarde, en 1958, Gill [39] ya escribía sobre la idea de programar en paralelo. En 1959 Holland [53] describía "una computadora capaz de ejecutar un número arbitrario de subprogramas simultáneamente" . En 1963, Conway [24] describe el diseño de una computadora paralela y la forma de programarla. En 1996 Flynn y Rudd [34] concluyeron: "la continua tendencia hacia sistemas de alto rendimiento... nos conduce a la siguiente conclusión: el futuro es paralelo".

El desarrollo de aplicaciones paralelas es una tarea compleja. Los desarrolladores de software paralelo deben enfrentarse a problemas que no aparecen en la programación secuencial, tales como la comunicación, la sincronización, el particionado y distribución de los datos, el reparto de carga, la tolerancia a fallos, la heterogeneidad de la arquitectura, los interbloqueos o las condiciones de carrera, entre otros.

3.2.1. Clasificación de las computadoras paralelas

En la literatura se registran distintas clasificaciones de las computadoras paralelas, sin embargo quizá la mas aceptada es la taxonomía de Flynn [34], véase la *figura 3.1*. La cual se basa en la multiplicidad del flujo de instrucciones y del flujo de datos en una computadora. El flujo de instrucciones es la secuencia de instrucciones ejecutadas por la computadora y el flujo de datos es la secuencia de datos sobre los cuales operan las instrucciones. Así, se puede tener un flujo único (*S=Single*) o múltiple (*M=Multiple*) de instrucciones, ejecutándose sobre un conjunto único (*S=Single*) o sobre varios (*M=Multiple*) conjuntos de datos. Con lo cual aparecen cuatro clases de computadoras:

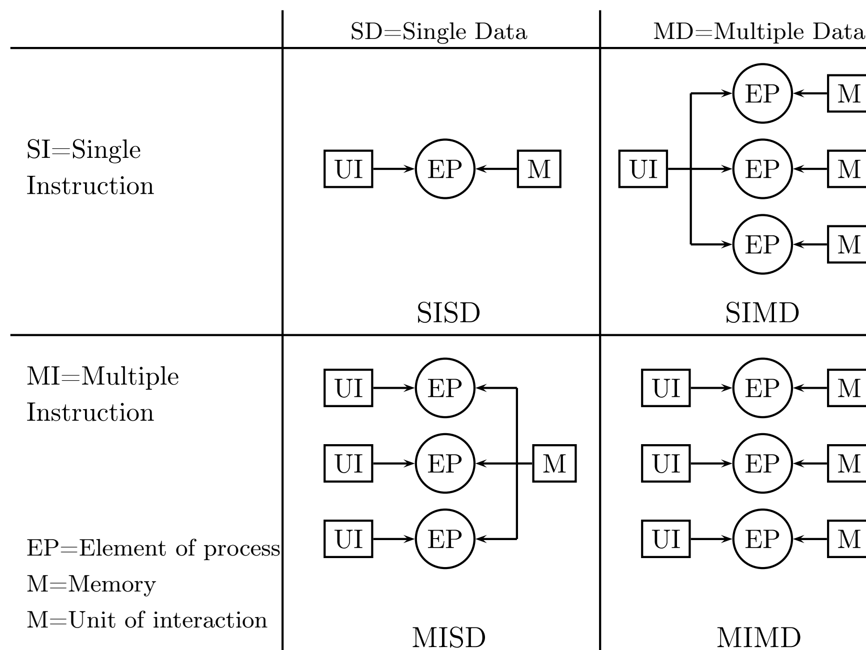


Figura 3.1: Clasificación de Flynn.

Single Instruction Single Data (SISD) Representa a la mayoría de las computadoras secuenciales, las cuales tienen una CPU que ejecuta una instrucción en un momento dado y busca o guarda un dato en un momento dado.

Single Instruction Multiple Data (SIMD) Un flujo único de instrucciones se ejecuta sobre diferentes conjuntos de datos. Se supone que la ejecución de las instrucciones se hace de forma síncrona garantizando así la perfecta unicidad del flujo.

Multiple Instruction Single Data (MISD) Diferentes flujos de instrucciones se ejecutan sobre el mismo conjunto de datos. Aunque es posible imaginar distintas aplicaciones para este tipo de máquina, en realidad es difícil encontrar computadoras que se correspondan con este modelo, debiéndose considerar su existencia exclusivamente a nivel conceptual.

Multiple Instruction Multiple Data (MIMD) Diferentes flujos de instrucciones se ejecutan sobre diferentes conjuntos de datos. Desde el punto de vista teórico representa el modelo más versátil, correspondiendo a una computadora con distintos elementos de procesos, tal que cada uno de ellos ejecuta su propio conjunto de instrucciones de forma asíncrona.

3.2.2. Principales Arquitecturas Paralelas

Actualmente es muy probable que las dos configuraciones de computadoras paralelas mas difundidas sean: los multiprocesadores con memoria compartida y los multiprocesadores con memoria distribuida, ambas configuraciones pertenecen claramente a la clase MIMD según la clasificación de Flynn.

Multiprocesadores de Memoria Compartida

En esta arquitectura todos los Elemento de Proceso (EP) tienen acceso a una memoria común. Las restricciones de acceso a la memoria son únicamente de tipo físico y no de tipo lógico, en el sentido de que el tiempo de acceso puede ser diferente en función del módulo en el que se encuentre la posición de memoria. Una red de interconexión une los distintos EP con los módulos de memoria que es compartida por todos ellos. La comunicación entre distintos EP se hace utilizando la memoria. Para ello, un determinado procesador escribe una variable en memoria y ésta puede ser leída por cualquier otro procesador.

La *figura 3.2* muestra un esquema general del paradigma de memoria compartida, donde todos los procesadores comparten una misma memoria global que es accedida por medio de la red de interconexión.

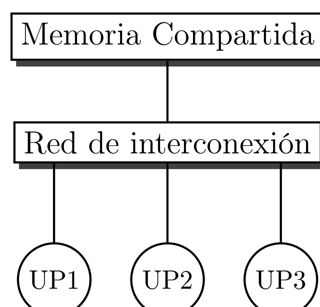


Figura 3.2: Modelo de memoria compartida.

El principal problema de las máquinas de memoria compartida suele ser su elevado coste y su limitada escalabilidad.

Multicomputadoras de Memoria Distribuida

En esta arquitectura cada EP tiene su propia memoria local donde almacena sus datos, no existiendo una memoria global común. Una red de interconexión une los diferentes EP entre sí. La comunicación entre los diferentes procesadores se realiza mediante el paso de mensajes. En la *figura 3.3* se muestra un esquema general de este paradigma.

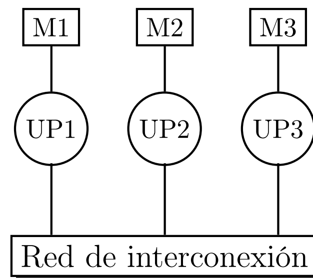


Figura 3.3: Modelo de memoria distribuida.

Ventajas e inconvenientes de las multicomputadoras sobre los multiprocesadores

Ventajas de las multicomputadoras basadas en el modelo de memoria distribuida frente al modelo de memoria compartida:

- No existen conflictos de acceso a memoria o son muy reducidos.
- Admiten un número de procesadores muy elevado permitiendo así un paralelismo masivo.
- Son escalables, en el sentido de que es posible aumentar de forma sencilla el número de procesadores y con ello las prestaciones de la máquina aumentan también de forma lineal, sin que sea necesario incrementar excesivamente el costo de la red de interconexión.
- La tecnología de diseño y fabricación suele ser más asequible.
- Permiten implementaciones sencillas y no excesivamente caras construidas a partir de computadoras personales de propósito general, dando lugar a lo que se conoce comúnmente como clusters de PCs.

Desventajas de las multicomputadoras basadas en el modelo de memoria distribuida frente al modelo de memoria compartida:

- Son más complejos de programar y esta muy alejada de los hábitos y métodos de la programación secuencial.
- Es difícil concebir algoritmos paralelos eficientes.
- Los mensajes ocasionan un tiempo adicional en la ejecución de los programas, generalmente difícil de evitar.
- Es difícil equilibrar la carga computacional entre los distintos procesadores. Más aún, generalmente equilibrar la carga y disminuir el tiempo de comunicación son objetivos contrapuestos.

- A pesar del incremento de memoria asociado con un mayor número de procesadores, debe replicarse el núcleo del sistema operativo y el código de los programas en cada procesador, limitando así, en parte, el aprovechamiento de la memoria.

3.2.3. Modelos de programación

En la actualidad, existen dos grandes paradigmas de programación dedicados a la construcción de algoritmos paralelos: *programación en memoria compartida* y *programación en memoria distribuida*, los cuales se corresponden con las dos principales arquitecturas de computadoras, a saber: multiprocesadores de memoria compartida y multicomputadoras de memoria distribuida.

Programación en Memoria Compartida

En este paradigma la comunicación entre procesos se hace mediante áreas de memoria compartidas. Es responsabilidad del programador la gestión de las estructuras de datos para realizar la compartición de datos entre los diferentes procesos.

Tradicionalmente cada fabricante de multiprocesadores había generado sus propios compiladores con sintaxis diferente. Aunque en 1994 se terminó un borrador de estándar conocido como ANSI X3H5, que trataba de solucionar el problema, nunca fue adoptado de manera formal.

Sin embargo, en Octubre de 1997 surge la primera especificación de OpenMP, donde se definen una serie de directivas que se especifican en el código fuente de la aplicación del usuario. Estas son posteriormente transformadas en código por parte de un compilador que admita la especificación OpenMP. De esta manera, OpenMP se constituye como un estándar en las comunicaciones entre procesos bajo el paradigma de memoria compartida [26].

La especificación de OpenMP soporta programación paralela en memoria compartida en múltiples arquitecturas. OpenMP es una especificación de directivas de compilador, librerías de rutinas y variables de entorno que pueden ser utilizadas para especificar paralelismo de memoria compartida en programas escritos tanto en C/C++ como en Fortran.

Programación en Memoria Distribuida

Al modelo de *Memoria Distribuida* comúnmente se le conoce como el modelo de *Paso de Mensajes*. Bajo este esquema, cada EP dispone de una memoria local que únicamente puede ser accedida por él. La comunicación entre procesadores se realiza por medio de la red de interconexión, mediante el paso de mensajes entre los diferentes EP.

En este esquema, los datos del usuario se encuentran típicamente repartidos entre las diferentes memorias locales a cada EP. Esta repartición no es transparente al usuario y, por lo tanto, la aplicación es responsable de la distribución de los datos y la comunicación de los mismos intercambiando mensajes entre los EP.

El paso de mensajes es un modelo de comunicación muy extendido en la computación paralela. Permite la implementación de programas paralelos en sistemas de memoria distribuida. La programación de este tipo de sistemas se puede realizar de tres formas diferentes:

1. Utilizando un lenguaje especial de programación paralela.
2. Utilizando una extensión de un lenguaje secuencial de alto nivel existente.
3. Utilizando un lenguaje secuencial de alto nivel existente y una biblioteca de paso de mensajes.

Respecto a la primera opción, tal vez el único lenguaje de programación especial que permite paso de mensajes sea Occam, diseñado para su utilización en el procesador Transputer [66].

Algunos ejemplos de la segunda opción son el lenguaje CC++, que es una extensión del lenguaje C++ y Fortran M, que es una extensión del lenguaje FORTRAN. Ambas extensiones permiten programación paralela en general [35].

La tercera opción es la más común, debido a que utiliza lenguajes de alto nivel sin modificar. Para proporcionar las características de paso de mensajes utiliza bibliotecas que proporcionan dicha funcionalidad. Las principales tecnologías que implementan el paso de mensajes son MPI y PVM.

MPI

Message Passing Interface (MPI) [75] es una interfaz estándar de paso de mensajes, que ha sido desarrollada por un consorcio de laboratorios de investigación, universidades y organizaciones comerciales, bajo el nombre de MPI Forum [74].

MPI es un paradigma de paso de mensajes, donde las tareas se comunican a través del envío de mensajes. El objetivo principal de MPI es lograr la portabilidad en un entorno distribuido, de forma similar al de un lenguaje de programación que permita la ejecución transparente de aplicaciones sobre sistemas heterogéneos. Al tratarse de una interfaz, MPI define la sintaxis y semántica de las operaciones que pueden utilizarse para llevar a cabo la comunicación por paso de mensajes. Todas las implementaciones existentes de MPI proporcionan dicha interfaz, manteniendo el comportamiento básico de las operaciones.

La primera versión de MPI [72] pasó a ser estándar en 1994. Esta versión proporciona primitivas de comunicación punto a punto bloqueantes y no bloqueantes. Da soporte a tipos de datos derivados y permite realizar operaciones colectivas. Otras operaciones permiten establecer diferentes topologías de comunicación o modificar el entorno de los procesos.

MPI [73] fue desarrollado en 1997 y añade algunas novedades a la primera versión de MPI. Tal vez las características más destacables sean la gestión dinámica de procesos y el acceso a operaciones de E/S paralela. De hecho, MPI-2 proporciona un conjunto bastante rico de operaciones de E/S, englobadas bajo el nombre de MPI-IO [105], [25]. Estas utilidades están totalmente integradas con el resto de funciones de MPI.

MPI ha sido aceptado como un estándar de paso de mensajes por muchos fabricantes, entre los que se incluye DEC, Hitachi, IBM o Sun Microsystems. Algunas implementaciones son soluciones propietarias; otras, por el contrario, son portables a diferentes arquitecturas. Las implementaciones también se pueden caracterizar por ser de libre distribución o no. Además de implementaciones comerciales, existe un conjunto de implementaciones realizadas por centros de investigación o universidades, que son de libre distribución, entre las que destacan MPICH [76] y Open MPI [80].

PVM

La Parallel Virtual Machine (PVM) consiste en un conjunto de bibliotecas que permite la ejecución de aplicaciones concurrentes o paralelas, en un entorno distribuido y heterogéneo de computadoras con sistema operativo tipo UNIX. Al igual que MPI, se basa en el mecanismo de paso de mensajes.

PVM fue desarrollado para un proyecto de computación distribuida de la NASA llamado Beowulf [83], que consiste en la creación de un cluster de computadoras personales comunes mediante el uso de varios canales de comunicación, que permitían tener una red de alta velocidad para la comunicación de los distintos componentes del cluster. Las características más importantes de PVM son su simplicidad, debido a su fácil instalación y su interfaz de programación sencilla, y su flexibilidad, debido a que se adapta a diferentes arquitecturas y distintas redes y, además, que se trata de un software de dominio público.

A diferencia de PVM, MPI consiste en una especificación que debe guiar las diferentes implementaciones que existen. Esto constituye una primera diferencia entre PVM y MPI, ya que la filosofía bajo la cual se construyeron ambos sistemas es diferente. No obstante, PVM también se puede considerar una especificación de una biblioteca para computación paralela, si se toma como tal la versión Oak Ridge de PVM [38]. Normalmente las especificaciones estándares especifican el mínimo número de características, ya que éstas deben ser obligatoriamente desarrolladas en las diferentes implementaciones. Por el contrario, las implementaciones ofrecen mucha más funcionalidad.

A pesar de que MPI es el estándar de la computación paralela, no se presenta como competencia de otras bibliotecas de este estilo, tipo PVM. Se pueden tener ambas tecnologías ejecutando sobre una misma plataforma hardware. De hecho, existe una implementación denominada Unify [108], que consiste en un subconjunto portable de operaciones pertenecientes a la especificación MPI, construidas encima de PVM. Unify permite que los programadores puedan desarrollar aplicaciones en las que convivan código MPI y código PVM. Se ha diseñado para que un programador pueda migrar de forma gradual las aplicaciones PVM a MPI.

PVM ofrece una máquina virtual, que se implementa mediante los denominados demonios PVM. Esta máquina virtual le proporciona un sistema operativo distribuido sencillo y útil. Además hace uso de interfaces especiales, para interactuar con otros sistemas de gestión de recursos.

A diferencia de MPI, PVM no ofrece un conjunto de operaciones de E/S paralela integrado dentro del resto del sistema.

3.2.4. Diseño de algoritmos paralelos

Diseñar algoritmos paralelos es un proceso altamente creativo. Inicialmente se deben explorar los aspectos independientes de la máquina y los dependientes se deben dejar para el final. El diseño involucra las siguientes cuatro etapas: (1) Particionar el problema en tareas. (2) Se establecen los mecanismos de comunicación entre procesos. (3) Agrupar tareas pequeñas en otras mas grandes. (4) Reparto de trabajo entre los procesadores disponibles.

Al particionar el problema se deben tener en cuenta los siguientes aspectos:

1. El número de tareas debe ser por lo menos un orden de magnitud superior al número de procesadores disponibles para tener flexibilidad en las etapas siguientes.

2. Hay que evitar cálculos y almacenamientos redundantes; de lo contrario el algoritmo puede ser no extensible a problemas más grandes.

3. Hay que tratar de que las tareas sean de tamaños equivalentes ya que facilita el balanceo de la carga de los procesadores.

4. El número de tareas debe ser proporcional al tamaño del problema. De esta forma el algoritmo será capaz de resolver problemas más grandes cuando se tenga más disponibilidad de procesadores. En otras palabras, se debe tratar de que el algoritmo sea escalable; que el grado de paralelismo aumente al menos linealmente con el tamaño del problema.

La comunicación requerida por un algoritmo puede ser definida en dos fases. Primero se definen los canales que conectan las tareas que requieren datos con las que los poseen. Segundo se especifica la información o mensajes que deben ser enviados y recibidos en estos canales.

Dependiendo del tipo de máquina en que se implementará el algoritmo, memoria distribuida o memoria compartida, la forma de definir el modelo de comunicación variará.

En equipos con memoria distribuida, cada tarea tiene una identificación única y las tareas interactúan enviando y recibiendo mensajes hacia y desde tareas específicas. Las librerías más conocidas para implementar el pase de mensajes en ambientes de memoria distribuida son: MPI y PVM.

En equipos con memoria compartida no existe la noción de pertenencia y el envío de datos no se da como tal. Todas las tareas comparten la misma memoria. Semáforos, semáforos binarios, barreras y otros mecanismos de sincronización son usados para controlar el acceso a la memoria compartida y coordinar las tareas, OpenMP es la librería mas común para estos equipos.

Hasta ahora, en la fase de partición se trató de establecer el mayor número posible de tareas con la intensión de explorar al máximo las oportunidades de paralelismo. Esto no necesariamente produce un algoritmo eficiente ya que el costo de comunicación puede ser significativo. En la mayoría de las computadoras paralelas la comunicación es mediante el pase de mensajes y frecuentemente hay que detener los cálculos para enviar o recibir mensajes. Mediante la agrupación de tareas se puede reducir la cantidad de datos a enviar y así reducir el número de mensajes y el costo de comunicación.

La comunicación no solo depende de la cantidad de información enviada. Cada comunicación tiene un costo fijo por el hecho de iniciarse. Reducir el número de mensajes, a pesar de que se envíe la misma cantidad de información, puede ser útil. Así mismo se puede intentar replicar cómputos y/o datos para reducir los requerimientos de comunicación. Por otro lado, también se debe considerar el costo de creación de tareas y el costo de cambio de contexto en caso de que se asignen varias tareas a un mismo procesador.

En caso de tener distintas tareas corriendo en diferentes computadoras con memorias privadas, se debe tratar de que la granularidad sea gruesa: es decir, que exista una cantidad de computo significativa antes de tener necesidades de comunicación. Se puede tener granularidad media si la aplicación corre sobre una máquina de memoria compartida. En estas máquinas el costo de comunicación es menor que en las anteriores, siempre y cuando el número de tareas y procesadores se mantenga dentro de cierto rango. A medida que la granularidad decrece y el número de

procesadores se incrementa, se intensifican la necesidades de altas velocidades de comunicaciones entre los nodos. Esto hace que los sistemas de grano fino por lo general requieran máquinas de propósito especifico.

Por último sólo queda hacer el reparto de tareas, en esta etapa se determina en que procesador se ejecutará cada tarea. Este problema no se presenta en máquinas de memoria compartida, estas proveen asignación dinámica de procesos y los procesos que necesitan de una CPU están en una cola de procesos listos. Cada procesador tiene acceso a esta cola y puede correr el próximo proceso. Hasta ahora no hay mecanismos generales de asignación de tareas para máquinas distribuidas. Esto continúa siendo un problema difícil y que debe ser atacado explícitamente a la hora de diseñar algoritmos paralelos.

La asignación de tareas puede ser estática o dinámica. En la asignación estática, las tareas son asignadas a un procesador al comienzo de la ejecución del algoritmo paralelo, y corren ahí hasta el final. La asignación estática en ciertos casos puede resultar en un tiempo de ejecución menor respecto a asignaciones dinámicas, y también puede reducir el costo de creación de procesos, sincronización y terminación.

En la asignación dinámica se hacen cambios en la distribución de las tareas entre los procesadores en tiempo de ejecución, o sea, hay migración de tareas en tiempo de ejecución. Esto es con el fin de balancear la carga del sistema y reducir el tiempo de ejecución. Sin embargo, el costo de balanceo puede ser significativo y por ende incrementar el tiempo de ejecución.

3.2.5. Aceleración y eficiencia

La aceleración y la eficiencia son dos de las medidas más importantes para apreciar la calidad de la implementación de un algoritmo paralelo en multicomputadoras y multiprocesadores. La ***aceleración*** de un algoritmo paralelo ejecutado usando p procesadores es la razón entre el tiempo que tarda el mejor algoritmo secuencial en ser ejecutado usando un procesador en una supercomputadora y el tiempo que tarda el correspondiente algoritmo paralelo en ser ejecutado en la misma supercomputadora usando p procesadores. La ***eficiencia*** de un algoritmo paralelo ejecutado en p procesadores es la aceleración dividida por p. La ley de Amdahl permite expresar la aceleración máxima obtenible como una función de la fracción del código del algoritmo que sea paralelizable.

Ley de Amdahl

Sea T el tiempo que tarda el mejor algoritmo secuencial en correr en una computadora. Si f es la fracción de operaciones del algoritmo que hay que ejecutar secuencialmente ($0 \leq f \leq 1$), entonces la aceleración máxima que se obtendrá ejecutando el algoritmo en p procesadores se calcula en base a la *fórmula 3.2*

$$A_p = \frac{T}{T(f + (1-f)/p)} + \frac{1}{f + (1-f)/p} \tag{3.2}$$

En el límite, cuando $f \to 0$, $A = p$. Excepto casos muy particulares, este límite nunca es obtenible debido a las siguientes razones:

- Inevitablemente parte del algoritmo es secuencial.
- Hay conflictos de datos y memoria.
- La comunicación y sincronización de procesos consumen tiempo.
- Es muy difícil lograr un balanceo de carga perfecto entre los procesadores.

En la *tabla 3.1* se muestra la aceleración máxima a obtener en términos del porcentaje de código paralelizable y en la *figura 3.4* se muestra la gráfica correspondiente. Se puede observar que la aceleración no es significativa a menos que un gran porcentaje del código sea paralelizable; al menos el 90 % y preferiblemente más del 95 %. Si el 100 % del código es paralelizable, entonces se tendrá lo que se conoce como *aceleración lineal*.

Tabla 3.1: Aceleración en términos del porcentaje de código paralelizable.

$A_p(\%)$	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$	$p=\infty$
10	1.00	1.05	1.08	1.10	1.10	1.11	1.11	1.11	1.11
30	1.00	1.18	1.29	1.36	1.39	1.41	1.42	1.42	1.43
50	1.00	1.33	1.60	1.78	1.88	1.94	1.97	1.98	2.00
70	1.00	1.54	2.11	2.58	2.91	3.11	3.22	3.27	3.33
80	1.00	1.67	2.50	3.33	4.00	4.44	4.71	4.85	5.00
90	1.00	1.82	3.08	4.71	6.40	7.80	8.77	9.34	10.00
100	1.00	2.00	4.00	8.00	16.00	32.00	64.00	128.00	∞

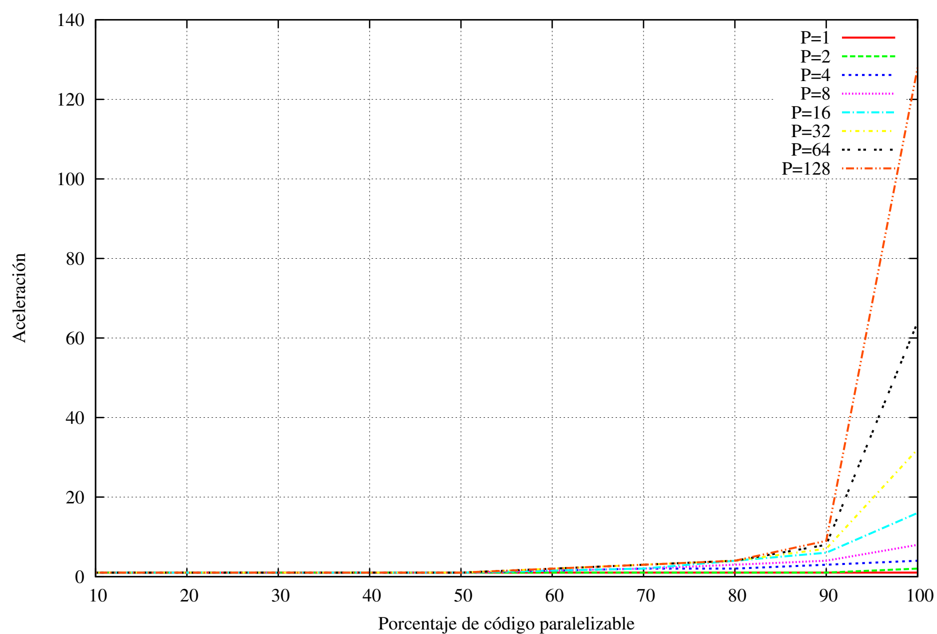


Figura 3.4: Aceleración en términos del porcentaje de código paralelizable.

Aceleración superlineal

Tomando como base la ley de Amdahl, no es posible obtener una aceleración superior a la lineal. Esto esta basado en el supuesto de que un único procesador siempre puede emular procesadores paralelos. Suponga que un algoritmo paralelo A resuelve una instancia del problema π en T_p unidades de tiempo en una computadora paralela con p procesadores. Entonces el algoritmo A puede resolver la misma instancia del problema en $p \times T_p$ unidades de tiempo en la misma computadora pero usando un solo procesador. Por lo tanto la aceleración no puede ser superior a p. Dado que usualmente los algoritmos paralelos tienen asociados costos adicionales de sincronización y comunicación, es muy probable que exista un algoritmo secuencial que resuelva el problema en menos de $p \times T_p$ unidades de tiempo, haciendo que la aceleración sea menos que lineal.

Sin embargo, hay circunstancias algorítmicas especiales en las que una aceleración mayor que la lineal es posible. Por ejemplo, cuando se esta resolviendo un problema de búsqueda, un algoritmo secuencial puede perder una cantidad de tiempo considerable examinando estrategias que no llevan a la solución. Un algoritmo paralelo puede revisar muchas estrategias simultáneamente y se puede dar el caso de que una de ellas dé con la solución rápidamente. En este caso el tiempo del algoritmo secuencial comparado con el paralelo es mayor que el número de procesadores empleados p.

También existen circunstancias de arquitectura que pueden producir aceleraciones superlineales. Por ejemplo, considere que cada CPU en un multiprocesador tiene cierta cantidad de memoria cache. Comparando con la ejecución en un solo procesador, un grupo de p procesadores ejecutando un algoritmo paralelo tiene p veces la cantidad de memoria cache. Es fácil construir ejemplos en los que la tasa colectiva de hits de cache para los p procesadores sea significativamente mayor que los hits de cache del mejor algoritmo secuencial corriendo en un solo procesador [50]. En estas condiciones el algoritmo paralelo puede correr más de p veces más rápido. Estas circunstancias son especiales y se dan raras veces. Lo que aplica en la mayoría de los casos es la ley de Amdahl.

Eficiencia

Si se normaliza la aceleración dividiéndola por el número de procesadores se obtiene la eficiencia, véase la *fórmula 3.3*.

$$\eta = \frac{A_p}{p} \tag{3.3}$$

En la *tabla 3.2* se puede observar que cuando se incrementa el número de procesadores p, la eficiencia η decrementa. La *figura 3.5* muestra el gráfico respectivo. Esto refleja el hecho de que a medida que el código es dividido en trozos más y más pequeños, eventualmente algunos procesadores deben permanecer ociosos esperando que otros terminen ciertas tareas. En la práctica esto no es así, ya que solo a medida que los problemas se hacen más grandes es que se asignan a más procesadores.

Tabla 3.2: Eficiencia η para p procesadores y los distintos porcentajes de código paralelizable.

$A_p(\%)$	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$	$p=128$
0	1,00	0,50	0,25	0,13	0,06	0,03	0,02	0,01
10	1,00	0,53	0,27	0,14	0,07	0,03	0,02	0,01
30	1,00	0,59	0,32	0,17	0,09	0,04	0,02	0,01
50	1,00	0,67	0,40	0,22	0,12	0,06	0,03	0,02
70	1,00	0,77	0,53	0,32	0,18	0,10	0,05	0,03
80	1,00	0,83	0,63	0,42	0,25	0,14	0,07	0,04
90	1,00	0,91	0,77	0,59	0,40	0,24	0,14	0,07
100	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00

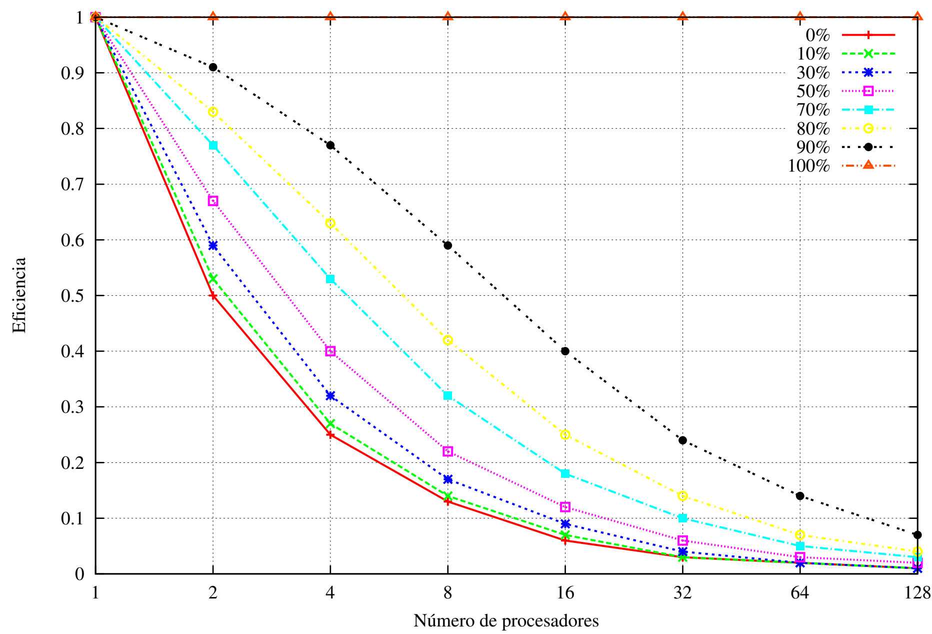


Figura 3.5: Eficiencia η para p procesadores y los distintos porcentajes de código paralelizable.

Costo de comunicación

En el supuesto que el código sea 100 % paralelizable y por lo tanto la aceleración es $A_p = p$; despreciando el tiempo de comunicación entre procesadores. Considerando el tiempo de comunicación o latencia T_c, la aceleración decrece aproximadamente según la *fórmula 3.4* y para que la aceleración no sea afectada por la latencia de comunicación se necesita: véase la *fórmula 3.5*.

$$A_p = \frac{T}{T/p + T_c} < p \tag{3.4}$$

$$\frac{T}{p} \gg T_c \Rightarrow p \ll \frac{T}{T_c} \tag{3.5}$$

Esto significa que a medida que se divide el problema en partes más y más pequeñas para poder correr el problema en más procesadores, llega un momento en que el costo de comunicación Tc se hace muy significativo y desacelera el cómputo.

3.3. Computación Grid

El Grid es una tecnología que tiene como objetivo compartir recursos en Internet de forma uniforme, transparente, segura, eficiente y fiable. La tecnología Grid nace dentro de la comunidad de supercomputación. El concepto es análogo a las redes de suministro eléctrico que ofrecen un único punto de acceso a un conjunto de recursos distribuidos (supercomputadoras, clusters, almacenamiento, fuentes de información, personal, bases de datos, entre otros).

Esta tecnología permite interconectar recursos en diferentes dominios de administración respetando sus políticas internas de seguridad y su software de gestión de recursos en la Intranet. Coordina recursos que no están sujetos a un control centralizado, usando protocolos e interfaces basados en estándares abiertos y de propósito general. La tecnología Grid propone agregar y compartir recursos de computación distribuidos entre diferentes organizaciones e instituciones, a través de redes de alta velocidad, de modo que el acceso a los mismos sea sencillo, flexible y fiable.

3.3.1. Antecedentes y perspectivas de la computación Grid

En 1969, Leonard Kleinrock [57], ya asemejaba los servicios computacionales a los servicios de suministro eléctrico, de ahí el término Grid, con el que se denomina en inglés la red eléctrica. Entre un Grid eléctrico y uno computacional, existen algunas diferencias: mientras que la electricidad es producida más económicamente desde una gran central, los recursos computacionales son producidos más económicamente de forma distribuida a través de servidores o clusters.

Tras el desarrollo inicial de Internet durante los últimos treinta años, y la revolución que ha supuesto el World Wide Web, se desarrolla el concepto *Grid de cálculo* con el fin de compartir y utilizar recursos de computación; este concepto se amplía con la utilización de recursos de almacenamiento masivo distribuido, dando lugar al *Grid de datos*. En 1995 durante el congreso SuperComputing'95[1], la experiencia I-WAY demostró la posibilidad de ejecutar aplicaciones distribuidas de diferentes áreas científicas en una red de 17 centros de USA conectados con una red de alta velocidad (155 Mbps). Este fue el punto de partida de varios proyectos en diferentes áreas, con un denominador común: compartir recursos distribuidos de computación.

Años más tarde, en 1998, esta idea empezaba a materializarse con el inicio de la Globus Alliance, creada para organizar la investigación y el desarrollo de tecnologías básicas para el Grid y con el libro "The Grid: Blueprint for a Future Computing Infrastructure", editado por Ian Foster y Carl Kesselmann. En el prólogo, Foster y Kesselmann plantean la analogía con la "electrical power grid": el usuario debe tener acceso a los recursos computacionales en condiciones similares a las que tiene para utilizar la energía eléctrica: desde cualquier sitio (pervasive), con un interfaz uniforme (consistent), pudiendo confiar en su funcionamiento (dependable), y a un coste asequible (inexpensive).

El siguiente paso fue la extensión a recursos conectados a través de Internet: la red Entropía2 agregó en dos años 30.000 computadoras, logrando por ejemplo calcular el mayor número primo conocido; el sistema SETI funciona en más de medio millón de PCs analizando los datos del radio telescopio Arecibo para la búsqueda de señales de inteligencia extraterrestre. La computación en

[1] SuperComputing 2005: Seattle, WA, USA
Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing, November 12-18, 2005, Seattle, WA, USA. IEEE Computer Society 2005.

Física de Partículas es un ejemplo claro de esta evolución: los experimentos del anterior acelerador LEP del Consejo Europeo para la Investigación Nuclear (CERN) pasaron de usar computadoras Cray a clústers de computadoras en la primera parte de la década de los 90. Un ejemplo de las posibilidades de compartir estos recursos, lo proporcionó la simulación en sólo un fin de semana utilizando los recursos completos del CERN de más de cinco millones de colisiones e^+e^-, para mejorar los resultados de la búsqueda del *Bosón de Higgs* del experimento DELPHI.

En el año 2000, entre los primeros proyectos Grid surge la Information Power Grid (IPG) de la NASA y la iniciativa de la Nacional Science Foundation (NSF). En Europa, el programa comunitario Information Society Technologies (IST) lanza el proyecto European DataGrid (EDG) coordinado por el CERN, con el objetivo de "construir la próxima generación de infraestructura de computación que permita cálculo intensivo y análisis de bases de datos compartidas a gran escala, desde cientos de Terabytes a Petabytes, entre comunidades científicas ampliamente distribuidas".

En el año 2001 se lanzan varios proyectos IST en el área científica, como GridLab, EGSO, DataTag y CrossGrid; este último, enfocado al desarrollo de aplicaciones interactivas en el entorno Grid.

El interés se ha extendido rápidamente a la aplicación de esta tecnología en el entorno de grandes compañías. IBM considera "Grid Computing" como la opción para integrar múltiples computadoras distribuidos sobre una red (en particular Internet), de modo que actúen como una gran supercomputadora.

Gracias a la evolución de las tecnologías en las redes de comunicación (RedIRIS2, Gèant2), se creó la iniciativa IRISGrid, como la primera propuesta española para el desarrollo de esta tecnología, en la que participan numerosos grupos de investigación con experiencia en tecnología Grid. Esta iniciativa, reforzada tras la reciente acreditación de RedIRIS como autoridad de certificación EUGridPMA, propone un entorno colaborativo (*e-Ciencia*) entre los diferentes grupos de investigación.

3.3.2. Tecnología Grid

La idea básica de la tecnología Grid es aprovechar de modo óptimo los recursos proporcionados por equipos de computación distribuidos, conectados a una red de banda ancha, mediante el uso de un software adecuado para planificar su utilización ("scheduler" y "resource broker") que tenga en cuenta las prioridades y diferentes periodos de demanda de los usuarios. Debe incluir mecanismos de autenticación y autorización basados en certificados digitales, y contar con una gestión basada en una instalación automatizada, flexible y dinámica, acompañada de monitorización en tiempo real de todos los recursos (tanto de red como de equipos de cálculo).

Los servicios e interfaces de programación de aplicaciones y herramientas de desarrollo de software son clasificados de acuerdo con su función en la capacidad de compartir recursos; se establecen los requerimientos que deben satisfacer y se plantea la importancia de definir un conjunto concreto de protocolos "intergrid" para habilitar la interoperatibilidad a través de diferentes sistemas Grid. El establecimiento, gestión y explotación de recursos y transferencia de resultados entre diferentes Virtual Organization (VO), requiere de una nueva tecnología. En términos de arquitectura Grid, se identifican los componentes fundamentales del sistema, se especifica el propósito y la función de estos componentes, y cómo interactúa un componente con el resto de componentes.

En un entorno de red, la interoperatibilidad significa protocolos comunes; por tanto, una arquitectura Grid es ante todo una arquitectura de protocolos, que definen los mecanismos básicos con los que los usuarios de VOs y los recursos son administrados. Una arquitectura abierta, facilita la interoperatibilidad, extensión, portabilidad y compartición de código; los protocolos estándar facilitan la definición de servicios estándar que proporcionan capacidades mejoradas.

Para crear un Grid operativo, se construyen Application Programming Interface (API) y Software Development Kit (SDK) para proporcionar las reglas de programación necesarias.

Por otra parte, para que una arquitectura Grid sea posible, debe haber una capa de software que se sitúe lógicamente entre las aplicaciones que hagan uso de su potencial, y los recursos a compartir. Esta capa es lo que se conoce como el middleware Grid, que abstrae de los aspectos de más bajo nivel, proporcionando un acceso transparente a las aplicaciones que requieran un uso intensivo de recursos. Un ejemplo de middleware Grid es el Globus Toolkit [40] desarrollado de forma comunitaria y en licencia de Open Source, y que hoy en día es una referencia de las tecnologías Grid en todo el mundo.

Estas herramientas, basadas e implementadas sobre arquitecturas Grid, deben proporcionar la siguiente funcionalidad básica:

- **Gestión compartida de los recursos.** Los recursos deben verse como algo compartido por todas las entidades participantes, de manera que su gestión debe ser manejada por los administradores de la VO.
- **Entorno de Seguridad a nivel de usuarios, recursos y procesos.** Deben de garantizarse las identidades de los usuarios y de los recursos, además de asegurar que las comunicaciones se realizan a través de canales seguros.
- **Gestión eficiente de los recursos.** El acceso a estos recursos debe ser controlado de manera eficiente.
- **Transmisión rápida y fiable de datos.** Se definen e implementan protocolos y redes adecuados a las necesidades.
- **Estándares abiertos.** Con el objetivo de permitir una mejor integración de nuevos componentes.

3.3.3. Arquitecturas Grid

Los diferentes protocolos, servicios y aplicaciones de una arquitectura basada en tecnologías Grid se estructuran en diferentes capas como las que se describen a continuación:

- **Infraestructura:** Corresponde a los recursos que se van a desplegar para su uso por los usuarios de cada VO. Los recursos pueden ser computadoras, clusters, supercomputadoras, bases de datos, sistemas de almacenamiento en red, etc.
- **Sistemas de Información y Monitorización:** En un entorno Grid se requiere de un sistema que monitorice y mantenga información actualizada sobre los recursos disponibles en la infraestructura.

- **Conectividad:** Se refiere tanto a los medios existentes para comunicarse con los recursos (por ejemplo Internet), como a los protocolos utilizados.
- **Seguridad:** El sistema debe garantizar el acceso a los recursos de una manera fiable, asegurando la privacidad de los datos, la autenticación de los usuarios y la gestión de recursos a nivel de una VO.
- **Aplicaciones:** Aquéllas que hacen uso de las infraestructuras Grid mediante un Middleware Grid.

3.3.4. Open Grid Services Architecture

Como se ha visto en el apartado anterior, el primero de los puntos clave a la hora de definir una arquitectura Grid es la infraestructura a emplear, es decir, los recursos a desplegar y de qué manera estos recursos pueden ser definidos y como se ha de interactuar con ellos. Por ello, en este apartado se incluye una visión de las tecnologías actuales que se utilizan para la definición de los recursos a desplegar. Entre las tecnologías más utilizadas en la actualidad destacan los Web Services (WS), los servicios Grid (OGSI o WSRF) y CORBA, entre otros.

Open Grid Services Architecture (OGSA) soporta la creación, mantenimiento y agrupación de servicios integrados dentro de las VOs. Pese a esto, el estándar OGSA no se compromete con la forma en que se implementarán los servicios que sustenta, especificando únicamente las interfaces y no el comportamiento. Sobre la arquitectura OGSA se definieron entonces especificaciones de servicios que cada vez más son aceptadas en la comunidad informática. En este marco, se identificaron los WS como la base de los nuevos Servicios Grid. Dado que OGSA define una arquitectura estándar y abierta para aplicaciones basadas en tecnologías Grid, el objetivo principal del Open Grid Forum (OGF) está siendo estandarizar los servicios que se necesitan en el desarrollo de una aplicación Grid (seguridad, servicios para gestionar recursos, servicios para el manejo de trabajos, etc.), desarrollando y especificando un conjunto de interfaces estándar genéricos para toda aplicación Grid que pretenda beneficiarse de los servicios que la arquitectura pueda ofrecer.

Para el desarrollo de los requisitos comunes de una arquitectura Grid, se podría haber tomado como base los middlewares anteriores al Grid relacionados con la computación distribuida (como CORBA, RMI, o incluso RPC), ya que son estándares altamente aceptados por la comunidad informática. Algunas de las razones más importantes por las que se eligió la tecnología de WS se exponen a continuación.

Los WS permiten crear aplicaciones cliente/servidor al igual que otros midlewares como CORBA o RMI. En este caso, una aplicación basada en WS consta de la parte cliente que ejecuta los servicios ofrecidos por los servidores, permitiendo el descubrimiento del servicio, de modo que el cliente busca el servicio que necesita y una vez lo encuentra lo ejecuta. Los WS utilizan Web Services Description Language (WSDL) para permitir a los clientes poder descubrir e interactuar con los servicios. De nuevo, en CORBA existe un lenguaje análogo que es el Interface Definition Language (IDL). Sin embargo, la principal ventaja de WSDL es que es estándar, por lo que descubrir los servicios resulta más fácil, y por tanto las herramientas para poder tratar e interpretar este tipo de información son mucho más extensas.

Otra de las ventajas de los WS, y que otras tecnologías como CORBA no ofrecen, es que los WS utilizan para la comunicación el protocolo de alto nivel Simple Object Access Protocol (SOAP)

[77], que puede funcionar sobre otros protocolos ya existentes como por ejemplo http, ftp, https y ftps. Al ser estos protocolos ampliamente utilizados y soportados, este aspecto aporta una gran ventaja respecto a otros middlewares como CORBA, evitando numerosos problemas organizativos asociados (como cortafuegos, proxies de Internet, etc.).

También existen puntos negativos en los WS con respecto a las necesidades de OGSA. Uno de los aspectos más notorios es la carencia de la persistencia del estado de un servicio, aunque también existen otras como la carencia de notificaciones de eventos, manejo del ciclo de vida de un servicio, transacciones etc. Si bien es posible implementar de forma particular soluciones a estos problemas, se hace necesaria la extensión de un estándar. Estas carencias son recogidas por OGSA y se corrigen en las especificaciones de Web Services Resource Framework (WSRF) [79].

3.3.5. Open Grid Service Infraestructure

Open Grid Service Infraestructure (OGSI) es una especificación de OGSA desarrollada por los miembros del Global Grid Forum (GGF). OGSI fue creado con la esperanza de que acabara convergiendo a WS, lo que no ha ocurrido principalmente por diversos motivos [88]:

- La especificación OGSI es extremadamente densa y compleja lo cual dificulta los desarrollos.
- OGSI no interactúa bien con herramientas que trabajan con WS, se requiere de herramientas específicas como Globus 3.X o OGSI.NET.
- Su intensa orientación a objetos, a pesar de las ventajas que este paradigma proporciona, implica ciertas incompatibilidades con los WS que no siempre se ciñen a este modelo.
- Por tanto, para garantizar la convergencia a WS, se substituyó esta especificación por el WSRF.

3.3.6. Web Services Resource Framework

La Organization for the Advancement of Structured Information Standards (OASIS) ha sido la encargada de definir WSRF, la cual básicamente especifica como proporcionar estado a los WS y poder agregar las características definidas en OGSA a los WS. Se puede decir que WSRF es una extensión de los WS para ser aceptado por OGSA.

WSRF garantiza la convergencia a WS, pues estos servicios son utilizados como plataforma de comunicación de los Servicios Grid, por el contrario OGSI no utilizaba los WS e interactuaba directamente con los Servicios Grid.

3.3.7. Diferencias entre OGSI y WSRF

Las principales ventajas que aporta WSRF frente a OGSI son las siguientes [88]:

Las dos especificaciones abarcan los mismos requisitos determinados por OGSA, pero la especificación WSRF es mucho menos densa que la de OGSI, conteniendo sólo 5 documentos que se corresponden estrictamente con WSRF.

OGSI no se integra bien con las herramientas actuales de los WS, mientras que WSRF sí interactúa con estas herramientas correctamente al estar basado puramente en WS.

OGSI está muy orientado a objetos, lo cual provoca ciertas incompatibilidades con los WS. Por el contrario, WSRF separa claramente la parte de WS con la parte de gestión de recursos (como creación de estancias y estado).

3.3.8. Uniendo las partes

Hasta ahora se tiene el software (Web y Grid Services) y los estándares (OGSA) que permiten que la computación Grid trabaje satisfactoriamente, sin embargo tiene que haber un software intermediario (Middleware) que permita la comunicación, entre el sistema operativo y las aplicaciones en cada sistema. En la actualidad se han desarrollado diferentes softwares Middleware para la implementación de Grid. En la *figura 3.6* se muestra como Globus Toolkit 4 cumple con lo establecido por OGSA e implementa WSRF.

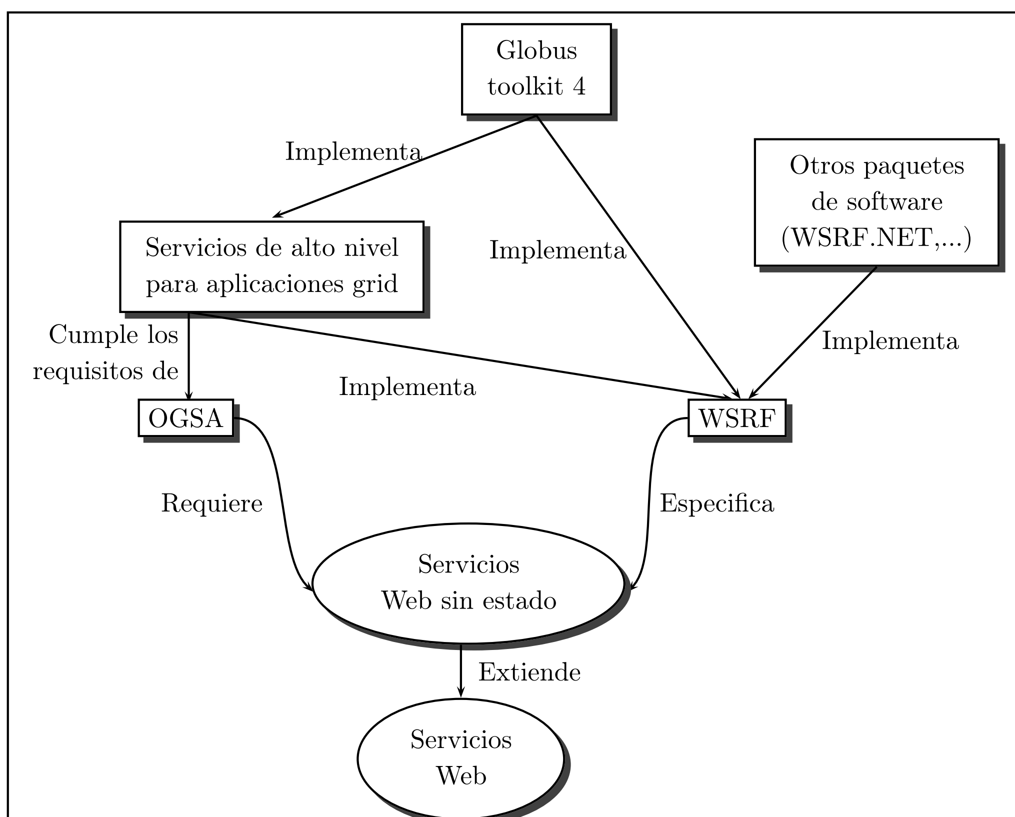


Figura 3.6: Globus Toolkit 4

3.4. Resumen del capítulo

En este capítulo se presentó una reseña sobre la supercomputación y su aplicación a los problemas de gran desafío como la física de altas energías, la genética, predicción climática, etc.

Se han descrito las principales estrategias de paralelización, basadas en las arquitecturas de

computadoras paralelas. Y se ha dado una reseña de la computación Grid y su aplicación a los problemas de gran desafío.

Como el problema de construir *covering arrays* es un problema *NP-Completo* y además en muchos casos es difícil de tratar por las capacidades limitadas de memoria, almacenamiento y poder de cálculo, de las computadoras convencionales, en este trabajo se propone su construcción utilizando supercomputadoras.

En el siguiente capítulo se describe la metodología para construir una herramienta que verifique si un arreglo es o no un CA utilizando supercomputación y computación GRID.

Capítulo 4

Metodología propuesta

En este capítulo se describe como verificar CA. Se presenta un algoritmo que lleva a cabo esta verificación y su implementación en secuencial; sin embargo, cuando los CA son grandes este algoritmo tiene el problema del tiempo que tarda en obtener una solución. Para poder hacer frente a esta situación se presentan dos soluciones, la primera de ellas hace uso de computación paralela y la segunda de computación Grid.

4.1. Proceso para validar CAs

Los CAs son objetos combinatorios que no necesitan estar balanceados, es decir en un CA que satisface una cobertura de fuerza t, cada posible combinación de valores de variables (*t-ada*) debe estar presente al menos una vez pero no necesariamente el mismo número de veces. En la sección 2.11 se muestran los detalles específicos y formales.

Un CA está representado por una matriz C de tamaño $N \times k$, donde cada elemento $c_{i,j}$ toma como valor un símbolo del conjunto $S = \{0, 1, 2, ..., v-1\}$, tal que cada $N \times t$ subarreglo contiene al menos una vez todas las posibles combinaciones de los v^t símbolos para alfabetos uniformes y $\prod_{i=0}^{i=k-1} v_i$ para alfabetos mixtos. El número de subarreglos Λ de tamaño $N \times t$ está determinado por las Λ.

Para validar que todas las posibles combinaciones de los $\prod_{i=0}^{i=k-1} v_i$ símbolos aparecen al menos una vez en cada una de las Λ *t-adas* del CA, se propone el uso de una matriz P de tamaño $\Lambda \times \prod_{i=0}^{i=t-1} maxV_i$, donde $maxV_i$ contiene las t cardinalidades de mayor valor, un vector V de tamaño k con las cardinalidades de las variables, y una matriz J de tamaño $\Lambda \times t$ donde cada renglón representa una *t-ada* del CA.

En P se lleva el registro de las combinaciones de símbolos existentes por cada *t-ada* del CA. Cada una de las Λ *t-adas* se mapea a un renglón, en P, siguiendo la *expresión 4.1*.

$$\sum_{i=0}^{t-1} \binom{x_i}{i+1} \tag{4.1}$$

Para verificar que todas las combinaciones de símbolos de cada t columnas de C aparezcan al menos una vez, es necesario mapear cada t combinaciones de símbolos a un valor decimal que

corresponde a una columna en P. Para ello, al vector V se le puede ver como un sistema de numeración donde cada uno de sus elementos representa el valor máximo (o la base del sistema de numeración) que puede tomar este elemento, por lo tanto, para contar los elementos de una *t-ada*, se realiza la transformación de los t símbolos correspondientes a la *t-ada* representada por J_i, a un valor decimal, siguiendo la expresión 4.2.

$$\begin{aligned} &\sum_{i=1}^{t-1} J_i, \\ &\text{s.t. } J_i = J_{i-1} \times V_{J_i} + J_i \end{aligned} \tag{4.2}$$

Para alfabetos uniformes se validará que es un CA, si al final no existe ningún cero en P. Para alfabetos mixtos se validará que es un MCA, si para cada *t-ada* (J_i) no existe ningún cero en las primeras $\prod_{i=0}^{i=t-1} V_{J_i}$ columnas de su correspondiente renglón en P.

Para comprender mejor este proceso de validación, en la siguiente sección se expone un ejemplo.

4.2. Validando un $MCA(6; 4, 2^3\ 3^1, 2)$

En la *tabla 4.1* se muestra un MCA de $N = 6$, $k = 4$, $v = 2^3\ 3^1$, $t = 2$. Para validar que realmente sea un MCA, primero se inicializan el vector de cardinalidades V, el vector $maxV$ con las t cardinalidades de mayor valor, la matriz de *t-adas* J y la matriz de control P. Para este caso V es de tamaño 4, $maxV$ de tamaño 2, J de tamaño 6×2 y P de tamaño 6×6, como se muestra en la *tabla 4.2*.

Tabla 4.1: $MCA(6; 4, 2^3\ 3^1, 2)$.

0	0	0	0
1	1	1	0
0	1	1	1
1	0	0	1
0	1	0	2
1	0	1	2

Tabla 4.2: Inicialización de variables.

$V = \{2, 2, 2, 3\}$

$maxV = \{2, 3\}$

J:

0	1
0	2
0	3
1	2
1	3
2	3

P:

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

El siguiente paso es verificar que todas las combinaciones de símbolos para cada *t-ada* existan; para ello se toma la primer *t-ada* de la matriz J, se calcula cual es su renglón correspondiente en P usando la *expresión 4.1*, posteriormente se leen renglón a renglón sus correspondientes símbolos en C, con cada t símbolos se obtiene el número de columna al que corresponden en P aplicando la *expresión 4.2*, este proceso se repite por cada *t-ada*. Para ver el proceso completo vea las *tablas 4.3, 4.4, 4.5, 4.6, 4.7 y 4.8*.

Tabla 4.3: Validando un MCA, *t-ada* {0,1}.

J		Renglon en P, Expresin 4.1				C				Columnas en P, Expresin 4.2				P					
0	1	0	1	=	$\binom{0}{1}+\binom{1}{2}=0$	0	0	0	0	0	0	=	$0\times2+0=0$	1	2	2	1	0	0
0	2	0	2	=	$\binom{0}{1}+\binom{2}{2}=1$	1	1	1	0	1	1	=	$1\times2+1=3$	0	0	0	0	0	0
0	3	0	3	=	$\binom{0}{1}+\binom{3}{2}=3$	0	1	1	1	0	1	=	$0\times2+1=1$	0	0	0	0	0	0
1	2	1	2	=	$\binom{1}{1}+\binom{2}{2}=2$	1	0	0	1	1	0	=	$1\times2+0=2$	0	0	0	0	0	0
1	3	1	3	=	$\binom{1}{1}+\binom{3}{2}=4$	0	1	0	2	0	1	=	$0\times2+1=1$	0	0	0	0	0	0
2	3	2	3	=	$\binom{2}{1}+\binom{3}{2}=5$	1	0	1	2	1	0	=	$1\times2+0=2$	0	0	0	0	0	0

Tabla 4.4: Validando un MCA, *t-ada* {0,2}.

J		Renglon en P, Expresin 4.1				C				Columnas en P, Expresin 4.2				P					
0	1	0	1	=	$\binom{0}{1}+\binom{1}{2}=0$	0	0	0	0	0	0	=	$0\times2+0=0$	1	2	2	1	0	0
0	2	0	2	=	$\binom{0}{1}+\binom{2}{2}=1$	1	1	1	0	1	1	=	$1\times2+1=3$	2	1	1	2	0	0
0	3	0	3	=	$\binom{0}{1}+\binom{3}{2}=3$	0	1	1	1	0	1	=	$0\times2+1=1$	0	0	0	0	0	0
1	2	1	2	=	$\binom{1}{1}+\binom{2}{2}=2$	1	0	0	1	1	0	=	$1\times2+0=2$	0	0	0	0	0	0
1	3	1	3	=	$\binom{1}{1}+\binom{3}{2}=4$	0	1	0	2	0	0	=	$0\times2+0=0$	0	0	0	0	0	0
2	3	2	3	=	$\binom{2}{1}+\binom{3}{2}=5$	1	0	1	2	1	1	=	$1\times2+1=3$	0	0	0	0	0	0

Tabla 4.5: Validando un MCA, *t-ada* {0,3}.

J		Renglon en P, Expresin 4.1				C				Columnas en P, Expresin 4.2				P					
0	1	0	1	=	$\binom{0}{1}+\binom{1}{2}=0$	0	0	0	0	0	0	=	$0\times3+0=0$	1	2	2	1	0	0
0	2	0	2	=	$\binom{0}{1}+\binom{2}{2}=1$	1	1	1	0	1	0	=	$1\times3+0=3$	2	1	1	2	0	0
0	3	0	3	=	$\binom{0}{1}+\binom{3}{2}=3$	0	1	1	1	0	1	=	$0\times3+1=1$	0	0	0	0	0	0
1	2	1	2	=	$\binom{1}{1}+\binom{2}{2}=2$	1	0	0	1	1	1	=	$1\times3+1=4$	1	1	1	1	1	1
1	3	1	3	=	$\binom{1}{1}+\binom{3}{2}=4$	0	1	0	2	0	2	=	$0\times3+2=2$	0	0	0	0	0	0
2	3	2	3	=	$\binom{2}{1}+\binom{3}{2}=5$	1	0	1	2	1	2	=	$1\times3+2=5$	0	0	0	0	0	0

Tabla 4.6: Validando un MCA, *t-ada* {1,2}.

J		Renglon en P, Expresin 4.1				C				Columnas en P, Expresin 4.2				P					
0	1	0	1	=	$\binom{0}{1}+\binom{1}{2}=0$	0	0	0	0	0	0	=	$0\times2+0=0$	1	2	2	1	0	0
0	2	0	2	=	$\binom{0}{1}+\binom{2}{2}=1$	1	1	1	0	1	1	=	$1\times2+1=3$	2	1	1	2	0	0
0	3	0	3	=	$\binom{0}{1}+\binom{3}{2}=3$	0	1	1	1	1	1	=	$1\times2+1=3$	2	1	1	2	0	0
1	2	1	2	=	$\binom{1}{1}+\binom{2}{2}=2$	1	0	0	1	0	0	=	$0\times2+0=0$	1	1	1	1	1	1
1	3	1	3	=	$\binom{1}{1}+\binom{3}{2}=4$	0	1	0	2	1	0	=	$1\times2+0=2$	0	0	0	0	0	0
2	3	2	3	=	$\binom{2}{1}+\binom{3}{2}=5$	1	0	1	2	0	1	=	$0\times2+1=1$	0	0	0	0	0	0

Tabla 4.7: Validando un MCA, *t-ada* {1,3}.

J		Renglon en P, Expresin 4.1				C				Columnas en P, Expresin 4.2				P					
0	1	0	1	=	$\binom{0}{1}+\binom{1}{2}=0$	0	0	0	0	0	0	=	$0\times3+0=0$	1	2	2	1	0	0
0	2	0	2	=	$\binom{0}{1}+\binom{2}{2}=1$	1	1	1	0	1	0	=	$1\times3+0=3$	2	1	1	2	0	0
0	3	0	3	=	$\binom{0}{1}+\binom{3}{2}=3$	0	1	1	1	1	1	=	$1\times3+1=4$	2	1	1	2	0	0
1	2	1	2	=	$\binom{1}{1}+\binom{2}{2}=2$	1	0	0	1	0	1	=	$0\times3+1=1$	1	1	1	1	1	1
1	3	1	3	=	$\binom{1}{1}+\binom{3}{2}=4$	0	1	0	2	1	2	=	$1\times3+2=5$	1	1	1	1	1	1
2	3	2	3	=	$\binom{2}{1}+\binom{3}{2}=5$	1	0	1	2	0	2	=	$0\times3+2=2$	0	0	0	0	0	0

Al final se puede validar que el array mostrado en la *tabla 4.1* es un MCA, debido a que para cada *t-ada* no existe ningún cero en las primeras $\prod_{i=0}^{i=t-1} V_{J_i}$ columnas de su correspondiente renglón en P (*véase la tabla 4.9*).

Tabla 4.8: Validando un MCA, *t-ada* {2,3}.

J		Renglon en P, Expresin 4.1				C				Columnas en P, Expresin 4.2				P					
0	1	0	1	=	$\binom{0}{1}+\binom{1}{2}=0$	0	0	0	0	0	0	=	$0\times3+0=0$	1	2	2	1	0	0
0	2	0	2	=	$\binom{0}{1}+\binom{2}{2}=1$	1	1	1	0	1	0	=	$1\times3+0=3$	2	1	1	2	0	0
0	3	0	3	=	$\binom{0}{1}+\binom{3}{2}=3$	0	1	1	1	1	1	=	$1\times3+1=4$	2	1	1	2	0	0
1	2	1	2	=	$\binom{1}{1}+\binom{2}{2}=2$	1	0	0	1	0	1	=	$0\times3+1=1$	1	1	1	1	1	1
1	3	1	3	=	$\binom{1}{1}+\binom{3}{2}=4$	0	1	0	2	0	2	=	$0\times3+2=2$	1	1	1	1	1	1
2	3	2	3	=	$\binom{2}{1}+\binom{3}{2}=5$	1	0	1	2	1	2	=	$1\times3+2=5$	1	1	1	1	1	1

Tabla 4.9: Validación del MCA

J		Renglon en P, Expresin 4.1				$maxV$				P					
0	1	0	1	=	$\binom{0}{1}+\binom{1}{2}=0$	0	1	$=2\times2=$	4	1	2	2	1	0	0
0	2	0	2	=	$\binom{0}{1}+\binom{2}{2}=1$	0	2	$=2\times2=$	4	2	1	1	2	0	0
0	3	0	3	=	$\binom{0}{1}+\binom{3}{2}=3$	0	3	$=2\times3=$	6	2	1	1	2	0	0
1	2	1	2	=	$\binom{1}{1}+\binom{2}{2}=2$	1	2	$=2\times2=$	4	1	1	1	1	1	1
1	3	1	3	=	$\binom{1}{1}+\binom{3}{2}=4$	1	3	$=2\times3=$	6	1	1	1	1	1	1
2	3	2	3	=	$\binom{2}{1}+\binom{3}{2}=5$	2	3	$=2\times3=$	6	1	1	1	1	1	1

4.3. Validando un CCA(7;7,2,2)

Como se explico en la *sección 2.3* un CCA es una CM de tamaño $k \times k$ que es construida mediante $k-1$ rotaciones de un vector inicial s de tamaño k. La característica principal de un CCA es que por medio de automorfismos se reduce el espacio de búsqueda para verificar que sea un CA. En la *tabla 4.10* se muestra un $CCA(7;7,2,2)$ creado a partir de $s = \{0,0,0,1,0,1,1\}$.

Tabla 4.10: Un $CCA(7;7,2,2)$.

0	0	0	1	0	1	1
0	0	1	0	1	1	0
0	1	0	1	1	0	0
1	0	1	1	0	0	0
0	1	1	0	0	0	1
1	1	0	0	0	1	0
1	0	0	0	1	0	1

En la *tabla 4.11* muestra el conjunto Λ de t-adas requerido para verificar $CCA(7;7,2,2)$; esto es el conjunto completo de las diferentes t-adas.

Tabla 4.11: Conjunto Λ para validar un $CCA(7;7,2,2)$ completamente.

$\{0,1\}$	$\{1,3\}$	$\{2,6\}$
$\{0,2\}$	$\{1,4\}$	$\{3,4\}$
$\{0,3\}$	$\{1,5\}$	$\{3,5\}$
$\{0,4\}$	$\{1,6\}$	$\{3,6\}$
$\{0,5\}$	$\{2,3\}$	$\{4,5\}$
$\{0,6\}$	$\{2,4\}$	$\{4,6\}$
$\{1,2\}$	$\{2,5\}$	$\{5,6\}$

4.3.1. Automorfismo aditivo

Según la propiedad aditiva, dos vectores $v = \{v_0, v_1, ..., v_{t-1}\}$ y $w = \{w_0, w_1, ..., w_{t-1}\}$, $v, w \in \Lambda$, son isomórficos si y solo si hay una constante a tal que $s_i = (w_i + a)$ **mod** k, para

todo $0 \leq i \leq t-1$. Entonces, los elementos del grupo de automorfismos de las t-adas $v = \{v_0$, ..., $v_{t-1}\}$ obtenidos mediante la propiedad aditiva son el conjunto definido en la expresión 4.3.

$$\{w = \{w_0, ..., w_{t-1}\} | w_i = (v_i + a) \textbf{ mod } k, \forall\ 0 \leq i \leq t-1, \forall\ 0 \leq a \leq k-1\} \tag{4.3}$$

Al aplicar la propiedad aditiva al grupo de automorfismos se crea un conjunto $\Delta \subset \Lambda$ donde cada t-ada $v = \{v_0, ..., v_{t-1}\}$ inicia con el mismo valor, por ejemplo, $v_0 = 0$. Para ilustrar lo anterior, en la *tabla 4.12* se muestra el conjunto de los automorfismos de la t-ada $\{0, 1\}$ derivados de la propiedad aditiva; todas las t-adas isomórficas en esta tabla muestran la misma distribución en la combinación de símbolos pero con una rotación en los renglones.

Tabla 4.12: Ejemplo de automorfismo aditivo.

{0,1}	{1,2}	{2,3}	{3,4}	{4,5}	{5,6}	{6,0}
{0,0}	{0,0}	{0,1}	{1,0}	{0,1}	{1,1}	{1,0}
{0,0}	{0,1}	{1,0}	{0,1}	{1,1}	{1,0}	{0,0}
{0,1}	{1,0}	{0,1}	{1,1}	{1,0}	{0,0}	{0,0}
{1,0}	{0,1}	{1,1}	{1,0}	{0,0}	{0,0}	{0,1}
{0,1}	{1,1}	{1,0}	{0,0}	{0,0}	{0,1}	{1,0}
{1,1}	{1,0}	{0,0}	{0,0}	{0,1}	{1,0}	{0,1}
{1,0}	{0,0}	{0,0}	{0,1}	{1,0}	{0,1}	{1,1}

En conclusión, mediante automorfismos aditivos se reduce el espacio de búsqueda para verificar que un arreglo sea un CA, la cardinalidad del conjunto Δ es $\binom{k-1}{t-1}$. Sólo será necesario validar aquellas t-adas donde $v = 0$. En la *tabla 4.13* se muestran las t-adas necesarias para validar el $CCA(7; 7, 2, 2)$.

Tabla 4.13: Conjunto Δ para validar el $CCA(7; 7, 2, 2)$ usando automorfismos aditivos.

$\{v_0, v_1\}$
{0,1}
{0,2}
{0,3}
{0,4}
{0,5}
{0,6}

4.3.2. Automorfismo multiplicativo

Según la propiedad multiplicativa, dos elementos $v, w \in \Lambda$ son isomórficos si hay una constante c tal que $v_i = (w_i \cdot c)$ **mod** k y c sea un cuadrado perfecto. La *expresión 4.4* describe el grupo de automorfismos obtenido al aplicar esta propiedad. En la *tabla 4.14* se muestran los residuos cuadráticos para el caso $CCA(7; 7, 2, 2)$.

$$\{w = \{w_0, ..., w_{t-1}\} | w_i = (v_i \cdot c) \textbf{ mod } k, \forall\ 0 \leq i \leq t-1, c = j^2, \forall\ 1 \leq j \leq k-1\} \tag{4.4}$$

En la *tabla 4.15* se muestran las submatrices correspondientes a los automorfismos de la t-ada $\{0, 1\}$ derivados de la propiedad multiplicativa; note que las t-adas isomórficas $\{0, 2\}$ y $\{0, 4\}$ contienen la misma distribución de la combinación de diferentes símbolos pero con una rotación en los renglones.

Tabla 4.14: Residuos cuadráticos para el caso $CCA(7; 7, 2, 2)$.

b	b^2	a
0	0	0
1	1	1
2	4	4
3	9	2
4	16	2
5	25	4
6	36	1

Tabla 4.15: Ejemplo de automorfismo multiplicativo.

$\{0,1\}$	$\{0,2\}$	$\{0,4\}$
$\{0,0\}$	$\{0,0\}$	$\{0,0\}$
$\{0,0\}$	$\{0,1\}$	$\{0,1\}$
$\{0,1\}$	$\{0,0\}$	$\{0,1\}$
$\{1,0\}$	$\{1,1\}$	$\{1,0\}$
$\{0,1\}$	$\{0,1\}$	$\{0,0\}$
$\{1,1\}$	$\{1,0\}$	$\{1,0\}$
$\{1,0\}$	$\{1,0\}$	$\{1,1\}$

Al combinar las propiedades aditiva y multiplicativa se crea un conjunto $\mathcal{C} \subset \Delta$ suficiente para verificar si se trata de un CA. Cada elemento $v \in \mathcal{C}$ es una t-ada $v = \{v_0, v_1, ..., v_{t-1}\}$ que contiene $v_0 = 0$ y $v_1 = \{1, \alpha\}$. El valor α representa el mínimo de los residuos no cuadráticos, caracterizado por la *expresión 4.5*.

$$\alpha = \text{mín}\{a | \nexists_{1 \leq b \leq k} b^2 \equiv a(\mathbf{mod}\ k)\} \tag{4.5}$$

Mediante la combinación de las propiedades aditiva y multiplicativa se reduce el espacio de búsqueda para verificar que un arreglo sea un CA, la cardinalidad del conjunto $\mathcal{C} = \binom{k-2}{t-2} + \binom{k-\alpha-1}{t-2}$. La *tabla 4.16* muestra las únicas t-adas $v \in \mathcal{C}$ necesarias para verificar el $CCA(7; 7, 2, 2)$ como CA.

Tabla 4.16: Conjunto $\mathcal{C}$ para validar el $CCA(7; 7, 2, 2)$ usando automorfismos multiplicativos.

$\{v_0, v_1\}$
$\{0,1\}$
$\{0,3\}$

Una vez mostrada la metodología necesaria para verificar un CA, en las siguientes secciones se mostrará el algoritmo secuencial propuesto para hacer el proceso de verificación de un CA, así como sus implementaciones en paralelo y Grid.

4.4. Algoritmos propuestos

4.4.1. Algoritmo secuencial (AS)

Por lo general a las computadoras se las visualiza como máquinas secuenciales, en las cuales la CPU lee instrucciones de la memoria y las va ejecutando una por una; cuando se especifican algoritmos que siguen esa misma ideología se dice que se esta programando secuencialmente.

Este modelo de programación sigue siendo el mas difundido hasta la fecha y nos servirá como punto de partida para resolver el problema de validación de CA.

El Algoritmo Secuencial (AS) *A.0.1* es nuestra propuesta para llevar acabo el proceso de verificación descrito en la sección 4.1. Para implementar este algoritmo, es de vital importancia tomar en cuenta aspectos como: el lenguaje a utilizar, el compilador, la arquitectura donde se va a ejecutar y la forma de escribir las expresiones, de tal manera que se puedan obtener las mejores prestaciones.

Este algoritmo se puede dividir en cuatro etapas (inicialización de variables, obtención de las combinaciones de símbolos por *t-ada*, conteo de símbolos faltantes y el cálculo de la siguiente *t-ada* a verificar.

En la primera etapa se inicializa el vector J con la primer *t-ada* a validar (líneas $3-4$), después de las líneas $5-8$ se calcula la variable $iMax$ la cual controla el valor máximo para cada uno de los elementos del vector J, de tal manera que siempre se pueda calcular correctamente la siguiente *t-ada* a validar. De las líneas $9-10$ se inicializa el vector P a 0's.

La segunda etapa y principal del proceso esta representada de las lineas $14-28$ donde se van marcando las combinaciones de símbolos existentes para una *t-ada* en particular (se aplica el método de Ruffini).

La tercera etapa consiste en el conteo de las combinaciones faltantes, líneas $23-26$.

Por último se encuentra la etapa de calcular la siguiente *t-ada* a validar; este paso esta representado en la *función A.0.2*.

Niveles de optimización aplicados

Según Hoste [55], una de las maneras mas simples de mejorar el desempeño de una aplicación es utilizando correctamente las opciones de optimización del compilador ($-O0$, $-O1$, $-O2$, $-O3$).

También se puede mejorar el rendimiento de las aplicaciones modificando la forma en como se recorren las matrices [42]. En lenguaje C las matrices se almacenan en memoria *por filas* por lo cual una distribución de los datos del CA por columnas teóricamente debería ser más eficiente debido a que los datos a utilizar son cargados consecutivamente en memoria. El *algoritmo A.0.3* es una implementación por columnas del *algoritmo A.0.1*.

Otra manera de obtener mejores resultados, es escribiendo las expresiones de tal manera que el procesador pueda ejecutar instrucciones paralelamente [106], (ver el *algoritmo A.0.4*).

Tabla 4.17: Optimizaciones.

Implementación	$CA(7500;20,2^{20},10)$	$CA(256;128,2^{128},5)$
Por renglones	369.21	5332.35
Por renglones $-O3$	148.18	2161.74
Por columnas $-O3$	46.11	668.07
Por columnas $-O3$ modificando expresiones	42.88	650.28

Para ejemplificar los beneficios de la optimización, en la *tabla 4.17* se muestran los tiempos en segundos para el CA$(7500;20,2^{20},10)$ y el CA$(256;128,2^{128},5)$. En la primera fila se muestran los tiempos del Algoritmo A.0.1 compilado sin ninguna optimización por parte del compilador. En la segunda línea se muestran los tiempos de la ejecución del mismo algoritmo pero ahora

utilizando la opción $-O3$ del compilador *gcc*; como se puede notar se ha obtenido en ambos casos una aceleración $\approxeq 250\,\%$. En la tercer línea se muestra como es que al utilizar el enfoque por columnas del Algoritmo A.0.3 se obtiene una aceleración $\approxeq 800\,\%$ con respecto al primer algoritmo. Por último en la línea 4 se muestra como al modificar la manera de escribir las expresiones (Algoritmo A.0.4) se puede obtener aún mejores prestaciones ($> 800\,\%$).

Cuando los parámetros del CA empiezan a crecer poco a poco el algoritmo secuencial deja de ser una buena opción porque el tiempo que tarda en hacer los cálculos se va incrementando rápidamente, por lo cual, es necesario mas poder de cómputo. En la siguiente sección se presenta la implementación paralela del algoritmo para verificar CAs.

4.4.2. Algoritmo paralelo (AP)

Cuando llega el momento en que las aplicaciones que estamos ejecutando demandan mas poder de cómputo del que puede proporcionar una computadora tradicional, es momento de buscar nuevas alternativas y la computación paralela suele ser la solución a muchos de los problemas que demandan mas poder computacional.

Existen dos grandes clases de problemas que requieren técnicas de computación avanzada para gestionar el coste computacional de las mismas [4]:

- Las aplicaciones de alto rendimiento (High Performance Computing (HPC)), requieren la ejecución de una aplicación en el menor tiempo posible, mediante la paralelización de la misma y su ejecución en un multiprocesador o un cluster de PCs (simulaciones de procesos físicos, biomédicos, climatológicos, etc.).
- Las aplicaciones de alta productividad (High Throughput Computing (HTC)), requieren la ejecución de un número mayor de aplicaciones por unidad de tiempo. Para ello se utilizan múltiples recursos de manera coordinada para la ejecución concurrente de las diferentes aplicaciones.

Por otra parte, existen dos estilos principales de programación en paralelo, el paralelismo en los datos (data parallelism, DP [51]) y el paralelismo en el flujo de control del programa (control parallelism, CP [3]). El paralelismo en los datos se basa en realizar múltiples operaciones en paralelo sobre un gran conjunto de datos, mientras que el paralelismo en el control del programa se basa en tener simultáneamente varios flujos de control ejecutándose en paralelo.

Al dividir el trabajo en varios procesadores surge la necesidad de calcular la primer t-ada con la cual cada uno de ellos iniciará su trabajo, el cálculo de la primer t-ada no es trivial debido a que las t-adas no están ordenadas lexicográficamente. Una solución sería que cada procesador calculara su primer t-ada de la misma manera como se hace en el algoritmo secuencial (líneas $3-4$) y después llamar a la función *siguiente_tada* tantas veces como $\Lambda \times p$, donde p es identificador del procesador. Obviamente esta solución no es muy buena debido a la repetición excesiva de cálculos, porque el procesador con el mayor p hará los todos los cálculos que sus antecesores para encontrar su t-ada inicial.

Para encontrar la t-ada inicial se propone el *algoritmo A.0.5* el cual la obtiene a partir de un número entero $\Theta = p \cdot bloque$. Una llamada a este algoritmo remplaza las líneas $3 - 4$ del *algoritmo A.0.1*.

Ejemplo 4.1 *Para ilustrar el funcionamiento del algoritmo A.0.5, usaremos un* $CA(10; 7, 2, 4)$ *y 5 procesadores (size). Se utilizará una tabla de tamaño* $k \times t$*, donde los renglones representaran los valores de* k *y las columnas los valores de* t *(ver tabla 4.18); en cada casilla de la tabla se guardan las* $\binom{k}{t}$ *según corresponda. Los iteradores* iK *e* iT *son usados para indicar que casilla se va a seleccionar. El ciclo* $foreach$ *mostrado en la línea 4 se repetirá* t *veces, obteniendo en cada una de ellas el iésimo valor del polinomio que representa a la* t*-ada que se esta buscando. Toda búsqueda iniciará en la casilla* $k-1, t-1$*, el ciclo while mostrado en las lineas* $5-8$ *controla los movimientos por columnas y mientras el valor de la* t*-ada buscada* Θ *sea mayor al valor de la casilla* $(k-iK, t-iT)$ *el valor buscado será igual a* $\Theta - \binom{k-iK}{t-iT}$ *cuando este ciclo no se cumpla entonces se ha encontrado el iésimo valor de la* t*-ada el cual corresponderá al número de renglón donde se produjo la terminación del ciclo* $(iK-1)$*, a partir de este momento se continuara con el proceso de búsqueda en las siguientes columnas hasta completar los* t *valores.*

Tabla 4.18: Ejemplo $CA(10;\ 7,\ 2,\ 4)$, combinaciones de $\binom{k}{t}$.

k \ t	$t-1$	$t-2$	$t-3$	$t-4$	
$k-1$	20				0
$k-2$	10	10			1
$k-3$	4	6	4		2
$k-4$	1	3	3	1	3
$k-5$	0	1	2	1	4
$k-6$	0	0	1	1	5
$k-7$	0	0	0	1	6

En la tabla 4.19 se muestran las t*-adas iniciales correspondientes a cada uno de los 5 procesadores; en la tabla 4.20 se hace un listado de todas las* t*-adas y se indica a que procesador serán asignadas.*

Tabla 4.19: Ejemplo $CA(10; 7, 2, 4)$, t-adas iniciales de cada procesador.

p	Θ	$t-ada\ inicial$
0	0	{0,1,2,3}
1	7	{0,1,4,5}
2	14	{0,2,4,6}
3	21	{1,2,3,5}
4	28	{1,3,5,6}

En el *algoritmo A.0.5* se pueden optimizar los cálculos combinatorios, si al inicio se calculan las $\binom{k}{t}$ y se almacenan en $kint$, después el valor de $\binom{k-1}{t}$ estará definido por $(kint \cdot ((k-iK)-(t-iT)))/(k-iK)$; y el valor de $\binom{k-1}{t-1}$ estará definido $kint \cdot (t-iT)/(k-iK)$. El *algoritmo A.0.6* es la versión optimizada para el cálculo de la t-ada inicial.

El Algoritmo Paralelo (AP) *A.0.7* es nuestra propuesta para validar CA paralelamente, podríamos decir que este algoritmo es de la clase HPC y que utiliza un estilo de programación DP. A cada procesador se le asignan un conjunto de t-adas a validar, donde cada t-ada utiliza una combinación diferente de columnas del CA, de tal manera que todo el proceso se puede dividir en muchos trabajos independientes donde la comunicación entre procesos es casi nula.

En las primeras 4 líneas del algoritmo se hace el cálculo del bloque de *t-adas* que le corresponderán a cada procesador; si dicho reparto no puede ser exacto el nodo maestro se encargará de realizar el trabajo con menor número de *t-adas*. Las líneas de la $8-10$ serán ejecutadas por

Tabla 4.20: Ejemplo $CA(10;7,2,4)$, t-adas asignadas a cada procesador.

t-ada	p	Θ	J_0	J_1	J_2	J_3
0	0	0	0	1	2	3
1			0	1	2	4
2			0	1	2	5
3			0	1	2	6
4			0	1	3	4
5			0	1	3	5
6			0	1	3	6
7	1	7	0	1	4	5
8			0	1	4	6
9			0	1	5	6
10			0	2	3	4
11			0	2	3	5
12			0	2	3	6
13			0	2	4	5
14	2	14	0	2	4	6
15			0	2	5	6
16			0	3	4	5
17			0	3	4	6
18			0	3	5	6
19			0	4	5	6
20			1	2	3	4
21	3	21	1	2	3	5
22			1	2	3	6
23			1	2	4	5
24			1	2	4	6
25			1	2	5	6
26			1	3	4	5
27			1	3	4	6
28	4	28	1	3	5	6
29			1	4	5	6
30			2	3	4	5
31			2	3	4	6
32			2	3	5	6
33			2	4	5	6
34			3	4	5	6

los nodos esclavos, los cuales calcularan el número de *t-adas* faltantes y le enviaran los resultados al nodo maestro. Las líneas $12-16$ corresponden al trabajo del nodo maestro, el cual, primero hará el cálculo de las *t-adas* que le tocaron según su número de nodo y posteriormente se pone a recibir los resultados de los nodos esclavos. En este algoritmo cada nodo, en base a su número de identificador y al tamaño del bloque de trabajo, calcula cual será la *t-ada* desde donde iniciará su trabajo, a partir de este momento su trabajo es similar al *algoritmo A.0.1*.

El *algoritmo A.0.7* fue implementado en lenguaje *C* con MPI. Se optó por este modelo para poder sacar el máximo provecho a las memoria distribuida debido a que el paso de mensajes entre nodos es casi nulo, de tal manera que el algoritmo pueda escalar aumentando el número de procesadores.

Se utilizó un reparto por bloques para mantener la sencillez del código y porque las implementaciones de un reparto cíclico de *t-adas* y reparto cíclico de bloques de *t-adas* no aumentaban significativamente las prestaciones.

4.4.3. Algoritmo Grid (AG)

La idea principal que subyace a las tecnología Grid es ofrecer una interfaz homogénea y estándar para poder acceder a los recursos de distintas Organizaciones. Estos recursos pueden ser

máquinas de cálculo como supercomputadoras, clusters de PCs o sistemas de almacenamiento. Conceptualmente, es una solución muy sencilla, un "divide y vencerás", si no podemos resolver el problema en un único lugar, pues lo repartimos entre centros de cálculo distribuidos geográficamente. Esta unión de recursos de distintas organizaciones es lo que habitualmente se denomina Grid.

Los beneficios del Grid, gracias a la integración de recursos distribuidos, están teniendo repercusión en muchísimos campos, de entre los que cabe destacar: medicina (imágenes, diagnosis y tratamiento), ingeniería genética y biotecnología (estudios en genómica y proteómica), nanotecnología (diseño de nuevos materiales a escala molecular), ingeniería (diseño, simulación, análisis de fallos y acceso remoto a instrumentos de control), y recursos naturales y medio ambiente (previsión meteorológica, observación del planeta, modelos y predicción de sistemas complejos).

Este modelo de computación también se puede aplicar al proceso de validación de CA cuando la fuerza de los mismos es mayor de 5 y el número de columnas es superior a 1000. El Algoritmo Grid (AG) A.0.8 es nuestra propuesta para validar CA de gran tamaño utilizando computación Grid.

4.5. Adaptaciones para CCA

El algoritmo presentado para validar CA puede ser fácilmente extendido para usarse en la validación de los CCA. Para calcular el número de t-adas a verificar se debe remplazar Λ por $\mathcal{C}$ en los algoritmos A.0.1, A.0.7, A.0.8; en la *función A.0.2* sólo se deben tomar en cuenta aquellas t-adas donde $v_1 = 1,\ \alpha$.

4.6. Resumen del capítulo

En este capítulo se planteo un algoritmo para resolver el problema de validar CAs y su implementación secuencial, paralela y GRID. Es de esperar que cuando una aplicación no necesita mucho poder de cómputo se utilice la versión secuencial, cuando las necesidades de poder de cálculo aumentan considerablemente se puede utilizar la computación paralela y finalmente cuando los problemas crecieron gigantescamente llega el momento para la computación Grid.

En el problema de validar CA se pueden aplicar los tres tipos de tecnologías, debido a que hay instancias de CA que van desde necesitar unos segundos de un procesador hasta el punto de necesitar varios días de cómputo y miles de procesadores. En el siguiente capítulo se muestran los resultados del proceso de validación de CA utilizando los algoritmos descritos en este capítulo.

Capítulo 5

Experimentación y Resultados

En capítulo anterior se describieron los algoritmos necesarios para validar si una matriz es un CA. En este capítulo se describe la experimentación realizada y se muestran los resultados obtenidos.

Nosotros partimos del supuesto de que el proceso de validar CA es altamente paralelizable, en este apartado se muestra el grado de validez de dicha suposición.

5.1. Experimentación

En la *tabla 5.1* se muestra una lista de los CA que fueron utilizados en la experimentación. Estos CA fueron creados utilizando construcciones directas a través de campos finitos de Galois [10].

Los algoritmos planteados en el capítulo anterior fueron implementados en lenguaje C. Se hizo uso del cluster “Odin” y de la supercomputadora “Tirant” para lanzar los trabajos secuenciales y paralelos (10 veces cada trabajo); los trabajos en el Grid fueron ejecutados sobre la infraestructura Grid de EGEE (véase el anexo B para una breve descripción de la infraestructura).

El tiempo máximo por experimento fue de tres días debido a las políticas de la infraestructura donde se lanzaron los trabajos. Se usaron hasta 16 procesadores para experimentación paralela y hasta 512 en el Grid. Los tiempos utilizados para hacer las gráficas representan la mediana de los 10 trabajos lanzados por experimento.

5.2. Resultados

5.2.1. Coverings Arrays Generales

El número de operaciones necesarias para validar un CA tiene como cota superior $\binom{k}{t} \times N$ y como cota inferior $\binom{k}{t} \times \prod_{i=0}^{i=k-1} v_i$; por lo tanto cualquier incremento en los valores de alguna de esas variables repercute directamente en el tiempo de cómputo necesario para hacer la validación.

En las siguientes dos subsecciones mostramos los resultados de la experimentación realizada utilizando computación paralela y Grid respectivamente.

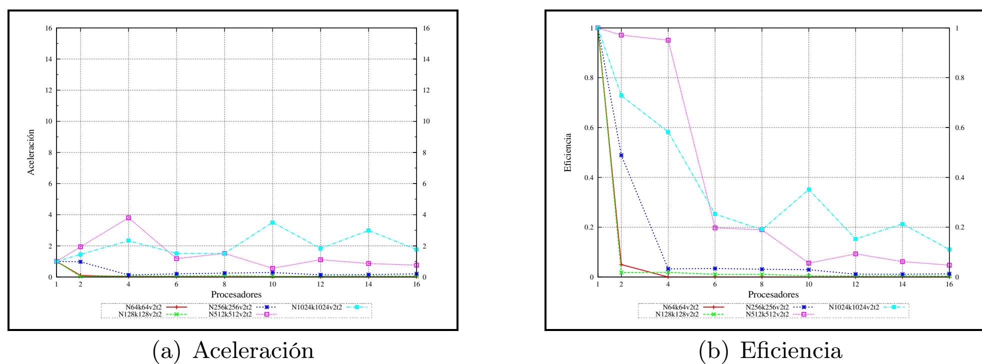
Tabla 5.1: CAs utilizados en la experimentación.

N	k	v	t
64	64	2	2
64	64	2	3
64	64	2	4
64	64	2	5
64	64	2	6
128	128	2	2
128	128	2	3
128	128	2	4
128	128	2	5
128	128	2	6
256	256	2	2
256	256	2	3
256	256	2	4
256	256	2	5
256	256	2	6
512	512	2	2
512	512	2	3
512	512	2	4
512	512	2	5
512	512	2	6
1024	1024	2	2
1024	1024	2	3
1024	1024	2	4
1024	1024	2	5
1024	1024	2	6

Usando computación paralela

En la sección 4.4.2 se mencionó que el proceso de validación de CA es del tipo HTC es decir, requiere la ejecución de un número mayor de procesos por unidad de tiempo. El *algoritmo A.0.7* que propusimos para la validación de CA es del tipo DP cuyo propósito es de realizar múltiples operaciones en paralelo sobre un gran conjunto de datos. Derivado de lo anterior se puede concluir entonces que el algoritmo paralelo no es una buena opción para los CA pequeños.

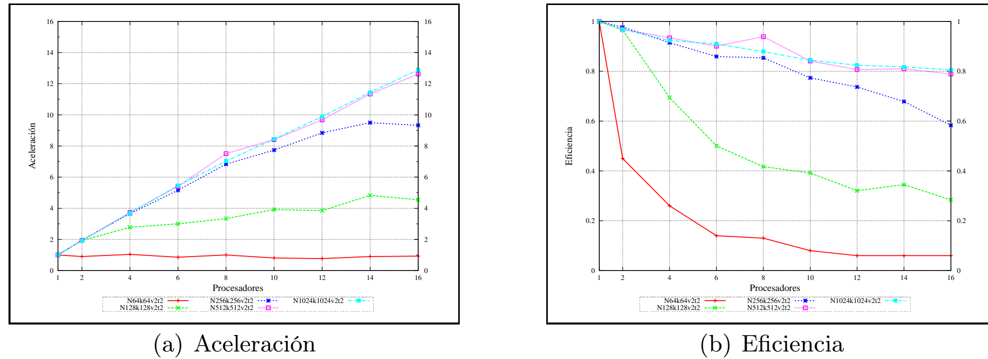
En la *figura 5.1* se muestra, en términos de aceleración y eficiencia, que para los casos donde la $t = 2$ y las $k \leq 1024$ no es rentable usar computación paralela, porque aunque las columnas vayan aumentando de tamaño nunca se logra alcanzar aceleraciones superiores a 4; en cuanto a la eficiencia se puede ver que al aumentar el número de procesadores inmediatamente cae la eficiencia; como dato adicional se debe tomar en cuenta que para el caso mas grande de fuerza 2 probado $(N = 1024,\ k = 1024,\ v = 2,\ t = 6)$ el tiempo de cómputo utilizando un procesador fue de 0,083501 *seg*.



(a) Aceleración (b) Eficiencia

Figura 5.1: Análisis para $t = 2$ y $k \leq 1024$.

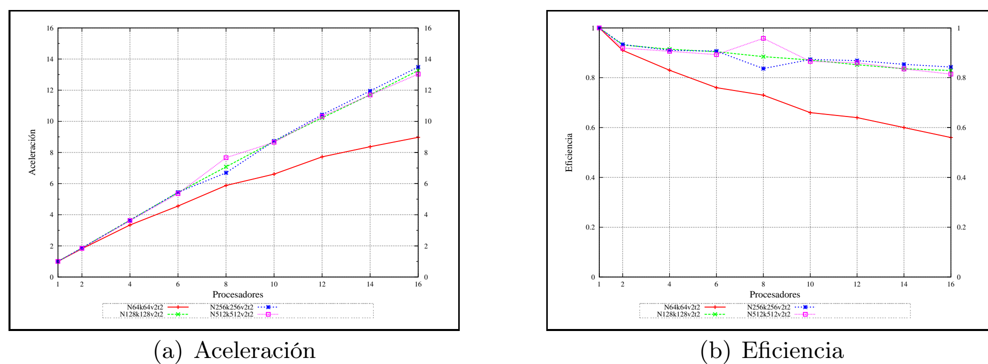
Cuando $t = 3$ se necesita de una $k > 256$ para que sea rentable el uso de la computación paralela (*véase la figura 5.2*), se necesitaron de 83,55 *seg* para terminar de procesar el mayor de los casos ($N = 1024,\ k = 1024,\ v = 2\ y\ t = 3$) con un solo procesador.



(a) Aceleración (b) Eficiencia

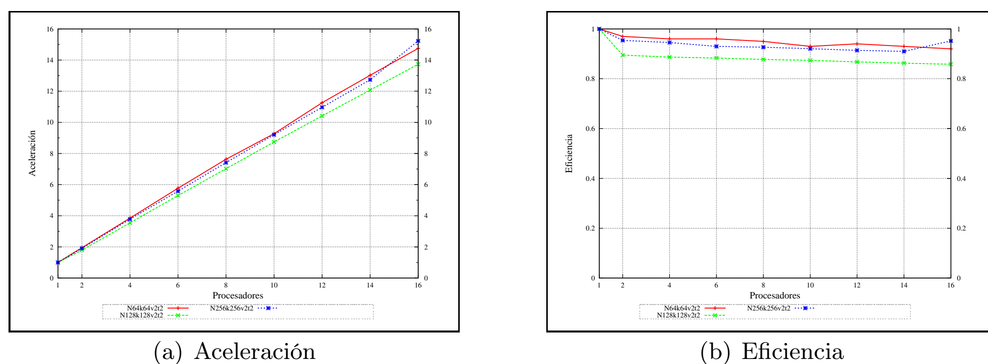
Figura 5.2: Análisis para $t = 3$ y $k \leq 1024$.

A partir de $t = 4$ y $k > 64$ es buena idea hacer uso del algoritmo paralelo, en las *figuras 5.3, 5.4 y 5.5* se muestran las gráficas de aceleración y eficiencia para estos casos.



(a) Aceleración (b) Eficiencia

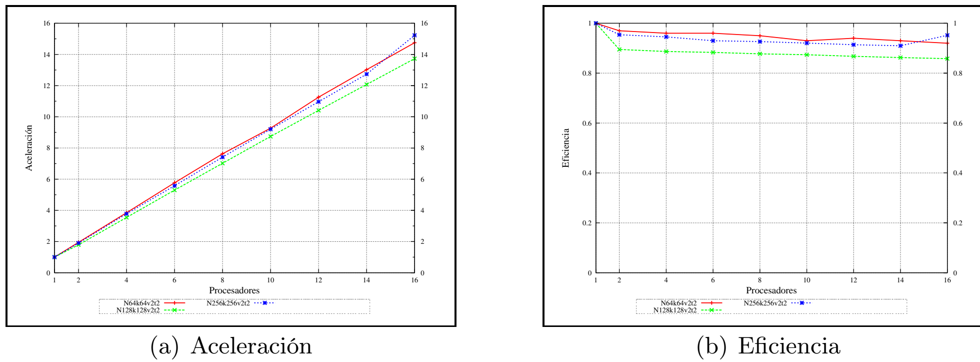
Figura 5.3: Análisis para $t = 4$ y $k \leq 1024$.



(a) Aceleración (b) Eficiencia

Figura 5.4: Análisis para $t = 4$ y $k \leq 1024$.

De estas figuras se puede concluir que es conveniente usar el algoritmo paralelo a partir de $t = 3$ siempre y cuando $k \geq 512$.



(a) Aceleración (b) Eficiencia

Figura 5.5: Análisis para $t = 4$ y $k \leq 1024$.

Usando computación Grid

En la *tabla 5.2* se muestra el rendimiento de los algoritmos AS y AG. En las primeras cuatro columnas se presentan las características de cada instancia. El número de procesadores por instancia fue de 512. En las columnas seis y siete se muestra el tiempo en horas que se necesito para correr cada instancia. El $CA(512; 512, 2, 6)$ y el $CA(1024; 1024, 2, 6)$ son solo estimaciones debido a que por tiempo no se pueden ejecutar en un plazo máximo de tres días usando 512 procesadores.

Tabla 5.2: Comparación de SA y GA para CA.

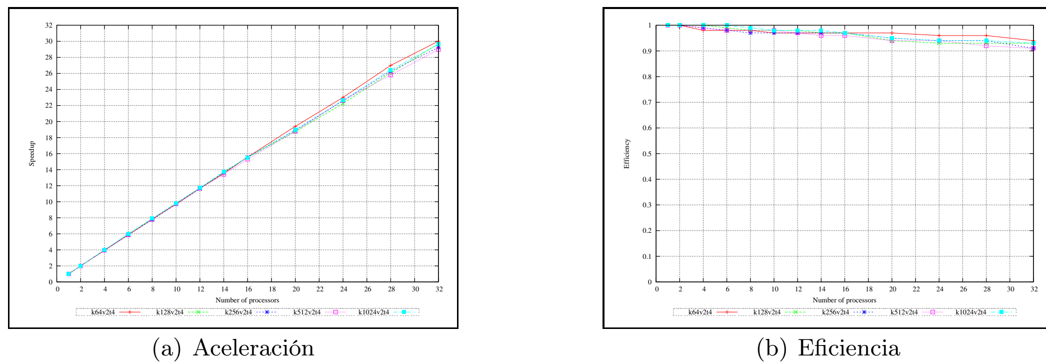
N	k	v	t	Procesadores	**Grid**	**Secuencial**
256	256	2	6	512	3.25	1669.6
512	512	2	5	512	1.1	559.4
512	512	2	6	512	232.4	119050.2
1024	1024	2	5	512	36.4	18631.7
1024	1024	2	6	512	15718.4	8045199.4

En base a los ejemplos mostrados, podemos validar que el proceso de verificación de CA se puede correr en un entorno Grid resolviendo problemas que en una computadora secuencial son inviables. Sin embargo, queda el hecho de que casos como: $CA(512; 512, 2, 6)$ y $CA(1024; 1024, 2, 6)$ no hayan podido ser verificados.

5.2.2. Coverings Arrays Cíclicos

Usando computación paralela

En las *figuras 5.6(a) y 5.6(b)* se ilustra el rendimiento del AP. Se tomaron como casos de uso los CCA de $t = 4$; en la *figura 5.6(a)* se puede observar que la aceleración es casi lineal, y en la *figura 5.6(b)* se muestra la eficiencia del AP, la cual se mantiene cercana al 100 % independientemente del CA utilizado; por lo cual, se puede concluir que el problema de verificar CCA es factible para ser abordado haciendo uso de algoritmos paralelos.



(a) Aceleración

(b) Eficiencia

Figura 5.6: Ejemplo del rendimiento del AG cuando $t = 4$.

Usando computación grid

En la *tabla 5.3* se muestra el rendimiento de los algoritmos AS y AG. En las primeras cuatro columnas se presentan las características de cada instancia. El número de procesadores por instancia fue de 64. En las columnas seis y siete se muestra el tiempo en minutos que se necesito para correr cada instancia.

Tabla 5.3: Comparación de SA y GA para CCA.

N	k	v	t	Procesadores	**Grid**	**Secuencial**
256	256	2	6	64	1.24	78.13
512	512	2	5	64	0.07	7.75
512	512	2	6	64	23.96	1589.58
1024	1024	2	5	64	0.6	38.85
1024	1024	2	6	64	407.96	26725.61

En base a lo anterior queda demostrado que el aplicar los automorfismos reduce considerablemente el tiempo de verificación de un CA, tanto es así, que los casos ($CA(512; 512, 2, 6)$ y $CA(1024; 1024, 2, 6)$) con este enfoque se pueden validar en cuestión de minutos a diferencia de los resultados mostrados en la *tabla 5.2*.

5.3. Resumen del capítulo

En el presente capítulo se mostraron los resultados del proceso de validación de CA utilizando computación secuencial, paralela y Grid. Quedó demostrado que el proceso de validación de CA es altamente paralelizable tanto para CA como para CCA. En el próximo capítulo se presentan las conclusiones generales y el trabajo que queda por hacer.

Capítulo 6

Conclusiones y trabajos futuros

El estado actual del avance del DOE demanda el poder construir diseños donde el esfuerzo computacional de validar que una matriz dada sea un DOE, es muy grande; la presente tesis presenta un enfoque original a través del uso Supercomputación y de Computación Grid para la verificación de DOE.

6.1. Conclusiones

En base a los resultados mostrados en el capítulo anterior se puede concluir que el proceso de validar CA es naturalmente paralelizable, de tal manera que puede beneficiarse no solo de la computación paralela a nivel de un cluster de computadoras; sino que además, puede descomponerse en una serie de trabajos totalmente independientes que pueden ejecutarse en un entorno Grid.

El tiempo necesario para validar un CA se puede acelerar casi linealmente, incrementando el número de procesadores en el trabajo.

6.2. Trabajos futuros

- Desarrollar nuevos algoritmos para construir CA haciendo un buen uso de la computación de altas prestaciones y la computación GRID.
- Diseñar nuevos algoritmos que puedan construir CA a partir de un vector inicial.
- Buscar nuevas técnicas que permitan construir CA con mejores *upper bounds* cada vez mas cercanos al valor óptimo.
- Realizar casos de estudio donde se aplique el uso de CA en el diseño de experimentos.
- Diseñar un sitio Web donde se ponga a disposición de la comunidad investigadora los CA construidos.

6.3. Publicaciones

6.3.1. Aceptadas

- **Himer Avila-George**, Jose Torres-Jimenez, Vicente Hernández and Nelson Rangel-Valdez, "Verification of General and Cyclic Covering Arrays Using Grid Computing", to appear in the *3rd International Conference on Data Management in Grid and P2P Systems (Globe 2010).* Conference Proceedings to be published by LNCS - Springer Verlag.
- Jose Torres-Jimenez, Nelson Rangel-Valdez, **Himer Avila-George** and Loreto Gonzalez-Hernandez, "Construction of Logarithm Tables for Galois Fields", to appear in the *International Journal of Mathematical Education in Science and Technology.*

6.3.2. En proceso

- **Himer Avila-George**, Jose Torres-Jimenez, Vicente Hernández and Nelson Rangel-Valdez, "Grid and Supercomputing on the verification of Covering Arrays", *The Journal of Supercomputing.*

Apéndices

Apéndice A

Algoritmos

Algoritmo A.0.1: TestCA, algoritmo secuencial para validar CA.

```
   Input: C_{N,k}, N, k, v, t, V_i, vtMax
   Output: Número de t-adas faltantes
 1 begin
 2     Miss ← 0, iMax ← 0;
 3     for i ← 0; i < t; i ← i + 1 do
 4         J_i ← i;
 5     for iMax ← t − 1, i ← 0; i < t; i ← i + 1 do
 6         if J_i == k − t + i then
 7             iMax ← i;
 8             break;
 9     for i ← 0; i < vtMax; i ← i + 1 do
10         P_i ← 0;
11     for nt ← 0; nt < Λ; nt ← nt + 1 do
12         for i ← 0, vt ← 1; i < t; i ← i + 1 do
13             vt ← vt × V_{J_i};
14         for Covered ← 0, r ← 0; r < N; r ← r + 1 do
15             aux ← C_{r,J_0} MOD V_{J_0};
16             for i ← 1; i < t; i ← i + 1 do
17                 aux ← aux × V_{J_i} + (C_{r,J_i} MOD V_{J_i});
18                 if P_aux == 0 then
19                     P[aux] ← 1;
20                     Covered ← Covered + 1;
21                     if Covered == vt then
22                         break;
23         if Covered < vt then
24             for i ← 0; i < vt; i ← i + 1 do
25                 if P_i == 0 then
26                     Miss ← Miss + 1;
27         for i ← 0; i < vt; i ← i + 1 do
28             P_i ← 0;
29         J ← Siguiente_t-ada(J);
30     return Miss;
31 end
```

Algoritmo A.0.2: Siguiente_t-ada.

```
    Input: t-ada actual
    Output: siguiente t-ada
 1  begin
 2      $J_{t-1} \leftarrow J_{t-1} + 1$;
 3      if $J_{t-1} == k$ then
 4          if $iMax == 0$ then
 5              return 0;
 6          $J_{iMax-1} \leftarrow J_{iMax-1} + 1$;
 7          for $i \leftarrow iMax;\ i < t;\ i \leftarrow i + 1$ do
 8              $J_i \leftarrow J_{i-1} + 1$;
 9          if $J_{iMax-1} == k - t + iMax - 1$ then
10              $iMax \leftarrow iMax - 1$;
11          else
12              $iMax \leftarrow t - 1$;
13      Return $J$
14  end
```

Algoritmo A.0.3: Algoritmo secuencial por columnas para validar CA.

```
    Input: $C_{N,k},\ N,\ k,\ v,\ t,\ V_i,\ vtMax$
    Output: Número de t-adas faltantes
 1  begin
 2      $Miss \leftarrow 0,\ iMax \leftarrow 0$;
 3      for $i \leftarrow 0;\ i < t;\ i \leftarrow i + 1$ do
 4          $J_i \leftarrow i$;
 5      for $iMax \leftarrow t - 1, i \leftarrow 0;\ i < t;\ i \leftarrow i + 1$ do
 6          if $J_i == k - t + i$ then
 7              $iMax \leftarrow i$;
 8              break;
 9      for $i \leftarrow 0;\ i < vtMax;\ i \leftarrow i + 1$ do
10          $P_i \leftarrow 0$;
11      for $nt \leftarrow 0;\ nt < \Lambda;\ nt \leftarrow nt + 1$ do
12          for $i \leftarrow 0, vt \leftarrow 1;\ i < t;\ i \leftarrow i + 1$ do
13              $vt \leftarrow vt \times V_{J_i}$;
14          for $Covered \leftarrow 0, r \leftarrow 0;\ r < N;\ r \leftarrow r + 1$ do
15              $aux \leftarrow C_{J_0,r}\ MOD\ V_{J_0}$;
16              for $i \leftarrow 1;\ i < t;\ i \leftarrow i + 1$ do
17                  $aux \leftarrow aux \times V_{J_i} + (C_{J_i,r}\ MOD\ V_{J_i})$;
18                  if $P_{aux} == 0$ then
19                      $P[aux] \leftarrow 1$;
20                      $Covered \leftarrow Covered + 1$;
21                      if $Covered == vt$ then
22                          break;
23          if $Covered < vt$ then
24              for $i = 0;\ i < vt;\ i = i + 1$ do
25                  if $P_i == 0$ then
26                      $Miss = Miss + 1$;
27          for $i = 0;\ i < vt;\ i = i + 1$ do
28              $P_i = 0$;
29          Siguiente_t-ada();
30      return $Miss$;
31  end
```

Algoritmo A.0.4: Algoritmo secuencial por columnas con modificación de expresiones.

```
   Input: C_{N,k}, N, k, v, t, V_i, vtMax
   Output: Número de t-adas faltantes
1  begin
2      Miss ← 0, iMax ← 0, K_{t-1} ← 1;
3      for i ← 0; i < t; i ← i + 1 do
4          J_i ← i;
5      for iMax ← t − 1, i ← 0; i < t; i ← i + 1 do
6          if J_i == k − t + i then
7              iMax ← i;
8              break;
9      for i ← 0; i < vtMax; i ← i + 1 do
10         P_i ← 0;
11     for nt ← 0; nt < Λ; nt ← nt + 1 do
12         for i ← 0, vt ← 1; i < t; i ← i + 1 do
13             K_i ← K_{i+1} × vV_{J_{i+1}};
14         for i ← 0, vt ← 1; i < t; i ← i + 1 do
15             vt ← vt × V_{J_i};
16         for Covered ← 0, r ← 0; r < N; r ← r + 1 do
17             aux ← 0;
18             for i ← 1; i < t; i ← i + 1 do
19                 aux ← aux + (C_{J_i,r}) ∗ K_i;
20                 if P_aux == 0 then
21                     P[aux] ← 1;
22                     Covered ← Covered + 1;
23                     if Covered == vt then
24                         break;
25         if Covered < vt then
26             for i = 0; i < vt; i = i + 1 do
27                 if P_i == 0 then
28                     Miss = Miss + 1;
29         for i = 0; i < vt; i = i + 1 do
30             P_i = 0;
31         Siguiente_t-ada();
32     return Miss;
33 end
```

Algoritmo A.0.5: Calcular la $t - ada$ inicial.

```
   Input: k, t, Θ
   Output: t-ada inicial
1  iK ← 1;
2  iT ← 1;
3  kint ← Δ;
4  foreach i ← 0; i < t; i ← i + 1 do
5      while Θ ≥ kint do
6          Θ ← Θ − kint;
7          kint ← binom(k−iK, t−iT);
8          iK ← iK + 1;
9      J_i ← iK − 1;
10     kint ← binom(k−iK, t−iT);
11     iK ← iK + 1;
12     iT ← iT + 1;
13 return J
```

Algoritmo A.0.6: Algoritmo optimizado para calcular la $t - ada$ inicial.

Input: $k,\ t,\ \Theta$
Output: t-ada inicial

1 $iK \leftarrow 1$;
2 $iT \leftarrow 1$;
3 $kint \leftarrow \Delta$;
4 **foreach** $i \leftarrow 0;\ i < t;\ i \leftarrow i + 1$ **do**
5   **while** $\Theta \geq kint$ **do**
6         $\Theta \leftarrow \Theta - kint$;
7         $kint \leftarrow (kint \cdot ((k - iK) - (t - iT)))/(k - iK)$;
8         $iK \leftarrow iK + 1$;
9     $J_i \leftarrow iK - 1$;
10     $kint \leftarrow (kint \cdot (t - iT))/(k - iK)$;
11     $iK \leftarrow iK + 1$;
12     $iT \leftarrow iT + 1$;
13 **return** J

Algoritmo A.0.7: TestCA Paralelo.

Input: $C_{N,k},\ N,\ k,\ v,\ t,\ V_i,\ vtMax$
Output: Número de t-adas faltantes

1 **begin**
2     $Miss \leftarrow 0;\ parcialMiss \leftarrow 0;\ iMax \leftarrow 0$;
3     $kint \leftarrow \Lambda$;
4     $bloque \leftarrow kint/size$;
5   **if** $kint\ MOD\ (size) \neq 0$ **then**
6         $bloque \leftarrow bloque + 1$;
7   **if** $Pr \neq size - 1$ **then**
8         $\Theta \leftarrow myRank \cdot bloque$;
9         $parcialMiss \leftarrow TestCA(\mathcal{C}_{N,k}, N, k, v, t, V_i, vtMax, \Theta)$;
10         $Enviar\ parcialMiss\ al\ Pr_0$
11   **else**
12         $parcialMiss \leftarrow TestCA(\mathcal{C}_{N,k}, N, k, v, t, V_i, vtMax, \Theta)$;
13   **for** $i \leftarrow 1;\ i < size;\ i \leftarrow i + 1$ **do**
14             $parcialMiss \leftarrow RecibirResultadosProcesadores()$;
15   **if** $parcialMiss > 0$ **then**
16                 $Miss \leftarrow Miss + parcialMiss$;
17   **return** $Miss$;
18 **end**

Algoritmo A.0.8: TestCA_Grid

Input: $\mathcal{C}_{N,k}, N, k, v, t, V_i, vtMax, p, size$
Output: Número de t-adas faltantes

1 $bloque \leftarrow \Lambda/size$;
2 **if** $kint\ MOD\ (size) \neq 0$ **then**
3     $bloque \leftarrow bloque + 1$;
4 $\Theta \leftarrow p \cdot bloque$;
5 $partialMiss \leftarrow t_wise(\mathcal{C}_{N,k}, N, k, v, t, V_i, vtMax, \Theta)$;
6 **return** $partialMiss$;

Apéndice B

Recursos computacionales utilizados

En este apéndice se describe el equipo de cómputo utilizado para hacer el lanzamiento de los trabajos descritos a lo largo de este documento.

B.1. Tirant

En las instalaciones de la Universidad de Valencia (UV) se encuentra la supercomputadora Tirant[1] la cual es uno de los siete nodos que forman parte de la Red Española de Supercomputación. Está recibe su nombre en honor al protagonista de la novela valenciana "Tirant lo blanc", escrita por Joanot Martorell en 1490.

Figura B.1: Tirant.

Tirant [84] esta conformado por una serie de computadoras interconectados entre sí por una red de baja latencia. Dentro de las supercomputadoras, Tirant se clasificaría como una de memoria distribuida.

Tirant está formado por 256 blades JS20 de IBM, cada uno de ellos con 2 procesadores PowerPC 970+ y 4 GB de memoria RAM (1 TByte en total). La comunicación entre ellos se realiza por medio de una red Myrinet, que tiene un ancho de banda de 2+2 GBytes por segundo.

[1] http://www.uv.es/siuv/cas/zcalculo/res/informa.wiki

La supercomputadora dispone además de 5 servidores pSeries 515 de IBM, que son los encargados de gestionar casi 10 TB de disco duro, exportado a todo el clúster mediante un sistema de archivos compartido, así como numerosos servicios auxiliares.

Este hardware dota al Tirant de una potencia de 4,5 Tflops, lo que lo llevó a estar situado entre los 500 supercomputadores más potentes del mundo según la lista TOP500 (puesto 413, antes de noviembre de 2006). Tirant funciona con el sistema operativo Suse Linux Enterprise Server 9; en la *figura B.1* se puede ver una imagen de dicha supercomputadora.

B.2. Odin

El clúster Odin[2][46] *(figura B.2)* consta de 55 nodos biprocesadores Intel(R) Xeon(TM) a 2.8 GHz, interconectados mediante una red SCI con topología de Toro 2D en malla de 11x5. Uno de los nodos actúa de nodo principal (front-end) mientras que los 54 restantes están disponibles para cálculo científico.

Cada nodo lleva instalado el Sistema Operativo Red Hat Enterprise Linux ES release 3 (Taroon Update 2). El gestor de trabajos instalado es ScaTorque (Scali Packaging of Torque). El clúster dispone de una versión optimizada de MPI, llamada ScaMPI, que permite aprovechar las prestaciones de la red SCI.

Figura B.2: Clúster Odin.

Este clúster pertenece y es administrado por el grupo GRyCAP y actualmente se encuentra ubicado en las instalaciones del Instituto de Aplicaciones de las Tecnologías de la Información y de las Comunicaciones Avanzadas (ITACA) de la Universidad Politécnica de Valencia (UPV).

[2] http://www.grycap.upv.es/usuario/odin.htm

B.3. Enabling Grids for E-SciencE

El proyecto Enabling Grids for E-SciencE Project (EGEE)[3] reúne a científicos e ingenieros de más de 260 instituciones repartidas por 55 países de todo el mundo proporcionándolos una infraestructura Grid para e-Ciencia disponible las 24 horas del día.

Originalmente el proyecto EGEE se centró en dos campos científicos, Altas Energías y Ciencias de la Vida (Medicina, Biología...), actualmente EGEE integra aplicaciones de otros muchos campos científicos, tratando desde geología hasta química computacional. Generalmente, la infraestructura Grid de EGEE es ideal para cualquier investigación científica, especialmente para aquellas cuyas necesidades de tiempo y recursos para ejecutar sus aplicaciones es tan grande que usando las infraestructuras tradicionales no serían prácticas.

La infraestructura Grid de EGEE consiste aproximadamente de 150,000 cores disponibles para los usuarios 24 horas al día y 7 días a la semana, además de unos 5 Petabytes (5 millones de Gigabytes) + cintas MSS de almacenamiento de datos, y el mantenimiento de una media de 100,000 trabajos simultáneos. Tal cantidad de recursos disponibles, cambia la forma en la que se llevan a cabo las investigaciones científicas. El uso final de estos recursos depende de las necesidades de los usuarios: gran capacidad de almacenamiento, la banda ancha que ofrece la infraestructura, o la disponibilidad total de la potencia de computación.

[3] http://project.eu-egee.org/

Referencias

[1] Aarts, E., and Korst, J. *Simulated annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing.* John Wiley & Sons, Inc., New York, NY, USA, 1989.

[2] Almeida, F., Giménez, D., Mantas, J. M., and Vidal, A. M. *Programación Paralela.* Paraninfo, 2008.

[3] Balasundaram, V. Translating control parallelism to data parallelism. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing* (Philadelphia, PA, USA, 1992), Society for Industrial and Applied Mathematics, pp. 555–563.

[4] Barbosa, J., Tavares, J., and Padilha, A. Self-tuned parallel processing system for heterogeneous clusters. In *International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2001* (2001).

[5] Berling, T., and Runeson, P. Efficient evaluation of multifactor dependent system performance using fractional factorial design. *IEEE Trans. Softw. Eng. 29*, 9 (2003), 769–781.

[6] Bryce, R. C., and Colbourn, C. J. The density algorithm for pairwise interaction testing: Research articles. *Softw. Test. Verif. Reliab. 17*, 3 (2007), 159–182.

[7] Bryce, R. C., Colbourn, C. J., and Cohen, M. B. A framework of greedy methods for constructing interaction test suites. In *ICSE '05: Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ACM, pp. 146–155.

[8] Burr, K., and Young, W. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proceedings of the Intl. Conf. on Software Testing Analysis and Review* (1998), West, pp. 503–513.

[9] Burroughs, K., Jain, A., and Erickson, R. L. Improved quality of protocol testing through techniques of experimental design. In *Supercomm/ICC'94 Proceedings of the IEEE International Conference on Communications* (May 1994), IEEE, pp. 745–752.

[10] Bush, K. A. Orthogonal arrays of index unity. *The Annals of Mathematical Statistics 23*, 3 (1952), 426–434.

[11] Cawse, J. N. *Experimental Design for Combinatorial and High Throughput Materials Development.* John Wiley & Sons, New York, 2003.

[12] Chateauneuf, M. A. *Covering Arrays.* PhD thesis, Michigan Technological University, 2000.

[13] Chateauneuf, M. A., Colbourn, C. J., and Kreher, D. L. Covering arrays of strength three. *Des. Codes Cryptography 16*, 3 (1999), 235–242.

[14] Chateauneuf, M. A., and Kreher, D. L. On the state of strength-three covering arrays. *J. Combin. Des. 10*, 4 (2002), 217–238.

[15] Cohen, D. M., Dalal, S. R., Kajla, A., and Patton, G. C. The automatic efficient test generator (AETG) system. In *ISSRE'94 Proceedings of Fifth International Symposium on Software Reliability Engineering* (Nov. 1994), IEEE Computer Society, pp. 303–309.

[16] Cohen, D. M., Dalal, S. R., Parelius, J., and Patton, G. C. The combinatorial design approach to automatic test generation. *IEEE Trans. Softw. Eng. 13*, 5 (1996), 83–88.

[17] Cohen, D. M., and Fredman, M. L. New techniques for designing qualitatively independent systems. *Journal of Combinatorial Designs 6*, 6 (1998), 411–416.

[18] Cohen, M. B. *Designing Test Suites for Software Interaction Testing.* PhD thesis, The University of Auckland, New Zealand, 2004.

[19] Cohen, M. B., Colbourn, C. J., and Ling, A. C. Constructing strength three covering arrays with augmented annealing. *Discrete Mathematics 308*, 13 (2008), 2709–2722. Combinatorial Designs: A tribute to Jennifer Seberry on her 60th Birthday.

[20] Cohen, M. B., Colbourn, C. J., and Ling, A. C. H. Augmenting simulated annealing to build interaction test suites. In *Proc. 14th Int. Symp. Software Reliability Engineering ISSRE 2003* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 394–405.

[21] Colbourn, C. J. A deterministic density algorithm for pairwise interaction coverage. In *Proceedings of the IASTED Intl. Conference on Software Engineering* (2004), pp. 242–252.

[22] Colbourn, C. J. Covering arrays from cyclotomy. *Des. Codes Cryptography 55*, 2-3 (2010), 201–219.

[23] Colbourn, C. J., Chen, Y., and Tsai, W.-T. Progressive ranking and composition of web services using covering arrays. In *WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 179–185.

[24] Conway, M. E. A multiprocessor system design. In *AFIPS '63 (Fall): Proceedings of the November 12-14, 1963, fall joint computer conference* (New York, NY, USA, 1963), ACM, pp. 139–146.

[25] Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Nitzberg, B., Snir, M., Traversat, B., and Wong, P. Overview of the MPI-IO Parallel I/O Interface. In *Third Annual Workshop on I/O in Parallel and Distributed Systems, IPPS '95* (Apr. 1995), pp. 1–15.

[26] Dagum, L., and Menon, R. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science & Engineering 5*, 1 (1998), 46–55.

[27] Dalal, S. R., and Mallows, C. L. Factor-covering designs for testing software. *Technometrics 40*, 3 (1998), 234–243.

[28] Dorigo, M., Maniezzo, V., and Colorni, A. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B 26*, 1 (1996), 29–41.

[29] Dueck, G. New optimization heuristics: The great deluge algorithm and the record-to-record travel. *Journal of Computational Physics 104*, 1 (1993), 86–92.

[30] Dunietz, I. S., Ehrlich, W. K., Szablak, B. D., Mallows, C. L., and Iannino, A. Applying design of experiments to software testing: experience report. In *ICSE '97: Proceedings of the 19th international conference on Software engineering* (New York, NY, USA, 1997), ACM, pp. 205–215.

[31] Eiben, A. E., and Smith, J. E. *Introduction to Evolutionary Computing*. Springer. Springer, 2003.

[32] Fisher, R. A. The arrangement of field experiments. *Journal of the Ministry of Agriculture of Great Britain 33* (1926), 503–513.

[33] Fisher, R. A. *Statistical methods for research workers*. Hafner Publishing Co., New York, 1973. Fourteenth edition—revised and enlarged.

[34] Flynn, M. J., and Rudd, K. W. Parallel architectures. *ACM Comput. Surv. 28*, 1 (1996), 67–70.

[35] Foster, I. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[36] Garey, M. R., and Johnson, D. S. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co Ltd, January 1979.

[37] Gargano, L., Körner, J., and Vaccaro, U. Capacities: for information theory to extremal set theory. *Journal of Combinatorial Theory Series A 68*, 2 (1994), 296–316.

[38] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. Mathematical sciences section PVM 3 user's guide and reference manual. Tech. rep., Oak Ridge National Laboratory, may 1994.

[39] Gill, S. Parallel programming. *The Computer Journal 1*, 1 (1958), 2–10.

[40] Globus Toolkit 4. Open source software toolkit used for building grids. http://www.globus.org/toolkit.

[41] Glover, F. Tabu search part 1. *ORSA Journal on Computing 1*, 3 (1989), 190–206.

[42] Golub, G. H., and Van Loan, C. F. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[43] Golumbic, M. C., and Hartman, I. B.-A. *Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications (Operations Research/Computer Science Interfaces Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[44] Grama, A., Gupta, A., Karypis, G., and Kumar, V. *Introduction to Parallel Computing*. Addison Wesley, 2003.

[45] GRINDAL, M., OFFUTT, J., AND ANDLER, S. F. Combination testing strategies: a survey. *Software Testing, Verification and Reliability 15*, 3 (2005), 167–199.

[46] GRUPO DE GRID Y COMPUTACIÓN DE ALTAS PRESTACIONES. Clúster Odin. http://www.grycap.upv.es/usuario/odin.htm.

[47] HARTMAN, A. Software and hardware testing using combinatorial covering suites. In *Graph Theory, Combinatorics and Algorithms* (2005), vol. 34 of *Operations Research/Computer Science Interfaces*, Springer US, pp. 237–266.

[48] HARTMAN, A., AND RASKIN, L. Problems and algorithms for covering arrays. *Discrete Mathematics 284*, 1-3 (2004), 149–156. Special Issue in Honour of Curt Lindner on His 65th Birthday.

[49] HEDAYAT, A. S., SLOANE, N. J. A., AND STUFKEN, J. *Orthogonal Arrays: Theory and Applications*. Springer-Verlag, New York, 1999.

[50] HELMBOLD, D. P., AND MCDOWELL, C. E. Modelling speedup (n) greater than n. *IEEE Transactions on Parallel and Distributed Systems 1*, 2 (1990), 250–256.

[51] HILLIS, W. D., AND STEELE, J. G. L. Data parallel algorithms. *Commun. ACM 29*, 12 (1986), 1170–1183.

[52] HNICH, B., PRESTWICH, S. D., SELENSKY, E., AND SMITH, B. M. Constraint models for the covering test problem. *Constraints 11*, 2-3 (2006), 199–219.

[53] HOLLAND, J. A universal computer capable of executing an arbitrary number of sub-programs simultaneously. In *IRE-AIEE-ACM '59 (Eastern): Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference* (New York, NY, USA, 1959), ACM, pp. 108–113.

[54] HOSKINS, D. S., COLBOURN, C. J., AND MONTGOMERY, D. C. Software performance testing using covering arrays: efficient screening designs with categorical factors. In *WOSP '05: Proceedings of the 5th international workshop on Software and performance* (New York, NY, USA, 2005), ACM, pp. 131–136.

[55] HOSTE, K., AND EECKHOUT, L. Cole: compiler optimization level exploration. In *CGO '08: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization* (New York, NY, USA, 2008), ACM, pp. 165–174.

[56] KATONA, G. O. H. Two applications (for search theory and truth functions) of sperner type theorems. *Periodica Mathematica Hungarica 3*, 1-2 (March 1973), 19–26.

[57] KLEINROCK, L. Memo on grid computing. http://www.lk.cs.ucla.edu/LK/Bib/REPORT/press.html, 1969.

[58] KLEITMAN, D. J., AND SPENCER, J. Families of k-independent sets. *Discrete Mathematics 6*, 3 (1973), 255–262.

[59] KUHN, D. R., AND REILLY, M. J. An investigation of the applicability of design of experiments to software testing. In *SEW '02: Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02)* (Washington, DC, USA, 2002), IEEE Computer Society, p. 91.

[60] Kuhn, D. R., Wallace, D. R., and Gallo, A. M. Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng. 30*, 6 (2004), 418–421.

[61] Kuhn, R., Lei, Y., and Kacker, R. Practical combinatorial testing: Beyond pairwise. *IT Professional 10*, 3 (2008), 19–23.

[62] Lei, Y., Kacker, R., Kuhn, D. R., Okun, V., and Lawrence, J. IPOG: A general strategy for T-Way software testing. In *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 549–556.

[63] Lei, Y., and Tai, K.-C. In-Parameter-Order: A test generation strategy for pairwise testing. In *HASE '98: The 3rd IEEE International Symposium on High-Assurance Systems Engineering* (Washington, DC, USA, 1998), IEEE Computer Society, pp. 254–261.

[64] Lobb, J. Hybrid strength two covering array constructions: Using cover starters to create covering arrays. Master's thesis, Carleton University, Canada, 2010.

[65] Mandl, R. Orthogonal Latin squares: an application of experiment design to compiler testing. *Commun. ACM 28*, 10 (1985), 1054–1058.

[66] May, D., and Shepherd, R. Occam and the transputer. In *Proc. of the IFIP WG 10.3 workshop on Concurrent languages in distributed systems: hardware supported implementation* (New York, NY, USA, 1985), Elsevier North-Holland, Inc., pp. 19–33.

[67] Meagher, K., and Stevens, B. Covering arrays on graphs. *J. Comb. Theory Ser. B 95*, 1 (2005), 134–151.

[68] Meagher, K., and Stevens, B. Group construction of covering arrays. *J. Combin. Des. 13*, 1 (2005), 70–77.

[69] Montgomery, D. C. *Design and Analysis of Experiments*, sixth ed. Wiley, December 2004.

[70] Morrison, E., and Morrison, P. *Charles Babbage and his Calculating Engines*. Dover, New York, 1961.

[71] Moura, L., Stardom, J., Stevens, B., and Williams, A. W. Covering arrays with mixed alphabet sizes. *J. Combin. Des. 11*, 6 (2003), 413–432.

[72] MPI-1. The Message Passing Interface standard.
http://www.mpi-forum.org/docs/mpi-10.ps, 1994.

[73] MPI-2. The Message Passing Interface standard.
http://www.mpi-forum.org/docs/mpi-20.ps, 1997.

[74] MPI Forum. http://www.mpi-forum.org/docs/docs.html.

[75] MPI Forum. The Message Passing Interface (MPI) standard.
http://www.mcs.anl.gov/research/projects/mpi/.

[76] MPICH. Widely portable implemetation of MPI.
http://www.mcs.anl.gov/research/projects/mpich2/.

[77] Nilo Mitra, E. SOAP Version 1.2 Part 0, W3C Recommendation 27 April 2007. http://www.w3.org/TR/soap12-part0/.

[78] Nurmela, K. J. Upper bounds for covering arrays by tabu search. *Discrete Appl. Math. 138*, 1-2 (2004), 143–152.

[79] OASIS. OASIS WSRF Technical Committee. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf.

[80] Open MPI. A High Performance Message Passing Library. http://www.open-mpi.org/.

[81] Phadke, M. S. *Quality Engineering Using Robust Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1995.

[82] Phadke, M. S. Planning efficient software tests. *CrossTalk - Journal of Defense Software Engineering 10* (1997), 11–15.

[83] PVM. Parallel Virtual Machine. http://www.beowulf.org/.

[84] Red Española de Supercomputación. Supercomputadora Tirant. http://www.uv.es/siuv/cas/zcalculo/res/informa.wiki.

[85] Rényi, A. *Foundations of Probability.* Wiley, 1971.

[86] Richard Kuhn, D., and Okum, V. Pseudo-exhaustive testing for software. In *SEW '06: Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 153–158.

[87] Ronneseth, A. H., and Colbourn, C. J. Merging covering arrays and compressing multiple sequence alignments. *Discrete Appl. Math. 157*, 9 (2009), 2177–2190.

[88] Segrelles, D. *Diseño y Desarrollo de una Arquitectura Software Genérica Orientada a Servicios para la Construcción de un Middleware Grid Orientado a la Gestión y Proceso Seguro de Información en formato DICOM sobre un Marco Ontológico.* PhD thesis, Universidad Politécnica de Valencia, España, 2008.

[89] Seroussi, G., and Bshouty, N. H. Vector sets for exhaustive testing of logic circuits. 513–522.

[90] Shasha, D. E., Kouranov, A. Y., Lejay, L. V., Chou, M. F., and Coruzzi, G. M. Using combinatorial design to study regulation by multiple input signals: A tool for parsimony in the post-genomics era. *Plant Physiology 127*, 4 (2001), 1590–1594.

[91] Sherwood, G. B. On the construction of orthogonal arrays and covering arrays using permutations groups. http://home.att.net/~gsherwood/cover.htm.

[92] Sherwood, G. B. Optimal and near-optimal mixed covering arrays by column expansion. *Discrete Mathematics 308*, 24 (2008), 6022–6035.

[93] Shiba, T., Tsuchiya, T., and Kikuno, T. Using artificial life techniques to generate test cases for combinatorial testing. In *COMPSAC '04: Proceedings of the 28th Annual International Computer Software and Applications Conference* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 72–77.

[94] Sloane, N. J. A. Covering arrays and intersecting codes. *Journal of Combinatorial Designs 1* (1993), 51–63.

[95] Sloane, N. J. A., and Stufken, J. A linear programming bound for orthogonal arrays with mixed levels. *Journal of Statistical Planning and Inference 56*, 2 (1996), 295–305.

[96] Stardom, J. Metaheuristics and search for covering and packing arrays. Master's thesis, Simon Fraser University, Canada, May 2001.

[97] Stevens, B. *Transversal Covers and Packings*. PhD thesis, University of Toronto, Canada, 1998.

[98] Stevens, B., and Meldensohn, E. New recursive methods for transversal covers. *Journal of Combinatorial Designs 7*, 3 (1999), 185–203.

[99] Stevens, B., Moura, L., and Mendelsohn, E. Lower bounds for transversal covers. *Des. Codes Cryptography 15*, 3 (1998), 279–299.

[100] Taguchi, G. *System of experimental design: engineering methods to optimize quality and minimize costs*. UNIPUBKraus International Publications, New York, 1987.

[101] Taguchi, G., and Konishi, S. *Taguchi Methods: Orthogonal Arrays and Linear Graphs; Tools for Quality Engineering*. American Supplier Institute, Dearborn, MI., 1987.

[102] Tai, K. C., and Lie, Y. A test generation strategy for pairwise testing. *IEEE Trans. Softw. Eng. 28*, 1 (2002), 109–111.

[103] Tang, D. T., and Chen, C. L. Iterative exhaustive pattern generation for logic testing. *IBM J. Res. Dev. 28*, 2 (1984), 212–219.

[104] Tassey, G. The economic impacts of inadequate infrastructure for software testing. Tech. rep., National Institute of Standards and Technology, 2002.

[105] Thakur, R., Gropp, W., and Lusk, E. On implementing mpi-io portably and with high performance. In *IOPADS '99: Proceedings of the sixth workshop on I/O in parallel and distributed systems* (New York, NY, USA, 1999), ACM, pp. 23–32.

[106] Tomasulo, R. M. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.* (1995), 13–21.

[107] Vadde, K., and Syrotiuk, V. R. Factor interaction on service delivery in mobile ad hoc networks. *IEEE Journal on Selected Areas in Communications 22*, 7 (Sept. 2004), 1335–1346.

[108] Vaughan, P. L., Skjellum, A., Reese, D. S., and Cheng, F.-C. Migrating from pvm to mpi.i. the unify system. In *FRONTIERS '95: Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95)* (Washington, DC, USA, 1995), IEEE Computer Society, p. 488.

[109] Williams, A. W. Determination of test configurations for pair-wise interaction coverage. In *TestCom '00: Proceedings of the IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems* (Deventer, The Netherlands, The Netherlands, 2000), Kluwer, B.V., pp. 59–74.

[110] Williams, A. W., and Probert, R. L. A practical strategy for testing pair-wise coverage of network interfaces. In *ISSRE '96: Proceedings of the The Seventh International Symposium on Software Reliability Engineering* (Washington, DC, USA, 1996), IEEE Computer Society, p. 246.

[111] Williams, A. W., and Probert, R. L. A measure for component interaction test coverage. In *AICCSA '01: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications* (Washington, DC, USA, 2001), IEEE Computer Society, p. 304.

[112] Yan, J., and Zhang, J. Backtracking algorithms and search heuristics to generate test suites for combinatorial testing. In *COMPSAC '06 Proceedings of the 30th Annual International Computer Software and Applications Conference* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 385–394.

[113] Yan, J., and Zhang, J. A backtracking search tool for constructing combinatorial test suites. *Journal of Systems and Software 81*, 10 (2008), 1681–1693. Selected papers from the 30th Annual International Computer Software and Applications Conference (COMPSAC), Chicago, September 7-21, 2006.

[114] Yilmaz, C. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Trans. Softw. Eng. 32*, 1 (2006), 20–34. Member-Cohen,, Myra B. and Senior Member-Porter,, Adam A.

[115] Yu-Wen, T., and Aldiwan, W. S. Automating test case generation for the new generation mission software system. In *Proceedings of the IEEE Aerospace Conference* (2000), IEEE Computer Society, pp. 431–437.