FACULTAD DE CIENCIAS QUÍMICAS E INGENIERÍA
CENTRO DE INVESTIGACIÓN EN INGENIERÍA
Y CIENCIAS APLICADAS

Algoritmo Genético Cooperativo Paralelizado en Ambiente Grid para el Problema de Ruteo Vehicular con Ventanas de Tiempo

TESIS PROFESIONAL
PARA OBTENER EL GRADO DE:

DOCTOR EN INGENIERÍA Y CIENCIAS APLICADAS
OPCIÓN TERMINAL EN TECNOLOGÍA ELÉCTRICA

P R E S E N T A:

M.I.C.A. ALINA MARTÍNEZ OROPEZA

ASESOR: DR. MARCO ANTONIO CRUZ CHÁVEZ

CUERNAVACA, MOR. MARZO 2015

Resumen

En este trabajo de investigación se propone un algoritmo genético cooperativo que permita el tratamiento de instancias de diferentes tamaños del problema de ruteo vehicular con ventanas de tiempo, el cual por su naturaleza es considerado como un problema intratable. El algoritmo propuesto, denominado Algoritmo Genético Cooperativo consiste en la aplicación de una hibridación que incorpora la estructura de un algoritmo genético con un operador de mutación basado en un método de cooperación conocido como algoritmo colonia de hormigas. El Algoritmo Genético Cooperativo fue desarrollado en tres versiones, secuencial, distribuida y paralelo-distribuida, donde la versión distribuida incorpora un modelo de islas con migración implícita que reduce la latencia mediante el uso de paso de mensajes con MPI utilizando comunicación colectiva. Por su parte, la versión paralelo-distribuida implica el uso de programación híbrida MPI-CUDA, donde de acuerdo a los tiempos de procesamiento, los operadores de selección y cruzamiento son aplicados de forma distribuida. Por su parte, debido al esfuerzo computacional que conlleva el operador de mutación cooperativa, este fue paralelizado con CUDA para GPUs.

Para estos algoritmos se llevó a cabo un análisis de sensibilidad con la finalidad de encontrar los valores adecuados de los parámetros de control que garanticen el mejor comportamiento del algoritmo propuesto. Las pruebas experimentales se realizaron sobre la infraestructura de la Grid Morelos, para lo cual se utilizaron los benchmarks de Solomon de 100 clientes y los de Gehring y Homberger de 1000 clientes. Los resultados experimentales demostraron la eficacia del algoritmo propuesto al alcanzar la mayoría de las mejores cotas conocidas y obtener tres mejoras a las cotas de los benchmarks de Solomon y dos a los de Gehring y Homberger, por lo que el algoritmo propuesto cumple con las expectativas planteadas en este trabajo de investigación.

Abstract

In this research, a cooperative genetic algorithm for the treatment of instances of different sizes of the vehicle routing problem with time windows, which by its nature is considered intractable is proposed. The proposed algorithm known as Cooperative Genetic Algorithm is comprised of a hybridization of the structure of a genetic algorithm with a mutation operator based on a cooperative method known as ant colony algorithm. The cooperative genetic algorithm was developed in three versions: sequential, distributed and parallel-distributed, where distributed version implements an island model with implicit migration that reduces latency using message passing with MPI based on collective communication. Meanwhile, the parallel-distributed version applies hybrid programming MPI-CUDA. According to processing times, selection and crossover operators are implemented in a distributed way. Meanwhile, the cooperative mutation operator that involves greater computational effort was parallelized with CUDA for GPUs.

For these algorithms, a distributed sensitivity analysis is performed in order to find the right values of the control parameters that guarantee the best possible behavior of the proposed algorithm. Experimental tests were carried out on the infrastructure Grid Morelos using the Solomon of 100 customers and Gehring and Homberger of 1000 customers benchmarking. Experimental results showed the efficacy of the proposed algorithm reaching most of the best-known solutions reported in the literature and obtaining three improvements to the Solomon benchmarking and two for the Gehring and Homberger's. Therefore, the proposed algorithm meets the considered expectations raised for this research.

Agradecimientos

A CONACYT por brindarme el apoyo económico sin el cuál no hubiera sido posible la realización de este trabajo de investigación

A FUNDACIÓN TELMEX por proporcionerme apoyo complementario para material didáctico.

Al Centro de Investigación en Ingeniería y Ciencias Aplicadas por ser mi segunda casa durante estos años de arduo trabajo.

Muy especialmente al Dr. Marco Antonio Cruz Chávez, director de esta tesis, por alentarme a lograr nuevos retos, además por su orientación y apoyo para la culminación de este trabajo.

A los integrantes de mi comité tutoral y revisores de tesis: Dra. Margarita Tecpoyotl Torres, Dr. Martín G. Martínez Rangel, Dr. Martín Heriberto Cruz Rosales, Dr. Federico Alonso Pecina, Dr. David Juárez Romero y al Dr. Alvaro Zamudio Lara por sus siempre acertados consejos y comentarios para la culminar este trabajo de investigación, así como para mi crecimiento personal e intelectual.

Dedicatorias

A Dios en primer lugar, por las oportunidades y ayuda que me ha brindado en el transcurso de mi vida.

A mis padres María Oropeza Díaz y Abel Martínez Pliego y a mi hermano Rogelio por su apoyo moral, comprensión y consejos, así como por siempre impulsarme a conseguir mis metas.

A J. Gerardo Vera Dimas, por su ayuda y apoyo incondicional cuando más lo he necesitado, así como por su comprensión en tiempos difíciles y por los bellos momentos que hemos pasado juntos.

A mi asesor, el Dr. Marco Antonio Cruz, por su dirección, apoyo y sus acertados consejos para lograr la realización de este trabajo.

A mi familia que siempre me ha dado palabras de aliento para seguir adelante con las metas que me propongo.

A mis grandes amigos Pedrito, Erika, Mireya, Juanita, Mara y todos los integrantes del grupo de trabajo con los que he compartido muy gratos momentos y que me han apoyado y ayudado siempre que los he necesitado.

Nomenclatura

ACH	Siglas para Algoritmo Colonia de Hormigas
AG	Siglas para Algoritmo Genético
AGC-VRPTW	Algoritmo secuencial propuesto. Sus siglas significan Algoritmo Genético Cooperativo - Problema de Ruteo Vehicular con Ventanas de Tiempo
AGCD-VRPTW	Algoritmo Distribuido propuesto (por sus siglas Algoritmo Genetico Cooperativo Distribuido para el Problema de Ruteo Vehicular con Ventanas de Tiempo (por sus siglas en inglés Vehicle Routing Problem with Time Windows)), el cual se basa en la versión secuencial AGC-VRPTW.
AGCP-VRPTW	Algoritmo Paralelo-Distribuido propuesto (por sus siglas Algoritmo Genetico Cooperativo Paralelo para el Problema de Ruteo Vehicular con Ventanas de tiempo) basado en la versión distribuida AGCD-VRPTW.
Benchmark	Instancia de prueba de un problema que ha sido utilizada frecuentemente como punto de referencia en diversas investigaciones para probar el desempeño de los algoritmos desarrollados
Clúster	Conjunto de máquinas conectadas mediante una red de alta velocidad que trabajan de forma unificada
CUDA	Siglas en Inglés para Arquitectura de Cómputo de Datos Unificados (*Compute Unified Data Architecture*)
Fitness	Aptitud de un individuo, considerado en este trabajo como el costo de una solución.
GPU	Siglas en inglés para Unidad Gráfica de Proceso (*Graphic Processing Unit*)

Grid Conjunto de clústeres geográficamente dispersos, conectados mediante una red de alta velocidad.

MPI Sigla en inglés para Interfaz de Paso de Mensajes (*Message Passing Interface*).

OpenMPI Versión libre de la distribución de MPI (por sus siglas en inglés *Open Message Passing Interface*).

Speedup Ganancia en aceleración que tiene un algoritmo secuencial al ser paralelizado sobre un conjunto de núcleos de procesamiento.

VRPTW Problema de Ruteo Vehicular con Ventanas de Tiempo (por sus siglas en inglés *Vehicle Routing Problem with Time Windows*).

Problema de Ruteo Vehicular con Ventanas de Tiempo (VRPTW)

A	Aristas que conectan dos clientes en el grafo (Distancia entre dos clientes)
a_i	Apertura de la ventana de tiempo del cliente *i*
b_i	Cierre de la ventana de tiempo del cliente *i*
C	Capacidad máxima del vehículo
c_{ij}	Costo de ir de un cliente *i* a un clente *j*
d_i	Demanda del cliente *i*
E	Apertura de la ventana de tiempo del depósito
K	Tamaño total de la flota de vehículos
L	Cierre de la ventana de tiempo del depósito
m	Ruta perteneciente a una solución
N	Total de clientes a calendarizar (tamaño de la instancia)
S_i	Tiempo de servicio requerido para un cliente *i*
V	Cada uno de los vértices del grafo, correspondiente a un cliente
W_{ik}	Tiempo de inicio del servicio del vehículo *k* en el cliente *i*

Operador de Mutación Cooperativa (Algoritmo Colonia de Hormigas)

α	Importancia de los montos de feromona

$\boldsymbol{\beta}$	Importancia de la distancia heurística
Colonia	Conjunto de hormigas que intervienen en el proceso de optimización
$\boldsymbol{Colonia_h}$	Hormiga actual perteneciente a la colonia. Cada hormiga represenra una solución en el ACH.
$\boldsymbol{d_{ij}}$	Distancia heurística correspondiente al arco (*i, j*)
$\boldsymbol{f_h}$	Costo de la solución obtenida por la hormiga *h*
$\boldsymbol{\gamma}$	Máximo de iteraciones aplicadas en la mutación cooperativa
$\boldsymbol{P_{ij}}$	Probabilidad de transición de un cliente *i* a un cliente *j*
$\boldsymbol{q_0}$	Coeficiente de equilibrio entre exploración y explotación
$\boldsymbol{\rho}$	Coeficiente de evaporación de la feromona
$\boldsymbol{\tau_{ij}}$	Monto de Feromona en el arco (*i, j*)
$\boldsymbol{\tau_0}$	Valor inicial de los montos de feromona
$\Delta\boldsymbol{\tau_{ij}}^{\boldsymbol{h}}$	Incremento de feromona en el arco (*i, j*) realizado por la hormiga *h*

<u>Algoritmo Genético</u>

Cromosoma	Una ruta factible dentro del individuo. Un individuo está compuesto de varios cromosomas.
Gen	Cada cliente perteneciente a un cromosoma
Individuo	Una solución factible al VRPTW en el AG.
Individuo Madre e Individuo Padre	Son los individuos elegidos mediante el operador de selección por torneo aleatorio para combinar sus genes de acuerdo al operador de cruzamiento y dar lugar a la generación de dos nuevos individuos denominados hijos.
Población	Conjunto de individuos que interactúan durante el proceso de evolución

<u>Algoritmo Distribuido AGCD-VRPTW</u>

Isla	Proceso independiente que aplica los operadores genéticos a su

propia subpoblación.

Migración Intercambio de individuos entre islas para mantener la diversidad en las subpoblaciones.

Proceso Secuencia de instrucciones a ejecutar por cada una de las islas.

Subpoblación Conjunto de Individuos asignados a una isla. Una subpoblación es el resultado de dividir la Población total entre el total de procesos a ejecutar

<u>Modelo Paralelo-Distribuido</u>

Bloque Conjunto de hilos que se ejecutan en un mismo SM, compartiendo los recursos computacionales.

Device Nombre con el que se identifica a la GPU en CUDA

Divergencia Hilos de un mismo bloque realizan tareas distintas

grid Conjunto de bloques que se ejecutan en el device

Hilo Conjunto de instrucciones que pueden ser ejecutadas de forma concurrente.

Host Nombre con el que se identifica a la CPU en CUDA

Kernel Encargado de ejecutar la grid en el device.

multithreading Caracteristica que indica que todos los cores de un SM utilizan el modelo SIMT (*Single Instruction Multiple Threads*)

SM Siglas en inglés para *Streaming multiprocessor*. En el caso de la tarjeta C2070, cada SM se compone de 32 SPs.

SP Siglas en inglés para *Streaming processor*, también conocido como *cuda-core.*

warp Conjunto de 32 hilos que pueden ejecutarse concurrentemente en un SM.

Contenido

RESUMEN
ABSTRACT
INDICE DE FIGURAS.............................. XII
INDICE DE TABLAS.............................. XVII

Capítulo 1. Introducción **1**

1.1 Teoría de la Complejidad de los Algoritmos 3
1.2 Teoría de la Complejidad de los Problemas 6
1.3 Motivación 8
1.4 Problema de Ruteo Vehicular. Clasificación y Notación 10
1.5 Objetivos de la Investigación 17
1.5.1 Objetivo General 17
1.5.2 Objetivos Específicos 17
1.6 Alcance de la Investigación 18
1.7 Contribución de la Tesis 19
1.8 Organización de la Tesis 19

Capítulo 2. Problema de Ruteo Vehicular con Ventanas de Tiempo **21**

2.1 Descripción del Problema de Ruteo Vehicular con Ventanas de Tiempo (VRPTW) 21
2.2 Modelo Matemático del VRPTW 23
2.3 Modelo de Grafos disyuntivos 27
2.4 Descripción de una instancia del Problema 29
2.5 Representación Simbólica y Diagrama de Gantt 32

Capítulo 3. Estado del Arte del VRPTW **36**

3.1 Métodos Secuenciales 37
3.2 Métodos Paralelos en Supercomputadoras 40
3.3 Métodos Distribuidos en Cluster 43
3.4 Métodos Distribuidos en Grid 45
3.5 Métodos Paralelos con GPUs 46

Capítulo 4. Metodología de Solución **50**

4.1 Algoritmo Secuencial AGC-VRPTW 50
4.1.1 Generación de la Población Inicial 59
4.1.2 Algoritmo Genético 63
4.1.2.1. Operador de Selección 64
4.1.2.2 Operador de Cruzamiento 66
4.1.2.3 Algoritmo Colonia de Hormigas 68
4.1.2.3.1 Operador de Mutación Cooperativa 69
4.1.3 Evaluación de la Distancia Hamming 74
4.2 Modelo de Islas Distribuido con Comunicación Colectiva AGC**D**-VRPTW 77
4.2.1 Construcción de la Población Inicial 78
4.2.2 Modelo de Islas con Migración Implícita 81
4.2.2.1 Comunicación de Procesos 84
4.2.2.2 Distribución de Procesos 87
4.2.2.3 Sincronización de Procesos 89
4.2.2.4 Cooperación Implícita de Procesos 91
4.3 Algoritmo Genético Cooperativo Paralelo-Distribuido AGC**P**-VRPTW 91
4.3.1 Conceptos Básicos de Programación en CUDA 93
4.3.2 Programación Híbrida MPI-CUDA 96
4.3.3 Tarjeta Nvidia Tesla C2070 97
4.3.4 Operadores Genéticos Distribuidos 102
4.3.5 Mutación Cooperativa Paralela con GPUs 105
4.3.5.1 Modelo de Memoria 108
4.3.5.2 Kernel "*Mutación*" 114
4.3.5.3 Kernel "*Feromona_Local*" 124
4.3.5.4 Kernel "*Actualización_Global*" de la Feromona 127
4.3.6 Uso de Multiples GPUs 131
4.3.7 Ejecución MPI-CUDA 134

Capítulo 5. Análisis de Sensibilidad **139**

5.1 Algoritmo Secuencial AGC-VRPTW 140
5.2 Algoritmo Genético-Cooperativo Distribuido AGC**D**-VRPTW 150

Capítulo 6. Pruebas Experimentales y Análisis de Resultados **154**

6.1 Infraestructura Grid Morelos 154
6.2 Instancias de Prueba 158
6.3 Algoritmo Secuencial AGC-VRPTW 159
6.3.1 Convergencia del Algoritmo 159
6.3.2 Análisis de Eficacia 163
6.3.3 Análisis de Eficiencia 169
6.3.4 Calculo de la Complejidad del Algoritmo 173
6.4 Algoritmo Distribuido AGC**D**-VRPTW 174
6.4.1 Convergencia del Algoritmo 175
6.4.2 Análisis de Eficacia 177
6.4.3 Análisis de Eficiencia 186
6.4.3.1 Análisis de la Distribución de Procesos 187
6.4.3.2 Análisis de Aceleración (Speedup) 194
6.5 Algoritmo Paralelo-Distribuido AGC**P**-VRPTW 200
6.5.1 Análisis de Eficacia 200
6.5.2 Análisis de Eficiencia 207
6.5.2.1 Sobrecarga de núcleos de la Grid 210
6.5.2.2 Análisis de Ocupación de la GPU 212

Capítulo 7. Conclusiones y Trabajos Futuros **219**

7.1 Conclusiones 219
7.2 Trabajos Futuros 223

REFERENCIAS 224

APÉNDICES

A. Envío de un Tipo de Dato Estructura (*struct*) con Comunicación Colectiva en MPI 239

B. Código Fuente del Algoritmo AGC-VRPTW 242

C. Código Fuente para Cálculo de la Distancia Hamming 259

D. Características de la tarjeta NVidia Tesla C2070 263

Índice de Figuras

1.1	Representación del crecimiento de funciones utilizadas comúnmente en las estimaciones con notación *O*.	5
1.2	Clasificación de Problemas acorde a la Teoría de la Complejidad	6
1.3	Clasificación básica del VRP y sus relaciones.	11
2.1	Modelo de Programación Lineal Entera Binaria para el VRPTW	24
2.2	Grafo Disyuntivo para una instancia de 8 clientes.	27
2.3	Representación de una solución mediante un dígrafo para la instancia presentada en la figura 2.2.	28
2.4	Estructura general de la información presentada en los benchmarks utilizados para evaluar los algoritmos diseñados para tratar el VRPTW.	30
2.5	Estructura general de la información presentada en uno de los benchmarks de Solomon de 100 clientes, tipo 1 con distribución aleatoria.	31
2.6	Tipo de dato estructura (struct) generado para el almacenamiento de soluciones factibles al VRPTW.	32
2.7	Diagrama de Gantt de una solución factible	34
4.1	Pseudocódigo del Algoritmo Genético Canónico	52
4.2	a) Las hormigas siguen el camino del nido hasta la fuente de alimento.	54
4.3	Ejemplo de lista tabú de la hormiga *h*	55
4.4	Pseudocódigo del Algoritmo Colonia de Hormigas	57
4.5	Estructura General del AGC-VRPTW propuesto en esta tesis	58

4.6	Algoritmo de Agrupamiento aplicado al VRPTW	60
4.7	Algoritmo de inserción híbrido aplicado a la generación de soluciones factibles al VRPTW.	62
4.8	Visualización gráfica de la representación simbólica del tipo de dato “struct”.	64
4.9	Selección de individuos mediante el operador de torneo aleatorio para el AGC-VRPTW	65
4.10	Ejemplo de la elección aleatoria de los cromosomas a cruzar para cada uno de los individuos previamente seleccionados	67
4.11	Ejemplo copiado de los cromosomas seleccionados de cada individuo a cada uno de los descendientes.	67
4.12	Ejemplo de barrido realizado por cada uno de los descendientes para completar su estructura genética.	68
4.13	Algoritmo de la mutación cooperativa aplicada a los descendientes seleccionados	69
4.14	Ejemplo de la Selección de Cromosomas a mutar, para un valor “muta=3”	71
4.15	Arreglo que almacena los clientes de las rutas seleccionadas aleatoriamente en la figura 4.14.	71
4.16	Representación de la mutación cooperativa mediante un grafo disyuntivo	74
4.17	Algoritmo para Cálculo de la Distancia Hamming de una población de N individuos del VRPTW	76
4.18	Modelo de Islas Distribuido con Comunicación Colectiva y Migración Implícita para el AGC-VRPTW	77
4.19	Algoritmo que genera semillas aleatorias y las envía a los nodos de procesamiento utilizando comunicación colectiva	80
4.20	Algoritmo Distribuido AGCD-VRPTW con Comunicación Colectiva	82
4.21	Comunicación de procesos del algoritmo AGCD-VRPTW	84

realizada de forma colectiva en ambos sentidos

4.22 Comunicación Colectiva con MPI_Scatter. Ejemplo del envío de una fila de una matríz. 85

4.23 Comunicación Colectiva mediante MPI_Gather. Ejemplo de recolección de información de cada proceso 86

4.24 Matriz de Estructuras 89

4.25 Etapas de Sincronización con MPI_Barrier 90

4.26 Tarjeta NVidia Tesla C2070 98

4.27 Arquitectura de cada SM de la Tarjeta NVidia Tesla C2070 99

4.28 Lanzamiento de un Kernel a partir de un Proceso ejecutado desde el Host 101

4.29 Modelo paralelo-distribuido AGCP-VRPTW aplicado al VRPTW 103

4.30 Mapeo del proceso de muatción cooperativa a un modelo paralelo en CUDA 106

4.31 Modelo de memoria del device 109

4.32 Bancos de la memoria compartida de la tarjeta C2070 111

4.33 Conflicto de bancos en la memoria compartida 111

4.34 Acceso de un bloque de hilos a memoria compartida sin conflicto de bancos 112

4.35 Ejemplo de un patrón de acceso coherente a la memoria global 113

4.36 Pseudocódigo del wrapper, utilizado para reservar memoria, copiado de datos, lanzamiento de kernels y liberación de memoria 116

4.37 Pseudocódigo del kernel “Mutación” 120

4.38 Patrón de acceso coherente al individuo a mutar para el llenado de la lista tabú 121

4.39 Cálculo paralelo de la probabilidad de los clientes contenidos en los cromosomas a mutar 123

4.40 Pseudocódigo del proceso de actualización de la feromona local aplicando operaciones atómicas en la memoria global 125

4.41 Comportamiento de la función *atomicAdd(…)* 127

4.42	Pseudocódigo del proceso de evaporación y actualización de la feromona del mejor individuo	128
4.43	Representación del método de reducción aplicado para la identificación del individuo con el mejor fitness	130
4.44	Pseudocódigo del manejo de multiGPUs para el AGCP-VRPTW	132
4.45	Diagrama de ejecución de la mutación cooperativa en Grid sobre cuatro tarjetas tesla C2070	134
4.46	Diagrama de ejecución multiGPU en el clúster cuexcomate	134
4.47	Proceso de ejecución de un programa en la GPU	136
4.48	Diagrama de ejecución de un programa MPI-CUDA	137
4.49	Variable de entorno para el uso del compilador OpenMPI	138
5.1	Esquema de distribución de procesos en la Grid utilizafo para llevar a cabo el análisis de sensibilidad del AGC-VRPTW	148
6.1	Gráficas de Convergencia de Instancias Representativas de Benchmarks de Solomon	160
6.2	Gráficas de Convergencia de Instancias Representativas de Benchmarks de Gehring y Homberger	162
6.3	Gráficas de Convergencia del algoritmo distribuido AGC**D**-VRPTW para lo benchmarks de Gehring y Homberger	176
6.4	AGCD-VRPTW trabaja con 132 islas, por lo que se muestra la ocupación de la Grid Morelos al 100%	180
6.5	Sobrecarga de Procesos del AGCD-VRPTW en la Grid Morelos con Ocupación al 300%	185
6.6	Resultados del análisis de eficiencia realizado a la instancia C1_10_1 los datos de la tabla 6.13	190
6.7	Resultados del análisis de eficiencia realizado a la instancia R1_10_1 los datos de la tabla 6.14	190
6.8	Resultados del análisis de eficiencia realizado a la instancia RC1_10_1 los datos de la tabla 6.15	191
6.9	Resultados del análisis de eficiencia realizado a la instancia	191

	C2_10_1 los datos de la tabla 6.16	
6.10	Resultados del análisis de eficiencia realizado a la instancia R2_10_1 los datos de la tabla 6.17	192
6.11	Resultados del análisis de eficiencia realizado a la instancia RC2_10_1 los datos de la tabla 6.13	192
6.12	Cálculo del Speedup	197
6.13	Comportamiento de la Comunicación durante la ejecución del AGCD-VRPTW usando el 100% de los núcleos de la Grid	198
6.14	Efecto del Número de Bloque sobre el valor de la función objetivo para los benchmarks de Gehring y Homberger de 1000 clientes	205
6.15	Gráfica comparativa de los tiempos obtenidos por el algoritmo paralelo-distribuido AGCP-VRPTW vs la versión distribuida AGCD-VRPTW para instancias tipo 1	209
6.16	Gráfica comparativa de los tiempos obtenidos por el algoritmo paralelo-distribuido AGCP-VRPTW vs la versión distribuida AGCD-VRPTW para instancias tipo 2.	210
6.17	Comportamiento del promedio de los Tiempos de Ejecución conforme aumenta la cantidad de bloques	212
6.18	Impacto de Variar el Tamaño de Bloque con respecto a la ocupación del SM	216
6.19	Impacto de Variar la cantidad de Registros por Hilo	217
6.20	Impacto de Variar el uso de Memoria Compartida utilizada por bloque	217

Índice de Tablas

1.1	Tipos de complejidad para el estudio de la eficiencia de los Algoritmos	4
3.1	Tabla comparativa de los métodos paralelizados en una supercomputadora, aplicados al VRPTW	42
3.2	Tabla comparativa de los métodos paralelizados en GPUs aplicados a variantes del problema de ruteo vehicular	47
4.1	Analogía de los elementos utilizados por las colonias de hormigas reales y la metaheurística ACH	53
4.2	Recursos de memoria utilizados en el modelo distribuido AGCD-VRPTW	78
4.3	Especificaciones de la Tarjeta NVidia Tesla C2070	98
4.4	Simbología utilizada en la representación del mapeo del proceso de mutación cooperativa para ser paralelizado con GPUs	106
4.5	Modelo de Memoria del Device de acuerdo a su alcance	110
4.6	Modelo de Memoria del Device de acuerdo a su jerarquía	110
4.7	Tipos de Calificadores para almacenamiento de variables en CUDA	114
4.8	Listado de Operaciones Atómicas disponibles en CUDA	124
4.9	Tabla comparativa de las caracteristicas de los algoritmos AGC-VRPTW, AGC**D**-VRPTW y AGC**P**-VRPTW	138
5.1	Parámetros de Control a Sintonizar para el AGC-VRPTW	140
5.2	Rangos de los Parámetros de Control del algoritmo colonia de hormigas	141
5.3	Rangos de los parámetros de control del algoritmo genético	142

5.4	Tamaño de las Muestras de los Parámetros de Control del algoritmo colonia de hormigas	142
5.5	Tamaño de las muestras de los parámetros de control del algoritmo genético	143
5.6	Muestras definidas para cada uno de los parámetros de control del algoritmo genético cooperativo	143
5.7	Ejemplo 1. Análisis de sensibilidad de α, manteniendo fijos los valores de los demás parámetros	144
5.8	Ejemplo 2. Análisis de sensibilidad de β, manteniendo fijos los valores de los demás parámetros	145
5.9	Resultados del análisis de sensibilidad del AGC-VRPTW para benchmarks de Solomon de 100 clientes	149
5.10	Resultados del análisis de sensibilidad del AGC-VRPTW para Benchmarks de Gehring y Homberger de 1000 clientes	150
5.11	Parámetros de control a sintonizar para el modelo de islas con migración implícita	151
5.12	Rangos establecidos para los parámetrsos de control utilizados durante la migración implícita	152
5.13	Tamaño y Cantidad de Muestras para Análisis de Sensibilidad de la Migración Implícita	152
5.14	Valores utilizados para cada uno de los parámetros de Control de la Migración Implícita	152
5.15	Ejemplo de análisis de sensibilidad de ProbCruce, manteniendo fijo el valor de ProbMuta	153
5.16	Valores Sintonizados para los parámetros mostrados en la tabla 5.11 para las instancias representativas de los benchmarks de Gehring y Homberger	153
6.1	Infraestructura de hardware de los clústeres Cuexcomate y Texcal	156
6.2	Distribución de Recursos en la Grid Morelos por clúster	157

6.3 Clasificación de las instancias utilizadas (Solomon y Gehring y Homberger) 158
6.4 Punto de convergencia de las instancias representativas de los benchmarks de Solomon, con base en lo mostrado en las figuras 6.1 161
6.5 Punto de convergencia de instancias representativas de Gehring y Homberger, con base a las gráficas de la figura 6.2 163
6.6 Análisis Estadístico de los resultados obtenidos por el algoritmo secuencial AGC-VRPTW para Benchmarks de Solomon de 100 clientes 164
6.7 Análisis Estadístico de Resultados Obtenidos por el algoritmo secuencial AGC-VRPTW para Benchmarks de Gehring y Homberger de 1000 clientes 166
6.8 Promedio de los tiempos de ejecución del AGC-VRPTW, aplicado a las instancias representativas de los benchmarks de Solomon de 100 clientes 170
6.9 Tabla comparativa de tiempos de ejecución para las instancias representativas de Solomon de 100 clientes 171
6.10 Promedio de los tiempos de ejecución del algoritmo secuencial AGC-VRPTW, aplicado a las instancias representativas de los benchmarks de Gehring y Homberger 173
6.11 Punto de Convergencia del AGCD-VRPTW para las instancias representativas de los benchmarks de Gehring y Homberger 177
6.12 Análisis estadístico de los resultados obtenidos por el algoritmo distribuido AGCD-VRPTW para Benchmarks de Gehring y Homberger de 1000 clientes 181
6.13 Análisis del comportamiento de la eficacia al variar la cantidad de procesos a ejecutar 184
6.14 Resultados del análisis de eficiencia realizado a la instancia 187

C1_10_1
6.15 Resultados del análisis de eficiencia realizado a la instancia R1_10_1 187
6.16 Resultados del análisis de eficiencia realizado a la instancia RC1_10_1 188
6.17 Resultados del análisis de eficiencia realizado a la instancia C2_10_1 188
6.18 Resultados del análisis de eficiencia realizado a la instancia R2_10_1 188
6.19 Resultados del análisis de eficiencia realizado a la instancia RC2_10_1 189
6.20 Tabla Comparativa entre los resultados obtenidos por el algoritmo secuencial AGC-VRPTW y la versión distribuida AGC**D**-VRPTW 195
6.21 Cálculo del Speedup para R1_10_1 de 1000 clientes 197
6.22 Resultados obtenidos por el algoritmo paralelo-distribuido AGC**P**-VRPTW para los benchmarks de Gehring y Homberger 201
6.23 Puntos de Convergencia para cada una de las instancias representativas de los benchmarks de Gehring y Homberger de 1000 clientes 206
6.24 Relación entre la cantidad de bloques ejecutados por el AGC**P**-VRPTW con respecto al tiempo de ejecución para instancias representativas de Gehring y Homberger 208
6.25 Muestras utilizadas para evaluar el desempeño con sobrecarga del algoritmo paralelo-distribuido AGC**P**-VRPTW 211
6.26 Cálculo de la ocupación del algoritmo paralelo-distribuido AGC**D**-VRPTW 215
6.27 Tabla comparativa de los tiempos de ejecución requeridos por cada uno de los algoritmos propuestos en esta tesis. 218

Capítulo 1

Introducción

La calendarización de recursos tiene gran repercusión en problemas de diversas áreas, como es el caso de problemas de manufactura donde se requiere determinar la secuencia adecuada de operaciones que permita reducir los tiempos de procesamiento de tareas o productos. Otro caso particular es la calendarización de horarios universitarios, donde se busca obtener la distribución adecuada de materias, profesores y alumnos en los salones disponibles en la institución, tratando de proporcionar, en la medida de lo posible, los requerimientos necesarios para que cada materia pueda ser impartida satisfactoriamente sin que se presenten traslapes entre los elementos involucrados [Cruz, Martínez, 2013]. Aunado a lo anterior, en la actualidad, algunos de los problemas que han despertado mayor interés en su estudio a nivel mundial debido a su fuerte relación con la industria y a las grandes pérdidas económicas que provocan anualmente, son los enfocados a distribución y logística, entre los que se encuentran problemas tan variados como los de distribución de redes hidráulicas, redes eléctricas y problemas de transporte, de este último se derivan múltiples variantes que tratan de modelar problemas reales mediante un enfoque general.

Dentro de los problemas de transporte, algunos de los más estudiados debido a su gran similitud con problemas reales presentados en la industria, son los problemas de Ruteo Vehicular, dentro de los que se encuentra una amplia cantidad de variantes, todas consideradas como problemas difíciles de resolver, debido a la complejidad y comportamiento de su espacio de soluciones, así como a la dureza de sus restricciones. Algunas de las variantes más estudiadas a nivel mundial son: el Problema de Ruteo Vehicular con Restricciones de Capacidad (*CVRP* por sus siglas en inglés *Capacited Vehicle Routing Problem*), el Problema de Ruteo Vehicular con Múltiples Depósitos (*MDVRP – Multi-depot Vehicle Routing Problem*), el Problema

de Ruteo Vehicular con Reutilización de Vehículos (*VRPB – Vehicle Routing Problem with Backhauls*) y el Problema de Ruteo Vehicular con Ventanas de Tiempo (*VRPTW – Vehicle Routing Problem with Time Windows*), el cual es abordado en esta tesis, entre otros.

El encontrar una solución que cumpla con todas las restricciones de este tipo de problemas no siempre es una tarea sencilla, ya que esto dependerá sustancialmente de la complejidad del problema y del tamaño de la instancia. Cabe mencionar que una instancia corresponde a los datos de entrada a procesar para un problema determinado. Por lo que, para instancias pequeñas resulta más sencillo encontrar una solución factible, debido a que el número de variables involucradas es menor, por el contrario, la complejidad que se va incrementando conforme crece el tamaño de la entrada.

En este trabajo de investigación se presenta una propuesta de solución al VRPTW basado en una hibridación de un algoritmo genético (AG) y un algoritmo colonia de hormigas (ACH), el cual se es adaptado para ser aplicado como un operador de mutación. El algoritmo obtenido es paralelizado como un modelo de programación distribuida para ser ejecutado en Grid sobre una infraestructura de alto rendimiento. Con la finalidad de obtener una mejora mayor en eficiencia, el algoritmo distribuido es paralelizado utilizando programación híbrida paralelo-distribuida, para ser ejecutada sobre una infraestructura Grid CPU-GPU.

Parte fundamental del diseño de algoritmos eficientes, consiste en el análisis de la complejidad del problema a tratar, ya que existen diversos métodos de optimización cuyas características condicionan su aplicación de acuerdo a la complejidad de los problemas. Para llevar a cabo este análisis es indispensable tomar en cuenta la teoría de la complejidad de los algoritmos y la complejidad de los problemas, la cual se encuentra descrita más ampliamente en los puntos 1.1 y 1.2 de este capítulo, respectivamente. Posteriormente, en el punto 1.3 se describe la motivación para el desarrollo de esta tesis doctoral, en el 1.4 se da una descripción de la clasificación del problema de Ruteo Vehicular, en el punto 1.5 se muestra el estado

del arte del Problema de Ruteo Vehicular con Ventanas de Tiempo, para finalmente dar el objetivo y alcance de la tesis, así como las contribuciones.

1.1 Teoría de la Complejidad de Algoritmos

La *Complejidad Algorítmica* estudia la eficiencia de los algoritmos basado en el tiempo y espacio necesarios para resolver un problema computacional, conocidos como Complejidad Temporal y Complejidad Espacial [Sipser, 2013].

La complejidad temporal se define como una función *T(n)*, misma que es dependiente de la implementación del algoritmo. Estos requerimientos pueden ser expresados en función del tamaño de la instancia, la cual refleja la cantidad de datos de entrada requeridos para abordar el problema.

El cálculo de la *complejidad temporal* de un algoritmo *T(n)* es medida asintóticamente. *T(n)* corresponde al número de pasos requeridos por un algoritmo para resolver un problema [Hartmanis, Hopcroft, 1971]; por lo que a mayor complejidad, mayor será el tiempo que requiera el algoritmo para obtener una solución a un problema dado, de modo que mientras incrementa la complejidad de un algoritmo, su eficiencia disminuye.

La complejidad temporal permite expresar matemáticamente la relación existente entre la cantidad de datos (tamaño de la entrada) y el tiempo de ejecución del algoritmo. De modo que a mayor tamaño de la entrada, mayor será el tiempo de ejecución requerido para su procesamiento. Independientemente del problema abordado, siempre existe el mejor y peor caso, por lo que el intervalo existente se puede definir como $T_{min}(n) < T(n) < T_{max}(n)$, tomando $T_{min}(n)$ como el *mejor de los casos*, es decir que se cumplen las características para el algoritmo requiera la menor cantidad de instrucciones para devolver una solución, por el contrario $T_{max}(n)$ corresponde a aquellas características que implican que el algoritmo debe realizar

todos los cálculos considerados por el algoritmo para obtener una solución, lo que se denomina el *peor de los casos*.

Por otro lado, para el cálculo de la *complejidad espacial*, se toma en cuenta la cantidad de memoria requerida para almacenar los datos utilizados por el algoritmo, debido a que la cantidad de datos que pueden ser almacenados está limitada por la cantidad de memoria con que cuenta el equipo. Otros de los factores que comúnmente afectan la eficiencia de un algoritmo son principalmente la programación, la velocidad del procesador y el tipo de compilador.

Un algoritmo eficiente se centra fundamentalmente en la simplicidad y en el manejo adecuado de los recursos, siendo la sencillez una característica fundamental para el diseño de algoritmos, debido a que facilita el cálculo de la función temporal, además de optimizar el número de instrucciones evaluadas por el procesador, lo cual se ve reflejado en el tiempo de ejecución.

En la tabla 1.1 se muestra la clasificación correspondiente al comportamiento de los algoritmos de acuerdo a la velocidad de crecimiento con base a una entrada n.

Tabla 1.1 *Tipos de complejidad para el estudio de eficiencia de los algoritmos [Aho, et al. 1974; Bisbal, 2009].*

Complejidad	Velocidad de Crecimiento
K	constante
$\log(n)$	Logarítmica
N	lineal
$n \log(n)$	Cuasi-lineal
n^2	cuadrática
n^k	polinomial
2^n	exponencial

De acuerdo a la clasificación mostrada en la tabla 1.1, el algoritmo más eficiente corresponde a aquel que presenta una complejidad constante $T(n) = k$, lo que significa que si el tamaño de la entrada aumenta, el tiempo de ejecución del algoritmo se mantiene constante. En el caso de una complejidad logarítmica $T(n) = log(n)$, si el tamaño de la entrada se incrementa 100 veces, el tiempo de ejecución del algoritmo únicamente se duplicará, por lo que se considera un algoritmo eficiente. Por el

contrario, si un algoritmo presenta una complejidad exponencial $T(n) = 2^n$, este se considera sumamente ineficiente debido a que su tiempo de ejecución aumenta exponencialmente conforme se incrementa el tamaño de la entrada. A continuación, la figura 1.1 muestra gráficamente el comportamiento de las funciones de complejidad mostradas en la tabla 1.1.

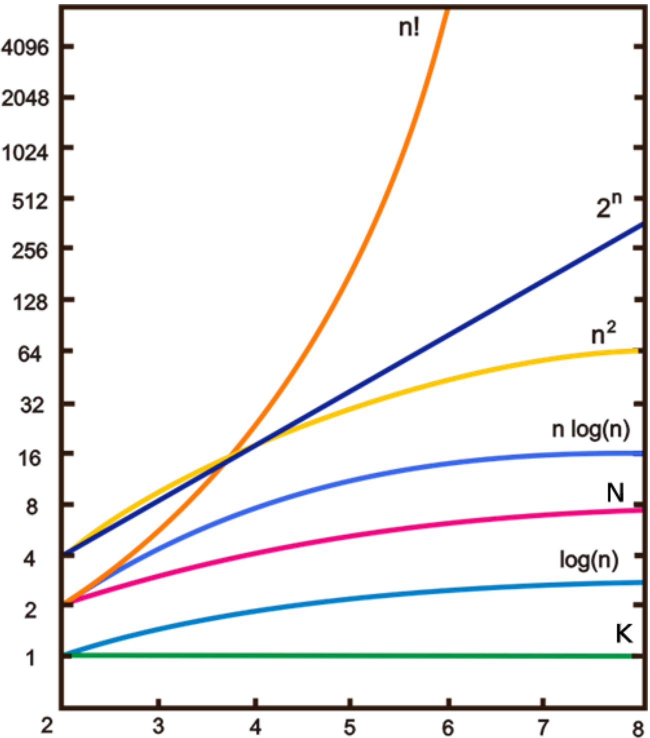


Figura 1.1 *Representación del crecimiento de funciones utilizadas comúnmente en las estimaciones con notación O. [Rosen, 2003]*

La importancia de conocer los tipos de complejidad algorítmica y su comportamiento con respecto al tamaño de la entrada radica en que permite diseñar algoritmos que en la medida de lo posible, cuenten con un desempeño eficiente que permitan obtener soluciones a un problema dado en un tiempo computacional logarítmico o polinomial, lo que implica que independientemente del equipo utilizado, el tiempo computacional requerido por el algoritmo será razonable. Por el contrario, un algoritmo con una complejidad superior a 2^n implica el algoritmo no es aplicable al problema dado, debido al gran incremento en el tiempo computacional con respecto a un pequeño cambio en el tamaño de la entrada.

1.2 Teoría de la Complejidad de los Problemas

La teoría de la complejidad [Papadimitriou, Steiglitz, 1998; Garey, Johnson, 1979; Hopcroft, Ullman, 1993; Cortéz, 2004] estudia la forma de clasificar los problemas de acuerdo a su complejidad para resolverlos. Esto involucra la cantidad de recursos computacionales requeridos por un algoritmo para resolver un problema determinado (explicado en el punto 1.1). De acuerdo a esto, se pueden determinar los recursos necesarios para diversos problemas de optimización y de esta forma poder clasificarlos dentro de las clases de complejidad existentes [Gács, Lovász, 1999], para así poder desarrollar algoritmos eficientes que puedan resolverlos. Esto es de vital importancia debido a que aún el algoritmo más eficiente para un problema determinado, bajo cierto tamaño de instancia puede resultar sumamente ineficiente para cantidades de datos mayores, ya que los recursos computacionales pueden llegar a sobrepasar las capacidades de un equipo de cómputo o bien, el tiempo requerido para encontrar una solución puede ser extremadamente largo.

Los problemas se encuentran divididos dentro de la clasificación más importante en la actualidad, correspondiente a tres clases de complejidad, identificadas como P, NP y NP-Completo, donde la clase P y NP-Completo son subconjuntos de la clase NP, como se muestra en la figura 1.2.

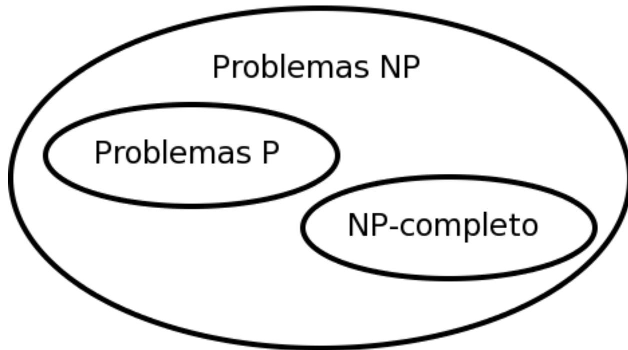


Figura 1.2 *Clasificación de los problemas acorde a la Teoría de la Complejidad.*

Es importante tomar en cuenta la complejidad de un problema antes de proponer un método de solución, ya que de acuerdo a su clasificación se puede identificar la cantidad de recursos requeridos para obtener una solución bajo cierto tamaño de instancia y con base en esto, proponer un método de solución que puede ser determinístico o no-determinístico. De modo que se puede decir que un algoritmo resuelve un problema, si este funciona de forma eficiente para cualquiera de sus instancias [Garey, Johnson, 1979].

De acuerdo a su clasificación [Garey, Johnson, 1979; Sipser, 2006], los problemas P son el conjunto de problemas que pueden ser resueltos en todas sus instancias en el peor de los casos por un algoritmo determinístico en tiempo polinomial. Los problemas NP o problemas intratables, son aquellos que únicamente pueden ser tratados por algoritmos no-determinísticos acotados en tiempo polinomial, debido a que resulta muy difícil y costoso obtener una solución óptima para todos los tamaños de instancia de un problema dentro de esta clasificación. Finalmente, la clase correspondiente a los NP-Completos engloba a los problemas más difíciles de resolver y que de acuerdo a sus características solo pueden ser abordados por algoritmos no-determinísticos.

Los algoritmos determinísticos son aquellos que a una misma entrada de datos devuelve la misma salida. Por otro lado, existen problemas tan difíciles, que se dice que encontrar un algoritmo que lo resuelva de forma eficiente en todas sus instancias es tan complicado como encontrar algoritmos eficientes para todos los problemas existentes dentro de la clase NP. Para este tipo de problemas, se desarrollaron los algoritmos no-determinísticos, los cuales son métodos estocásticos mejor conocidos como *heurísticos* [Colorni et al., 1998], generalmente basados en la experiencia y los *metaheurísticos*, mismos que sustentan su funcionamiento con base a fenómenos naturales. Este tipo de métodos no tiene prueba de optimalidad, por lo que el resultado que arrojan corresponde a una aproximación cercana al óptimo, ya que debido a la complejidad de los problemas a los que son aplicados, hasta el momento no se conoce la solución óptima para todas sus instancias.

1.3 Motivación

A través de los años, se han realizado gran cantidad de estudios en diversas áreas de la optimización, siendo los problemas de calendarización los más frecuentemente abordados debido a su importancia y amplia gama de aplicación. Los primeros estudios enfocados al área de calendarización fueron realizados durante los años 50s [Salveson, 1952] y han ido evolucionando con base en el desarrollo de diversos y novedosos métodos tanto heurísticos como metaheurísticos que han permitido el tratamiento de problemas considerados difíciles de resolver (NP-Completos) como los abordados en [Moreno et al., 2013; Martínez, et al., 2012; Morales et al., 2011; Martínez, 2010; Cruz et al., 2010b; Cruz et al., 2009]. Más recientemente, las nuevas tecnologías como el cómputo de alto rendimiento y la incorporación de tarjetas gráficas como unidades de procesamiento, han dado un nuevo auge al tratamiento de problemas de optimización, abriendo un nicho de oportunidad que permita mejorar el desempeño y escalabilidad de los algoritmos desarrollados para problemas clasificados como NP-Completos. De lo anterior surge la motivación para el desarrollo de un nuevo enfoque que involucre el tratamiento de uno de los problemas clave dentro del área de distribución y logística (el problema de ruteo vehicular) por medio de la explotación de la infraestructura híbrida de la Grid Morelos, la cual permite el uso de recursos CPU-GPU en Grid.

La problemática del transporte radica principalmente en su bien conocida complejidad y en las pérdidas millonarias que se ocasionan debido a un manejo poco eficiente de los recursos. Dichos problemas inciden principalmente en las áreas de distribución y logística, donde los costos de transportación de los productos llegan a generar incrementos considerables al costo final de un producto de hasta un 35% [Balseiro, 2007], por lo que se han convertido en problemas fundamentales dentro del área de investigación de operaciones. Por su naturaleza, los problemas de ruteo se clasifican como NP-Completos [Toth, Vigo, 2002; Lenstra, Rinnoy, 1981], por lo que incluso con los avances tecnológicos actuales, los métodos exactos proporcionan poca utilidad para la solución de instancias grandes de este tipo de problemas, debido a que

una de sus características en que sus requerimientos computacionales se incrementan de forma exponencial conforme crece el tamaño de la entrada [Garey, Johnson, 1979]. Debido a esto surge la necesidad de desarrollar métodos heurísticos y metaheurísticos que permitan obtener buenas soluciones cercanas al óptimo para tratar este tipo de problemas. La ventaja de trabajar con métodos heurísticos o metaheurísticos radica en que al tratarse de métodos estocásticos inspirados en la experiencia o en fenómenos de la naturaleza se facilita su adaptación de acuerdo a las necesidades del problema, con la finalidad de obtener mejores resultados en un menor tiempo.

Con base en lo anterior, surge un nicho de oportunidad basado en el manejo de programación paralelo-distribuida, enfocado en el uso y explotación de la infraestructura de alto rendimiento denominada Grid Morelos. La programación paralelo-distribuida, también conocida como programación híbrida, es aplicada a un método de solución híbrido basado en metaheurísticas que favorezca la distribución de recursos para el problema de ruteo vehicular con ventanas de tiempo mediante la implementación de nuevas tecnologías, como es la Grid Computing [Wilkinson, 2009; Magoulès, 2010] y el manejo de tarjetas GPU como unidades de procesamiento [Sanders, Kandrot, 2011]. Basado en esto, en este trabajo de investigación se plantean las siguientes preguntas e hipótesis:

- ¿Un algoritmo programado para GPUs permitirá un mejor rendimiento y por ende se podrán resolver instancias más grandes en menor tiempo?

Hipótesis

Un algoritmo Genético Cooperativo paralelizado con programación híbrida y ejecutado en ambiente GRID para el problema de Ruteo Vehícular con Ventanas de Tiempo mejorará considerablemente su eficiencia con respecto a las versiones secuencial y distribuida.

1.4 Problema de Ruteo Vehicular, Clasificación y Notación

El problema de Ruteo Vehicular (*VRP* por sus siglas en inglés) surge como una variante del bien conocido problema del Agente Viajero (*TSP* por sus siglas en inglés). El VRP ha sido estudiado por poco más de 50 años, tiempo en el que se ha convertido en uno de los problemas de optimización más estudiados y más importantes. Este problema debe su nombre a que es necesario determinar el mínimo número de rutas requeridas para atender a un conjunto específico de clientes, tomando en cuenta las restricciones definidas por el problema.

El VRP fue propuesto por [Dantzig, Ramser, 1959] con el objeto de describir un problema más real, con lo que se convierten en los primeros en proponer un modelo matemático y un enfoque algorítmico. Años después, [Clarke, Wright, 1964] propusieron un algoritmo voraz que mejoró los resultados obtenidos por Dantzig y Ramser. Posterior a esto, este problema tomó gran importancia debido a su relevancia en diversos enfoques de distribución y logística, principalmente, motivo por el cual ha sido abordado por diversos métodos de optimización, algunos de los cuales pueden consultarse en [Beasley, 1997; Bräysy et al., 2002]. Por lo que a la fecha, además de convertirse en uno de los problemas más importantes en el área de optimización, ha servido como base para el desarrollo de diversas variantes que tratan de modelar problemas generales con características específicas que permitan describir enfoques más cercanos a la realidad.

En la actualidad se han propuesto múltiples variantes del VRP, algunas de las más importantes así como su relación entre sí, se muestran en la figura 1.3 y se describen sus características e importancia a continuación.

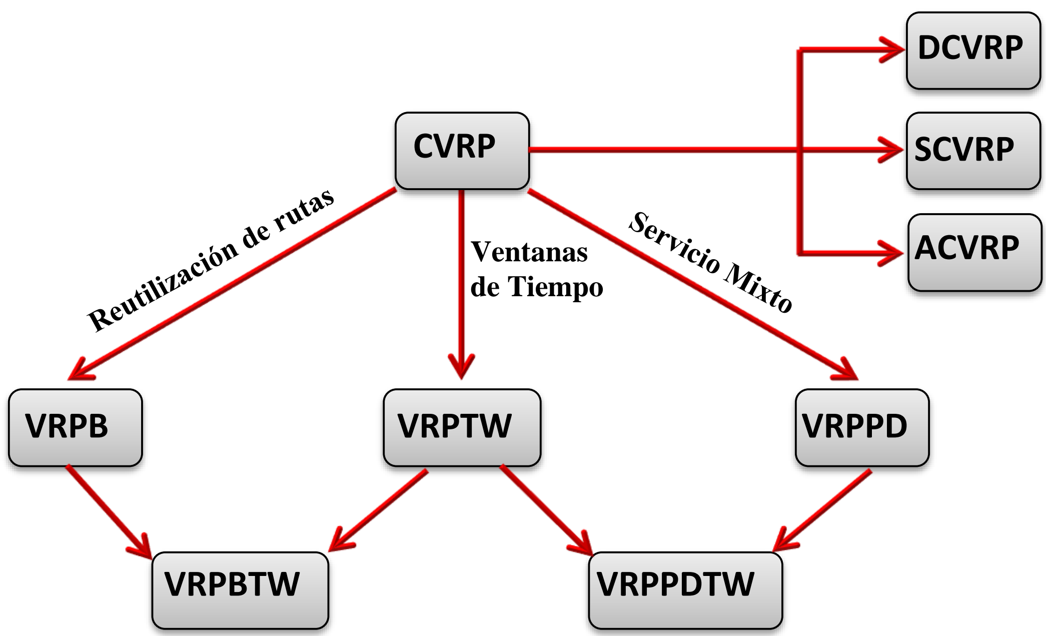


Figura 1.3 *Clasificación básica del VRP y sus relaciones [Toth, Vigo, 2001].*

Problema de Ruteo Vehicular con Restricciones de Capacidad

El problema de Ruteo Vehicular con Restricciones de Capacidad (*CVRP* por sus siglas en inglés *Capacited Vehicle Routing Problem*) es un problema clasificado como NP-Completo [Papadimitrious, Steiglitz, 1998] que puede ser descrito como un grafo tipo cliqué $G = (V, A)$, donde V es el conjunto de clientes $\{i = 1, 2, 3, \ldots, n\}$ y A es el conjunto de arcos que conectan un cliente i con un cliente j, donde cada arco se encuentra ponderado por un costo no-negativo c_{ij} correspondiente al cálculo de la distancia euclidiana existente entre los clientes i y j.

Cada *cliente* cuenta con una demanda específica d_i que debe ser entregada, por lo que será visitado por máximo un vehículo k de una *flota vehicular* K de vehículos homogéneos, de modo que a cada vehículo está vinculado una sola ruta.

Para asegurar la factibilidad de la solución, cada *vehículo* cuenta con una capacidad máxima, por lo que la suma de las demandas d_i de un conjunto de clientes asignados a una ruta $k \in K$, no debe exceder la capacidad máxima C del vehículo

asignado. Por su parte, cada *ruta* debe iniciar y terminar en el *depósito*, identificado como *0* al inicio de la ruta y $n+1$ al final.

Para el CVRP existe una sub-clasificación de acuerdo a las características esenciales del problema. La primera variante definida, fue el problema de Ruteo Vehicular con Restricciones de Distancia (*DCVRP* por sus siglas en inglés), donde la longitud de los arcos t_{ij} relacionada con cada arco $(i,j) \epsilon A$ es no-negativa, además de que cada ruta cuenta con una restricción dura que define el valor de la longitud máxima t que puede tener el recorrido. Cuando la distancia correspondiente a un arco define el tiempo de recorrido para todo arco $(i, j) \epsilon A,$ es importante tomar en cuenta un tiempo de servicio S_i para cada uno de los clientes asignados.

Otra variante se obtiene, si el grafo G del problema corresponde a un grafo dirigido, con lo que se considera que el costo c_{ij} correspondiente al arco $(i, j) \epsilon A$ es diferente al costo de recorrer el arco $(j, i) \epsilon A$. A este problema se le conoce como Problema de Ruteo Vehicular Asimétrico (*ACVRP* por sus siglas en inglés). Por el contrario, si se trata de un grafo G no-dirigido, significa que el costo correspondiente al arco (i, j) es igual al del arco (j, i) para todo $(i, j) \epsilon A$, por lo que a esta variante se le conoce como Problema de Ruteo Vehicular Simétrico (*SCVRP* por sus siglas en inglés).

Problema de Ruteo Vehicular con Reutilización de Rutas

El problema de Ruteo Vehicular con Reutilización de Rutas (*VRPB* por sus siglas en inglés *Vehicle Routing Problem with Backhauls*), se clasifica como NP-Completo [Toth, Vigo, 2001] y es considerado como una extensión del problema de Ruteo Vehicular, por lo que puede ser descrito mediante un grafo tipo cliqué $G = (V, A)$, donde el conjunto V de nodos es dividido en dos subconjuntos (V_1 y V_2). El subconjunto V_1 involucra el conjunto de clientes de V, los cuales esperan los productos que les serán entregados durante el proceso de distribución, por el contrario, el subconjunto V_2 corresponde a los vendedores del conjunto V que

requieren un vehículo que recoja sus productos para transportarlos al depósito. Por su parte, el conjunto *A* del grafo *G* involucra el conjunto de arcos ponderados por el cálculo de la distancia euclidiana que conectan a un cliente *i* con uno *j*.

Es importante tomar en cuenta que las demandas d_i a distribuir y a recolectar en cada punto son datos no-negativos, fijos y conocidos. Además de que la flota de vehículos *K* es homogénea y cuenta con un tamaño definido, donde cada vehículo $k \in K$ tiene especificada una capacidad máxima *C*, la cual no debe ser excedida por la suma de los productos asignados a un vehículo *k* en ninguno de los dos procesos (distribución/recolección). Una de las características esenciales del problema es que cualquier vehículo *k* que pertenezca a la flota *K* puede ser utilizado para hacer ambos servicios, tanto de distribución como de recolección, tomando en cuenta las siguientes restricciones:

- Cada ruta debe iniciar y terminar en el depósito (identificado como *0* al inicio y $n+1$ al final).
- El total de demandas correspondientes a distribución y recolección no deben exceder la capacidad máxima *C* de los vehículos, de forma independiente.

El punto crítico del problema se centra en la restricción de precedencia, que indica que para todo vehículo *k* que realice ambos servicios, el proceso de distribución deberá ser completado antes de comenzar con un proceso de recolección de productos, esto con la finalidad de evitar el costo que involucra un reacomodo de paquetes. Por lo que la función objetivo de este problema es encontrar el conjunto de rutas que minimice el costo total de la distancia recorrida tomando en cuenta todos los servicios requeridos.

Una variante de este problema es conocido como problema de Ruteo Vehicular con Recolección y Distribución (*VRPPD* por sus siglas en inglés), en este caso, cada cliente *i* tiene asociados dos valores d_i y p_i, donde d_i, corresponde a los paquetes que el cliente espera recibir del depósito y p_i son los paquetes que espera que el vehículo *k* recoja. La diferencia con el VRPB radica en que para el VRPPD, para cada cliente O_i específica el origen de donde proviene la demanda, y d_i indica el

nodo de destino para el producto recogido. En caso de que los servicios se lleven a cabo de forma simultánea, el problema se conoce como problema de Distribución y Recolección Simultanea (VRPSPD por sus siglas en ingles).

Problema de Ruteo Vehicular con Ventanas de Tiempo

Uno de los problemas más abordados a través de los años, debido a su conocida complejidad [Lenstra, Rinnoy, 1981; Savelsbergh, 1985; Homberger, Gehring, 1999; [Toth, Vigo, 2001] y a su fuerte relación con problemas reales enfocados principalmente al área de distribución y logística es el problema de Ruteo Vehicular con Ventanas de Tiempo (*VRPTW* por sus siglas en inglés *Vehicle Routing Problem with Time Windows*), el cual se describe de forma general como el conjunto de clientes N que deben ser atendidos por una flota de K vehículos idénticos que cuentan con una capacidad máxima C.

Para asegurar la factibilidad de la solución, es necesario tomar en cuenta las siguientes características:

- Cada cliente debe ser atendido por un solo vehículo.
- El tiempo de atención de cada cliente debe encontrarse dentro de su respectiva ventana de tiempo.
- Cada cliente cuenta con una demanda definida. De modo que la suma de las demandas de los clientes asignados a una ruta no debe acceder la capacidad máxima del vehículo.
- Toda ruta debe iniciar y terminar en el depósito
- Todas las rutas deben llevarse a cabo dentro de la ventana de tiempo del depósito.

Con base en esto, la función objetivo del problema es minimizar el costo total del recorrido c_{ij}, tratando de reducir el número de vehículos utilizados.

Problema de Ruteo Vehicular con Reutilización de Rutas y Ventanas de Tiempo

El problema de Ruteo Vehicular con Reutilización de Rutas y Ventanas de Tiempo (*VRPBTW* por sus siglas en inglés *Vehicle Routing Problem with Backhauls and Time Windows*) es una mezcla de las características del *VRPB* y del *VRPTW,* ambos descritos previamente, por lo que se clasifica como fuertemente NP-Completo [Toth, Vigo, 2001].

Este problema se representa por un grafo $G = (V, A)$, el cual describe un ambiente de doble servicio para la flota de vehículos K, donde el conjunto de vértices V cuenta con dos subconjuntos, correspondientes a clientes y vendedores, pero a diferencia del *VRPB*, cada uno de los clientes y vendedores cuenta con una ventana de tiempo definida dentro de la cual debe ser atendido y un tiempo de atención inamovible.

La complejidad de este problema radica en que todo vehículo k debe realizar el proceso de distribución a los clientes antes de iniciar con el proceso de recolección, para lo cual deben ser tomadas en cuenta las siguientes características:

- El tiempo de atención requerido para cada cliente o vendedor debe llevarse a cabo dentro su respectiva ventana de tiempo.
- Cada cliente y vendedor cuenta con una demanda definida para distribución o recolección, según corresponda. La suma de las demandas asignadas a un vehículo no debe exceder la capacidad máxima del mismo.
- Toda ruta inicia y termina en el depósito, identificado como 0 al inicio y $n+1$ al final.
- Todas las rutas deben realizarse dentro de la ventana de tiempo definida para el depósito.

Con base en esto, la función objetivo del problema es minimizar el costo total del recorrido y reducir el número de vehículos utilizados.

Problema de Ruteo Vehicular con Servicio Mixto y Ventanas de Tiempo

El problema de Ruteo Vehicular con Servicio Mixto y Ventanas de Tiempo, también conocido como problema de Distribución y Recolección con Ventanas de Tiempo (*VRPPDTW* por sus siglas en inglés *Vehicle Routing Problem with Pick-up and Delivery with Time Windows*), debe su nombre a que involucra las características de dos importantes variantes del problema de Ruteo Vehicular, el VRPB y el VRPTW, de modo que su complejidad se incrementa considerablemente, por lo que se le considera como un problema fuertemente NP-Completo [Toth, Vigo, 2001].

Este problema puede ser representado por medio de un grafo $G = (V, A)$, donde a diferencia del VRPBTW, en esta variante todo cliente $i \in V$ cuenta con dos valores asociados, d_i y p_i, donde d_i indica la demanda que espera recibir el cliente y p_i corresponde a la cantidad que deberá recoger el vehículo para trasladarlo al depósito. Otra diferencia significativa es que esta variante toma en cuenta O_i y D_i para identificar de donde provienen los pedidos y a dónde van los paquetes recolectados.

Para asegurar la factibilidad de la solución es importante tomar en cuenta las restricciones de capacidad, así como las características que se enlistan a continuación:

- Todo cliente debe ser atendido en sus dos partes (distribución y recolección).
- El tiempo de atención de cada cliente debe realizarse dentro de su respectiva ventana de tiempo.
- La suma de las demandas de los clientes asignados a un vehículo no debe exceder la capacidad máxima del mismo.
- Cada ruta debe iniciar y terminar en el depósito.

De acuerdo a las características enlistadas anteriormente, la función objetivo es minimizar el costo total del recorrido y minimizar el total de vehículos utilizados.

La importancia de este problema radica en su amplia aplicación a problemas reales, por lo que ha sido abordado por distintos métodos heurísticos y metaheurísticos sobre diferentes tipos de infraestructura, como se describe en el capítulo 3.

1.5 Objetivos de la Investigación

1.5.1 Objetivo General

Desarrollar un algoritmo híbrido paralelo-distribuido en ambiente Grid que aporte conocimiento al área de Optimización Combinatoria utilizando programación híbrida con MPI-CUDA para el Problema de Ruteo Vehicular con Ventanas de Tiempo.

1.5.2 Objetivos Específicos

- Desarrollar un Algoritmo híbrido secuencial utilizando algoritmos Genéticos y el algoritmo Colonia de Hormigas.
- Adaptar el algoritmo Colonia de Hormigas para ser aplicado como operador de Mutación Cooperativa.
- Realizar un análisis de Sensibilidad Distribuido para identificar los valores adecuados de los parámetros de control del algoritmo secuencial.
- Desarrollar la versión distribuida del algoritmo secuencial utilizando comunicación colectiva con MPI.
- Desarrollar la versión paralelo-distribuida del algoritmo secuencial utilizando programación híbrida mediante el uso de MPI-CUDA.
- Realizar el análisis de eficacia y eficiencia de cada una de las versiones del algoritmo para calcular el speed-up y obtener el análisis de las latencias al ser ejercutado en Grid

1.6 Alcance de la Investigación

En este trabajo de investigación se aborda una variante del problema de Ruteo Vehicular, conocida como problema de Ruteo Vehicular con Ventanas de Tiempo

(*VRPTW* por sus siglas en inglés), donde la función objetivo es minimizar el costo total del recorrido y de forma implícita minimizar el total de rutas utilizadas.

- Desarrollar un algoritmo secuencial que permita hibridar dos metaheurísticas: un algoritmo Genético y un algoritmo Colonia de Hormigas.
- Proponer un método distribuido que permita disminuir los tiempos requeridos para realizar el análisis de sensibilidad de los parámetros de control utilizando una infraestructura de alto rendimiento.
- Modificar el algoritmo secuencial con la finalidad de obtener el algoritmo distribuido implementando una modificación del modelo de islas, donde se maneja migración implícita. La comunicación y balanceo de los procesos se realiza mediante la aplicación de comunicación colectiva con MPI.
- Desarrollar el algoritmo paralelo-distribuido, donde los operadores de selección y cruzamiento se mantienen bajo el esquema del modelo de islas con migración implícita y paralelizando la mutación cooperativa para ser ejecutada sobre tarjetas gráficas.
- Tanto el algoritmo distribuido como el paralelo-distribuido utilizan el máximo número de procesadores de la Grid, cuya infraestructura involucra 120 procesadores CPU y 4 tarjetas tesla C2070 con 448 CUDA-cores cada una.
- Realizar el análisis de eficiencia y eficacia al modificar el número de procesos asignados a los núcleos de procesamiento, de modo que de manera inicial se realizó el análisis sin sobrecarga y posteriormente se llevó a cabo un estudio del efecto de la sobrecarga de procesos en los núcleos de procesamiento. Esto para identificar el número óptimo de procesos para el algoritmo propuesto.
- Realizar el estudio del efecto de las comunicaciones (latencia) conforme se incrementa el número de procesos utilizados.

1.7 Contribución de la Tesis

Las aportaciones de este trabajo de investigación se enlistan a continuación:

- Diseño y desarrollo del algoritmo híbrido secuencial AGC-VRPTW, compuesto de un algoritmo genético y un algoritmo colonia de hormigas que se aplica como operador de mutación, denominado "*operador de mutación cooperativa*".
- Desarrollo de la vesión distribuida del algoritmo AGC-VRPTW bajo el esquema de una modificación al modelo de islas, donde se implementa migración implícita para reducir la latencia de las comunicaciones.
- Modificación del algoritmo distribuido AGC**D**-VRPTW, adaptando la mutación cooperativa para ser ejecutada sobre tarjetas gráficas utilizando CUDA, con lo que se obtiene un algoritmo desarrollado con programación híbrida que permite la interacción de dos tipos de arquitecturas (CPU - GPU) para el problema de Ruteo Vehicular con Ventanas de Tiempo sobre una infraestructura de alto rendimiento.
- Aplicación de comunicación colectiva para el envío de arreglos de estructuras de arreglos, con la finalidad de reducir la cantidad de envíos y tiempos de comunicación en el algoritmo distribuido.

1.8 Organización de la Tesis

El presente trabajo de investigación se organiza de la siguiente manera; en el capítulo 1 se presenta una introducción a la teoría de la complejidad de problemas y algoritmos, además se detalla el desarrollo general del problema, basado en los objetivos, alcances y contribución de la tesis. El capítulo 2 muestra la descripción detallada del problema a tratar, su clasificación y modelo matemático, así como su representación por medio de teoría de grafos, la descripción general de una instancia del problema y la representación gráfica de una solución. En el capítulo 3 se presenta

el marco teórico del problema, correspondiente al estado del arte, dividido de acuerdo a la infraestructura utilizada: métodos secuenciales, paralelos en supercomputadoras, distribuidos en clúster, distribuidos en Grid y paralelos con GPUs. Por su parte, en el capítulo 4 se detalla la metodología de solución propuesta, que consiste en tres versiones con 4 pasos cada una. Las versiones son: secuencial, distribuida y paralelo-distribuida con GPUs. Los pasos corresponden al modelo a aplicar, el método propuesto, paralelización (según corresponda) y experimentación en ambiente Grid. En el capítulo 5 se presenta el análisis de sensibilidad realizado de forma distribuida para obtener la sintonización de los parámetros de control de cada una de las versiones del algoritmo propuesto. Los resultados experimentales y el análisis de resultados obtenido mediante la ejecución de cada una de las versiones del algoritmo sobre la infraestructura de la Grid Morelos, dividido en pruebas de eficacia y eficiencia para cada uno de los algoritmos, se presenta en el capítulo 6. Finalmente, en el sapítulo 7 se presentan las conclusiones de esta tesis doctoral, así como los trabajos futuros derivados de la presente investigación.

Capítulo 2

Problema de Ruteo Vehicular con Ventanas de Tiempo

En este capítulo se presenta un panorama generan del problema de Ruteo Vehicular con Ventanas de Tiempo, donde se hace énfasis en sus características y en la descripción del modelo matemático utilizado en esta investigación. Además, se muestra la representación del problema por medio de un modelo de grafos, así como la representación de una solución por medio de un diagrama de Gantt.

2.1 Descripción del Problema de Ruteo Vehicular con Ventanas de Tiempo

La calendarización de recursos constituye una de las áreas con mayor demanda en la actualidad, por lo que el problema del transporte es un punto clave en la toma de decisiones para el área de distribución y logística, principalmente. Dicho problema es considerado como parte medular dentro del área industrial, debido a que actualmente los costos de transporte y distribución impactan directamente en el costo de un producto, además de que el uso poco eficiente de los recursos genera grandes pérdidas económicas anuales a las empresas. De modo que el obtener una reducción en el costo de un producto, por mínimo que éste sea, genera un ahorro considerable, tomando en cuenta la cantidad de productos que maneja una empresa.

Este problema ha servido de inspiración para el desarrollo de diversos modelos teóricos que permitan obtener matemáticamente una representación más cercana a la realidad, y con esto permitir abordar el problema desde un enfoque computacional que facilite el tratamiento del mismo. Cabe mencionar que el problema del transporte enfocado a distribución y logística no es nuevo, ya que los problemas de transporte comenzaron a ser estudiados académicamente a finales de los

años 50's, para posteriormente convertirse en unos de los más importantes y estudiados dentro del área de investigación operativa y optimización combinatoria, debido a su bien conocida complejidad y a que se adaptan a gran cantidad de situaciones de la vida cotidiana en las empresas.

Uno de los modelos que más relevancia ha tomado a nivel científico a través de los años, debido a su complejidad y a sus características estructurales que permiten tomar en cuenta características fundamentales de las cadenas de suministros, como restricciones de capacidad y de tiempo, es el Problema de Ruteo Vehicular con Ventanas de Tiempo (por sus siglas en inglés *Vehicle Routing Problem with Time Windows*).

El *VRPTW* es un problema de optimización clasificado dentro de la Teoría de la Complejidad como NP-Completo en sentido estricto [Solomon, 1987; Toth, Vigo, 2001; Garey, Johnson1979], debido a que involucra características de dos problemas considerados NP-Completos, el problema de Ruteo Vehicular, el cual incrementa su complejidad involucrando el uso de restricciones de tiempo, conocidas como ventanas de tiempo, y el problema de la Mochila (conocido como *Backpacking*), del cual toma las restricciones de capacidad, consideradas como restricciones duras. Lo que significa que para encontrar la solución óptima de una instancia, el esfuerzo computacional crece de forma exponencial conforme se incrementa el tamaño de la entrada, esto debido a que el tamaño de su espacio de soluciones se encuentra acotado por $(\left(1+\frac{N}{K}\right)!)^K$ [Cruz et al., 2012], donde N corresponde al total de clientes a calendarizar y K al total de rutas disponibles.

El VRPTW puede definirse como un grafo ponderado tipo cliqué $G = (V, A)$, donde $V=\{i_0, i_1, i_2, \ldots, i_n\}$ corresponde a un conjunto de clientes y un depósito identificado como 0, tal que el conjunto de arista $A = \{(i, j)\}\ \forall\, i, j \in V, (i \neq j)$. Cada cliente i ubicado en un plano bidimensional (x_i, y_i), cuenta con una demanda $d_i > 0$, un tiempo de servicio $S_i > 0$ y una ventana de tiempo $[a_i, b_i]$ dentro de la cual, el cliente debe ser atendido. En el caso del depósito $d_0 = 0$ y $S_0 = 0$. Para cada arco $(i, j) \in A$ se asocia una distancia c_{ij}, ponderación calculada por medio de la fórmula 1.

$$c_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \qquad (1)$$

Dado que $K=\{k_1, k_2, k_3, ..., k_K\}$ es una flota de vehículos homogéneos con capacidad máxima *C* para cada vehículo. Una solución factible al VRPTW consiste en determinar una serie de rutas para un conjunto de vehículos, con la finalidad de satisfacer la demanda de un conjunto de clientes que se encuentran geográficamente dispersos, con el menor número de vehículos posibles [Dantzing, Ramser, 1959; Cruz et al., 2012], respetando las siguientes restricciones:

- Cada vehículo $k \in K$ puede tener asignada máximo una ruta.
- Cada ruta inicia y termina en el depósito, identificado como *0* al inicio y *n+1* al final del recorrido.
- Toda ruta debe llevarse a cabo dentro de la ventana de tiempo *[E, L]* del depósito, donde *E* indica el tiempo de apertura y *L* el tiempo de cierre.
- Cada cliente *i* debe ser atendido por un solo vehículo *k*, dentro de su respectiva ventana de tiempo $[a_i, b_i]$, durante un tiempo definido S_i.
- La demanda total de los clientes asignados a un vehículo *k* no debe exceder la capacidad máxima *C* del mismo.

Tomando en cuenta que una solución es un conjunto de *m* rutas, cada ruta *k* puede ser representada como la secuencia ordenada de clientes que deben ser atendidos por un solo vehículo. De modo que el costo total de la solución se define como la suma de los costos de cada una de las *m* rutas requeridas para atender al total de clientes. Teniendo como función objetivo minimizar el costo total del recorrido y de forma implícita minimizar el número de vehículos utilizados.

2.2 Modelo Matemático del VRPTW

El problema de Ruteo Vehicular con Ventanas de Tiempo se formula como un modelo de programación lineal entera binaria, el cual se muestra en la figura 2.1, donde el objetivo es minimizar el costo total del recorrido, de manera que el costo puede ser considerado como la distancia requerida para completar el recorrido, para

lo cual se cuenta con un conjunto de vehículos que salen de un depósito (lugar de reunión donde salen y llegan cada uno de los vehículos al inicio y término de cada ruta). Una ruta es la asignación de clientes que se hace a un vehículo para satisfacer la demanda de ciertos productos. La asignación para un vehículo se lleva a cabo tomando en cuenta tanto las restricciones de flujo y las de factibilidad, ambas expresadas en el modelo matemático y descritas a continuación. Por lo que los clientes son asignados a cada una de las rutas mientras se cumpla con su capacidad máxima; lo anterior se repite hasta satisfacer la demanda de todos los clientes. Una vez que todos los clientes hean sido asignados y los vehículos involucrados han terminado sus respectivas rutas, se dice que se ha encontrado una solución al problema.

$$\min \sum_{k \in K} \sum_{(i,j) \in A} c_{ij} X_{ijk} \tag{2}$$

Sujeto a:

$$\sum_{k \in K} \sum_{j \in \Delta^{+}(i)} x_{ijk} = 1 \qquad \forall i \in N \tag{3}$$

$$\sum_{j \in \Delta^{+}(0)} x_{0jk} = 1 \qquad \forall k \in K \tag{4}$$

$$\sum_{i \in \Delta^{-}(j)} x_{ijk} - \sum_{i \in \Delta^{+}(j)} x_{ijk} = 0 \qquad \forall k \in K, i \in N \tag{5}$$

$$\sum_{i \in \Delta^{-}(n+1)} x_{i,n+1,k} = 1 \qquad \forall k \in K \tag{6}$$

$$w_{ik} + s_i + t_{ij} - w_{jk} \leq (1 - x_{ijk}) M_{ij} \qquad \forall k \in K, (i,j) \in N \tag{7}$$

$$a_i \sum_{j \in \Delta^{+}(i)} X_{ijk} \leq w_{ik} \leq b_i \sum_{j \in \Delta^{+}(i)} X_{ijk} \qquad \forall k \in K, i \in N \tag{8}$$

$$E \leq w_{ik} \leq L \qquad \forall k \in K, i \in \{0, n+1\} \tag{9}$$

$$\sum_{i \in N} d_i \sum_{j \in \Delta^{+}(i)} x_{ijk} \leq C \qquad \forall k \in K \tag{10}$$

$$x_{ijk} \geq 0 \qquad \forall k \in K, (i,j) \in A \tag{11}$$

$$x_{ijk} \in \{0,1\} \qquad \forall k \in K, (i,j) \in A \tag{12}$$

Figura 2.1 *Modelo de Programación Lineal Entera Binaria para el VRPTW [Toth, Vigo, 2001].*

La expresión 2 del modelo matemático mostrado en la figura 2.1 representa la función objetivo, la cual corresponde a minimizar el costo total del recorrido así como reducir el número de vehículos requeridos para atender al total de clientes, donde el costo total es considerado como el total de unidades de distancia para realizar el recorrido. Tomando en cuenta que todo recorrido inicia y termina en el depósito, el cuál se encuentra representado por el nodo $i = 0$ cuando el vehículo k inicia su ruta, o bien, por $i = n+1$ cuando el vehículo k finaliza. La variable x_{ijk} indica si un arco (i, j) fue recorrido por un vehículo k, de modo que $x_{ijk} = 1$ sí el vehículo k atiende el arco que va del cliente i al cliente j, de lo contrario $x_{ijk} = 0$.

Para satisfacer la función objetivo es necesario cumplir las restricciones mostradas en el modelo por los conjuntos de restricciones 3 a la 12. Es importante mencionar que dichas restricciones se clasifican como restricciones de flujo de la 3 a la 6 y como restricciones de factibilidad de la 7 a la 10.

Las restricciones de flujo especifican las características de la ruta asignada a un vehículo k. Por otro lado, las restricciones de factibilidad garantizan que se cumplan tanto las restricciones de capacidad como las de tiempo para cada uno de los clientes. A continuación se detalla en que consiste cada una de las restricciones especificadas en el modelo matemático.

Conjunto de Restricciones en 3. Indica que un nodo debe ser asignado a una sola ruta, de modo que el recorrido x de i a j debe ser igual a uno. Lo que asegura que un cliente es atendido sólo por un vehículo k.

Conjunto de Restricciones en 4. Asegura que el número de clientes j directamente alcanzables por un vehículo k a partir del depósito al inicio del recorrido sea igual a uno, de modo que desde el depósito solo un cliente puede ser atendido.

Conjunto de Desigualdades en 5. Garantiza que el número de clientes j directamente alcanzables por el cliente i, con respecto al número de clientes i

directamente alcanzables por el cliente j es cero, por lo que la cantidad de vehículos que atienden a un cliente corresponde al mismo número de vehículos que salen.

Conjunto de Restricciones en 6. Asegura que para cada vehículo k el número de nodos que conectan con el depósito al final del recorrido es igual a uno.

Conjunto de Restricciones en 7. Garantiza que el cliente i será atendido antes que el cliente j. Considerando que la suma del tiempo de inicio del servicio w_{ik} para el cliente i, el tiempo de servicio asignado al cliente i y el tiempo de recorrido del cliente i al cliente j, menos el tiempo de inicio del servicio w_{jk} del nodo j, debe ser menor o igual a $(1-x_{ijk})$. Para garantizar esta desigualdad se da una penalización muy grande representada por M, la cual obliga a la factibilidad de la solución, de modo que sí $x_{ijk} = 0$, la desigualdad se cumple. Lo que significa que no se puede dar servicio al cliente j sí el cliente i no ha sido atendido previamente y el vehículo no ha llegado a j.

Conjunto de Restricciones en 8. Especifica que el tiempo de inicio del servicio w_{ik} del cliente i debe encontrarse dentro de los límites de la ventana de tiempo especificada por $[a_i, b_i]$, donde a_i es el límite inferior de la ventana de tiempo y b_i corresponde al límite superior, conocido como cierre de la ventana.

Conjunto de Restricciones en 9. Indica que cada vehículo k debe respetar el tiempo inicial E y de término de servicio L, correspondiente a la ventana de tiempo del depósito, es decir, todas las rutas deben iniciar y finalizar dentro de la ventana de tiempo definida para el depósito.

Conjunto de Restricciones en 10. Especifica que la suma de la demanda de todos los clientes asignados a un vehículo k no debe exceder la capacidad máxima C del vehículo.

Restricciones en 11. Garantizan la no negatividad de las variables x.

Restricciones en 12. Definen al modelo como de programación lineal entera binaria.

2.3 *Modelo de Grafos Disyuntivos*

Un grafo disyuntivo es una de las representaciones más utilizadas para este tipo de problemas. Un grafo disyuntivo puede ser definido como un grafo $G = (V, A)$, donde V corresponde a un conjunto de vértices y A al conjunto de aristas que unen los vértices en V.

Para el VRPTW, el conjunto V de vértices representa al conjunto de clientes a ser atendidos y el conjunto A representa los arcos ponderados por el valor de la distancia euclidiana que unen los clientes formando un cliqué, de modo que la suma del valor de los arcos recorridos por cada vehículo contribuye al costo total de la solución.

A continuación, en la figura 2.2 se muestra la representación de una instancia pequeña de 8 clientes por medio de un modelo de grafos disyuntivos.

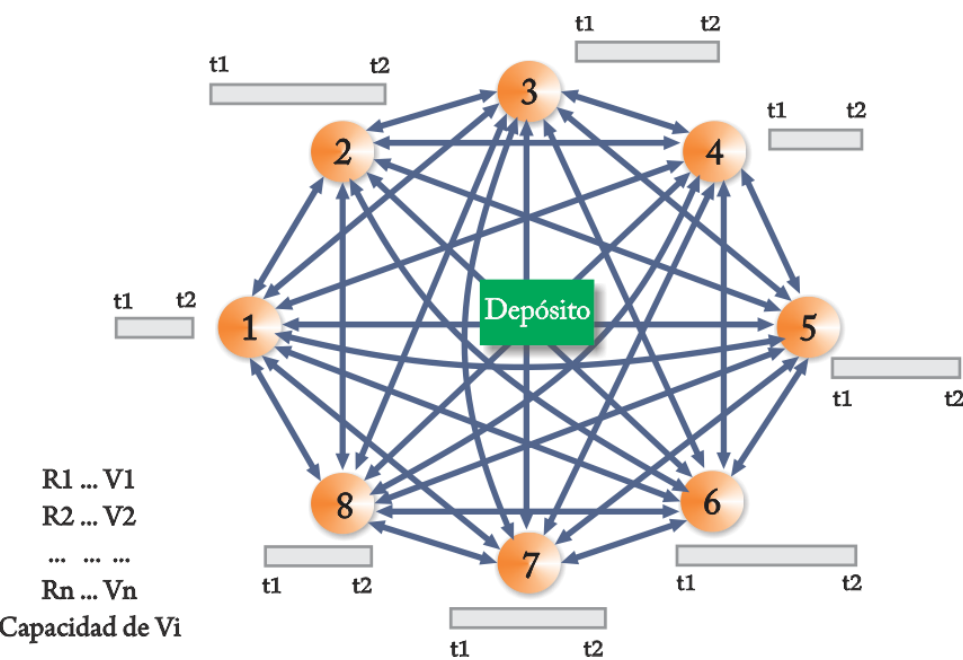


Figura 2.2 *Grafo disyuntivo para una instancia de 8 clientes.*

En el grafo de la figura 2.2, cada vértice corresponde a un cliente, donde cada uno cuenta con una ventana de tiempo [t_1, t_2] correspondiente al periodo durante el cual el

cliente debe recibir la mercancía transportada por el vehículo k. Tomando en cuenta que el objetivo del problema es minimizar el costo total del recorrido, así como la cantidad de vehículos a utilizar, es necesario identificar el número de rutas m que serán requeridas (cada vehículo puede tener a lo más una ruta), de modo que se cumplan los periodos establecidos por la ventana de tiempo de cada cliente y del depósito, así como no exceder la capacidad máxima de cada vehículo. De modo que una solución se encuentra dada por el conjunto de rutas menor o igual a K, que permitan satisfacer la demanda de todos los clientes cumpliendo las restricciones del problema. A continuación, en la figura 2.3 se presenta una posible solución a la instancia representada en la figura 2.2 por medio de un digrafo.

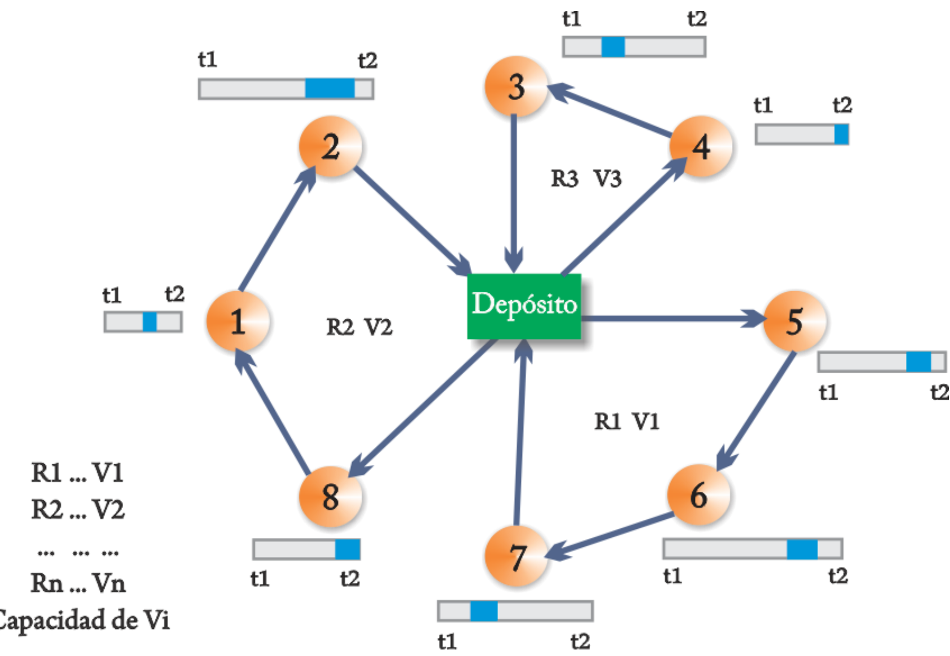


Figura 2.3 *Representación de una solución mediante un dígrafo para la instancia presentada en la figura 2.2.*

En el grafo conjuntivo mostrado en la figura 2.3, se cumple con las restricciones especificadas en el modelo matemático. En esta solución se definen 3 rutas, mismas que están a cargo de 3 vehículos. En cada ruta se puede observar el orden de atención de los clientes, así como los tiempos de atención a cada cliente (recuadro azul) dentro

de su respectiva ventana de tiempo, por lo que es considerada como una solución factible al VRPTW.

2.4 Descripción de una Instancia del VRPTW

Una *instancia* de un problema se define como el conjunto de valores específicos a ser evaluados. Donde un elemento de la instancia involucra un conjunto de parámetros del problema. De modo que el número de elementos definidos para una instancia, corresponde al tamaño de la misma y contribuye al costo computacional requerido por un algoritmo para obtener una solución.

Para el VRPTW se han generado instancias de diversos tamaños que permitan probar la eficiencia y eficacia de los algoritmos desarrollados para tratar este problema. A las instancias que son utilizadas recurrentemente por diversos investigadores para evaluar algoritmos computacionales se les denomina *benchmarks*. Algunos de los benchmarks más utilizados para la evaluación de algoritmos debido a sus características y estructura son los benchmarks de Solomon [Solomon, 1987] y los de Gehring y Homberger [Gehring, Homberger, 1999].

Los benchmarks de Solomon fueron diseñados por Marius M. Solomon en 1987 para evaluar diversas heurísticas de inserción aplicadas al VRPTW. Las instancias propuestas involucran tamaños que van de 25 a 100 clientes, respectivamente, las cuales se dividen en dos clases (*Tipo 1 y 2*) de acuerdo a su dureza y en 3 clases (*C, R y RC*) de acuerdo a su distribución. La clase Tipo 1 indica que las instancias involucradas cuentan con ventanas de tiempo pequeñas y capacidad de los vehículos reducida, lo que permite tener menor cantidad de clientes calendarizados por ruta, contrario a la clase Tipo 2, que maneja ventanas de tiempo más amplias y vehículos con mayor capacidad, permitiendo la asignación de un mayor número de clientes por ruta. En el caso de las clases enfocadas a distribución, las instancias clasificadas en la clase C (*Clustered*), cuentan con una distribución geográfica agrupada de los clientes dentro de un plano 2D. Las instancias correspondientes a la clase R (*Random*), cuentan con clientes distribuidos

aleatoriamente en un espacio euclidiano y finalmente, la clase RC (*Random - Clustered*) corresponde a una distribución mixta de los clientes, que involucra combinar la distribución Agrupada con la Aleatoria, lo que la convierte en el conjunto de instancias con mayor complejidad.

Los benchmarks de Gehring y Homberger fueron propuestos en [Homberger, Gehring, 1999] como una extensión de los benchmarks de Solomon, por lo que fueron diseñados bajo el mismo principio y características. La diferencia radica en que las instancias generadas por Gehring y Homberger son más grandes, involucrando tamaños de 200, 400, 600, 800 y 1000 clientes, bajo las 6 clases especificadas por Solomon.

Cada una de las instancias, independientemente del tamaño y clase a la que pertenezca, cuenta con la misma estructura por cliente. A continuación, en la figura 2.4 se muestra la estructura y datos contenidos en un benchmark de Solomon de 100 clientes.

```
Problem's name

VEHICLE
NUMBER     CAPACITY
  K            Q

CUSTOMER
CUST NO.  XCOORD.   YCOORD.    DEMAND   READY TIME  DUE DATE   SERVICE   TIME

    0        x0         y1         q0         e0          l0            s0
    1        x1         y2         q1         e1          l1            s1
   ...      ...        ...        ...        ...         ...           ...
   100      x100       y100       q100       e100        l100          s100
```

Figura 2.4 *Estructura general de la información presentada en los benchmarks utilizados para evaluar los algoritmos diseñados para tratar el VRPTW.*

La información contenida en un archivo correspondiente a un benchmark para el VRPTW incluye dos partes fundamentales. La primera parte muestra un encabezado con las características generales de la instancia, *nombre de la instancia* (donde se especifica el tamaño de la instancia, tipo y distribución utilizada), *tamaño de la flota de vehículos* (cantidad máxima de vehículos permitidos) y la capacidad máxima

soportada por vehículo (la cual es constante y no puede ser excedida bajo ninguna circunstancia). La segunda parte contempla los datos a evaluar, lo que involucra: *número de cliente ID* (este *id* permite identificar a que cliente corresponden los datos evaluados), *XCOORD* (indica la ubicación del cliente con respecto al eje *X* en un plano 2D), *YCOORD* (muestra la ubicación del cliente con respecto al eje *Y*), *DEMAND* (especifica la demanda requerida por cada cliente), *READY TIME* (corresponde a la apertura de la ventana de tiempo), *DUE DATE* (indica el punto de cierre de la ventana de tiempo) y *SERVICE_TIME* (define el tiempo de atención requerido para cada uno de los clientes). A continuación, en la figura 2.5 se muestra un ejemplo parcial de los datos contenidos por la instancia R101 (distribución aleatoria con 100 clientes y un depósito).

R101

CUST NO.	XCOORD.	YCOORD.	DEMAND	READY TIME	DUE DATE	SERVICE TIME
1	35.00	35.00	0.00	0.00	230.00	0.00
2	41.00	49.00	10.00	161.00	171.00	10.00
3	35.00	17.00	7.00	50.00	60.00	10.00
4	55.00	45.00	13.00	116.00	126.00	10.00
5	55.00	20.00	19.00	149.00	159.00	10.00
6	15.00	30.00	26.00	34.00	44.00	10.00
7	25.00	30.00	3.00	99.00	109.00	10.00
8	20.00	50.00	5.00	81.00	91.00	10.00

Figura 2.5 *Estructura general de la información presentada en uno de los benchmarks de Solomon de 100 clientes, tipo 1 con distribución aleatoria para el VRPTW.*

La estructura de los benchmarks de Solomon es la misma para todos los tipos de distribución y dureza en sus restricciones, representadas como Tipo 1 y 2. Bajo este contexto, Gehring and Homberger desarrollaron en 1999 un nuevo conjunto de benchmarks como una extensión de los benchmarks de Solomon. Esto con el objetivo de manejar conjuntos de datos de mayor tamaño. En el caso de los benchmarks de Solomon, la instancia más grande es de 100 clientes, por su parte, los bechmarks de Gehring and Homberger manejan instancias de 200 a 1000 clientes.

2.5 Representación Simbólica y Diagrama de Gantt

La representación de una solución juega un papel fundamental en el proceso de solución de un problema. Existen dos tipos de representaciones requeridas para una solución. La representación simbólica corresponde a la forma computacional de almacenar todas las características de una solución, para lo cual generalmente se utilizan estructuras de datos (estáticas o dinámicas) o se generan tipos de datos *struct*. Otra forma de representación corresponde al bien conocido *diagrama de Gantt.* Este diagrama ha sido frecuentemente utilizado en problemas de calendarización (*scheduling*) para hacer más comprensible y facilitar el análisis de las restricciones de una solución.

En este trabajo de investigación se generó un tipo de dato estructura (*struct*) como representación simbólica del problema. La estructura del tipo de dato *struct* utilizado se muestra en la figura 2.6.

```
struct Poblacion {
                        int SOLUCION[N];
                        int INICIO[K];
                        int FIN[K];
                        int DEMANDA[K];
                        float COSTO_VEH[K];
                        int NUMVEH;
                        float FITNESS;
                  } solucion;
```

Figura 2.6 *Tipo de dato estructura (struct) generado para el almacenamiento de soluciones factibles al VRPTW.*

Los campos definidos por el tipo de dato estructura mostrado en la figura 2.6 para representar las características de una solución se detallan a continuación.

- Un arreglo ***SOLUCION*** de tamaño ***N***. Permite almacenar la secuencia de cada una de las rutas requeridas para una solución de acuerdo al orden de

atención en cada vehículo, teniendo al final una calendarización de *N* clientes, tal y como se muestra a continuación.

$$\begin{pmatrix} i_{dep,0}, & i_{1,0}, & i_{2,0}, & \ldots, & i_{m,0}, & i_{dep,0}, \\ i_{dep,k+1}, & i_{1,k+1}, & i_{2,k+1}, & \ldots, & i_{m',k+1}, & i_{dep,k+1}, \\ \ldots, & & & & & \\ i_{dep,k<K}, & i_{1,k<K}, & i_{2,k<K}, & \ldots, & i_{m<N,k<K}, & i_{dep,k<K} \end{pmatrix}$$

- Arreglo ***INICIO*** de longitud ***K***. Almacena la posición de inicio (en el arreglo SOLUCION) de la ruta correspondiente a cada vehículo utilizado en la solución factible obtenida. Cabe mencionar que la longitud del arreglo corresponde al máximo número de vehículos existentes en la flota, como se presenta a continuación.

$$(INICIO_0, INICIO_1, INICIO_2, INICIO_3, \ldots, INICIO_K)$$

Es característica del problema tratar de reducir la cantidad de vehículos involucrados en una solución, aunque una solución con menor costo es preferible a una con menor cantidad de rutas y mayor costo.

- Arreglo ***FIN*** de longitud ***K***. Al igual que en el punto anterior, la longitud máxima del arreglo corresponde al tamaño de la flota vehicular. Este arreglo guarda la posición de término de cada ruta asignada a un vehículo en el arreglo SOLUCION, tal y como se muestra a continuación.

$$(FIN_0, FIN_1, FIN_2, FIN_3, \ldots, FIN_K)$$

- Arreglo ***DEMANDA*** de longitud ***K***. Cada elemento de este arreglo de longitud igual al tamaño de la flota vehicular, corresponde a la sumatoria de las demandas de los clientes asignados a una sola ruta (vehículo). Posteriormente, se evalúan las restricciones de capacidad con base al cálculo del costo por

vehículo, de modo que el valor almacenado no debe exceder la capacidad máxima establecida para cada vehículo.

- Arreglo ***COSTO_VEH*** de longitud ***K***. Similar al campo explicado anteriormente, cada elemento de este arreglo permite realizar una evaluación de las restricciones de tiempo del problema (por ruta), de modo que cada elemento del arreglo corresponderá al costo individual por vehículo requerido para atender a todos los clientes asignados a una misma ruta. Es importante mencionar que este valor afecta directamente el costo final de la solución.
- ***NUMVEH***. Este campo muestra el total de rutas (vehículos) utilizadas para atender a todos los clientes requeridos para obtener una solución factible.
- ***FITNESS (Aptitud)***. Corresponde a la suma de los costos de todas las rutas requeridas para obtener una solución. Este valor permite definir la calidad de la solución obtenida con base a la función objetivo definida en el modelo matemático.

Una vez que se cuenta con una solución factible al VRPTW, es importante verificar el cumplimiento de las restricciones del problema, para lo cual, además de realizarlo de forma computacional, se utilizan los diagramas de Gantt. Un diagrama de Gantt es una representación gráfica frecuentemente utilizada para visualizar el resultado de una calendarización. Por ejemplo, una solución representada por medio de una gráfica de Gantt se muestra en la figura 2.7.

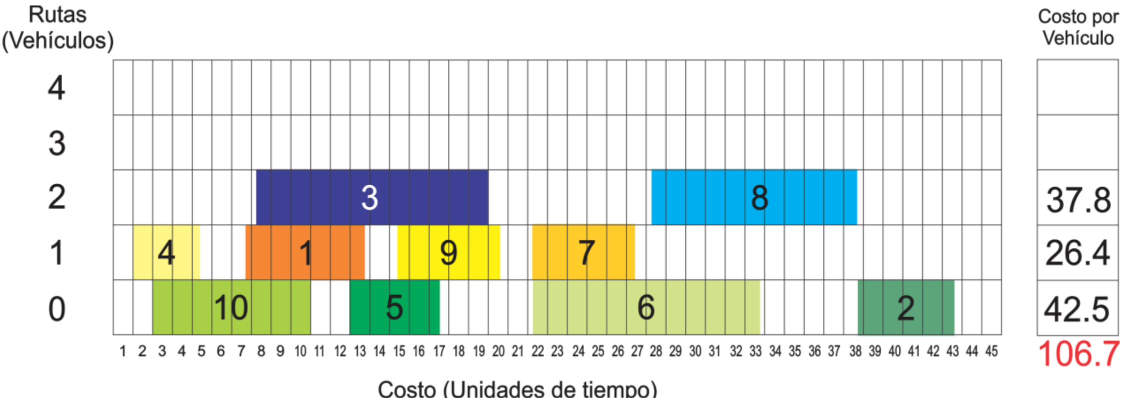


Figura 2.7 *Diagrama de Gantt de una solución factible.*

En esta solución el número de rutas es considerado en el eje Y y el costo de atender a cada uno de los clientes es mostrado en el eje X, teniendo en cada fila de la gráfica, el orden de calendarización de los clientes asignados a una misma ruta. De esta forma es fácilmente identificable cuando se trata de una solución factible al VRPTW, debido a que no se visualizan traslapes en la atención de los clientes. Por otro lado, los espacios en blanco existentes entre los clientes atendidos muestran las unidades de tiempo requeridas para ir de un cliente *i* a un cliente *j,* incluyendo los tiempos de espera requeridos cuando un vehículo llega a un cliente antes de que su respectiva ventana de tiempo se abra. Finalmente, en la columna de la derecha se calcula el costo de atender a los clientes asignados a una misma ruta, de modo que la suma de los costos por ruta corresponde al costo total de la solución.

Capítulo 3

Estado del Arte del VRPTW

El VRPTW es un problema encontrado frecuentemente en diversas empresas. Sus características que engloban aspectos fundamentales de los problemas de logística y distribución, aunado a su complejidad de tipo *NP-Completo,* han atraído la atención de diversos investigadores que tratan continuamente de encontrar métodos que permitan obtener buenas soluciones; esto con la final de poder aplicarlos en la práctica.

Debido a su alta complejidad, el VRPTW ha sido abordado mediante la aplicación de diversas técnicas computacionales que permitan obtener buenas soluciones de manera eficiente. Las técnicas más frecuentemente implementadas, corresponden principalmente a métodos estocásticos, como son heurísticos [Whitley, 1993] y metaheurísticos [Yang, 2010], los cuales faciliten el abordar instancias grandes, debido a que para problemas de tipo NP-Completo, el espacio de soluciones y los requerimientos computacionales crecen de manera exponencial conforme se incrementa el tamaño de la entrada.

Además del uso de métodos heurísticos y metaheurísticos, el rápido avance de la tecnología ha permitido adaptar dichas técnicas computacionales a diferentes arquitecturas que permiten el tratamiento de instancias más grandes. Las nuevas arquitecturas involucran el uso de programación distribuida con MPI (por sus siglas en inglés *Message Passing Interface*) para el uso de infraestructuras clúster y Grid. Más recientemente, las tarjetas gráficas han llamado la atención de investigadores de diversas áreas debido a su rapidez para realizar cálculos numéricos, debido a que permiten el manejo de programación paralela. En la actualidad, una de las plataformas de cómputo paralelo más populares para este propósito es CUDA (por

sus siglas en inglés *Compute Unified Data Architecture*), la cual fue desarrollada por NVidia.

Algunos de los trabajos de investigación más importantes enfocados al tratamiento del VRPTW pueden ser clasificados en métodos secuenciales (sección 3.1), métodos paralelos en supercomputadoras (sección 3.2), métodos distribuidos (sección 3.3) y métodos paralelos con GPUs (sección 3.4), los cuales se detallan a continuación.

3.1 Métodos Secuenciales

El VRPTW ha sido abordado mediante diversos métodos computacionales. Debido a su complejidad es necesario utilizar técnicas estocásticas mejor conocidas como heurísticas y metaheurísticas que permitan obtener soluciones aproximadas cercanas al óptimo global de una forma más eficiente.

Los primeros trabajos enfocados al tratamiento de lo que más tarde sería conocido como el VRPTW, son los presentados por [Pullen, Webb, 1967; Knight, Hofer, 1968; Madsen, 1976], donde se muestran enfoques generales para solucionar instancias pequeñas de una generalización del problema. En la actualidad, los métodos utilizados para abordar gran variedad de problemas de optimización derivados de diversas áreas del conocimiento considerados por la teoría de la complejidad como NP-Completos, han ido evolucionando, permitiendo la aplicación de algoritmos de aproximación como heurísticas, metaheuristicas y métodos híbridos que permitan obtener buenas soluciones de forma cada vez más eficiente. Algunos de los trabajos más relevantes reportados en la literatura se describen a continuación:

En [Bräysy, 2001] se propone un algoritmo híbrido evolutivo de dos fases para abordar el VRPTW. El algoritmo presentado involucra la hibridación de un algoritmo genético y un evolutivo, aplicando diversas búsquedas locales y heurísticas de construcción de rutas. En la primera fase se busca obtener soluciones factibles mediante la implementación de un algoritmo genético. En la segunda fase, el operador de cruzamiento selecciona aleatoriamente pares de rutas y les aplica los

cuatro operadores de búsqueda local utilizados o la heurística de construcción de rutas. Los individuos obtenidos por el cruzamiento son mutados de acuerdo a la probabilidad establecida. El algoritmo fue desarrollado en lenguaje C. Las pruebas experimentales se realizaron con base en los 56 benchmarks de Solomon de 100 clientes. Los resultados experimentales fueron comparados con los obtenidos por el algoritmo MACS-VRPTW [Gambardella et al., 1999] y por el algoritmo de búsqueda Tabú propuesto por [Cordeau et al. 2001], demostrando que el enfoque propuesto es competitivo con respecto a las mejores cotas reportadas. Tomando como base una metodología similar, [Homberger, Gehring, 2005] proponen una metaheurística híbrida de dos fases, donde la primera fase trata de minimizar el número de rutas y la segunda se enfoca en minimizar el costo del recorrido aplicando una búsqueda tabú. Las pruebas experimentales fueron realizadas con base a 356 instancias de prueba de 100 a 1000 clientes, mostrando obtener resultados competitivos. Con base a los trabajos explicados con anterioridad, [Tan et al., 2006] proponen el algoritmo evolutivo híbrido multiobjetivo HMOEA. La diferencia con la propuesta de Bräysy es que este algoritmo involucra el uso de varias heurísticas para mejorar la explotación en el proceso evolutivo. El algoritmo HMOEA propone el uso de operadores genéticos especializados donde en cada individuo se manejan cromosomas de longitudes variables para representar las rutas pertenecientes a una solución. El algoritmo propuesto trabaja con la optimización de la función multiobjetivo y el cumplimiento de las restricciones al mismo tiempo, lo cual permitió obtener mejoras en las soluciones tanto en el número de vehículos utilizados como en los costos. El HMOEA fue probado de forma experimental con base a los 56 benchmarks de Solomon de 100 clientes, donde los resultados mostraron 20 soluciones competitivas, entre las que se observan algunas mejorías con respecto a las mejores cotas publicadas en la literatura. De forma similar [Ombuki et al., 2006] considera el VRPTW como un problema multiobjetivo, minimizando de forma independiente el número de vehículos y el costo total del recorrido mediante la aplicación de un algoritmo genético. Se utiliza un operador de cruzamiento de intercambio en rutas y

aplica una mutación multimodal que considera el intercambio, segmentación e inserción de clientes.

Por su parte, en [Nagata, 2007] se hace uso de un algoritmo evolutivo (*EA* por sus siglas en inglés) al cual se aplica una modificación para asegurar el cumplimiento de las restricciones de capacidad y tiempo. En este trabajo se propone una extensión de dos fases para el operador de cruzamiento EAX diseñado para el TSP simétrico. La primera fase del operador EAX se enfoca en la obtención de soluciones intermedias por medio de la construcción de sub-recorridos. La segunda fase se encarga de calendarizar los sub-recorridos a cada una de las rutas tratando de minimizar la cantidad de vehículos utilizados y el costo total del recorrido. En caso de que existan soluciones infactibles, estas son modificadas por medio de mejoras locales; dicha solución es evaluada con una penalización. Las estructuras de vecindad utilizadas como mejoras locales son 2-exchange y λ-exchange. El algoritmo fue desarrollado en C++ y ejecutado de forma secuencial en un equipo con un procesador Xeon 3.2 GHz. para instancias de 200 clientes y en un Opteron 150 a 2.4 GHz. para 400 y 600 clientes. Las pruebas experimentales mostraron una mejora en el 67% de los benchmarks de Gehring y Homberger de 200 a 600 clientes, a pesar de que el tiempo de cómputo fue mayor que el reportado en la literatura por otros métodos heurísticos.

Un enfoque heurístico eficiente correspondiente a la hibridación de un recocido simulado con un algoritmo de ascenso de colinas (mejor conocido como *hill-climbing*) con reinicio aleatorio, es propuesto en [Brandão, Vasconcelos, 2008]. Este algoritmo presenta una mejora sustancial al resolver problemas de tipo 2, correspondientes a los benchmarks de Solomon, de los cuales se obtuvieron 17 mejoras y se alcanzaron 9 de las cotas reportadas en la literatura. En el caso de los benchmarks de tipo 1 se alcanzaron las cotas reportadas en solo en el caso de las instancias de tipo Agrupadas (*C* por sus siglas en inglés) y Agrupadas-Aleatorias (*RC* por sus siglas en inglés). En el mismo año [Qi et al., 2008] presentan una mejora al algoritmo MACS-VRPTW [Gambardella et. al, 1999], aplicando un método aleatorizado para el cálculo de probabilidades y una búsqueda local de Pareto (*PLS*

por sus siglas en inglés). Dicho algoritmo fue denominado RPACS-VRPTW (*Random Pareto Ant Colony System – Vehicle Routing Problem with Time Windows* por sus siglas en inglés). La mejora realizada al MACS-VRPTW consiste en aplicar el método aleatorizado dentro del proceso de construcción de soluciones de las hormigas, para posteriormente aplicar la búsqueda de Pareto. Las pruebas experimentales se realizaron con base a los benchmarks de Solomon y utilizando los mismos valores de los parámetros de control. Los resultados fueron comparados con los obtenidos por el MACS-VRPTW, donde se observaron algunas mejoras en las instancias de tipo 2.

Más recientemente, en [Yu et al., 2010] proponen un enfoque híbrido compuesto de un algoritmo colonia de hormigas (*ACO* por sus siglas en inglés *Ant Colony Optimization*) y una búsqueda tabú. Las pruebas experimentales se realizaron en una máquina con 512 MB en RAM y un procesador Pentium a 1 MHz. con base a los 56 benchmarks de Solomon. Los resultados obtenidos se compararon con las mejores cotas reportadas, llegando a la conclusión de que el algoritmo es competitivo. Utilizando la primera versión del algoritmo colonia de hormigas, AS (*Ant System* por sus siglas en inglés), [Wang, 2012] presenta un enfoque híbrido del algoritmo de sistema colonia de hormigas con una búsqueda local iterada, denominado IACS. Los resultados computacionales muestran la eficacia del algoritmo propuesto obteniendo resultados competitivos con las mejores cotas reportadas.

3.1 Métodos Paralelos en Supercomputadoras

Uno de los trabajos más relevantes enfocados al VRPTW debido a la eficacia de sus resultados, es el algoritmo paralelo MACS-VRPTW (por sus siglas en inglés *Multiple Ant Colony System – Vehicle Routing Problem with Time Windows*) propuesto en [Gambardella et. al, 1999]. Este enfoque modifica el algoritmo original propuesto en [Dorigo, 1992], debido a que la formulación matemática de Gambardella evalúa una función multi-objetivo, donde se busca minimizar tanto el

costo total del recorrido como el número de vehículos utilizados. Cada una de las partes de la función objetivo se evalúa con una colonia de hormigas independiente de la otra. Por lo que la solución inicial, encontrada por medio de una heurística del vecino más cercano, es evaluada en primera instancia por la colonia ACS-VEI tratando de minimizar el número de vehículos utilizados y posteriormente la solución previamente modificada, es recibida por la segunda colonia ACS-TIME para proceder a buscar una minimización en el costo del recorrido. El algoritmo fue desarrollado en C++ y probado con base en los 56 benchmarks de Solomon de 100 clientes. Los parámetros utilizados fueron 10 hormigas, $q0=0.9$, $\beta=1$, $\rho=0.1$. Las pruebas experimentales fueron realizadas en una Sun UltraSparc1 a 167 MHz., 70 Mflops/seg. Los resultados experimentales mostraron mejoras a algunos de los bechmarks tipo 2 de distribución aleatoria y aleatoria-agrupada. En ese mismo año [Gehring, Homberger, 1999] proponen una metaheuristica híbrida evolutiva paralela de grano grueso ejecutada en una workstation PC-LAN en dos partes: un proceso de control y uno de cooperación, donde el proceso de control se basa en un modelo maestro-esclavo y el proceso de cooperación sigue el concepto de cooperación autónoma que se caracteriza por tener una ejecución concurrente de procesos completamente autónomos, basado en una estratégia evolutiva y en una búsqueda tabú.

En [Gehring, Homberger, 2001] se presenta una metaheurística de dos-fases paralelizada sobre una Workstation PC-LAN, donde los procesos ejecutados cooperan a través del intercambio de soluciones. Las pruebas experimentales se basan en 358 problemas de la literatura con tamaños de 100 a 1000 clientes. Obteniendo resultados competitivos con los reportados en la literatura.

Años más tarde, en [Le Bouthillier, Crainic, 2005] presentan un método multibúsqueda cooperativo paralelo aplicado al VRPTW, donde múltiples hilos de búsqueda cooperan por medio de intercambio de información asíncrona para identificar las mejores soluciones. Cada uno de los procesos independientes implementa un algoritmo evolutivo o una búsqueda tabú. Los resultados obtenidos muestran que el algoritmo obtiene una aceleración lineal con respecto al tamaño de la

entrada. Además de que permite identificar soluciones competitivas con respecto a las mejores reportadas en la literatura.

Más recientemente, [Diaz, 2008] propone la aplicación de un algoritmo genético con estructuras de vecindad GA-VRPTW-PN (por sus siglas en inglés *Genetic Algorithm-Vehicle Routing Problem with Time Windows-Paralell Neighborhood*) implementado de forma paralela utilizando OpenMP. El algoritmo propuesto aplica selección por torneo, la técnica de cruzamiento-k y mutación-s, ambas propuestas en este trabajo de investigación. La estructura de vecindad utiliza una vecindad modificada tipo OR con movimientos 1-óptimos. El algoritmo fue ejecutado en los 32 procesadores de la supercomputadora paralela IBM PSeries p690 Regatta-Aleph utilizando los benchmarks de Solomon de 25 a 100 clientes, con lo que se obtuvo una mejora en 7 de los benchmarks utilizados con respecto a las mejoras cotas publicadas. [Blocho, Czech, 2013] posteriormente presentan un algoritmo memético paralelo *PMA* (*Parallel Memetic Algorithm* por sus siglas en inglés). El algoritmo se compone de dos partes, las cuales son ejecutadas en paralelo. El algoritmo presenta un proceso de cooperación entre procesos, el cual busca obtener mejores soluciones. La paralelización se llevó a cabo utilizando la librería de paso de mensajes MPI. Las pruebas experimentales se realizaron en la supercomputadora Galera con los 64 procesadores que la componen, utilizando los benchmarks de Solomon y de Gehring y Homberger. El análisis de eficacia mostró que en el 89% de las pruebas se obtuvo la mejor cota conocida

Lo descrito anteriormente se presenta concentrado en la tabla comparativa 3.1, la cual permite visualizar las características de cada uno de los algoritmos.

Tabla 3.1 *Tabla comparativa de los métodos paralelizados en una supercomputadora, aplicados al VRPTW*

Algoritmo	Autor	Método	Infraestructura	Benchmarks
MACS-VRPTW	Gambardella et al., 1999	Algoritmo Colonia de Hormigas	Sun UltraSparc1	56 benchmarks de Solomon de 100 clientes
Metaheurís-tica Evolutiva	Gehring, Homberger, 1999	Estrategia Evolutiva y Búsqueda	Workstation PC-LAN	56 benchmarks de Solomon de 100 clientes

Paralela		Tabú		
Metaheurística de dos fases	Gehring, Homberger, 2001	Estrategia Evolutiva y Búsqueda Tabú	Workstation PC-LAN	358 instancias de 100 a 1000 clientes
Multibúsqueda Cooperativa Paralela	Le Bouthillier, Crainic, 2005	Algoritmo Evolutivo y Búsqueda Tabú	Búsqueda Tabú	56 benchmarks de Solomon de 100 clientes 200 de Gehring y Homberger de 200 a 1000 clientes
GA-VRPTW-PN	Díaz, 2008	Algoritmo Genético con Estructuras de Vecindad	Supercomputadora paralela IBM PSeries p690 Regatta-Aleph	Benchmarkas de Solomon de 25 a 100 clientes
PMA	Blocho, Czech, 2013	Algoritmo Memético	Supercomputadora Galera con 64 procesadores	Benchmarks de Gehring y Homberger

3.3 Métodos Distribuidos en Clúster

Desde los inicios del paralelismo, las supercomputadoras han jugado un papel fundamental en el desarrollo de algoritmos enfocados al cálculo científico, debido a las ventajas que ofrecen sus cientos e incluso miles de núcleos de procesamiento dedicados. Por otro lado, cuentan con una desventaja, su elevado costo, lo que favoreció el surgimiento de nuevas tecnologías como es el caso de los clústeres de computadoras.

Los clústeres de computadoras han surgido como una alternativa de cómputo distribuido derivada de las tendencias tecnológicas actuales, así como de la disponibilidad de microprocesadores a bajo costo y de las mejoras en la velocidad de las redes. Por lo que un clúster puede ser definido como el conjunto de computadoras conectadas a través de una red de alta velocidad que permite tener acceso a todos los recursos como si se tratara de una sola computadora. Las ventajas de trabajar con clústeres de computadoras se basan en obtener un alto rendimiento y disponibilidad, así como mayor escalabilidad a menor costo que en el caso de una supercomputadora.

El desarrollo de algoritmos que exploten las ventajas de esta infraestructura se basan en programación distribuida utilizando el esquema de paso de mensajes. Uno de los primeros trabajos desarrollados bajo este esquema es el propuesto por [Arbelaitz, Rodríguez, 2000], donde se presenta el diseño y análisis de un algoritmo de recocido simulado paralelo de dos fases para mejorar la eficiencia en el tratamiento del VRPTW. El algoritmo se desarrolló utilizando el esquema de paso de mensajes para lograr la comunicación entre los núcleos de procesamiento. Las pruebas experimentales se realizaron con base a los 56 bien conocidos benchmarks de Solomon de 100 clientes y sobre un problema real. Los resultados mostraron que incluso para instancias pequeñas, los resultados son siempre cercanos al óptimo, además de que acorde al análisis de eficiencia, se observó que el incremento en el número de procesadores utilizados favorece la eficiencia del algoritmo.

Un esquema similar se presenta en [Wieczorek, 2011], donde se propone un algoritmo distribuido de búsquedas independientes con tamaño constante de la etapa de enfriamiento (*ISC* por sus siglas en inglés *Independent Search with Cooling*) para mejorar la eficacia de las soluciones obtenidas. El algoritmo fue desarrollado en lenguaje C haciendo uso de la librería de paso de mensajes MPI. Las pruebas experimentales se llevaron a cabo en un clúster con 11 nodos homogéneos. Cada nodo cuenta con un procesador Intel Core 2 Quad a 2.4 GHz. Las pruebas se realizaron con base a algunas de las instancias de Solomon como R109, R110 y R202, donde se observa un decremento en la eficacia para las instancias R109 y R110 conforme aumenta la cantidad de procesadores utilizados. Contrario a lo que sucede con la instancia R202 donde se obtienen buenos resultados independientemente del número de procesadores utilizados.

3.4 Métodos Distribuidos en Grid

La infraestructura Grid ofrece una forma distribuida de obtener mayor capacidad de cálculo gracias al acceso compartido a los recursos de diversos clústeres separados

geográficamente, los cuales se encuentran conectados mediante una red que permite acceder a todos los recursos de forma transparente. El funcionamiento de la Grid se enfoca en vincular y compartir información, así como en dar la idea de estar trabajando con una sola supercomputadora virtual, de donde surge el concepto de *"grid computing"*.

En la actualidad, investigadores de diversas áreas se han enfocado en el tratamiento de problemas de optimización considerados difíciles de resolver, donde el tamaño de las instancias evaluadas afecta directamente la eficiencia de los algoritmos. Por lo que el desarrollo de algoritmos distribuidos en CPUs corresponde a una alternativa que favorece la eficiencia, mediante la división de procesos distribuyéndolos en los recursos de la Grid. Aunque el desarrollo de este tipos de algoritmos aun es mínimo, uno de los trabajos de investigación más relevantes bajo dicho esquema es el propuesto por [Rodríguez et al., 2010] donde se presenta un esquema de dos etapas para el envío de segmentos de población del algoritmo genético propuesto por [Díaz, 2008] utilizando comunicación bloqueante, donde dicha modificación se denomina algoritmo Genético Paralelo (PGA). Los segmentos de la población son enviados a los nodos de la Grid llamada "*Tarántula*" para tratar el VRPTW. El envío de individuos se lleva a cabo mediante la interfaz de paso de mensajes MPI, a través de la tecnología VPN (*Virtual Private Network*), la cual crea un túnel entre redes separadas geográficamente utilizando Internet 2. Cada nodo de la Grid ejecuta el operador de mutación en el segmento de la población asignada. Este proceso continúa hasta que la mejor solución sea encontrada o se alcance el máximo de generaciones. El análisis de eficiencia muestra que el speed-up obtenido por el algoritmo es cercano al ideal.

3.5 Métodos Paralelos en GPUs

Las nuevas tecnologías han dado un nuevo auge al estudio de los problemas de Optimización, en donde destaca el tratamiento del problema de Ruteo Vehicular y sus

variantes. Debido a las características propias del VRPTW clasificado como NP-Completo, para el manejo de instancias cada vez más grandes, es necesario incrementar el poder de cómputo y utilizar las ventajas del paralelismo, de modo que la eficiencia de los algoritmos pueda ser mejorada. En la actualidad, el uso de tarjetas gráficas se ha convertido en parte integral del desarrollo de algoritmos que permitan abordar instancias cada vez más grandes de problemas considerados como difíciles de resolver. Para lo cual el uso de programación paralela ha jugado un papel fundamental, más aún con el desarrollo de Open CL y más recientemente del paradigma de programación CUDA, desarrollado por NVidia.

Actualmente, el número de trabajos de investigación que tratan el VRPTW y aplican la programación con GPUs, son pocos aunque se encuentran en constante incremento. Uno de los trabajos de investigación desarrollados para dicho problema, es el propuesto por [Jian-Ming et al., 2010], en donde se presenta un algoritmo de Recocido Simulado Paralelo (*PSA* por sus siglas en inglés) basado en la arquitectura GPU. El método propuesto implementa un algoritmo de reducción para la evaluación de las soluciones, haciendo uso de la memoria compartida. Las pruebas experimentales fueron realizadas con base en instancias de 100 a 500 clientes en un equipo con procesador Intel Core 2 Duo con una tarjeta GeForce 9800GT, y 3.25 GB en RAM. Los resultados demostraron que el algoritmo permite incrementar el tamaño de la población, lo que favorece el procesamiento de grandes cantidades de datos de forma simultánea. Además de que el algoritmo presenta una relación sub-lineal entre el tamaño de la entrada y el tiempo de ejecución.

Años más tarde, [Diego et al., 2012] propusieron un algoritmo paralelo de optimización por colonia de hormigas (ACO) para tratar una versión más generalizada del problema abordado en este trabajo de tesis, el CVRP (*Capacited Vehicle Routing Problem*, por sus siglas en inglés). La implementación se realizó en CUDA en una máquina con una tarjeta GeForce GTX46, un procesador Pentium Dual Core a 2.7 GHz. y 2 GB en RAM. Las pruebas experimentales se realizaron con las instancias de Augerat (36 a 80 ciudades), además se generaron nuevas instancias aleatorias más grandes (100 a 700 ciudades). De manera general, se obtuvo un

speedup del 12%. En 2013, el mismo problema fue abordado mediante la propuesta de [Szymon, Dominik, 2013], donde se desarrolla una búsqueda tabú paralelizada con CUDA (PPATS). Las pruebas experimentales fueron realizadas utilizando 2 tarjetas GPU, NVidia Tesla S2050 y una GeForce GTX480. Los resultados mostraron que el algoritmo obtiene resultados competitivos con respecto a las mejores cotas conocidas.

Otro antecesor del VRPTW que ha sido tratado mediante el desarrollo de algoritmos paralelos en GPUs, es el CVRP (*Capacited Vehicle Routing Problem,* por sus siglas en inglés). Este problema es abordado en [Wodecki et al., 2014], donde se presenta un algoritmo memetico paralelo en GPUs. Las pruebas experimentales se realizaron en una tarjeta Tesla S2050 y se utilizaron los benchmarks del TSPLIB. Las pruebas de eficiencia mostraron que el comportamiento del speedup experimental es similar al speedup teórico, además de que la eficacia mostrada para las instancias utilizadas es cercana al óptimo.

Las características de los algoritmos descritos anteriormente se presentan en a tabla comparativa 3.2, misma que permite visualizar las características de cada uno de los algoritmos.

Tabla 3.2 *Tabla comparativa de los métodos paralelizados en GPUS aplicados a variantes del problema de Ruteo Vehicular*

Algoritmo	Problema	Método	Infraestructura	Benchmarks	Memoria
PSA (Recocido Simulado Paralelo)	VRPTW	Recocido Simulado Paralelo	Procesador Intel Core 2 Duo con 14 ciclos de reloj. Tarjeta GeForce 9800GT con 112 cuda-cores	Instancias de 100 a 500 clientes	Global y Compartida
ACO para CVRP Paralelo	CVRP	Algoritmo Colonia de Hormigas	Procesador Intel Pentium Dual Core a 2.7 GHz Tarjeta GeForce GTX46, un procesador.	Instancias de Augerat (36 a 80 clientes). Instancias aleatorias más grandes (100 a 700 clientes)	Global, Compartida y registros

PPATS	CVRP	Búsqueda Tabú	2 tarjetas GPU, NVidia Tesla S2050 con 448 cuda-cores y una GeForce GTX480 con 448 cuda-cores	Instancias de 10 a 500 clientes	Global y Compartida
Memético Paralelo	CVRP	Algoritmo Genético y Búsqueda Local	Tarjeta Tesla S2050 con 448 cuda cores	Benchmarks de TSPLIB	Global y Compartida

Capítulo 4

Metodología de Solución

En este capítulo se presenta la metodología de solución desarrollada mediante la hibridación de un algoritmo genético con un algoritmo colonia de hormigas aplicados al VRPTW. Esta propuesta se presenta como un Algoritmo Genético Cooperativo (AGC-VRPTW), donde la propuesta de solución incluye el desarrollo de tres versiones del algoritmo, secuencial, distribuida y distribuida-paralela, las cuales fueron ejecutadas sobre la infraestructura de la Grid Morelos, misma que se describe en la sección 6.1.

La metodología seguida en esta tesis para tratar el VRPTW se explica a continuación:

- ***Modelo Matemático:*** Consiste en el análisis y aplicación del modelo matemático de programación lineal entera binaria explicado en la sección 2.2, donde su representación se describe en la sección 2.3 por medio de un grafo disyuntivo.
- ***Algoritmo Secuencial AGC-VRPTW:*** Se realiza una hibridación con un algoritmo genético y un algoritmo colonia de hormigas para encontrar soluciones de buena calidad al VRPTW en tiempo polinomial. El algoritmo denominado AGC-VRPTW se detalla en la sección 4.1.
- ***Algoritmo Distribuido AGCD-VRPTW:*** Se enfoca en el análisis, diseño y programación del algoritmo secuencial con base a una variante propuesta para el Modelo de Islas, mediante la Interfaz de Paso de Mensajes MPI con comunicación colectiva, mismo que se detalla en la sección 4.2.
- ***Algoritmo Paralelo-Distribuido AGCP-VRPTW:*** El método de solución obtenido en el paso 4 fue analizado para ser paralelizado utilizando programación híbrida paralelo-distribuida, esto con la finalidad de mejorar la

eficiencia del algoritmo, mediante el uso de núcleos de procesamiento CPU-GPU. Este modelo se explica detalladamente en la sección 4.3.

- ***Arquitectura Computacional:*** Para las pruebas experimentales de las tres versiones del algoritmo mencionadas anteriormente, así como para el análisis de sensibilidad del algoritmo secuencial y distribuido, se hizo uso de la plataforma denominada Grid-Morelos (sección 6.1), la cual se encuentra c
- Conformada por clústeres de alto rendimiento que se encuentran en diferentes puntos geográficos.
- ***Análisis de Sensibilidad (Sensitividad de los Parámetros de la Solución):*** Análisis de sensibilidad en clúster para identificar los valores de los parámetros de control del algoritmo propuesto. Se detalla en el capítulo 5.
- ***Pruebas Experimentales:*** Para el algoritmo secuencial, se utilizaron los benchmarks de Solomon y los de Gehring y Homberger y para las versiones distribuida y paralelo-distribuida únicamente se utilizaron los benchmarks de Gehring y Homberger debido a que se requiere explotar los recursos de la Grid, por lo que los benchmarks de Solomon son muy pequeños para ser utilizados sobre una infraestructura de alto rendimiento. El análisis de los resultados obtenidos se detalla en el capítulo 6.

4.1 Algoritmo Secuencial AGC-VRPTW

Una de las aportaciones de este trabajo de investigación es el método secuencial para tratar el VRPTW, ya que de acuerdo a su clasificación dentro de la teoría de la complejidad, su tratamiento solo se puede llevar a cabo mediante métodos no determinísticos, conocidos como heurísticas y metaheurísticas, debido a que no se conoce un algoritmo determinístico en tiempo polinomial que permita resolver problemas de tipo NP-Completos.

El método propuesto consiste en la hibridación de dos metaheurísticas, un algoritmo genético (*AG*) y un algoritmo colonia de hormigas (*ACH*) en un algoritmo denominado en este trabajo como AGC-VRPTW (por sus siglas *Algoritmo Genético*

Cooperativo – Vehicle Routing Problem with Time Windows). A continuación se describe cada uno de los métodos utilizados en la hibridación propuesta

Los algoritmos genéticos [Holland, 1962; Holland, 1962b; Holland, 1967; Holland, 1975] son métodos metaheurísticos adaptativos inspirados en la teoría de la evolución postulada por Darwin en 1859, los cuales se basan en una analogía del ciclo genético de los seres vivos, donde involucra la selección natural y la supervivencia de los individuos más aptos con base en la combinación de genes y la mutación que se presenta de forma natural entre los individuos. Este método es utilizado para el tratamiento de diversos problemas de la vida real, donde el proceso de evolución se refleja en la mejora de las soluciones basado en una función objetivo y la adaptación se presenta con un valor cuantitativo que indica la calidad de las soluciones obtenidas. De acuerdo a su naturaleza, los AGs se basan en poblaciones, donde los individuos de la población correspondientes a soluciones factibles de un problema cuentan con un valor que indica el grado de adaptación. De acuerdo al ciclo natural, mientras mayor sea la adaptación del individuo, mayor será la probabilidad de ser seleccionado para reproducirse, lo que conlleva mezclar sus genes con el de otro individuo seleccionado bajo los mismos términos. El proceso explicado anteriormente permite preservar los genes de los individuos mejor adaptados por un mayor número de generaciones a través de los descendientes, los cuales cooperan en la creación de una nueva población que reemplazará a la generación anterior.

Las ventajas de utilizar un AG corresponde a que se trata de un método robusto que puede ser adaptado para el tratamiento de diversos problemas de optimización, donde se ha comprobado que se obtienen buenas soluciones. Por otro lado, una característica fundamental para el buen desempeño de los AGs, además de los tipos de operadores aplicados, es la diversidad de la población [Martínez et al., 2012], ya que estos métodos son propensos a quedar atrapados en óptimos locales debido a la pérdida de la misma. A continuación se presenta en la figura 4.1 el pseudocódigo de un AG canónico [Holland, 1975], que corresponde a la versión más simple de los AGs.

```
1. INICIO
2. Genera Población Inicial
3. Evalúa Fitness          /*Con respecto a la función objetivo*/
4. G =0                    /*Inicializa contador de generaciones*/
5. Hacer                   /* Aplicación de operadores genéticos*/
          Selección
          Cruzamiento
          Mutación
          Evalúa Fitness
          Actualiza Población
   Mientras G<MAXGEN       /* Mientras G sea menor al máximo de
                              generaciones*/
6. FIN
```

Figura 4.1 *Pseudocódigo de un Algoritmo Genético Canónico*

De acuerdo a la estructura general de un AG, la generación de la población inicial y los operadores genéticos de Selección, Cruzamiento y Mutación juegan un papel fundamental en el buen desempeño del algoritmo, ya que de ellos depende el mantener la diversidad en la población hasta obtener la convergencia.

Por su parte, el algoritmo colonia de Hormigas [Dorigo et al., 1992], al igual que en el caso de los AGs, se presenta como una analogía del comportamiento de los seres vivos, en este caso, de las hormigas. En 1992, Marco Dorigo propuso en su tesis de doctorado una metaheurística cooperativa inspirada en el comportamiento de las hormigas en su búsqueda de alimento, donde se lleva a cabo una especie de comunicación indirecta entre las hormigas por medio de rastros de feromona denominada *stigmergy* [Grassé, 1959]. La analogía de los elementos requeridos por las hormigas reales con respecto al método constructivo para el problema tratado se muestra en la tabla 4.1.

De acuerdo a la analogía presentada en la tabla 4.1, el ACH es un método constructivo basado en el comportamiento colectivo de los agentes en sistemas autoorganizados.

Tabla 4.1 *Analogía de los elementos utilizados por las colonias de hormigas reales y la metaheurística ACH*

Proceso Natural	Método de Optimización ACH
Habitat	Grafo
Fuente de Alimento	Cada uno de los clientes a atender
Hormiga	Agente constructor de soluciones
Visibilidad	Inversa de la distancia
Feromona	Rastros químicos artificiales (Feromona artificial)
Colonia	Conjunto de agentes constructores de soluciones

Algunas de las características de estos algoritmos es que se componen de un conjunto de individuos homogéneos denominado "Colonia", los cuales son ciegos por lo que es necesario utilizar una sustancia química que les permita mantenerse comunicados con respecto a los caminos hacia la fuente de alimento. Dicha sustancia se denomina feromona y permite indicar a las demás hormigas de la colonia los mejores caminos hacia la fuente de alimento, de modo que los caminos más cortos concentrarán mayor cantidad de feromona, contrario a lo que sucede con los caminos más largos, como se observa en la figura 4.2.

Este método de solución fue aplicado inicialmente al problema del trasporte, aunque en la actualidad ha sido aplicado a problemas de diversas áreas, donde cada una de las hormigas de la colonia se encarga de construir una solución factible recorriendo un grafo $G= (V, A)$ respetando las restricciones del problema tratado, dejando cierta cantidad de feromona en los trayectos recorridos para indicar a la colonia la calidad del recorrido. Debido a las limitaciones de las hormigas, los rastros de feromona constituyen parte fundamental del proceso cooperativo, de modo, que de forma inicial, se asigna un monto de feromona idéntico a todos los arcos, conocido como τ_{ij}.

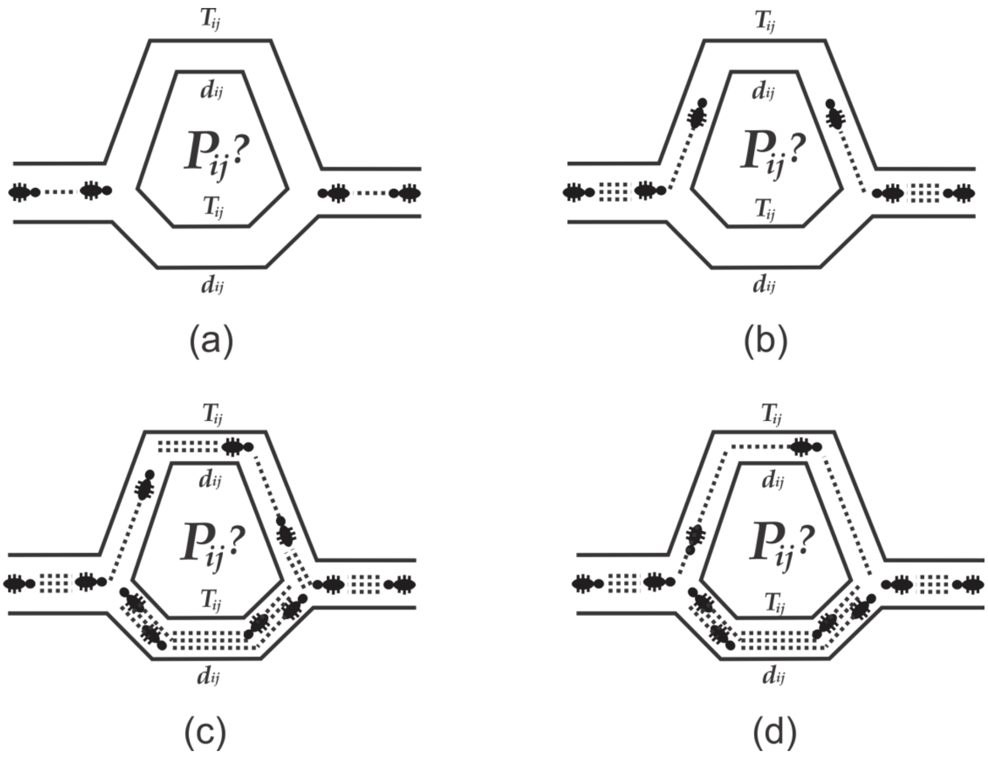


Figura 4.2 *a) Las hormigas siguen el camino del nido hasta la fuente de alimento. b) Encuentran una bifurcación, las hormigas deciden probabilísticamente qué camino tomar. c) Los caminos más cortos se ven favorecidos con la acumulación de feromona. d) La mayoría de las hormigas escogen los caminos con mayor cantidad de feromona. [Dorigo, 1996]*

El orden de asignación de cada uno de los clientes de la solución se lleva a cabo de forma probabilística, basado en la fórmula 13, tomando en cuenta la distancia entre los clientes y los rastros de feromona.

$$P_{ij}^{h} = \frac{[\tau_{ij}(t)]^{\alpha}\ [1/c_{ij}]^{\beta}}{\sum[\tau_{ij}(t)]^{\alpha}\ [1/c_{ij}]^{\beta}} \qquad (13)$$

Donde:

P_{ij}^{h} Representa la probabilidad de la hormiga *h* de ir del cliente *i* al cliente *j*.

τ_{ij} Monto de feromona en el arco (*i,j*)

$\frac{1}{c_{ij}}$ Visibilidad: Inversa de la distancia heurística en el arco (*i,j*)

α	Importancia relativa de los montos de feromona

β	Importancia de la distancia heurística

Para llevar un control sobre aquellos elementos que aún no han sido agregados a la solución, cada hormiga hace uso de una lista tabú, misma que se muestra en la figura 4.3, la cual tiene la función de trabajar como la memoria de la hormiga *h*.

índice →	0	1	2	3	4	5	6	7	8	9	10	11
	1	0	1	1	1	0	0	1	1	0	0	0

Figura 4.3 *Ejemplo de lista tabú de la hormiga* ***h.***

En la figura 4.3 el valor del *índice+1* corresponde al número de cliente a evaluar y el valor contenido en su posición indica con 0 que el cliente no ha sido asignado a una ruta y con 1 que ya fue calendarizado.

Cada que un cliente es asignado a una ruta por una hormiga *h,* ésta deposita feromona en el arco recorrido con base en la fórmula 14, lo que se conoce como actualización local de la feromona.

$$\Delta\tau_{ij}^{h} = \begin{cases} \dfrac{1}{f_{ij}^{h}} & Si\ el\ arco\ (i,j)\ es\ recorrido\ por\ la\ hormiga\ h \\ 0 & De\ lo\ contrario \end{cases} \quad (14)$$

Donde:

f_{ij}^{h}	Costo de recorrer el arco (*i,j*) por la hormiga *h.*

$\Delta\tau_{ij}^{h}$	Incremento aplicado al arco (*i,j*) por la hormiga *h*.

De modo que el monto total de feromona en el arco (*i,j*) corresponde a la suma del monto depositado por cada una de las hormigas que lo recorran el monto de feromona correspondiente a la fórmula 15.

$$\Delta\tau_{ij} = \sum_{h=1}^{Colonia} \Delta\tau_{ij}^{h} \quad (15)$$

Donde:

$\Delta\tau_{ij}$ Incremento total de feromona en el arco (i, j).

Al término de cada iteración del algoritmo, donde cada una de las hormigas de la colonia ha construido una solución, se aplica una evaporación global de la feromona mediante la fórmula 16, lo cual tiene como objetivo generar una especie de olvido en las hormigas, con lo que se permite continuar con proceso de exploración y explotación del espacio de soluciones.

$$\tau_{ij} = (1 - \rho)\tau_{ij}(t) + \Delta\tau_{ij} \quad (16)$$

Donde:

τ_{ij} Monto de feromona actual en el arco (i, j)

ρ Coeficiente de evaporación de la feromona

La hormiga que construyó la mejor solución de la iteración, deposita un extra de feromona, correspondiente a la fórmula 17 sobre los arcos que componen dicha solución, esto con la finalidad de hacer más deseables los arcos que proveen una mejor solución. El incremento corresponde a la inversa del costo de la solución construida.

$$\Delta\tau_{ij}{}^{best} = \frac{1}{f_{best}} \quad (17)$$

Donde:

f_{best} Costo total de la solución

El pseudocódigo general del algoritmo colonia de hormigas se presenta en la figura 4.4.

Las características específicas de los algoritmos AG y ACH serán detalladas en la siguiente sección, donde se explica la hibridación y adaptación de ambos algoritmos al método de solución propuesto para el VRPTW.

El algoritmo Genético Cooperativo secuencial denominado en este trabajo de investigación AGC-VRPTW, se encuentra compuesto por la hibridación de dos metaheurísticas que tienen como objetivo mejorar la exploración y explotación del

espacio de soluciones, aplicando búsquedas globales a través del AG y búsquedas locales con el ACH. La hibridación de estas dos técnicas trata de complementar

1. **INICIO**
2. Inicializa la feromona y el tamaño de la colonia
3. **Para** *h*=1 : *h<=COLONIA*

   hacer

   Calcula $\frac{1}{c_{ij}}$ $\forall$ $(i, j) \in N$ */*Inversa de la distancia heurística*/*

   Calcula $P_{ij}^{h} = \frac{[\tau_{ij}(t)]^{\alpha}\,[1/c_{ij}]^{\beta}}{\sum[\tau_{ij}(t)]^{\alpha}\,[1/c_{ij}]^{\beta}}$

   Actualiza lista tabú

   Actualización local $\Delta\tau_{ij}^{h} = \begin{cases} \frac{1}{f_h} & \text{Sí } (i,j) \text{ es recorrido por la hormiga } h \\ 0 & \text{De otro modo} \end{cases}$

   Hasta que *h* haya completado la solución

   Fin-para
4. **Para** cada solución *h*

   $\tau_{ij} = (1-\rho)\tau_{ij}(t) + \Delta\tau_{ij}$ */*Evaporación global*/*

   ${\Delta\tau_{ij}}^{best} = \frac{1}{f_{best}}$ */*Incremento de feromona en el mejor recorrido*/*

   Fin-para
5. **FIN**

Figura 4.4 *Pseudocódigo de un Algoritmo Colonia de Hormigas*

ambas metaheurísticas balanceando sus bondades y reduciendo sus desventajas, ya que los AGs son conocidos porque permiten realizar una buena exploración del espacio de soluciones pero llegan a sufrir de convergencia prematura. Por otro lado, el ACH es un algoritmo de búsqueda local que favorece la explotación de vecindades, por lo que son propensos a quedar atrapados en óptimos locales.

A diferencia de algunos algoritmos híbridos propuestos en la literatura, como es el caso de los presentados en [Tan et al., 2006; Yu et al., 2010] donde cada una de las heurísticas hibridadas corresponde a una parte independiente del algoritmo principal, el algoritmo propuesto en este trabajo de tesis integra el ACH con el AG como un solo método de solución denominado AGC-VRPTW, el cual se muestra en la figura 4.5. El algoritmo propuesto adapta el ACH para ser aplicado como un

operador de mutación, el cual por sus características es llamado en este trabajo como operador de mutación cooperativa.

El proceso desarrollado en el algoritmo secuencial AGC-VRPTW mostrado en la figura 4.5, corresponde a la analogía del ciclo evolutivo aplicado al VRPTW, donde una vez iniciado el proceso, se aplica el operador de selección que permitirá obtener los individuos candidatos para someterse al operador de cruzamiento, para posteriormente seleccionar de forma probabilística a cuales de los descendientes les será aplicado el operador de mutación cooperativa.

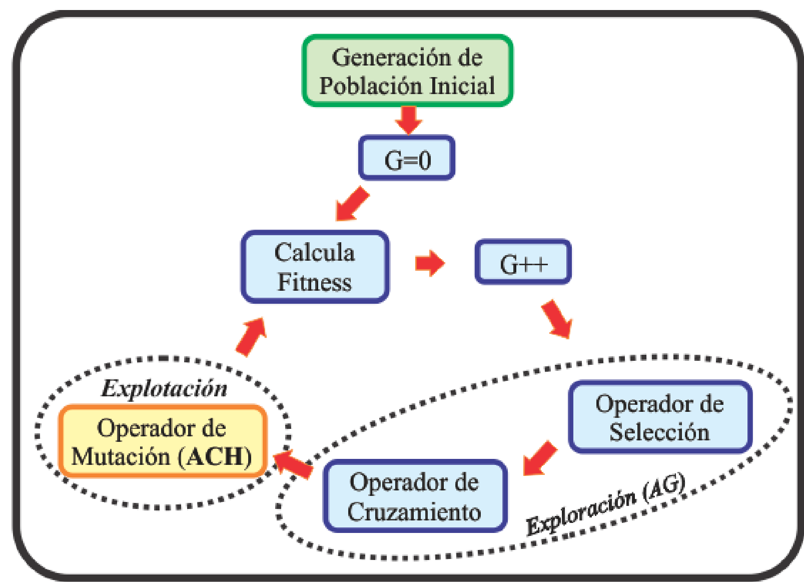


Figura 4.5 *Estructura general del AGC-VRPW propuesto en esta tesis*

Este proceso se repite hasta que se alcance el número máximo de generaciones. Cada una de las partes del algoritmo propuesto se explica a detalle en las siguientes secciones.

4.1.1 Generación de la Población Inicial

Los algoritmos basados en poblaciones, como es el caso de los AGs, en ocasiones llegan a sufrir de convergencia prematura debido a la pérdida de diversidad en la población durante el proceso evolutivo, lo que provoca convergencia prematura y por

ende quedar rápidamente atrapado en óptimos locales. Por lo que la diversidad en la generación de la población inicial así como la selección de los operadores genéticos a aplicar son parte fundamental para el buen desempeño de este tipo de algoritmos [Lozano et al., 2008; Gupta, Ghafir, 2012].

Para lograr una mayor diversidad en la población, se aplicó una modificación del algoritmo de agrupamiento k-means, así como la propuesta de una heurística de inserción híbrida, donde la aplicación aleatoria de 4 diferentes heurísticas de inserción favorecen la diversidad de la población generada. Lo explicado anteriormente se explica a continuación.

Algoritmo de Agrupamiento

El agrupamiento es un proceso de minería de datos que consiste en dividir un conjunto de elementos ubicados en un espacio euclidiano, en subgrupos que comparten características similares. La relación existente entre los elementos a agrupar se encuentra representada por una matriz de proximidad. En el caso del VRPTW dicha matriz corresponde a la distancia euclidiana existente entre los elementos.

En optimización combinatoria, los algoritmos de agrupamiento han sido frecuentemente utilizados para facilitar el tratamiento de problemas clasificados como NP-Completos basados en el paradigma *"Divide y Vencerás"*, donde el problema es dividido en *P* subproblemas que pueden ser tratados de forma independiente. Existen diversos algoritmos de agrupamiento que han sido aplicados a problemas de Ruteo Vehicular, pero uno de los más populares es el bien conocido *k-means* [Ochi et al., 1998]. Este método es frecuentemente aplicado debido a su simplicidad y flexibilidad, aunque como todos los métodos de agrupamiento, cuenta con ventajas y desventajas.

En este trabajo de investigación se realizaron algunas modificaciones al algoritmo *k-means*, mismas que se muestran en la figura 4.6, con la finalidad de

tomar sus fortalezas y reducir algunas de sus desventajas. Las modificaciones aplicadas se detallan en [Martínez, Cruz, 2011].

```
6.   INICIO
7.   Inicializa N, Veh, C
8.   Lee los datos de entrada
9.   Para i=0 : i<N          /* Genera la matriz de distancias*/
         Para j=0 : j<N
                 c_ij = sqrt((x_2 - x_1)^2 + (y_2 - y_1)^2)
         Fin-para
     Fin-para
10.  Inicializa Estructura Solución, vect_tabu, índice
11.  Repite                  /*Calcula la distancia hamming*/
         Mientras vect_tabu[centroide] == 1
             centroide = 1+(rand()%N)
         fin-mientras
         vect_tabu[centroide] = 1
         Mientras (dem_i + demAct_k) ≤ C   /* Verifica capacidad*/
             i = Cliente_cercano()
             Sí vect_tabu[i] == 0
                 vect_tabu[i] = 1
                 Solución[indice] = i
                 demAct_k += dem_i
                 indice++
             Fin-si
         Fin-mientras
         Sí (dem_i + demAct_k) > C     /* Si excede capacidad*/
             veh++
         fin-si
     Mientras  indice < N
12.  FIN
```

Figura 4.6 *Algoritmo de agrupamiento aplicado al VRPTW*

El proceso mostrado en el pseudocódigo de la figura 4.6 muestra las modificaciones aplicadas al algoritmo *k-means*. Una de las mejoras consiste en agregar la evaluación de las restricciones de capacidad como parte de los criterios para realizar el agrupamiento.

El resultado obtenido son agrupamientos que corresponden a una solución parcialmente factible para el VRPTW, donde cada agrupamiento es conocido como una *ruta*, la cual se encuentra asignada a un solo *vehículo*.

El algoritmo de agrupamiento consiste en un proceso iterativo, donde en cada iteración se selecciona el *centroide* de la ruta a contruir, el cual correspondiente a un cliente que no haya sido previamente calendarizado. Posteriomente, se van asignando a la ruta los clientes más cercanos al centroide que cumplan con las restricciones de capacidad del problema. En caso de que la capacidad del vehículo sea excedida, se habilita otro vehículo con la creación de una nueva ruta. Este proceso se repite hasta que todos los clientes hayan sido asignados a una ruta.

La solución obtenida se considera parcialmente factible debido a que para el agrupamiento se relajan las restricciones de tiempo. Esta solución parcial fungirá como entrada para la heurística híbrida de inserción, explicada a continuación, la cual se encarga de garantizar la factibilidad de las soluciones pertenecientes a la población inicial, así como de asegurar una buena diversidad entre las soluciones obtenidas.

Heurística Híbrida de Inserción para Clientes Aislados

Las heurísticas de inserción han demostrado ser métodos eficientes para el tratamiento de problemas de calendarización [Campbell, Savelsbergh, 2004] debido a que permiten la construcción de soluciones factibles mediante un proceso iterativo basado en la inserción de los elementos a calendarizar, tomando en cuenta las restricciones propias del problema tratado.

La mayoría de las heurísticas de inserción toman algunos parámetros específicos del problema o de la función objetivo a evaluar para llevar a cabo las inserciones de los elementos correspondientes. En el caso particular del VRPTW, se propone una heurística de inserción híbrida, compuesta de cuatro diferentes tipos de inserción, las cuales se aplican de forma aleatoria, como se muestra en la figura 4.7, lo que permite la generación de soluciones factibles a partir de una solución con factibilidad parcial.

En la figura 4.7 se observa que la entrada del algoritmo de inserción híbrido corresponde a una solución parcialmente factible generada por el algoritmo de agrupamiento descrito en la sección anterior. En primera instancia, dicha solución es

evaluada ruta por ruta con base a las restricciones de tiempo del problema, de modo que aquellos clientes que no cumplan con las restricciones serán separados de los agrupamientos originales. Posteriormente, se seleccionan uno po uno los clientes previamente aislados y se le aplica la heurística de inserción híbrida, la cual selecciona aleatoriamente un método de inserción de los cuatro implementados [Cruz et al., 2014; Cruz et al., 2014b].

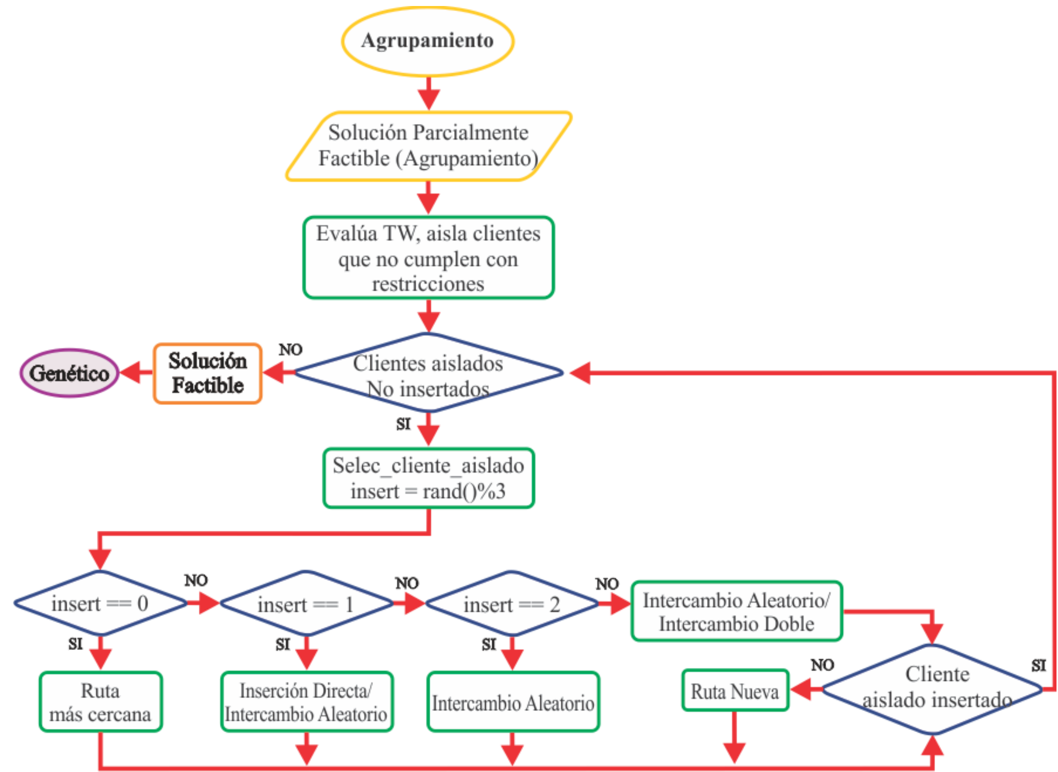


Figura 4.7 *Algoritmo de Inserción Híbrido aplicado a la generación de soluciones factibles al VRPTW*

Cada método de inserción corresponde a un proceso iterativo que durante un número definido de intentos, trata de insertar el cliente seleccionado en alguna de las rutas existentes. En caso de que la inserción no pueda ser realizada debido a las restricciones del problema, se abre una nueva ruta, donde el cliente es calendarizado. El proceso de inserción se repite hasta que todos los clientes hayan sido asignados a una ruta, lo que indica que se ha obtenido una solución factible al VRPTW.

Para la generación de una población inicial factible para el AGC-VRPTW, lo descrito anteriormente es repetido hasta que se cumpla con la cantidad de individuos definida para la población.

4.1.2 Algoritmo Genético

Los métodos heurísticos descritos en la sección 4.1 permiten obtener una población inicial factible de *m* individuos, la cual corresponde a la entrada para el AG implementado. Una solución factible corresponde a un individuo de la población, el cual se maneja en esta tesis por medio de un tipo de dato estructura (*struct),* mostrado en la figura 2.6.

Donde *N* corresponde a la cantidad de clientes a calendarizar y *K* al tamaño de la flota de vehículos, el cual es fijo y no debe de ser excedido. Por su parte, el campo *SOLUCION* es un arreglo que almacena la secuencia de los clientes (ruta) en el orden que serán atendidos por cada uno de los vehículos. Los campos *INICIO* y *FIN* identifican los puntos de inicio y término de cada una de las rutas asignadas a un vehículo, por medio del valor del índice del arreglo *SOLUCION* en el que se encuentran los clientes de la ruta. El campo correspondiente al arreglo *DEMANDA*, almacena la suma de las demandas de los clientes asignados a una ruta, por lo que su función es facilitar la evaluación de las restricciones de capacidad. Cada una de las posiciones del arreglo *COSTO_VEH* corresponde a un acumulador que va almacenando la suma del costo de atender a los clientes de una ruta específica. El campo *NUMVEH* indica el total de vehículos utilizados en la solución. Finalmente, el campo *FITNESS* indica el nivel de adaptación del individuo, el cual se encuentra representado por el costo total de la solución, es decir, la suma de los costos de cada ruta representada por cada vehículo utilizado para una solución.

La representación gráfica de un ejemplo del tipo de dato *struct* mostrado en la figura 2.6 puede ser visto gráficamente en la figura 4.8.

En el ejemplo mostrado en la figura 4.8 se observa claramente el manejo de los campos de la estructura con respecto a la distribución de clientes por ruta. En este

ejemplo, cada una de las rutas es identificada por un marco con color diferente, donde los campos *INICIO* y *FIN* corresponden a los límites de cada ruta asignada a un

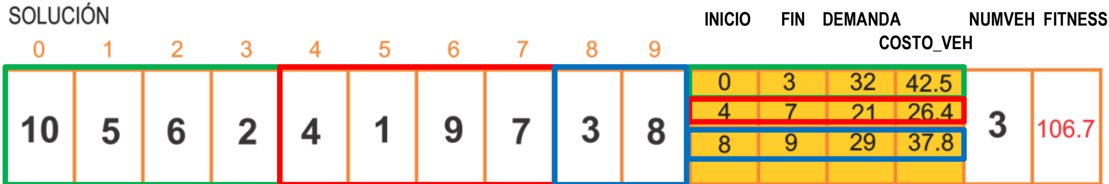


Figura 4.8 *Visualización gráfica de la representación simbólica del tipo de dato "struct" generado para el almacenamiento de soluciones factibles del VRPTW.*

vehículo y los campos *DEMANDA* y *COSTO_VEH* permiten la evaluación de las restricciones del problema. Finalmente, el campo *NUMVEH* muestra el total de rutas (vehículos) utilizadas en la solución y el campo *FITNESS* almacena de forma cuantitativa la calidad de la solución factible obtenida, misma que corresponde a la suma del costo individual de cada ruta recorrida.

De acuerdo a las características genéticas de un individuo, el cual es considerado como una solución factible, un *cromosoma* corresponde a una ruta asignada a un vehículo y un *gen* se encuentra representado por cada uno de los clientes calendarizados en la solución. Dichas características genéticas son tomadas en cuenta para la aplicación de los operadores genéticos, los cuales se describen a continuación.

4.1.2.1 Operador de Selección

La evolución se presenta a través de cambios aleatorios, los cuales son responsables del desarrollo de las especies, por lo que la selección juega un papel fundamental en la evolución, ya que es mediante este proceso que se eligen los individuos candidatos a reproducirse, así como aquellos que pasarán a la siguiente generación. Al igual que en la naturaleza, no solo los mejores individuos son seleccionados, por lo que la elección no se encuentra directamente en función del fitness de los individuos, sino que se hace uso de operadores probabilísticos y aleatorios que favorezcan la equidad en la selección de los individuos candidatos.

Para el algoritmo ACG-VRPTW, se probaron tres tipos de operadores de selección, el de ruleta, torneo elitista y torneo aleatorio, siendo este último el que permitió obtener mejores resultados. El proceso del operador de selección por torneo aleatorio aplicado al algoritmo propuesto, se muestra en la figura 4.9.

```
1. Recibe PoblacionIni, TamMuestra
2. Para i=0 : TamMuestra          /*Selecciona muestra*/
           ind1 = rand()%TamPop;
           Hacer  ind2 = rand()%TamPop
           Mientras ind1 == ind2
   fin-para
4. Select = rand()%1;
5. Sí Select == 0 entonces  IndM = ind1;
   de_lo_contrario  IndM = ind2;
   fin-sí
6. Para i=0 : TamMuestra          /*Selecciona muestra*/
           Hacer  ind1 = rand()%TamPop;
                  ind2 = rand()%TamPop;
           Mientras ind1 ==ind2 | | ind1==IndM | | ind2==IndM
   fin-para
7. Select = rand()%1;
8. Sí Select == 0 entonces  IndP = ind1;
   de_lo_contrario  IndP = ind2;
   fin-sí
9. /* Inicia Operador de Cruzamiento*/
```

Figura 4.9 *Selección de individuos mediante el operador de torneo aleatorio para el AGC-VRPTW.*

La selección por Torneo Aleatorio consiste en realizar una selección basada en la comparación directa de individuos pertenecientes a una muestra. Existen dos tipos de selección por torneo:

- ***Elitista:*** La muestra de *p* individuos es seleccionada al azar. De dicha muestra, el ganador del torneo será el individuo más apto de acuerdo al fitness (Aptitud), mismo que será cruzado y pasará sus genes a la siguiente generación.
- ***Aleatoria:*** La diferencia con la versión Elitista radica en la forma de seleccionar el ganador del torneo, donde en este caso se elige de forma

aleatoria sin influencia del fitness, por lo que cualquier individuo tiene la misma probabilidad de ser seleccionado.

De acuerdo a la literatura, el operador de selección por torneo ejerce presión sobre el algoritmo, misma que puede ser aumentada o disminuida variando la cantidad de individuos en la muestra que participa en el torneo, lo que repercute directamente en el desempeño del algoritmo.

Para este trabajo de tesis, el algoritmo genético cooperativo recibe la población inicial factible y define el tamaño del torneo, el cual de acuerdo a los resultados obtenidos en el análisis de sensibilidad (sección 5.1), fue fijado en 2. El segundo paso corresponde a la selección por torneo aleatorio de *TamMuestra* individuos para posteriormente elegir cada uno de los individuos que participarán en el cruzamiento.

4.1.2.2 Operador de Cruzamiento

Una vez que se han seleccionado los individuos, estos deben cooperar para generar nuevos individuos con la finalidad de mejorar la siguiente generación. Para esto son sometidos al proceso de cruzamiento, el cual toma los dos individuos seleccionados previamente y combina segmentos de su código genético de acuerdo al operador utilizado. El resultado es una descendencia artificial, donde los hijos cuentan con características de ambos individuos padres, similar a lo que sucede con el método de reproducción sexual de los seres vivos.

Tomando en cuenta las características del problema tratado, se desarrolló un operador de cruzamiento basado en cromosomas, donde cada cromosoma corresponde a una ruta, lo que permite mantener la factibilidad de los individuos en todo momento, evitando realizar ajustes post-cruzamiento. Para la implementación de dicho operador, lo primero a considerar fue la cantidad de cromosomas que se tomaría en cuenta durante el cruzamiento, por lo que se realizaron pruebas experimentales utilizando uno, dos, tres y un valor aleatorio entre 1 y la mitad de los cromosomas contenidos en el individuo, esto debido a que lo más diferente que un

individuo hijo puede ser con alguno de sus padres, es contener la mitad de sus cromosomas. De acuerdo a los resultados obtenidos correspondientes a la diversidad entre los individuos hijos, el cruzamiento aleatorio por cromosomas fue el que arrojó mejores resultados. El proceso realizado por el operador de cruzamiento utilizado en el AGC-VRPTW se explica por pasos a continuación:

1. Elegir de forma aleatoria la cantidad de cromosomas a mutar. Para esto se toma como límite inferior 1 y como límite superior el total de rutas del individuo seleccionado entre 2, tal y como se muestra en la fórmula 1.

```
cruza = rand()%(Veh/2)+1;
```
(18)

2. Seleccionar aleatoriamente *cruza* cromosomas en cada uno de los individuos a cruzar, como se observa en la figura 4.10.

cruza = 3

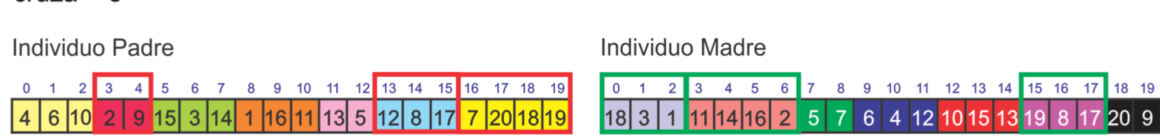


Figura 4.10 *Ejemplo de la elección aleatoria de los cromosomas a cruzar para cada uno de los individuos previamente seleccionados. Para este ejemplo se utilizó cruza=3.*

3. Los cromosomas seleccionados en cada individuo son copiados a dos nuevas estructuras que fungirán como descendientes o individuos hijo, lo cual puede observarse más claramente en la figura 4.11.

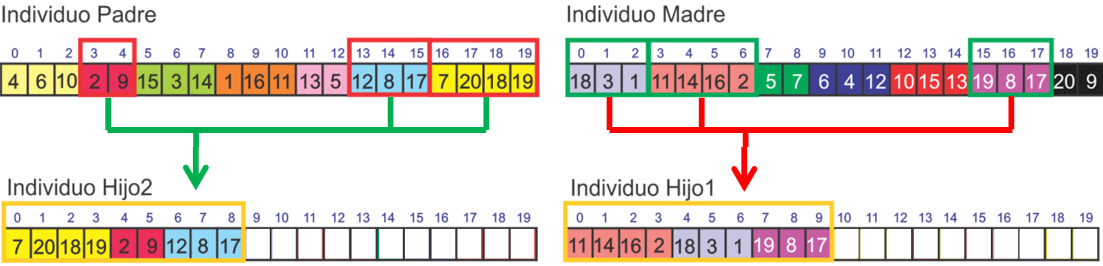


Figura 4.11*. Ejemplo de copiado de los cromosomas seleccionados de cada Individuo a cada uno de los descendientes.*

4. Para completar la estructura genética de los descendientes, cada uno de los individuos hijos realiza un barrido sobre el individuo padre / madre que falte de heredarle su material genético, copiando aquellos genes que no se encuentren dentro de los cromosomas que lo componen. Este proceso puede observarse en la figura 4.12.

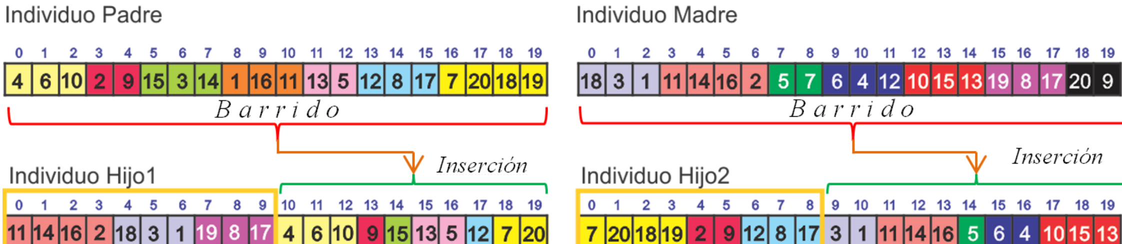


Figura 4.12. *Ejemplo del barrido realizado por cada uno de los descendientes para completar su estructura genética. Individuo hijo 1 recibió los cromosomas del Individuo Madre (Paso 3), realiza un barrido sobre los cromosomas del individuo Padre y copia aquellos genes que no se encuentren en su estructura.*

5. Una vez que cada descendiente ha copiado los genes faltantes en su estructura, se actualizan los límites de las rutas, las demandas, costos y fitness de cada uno de los individuos hijo y quedan en espera del operador de Mutación.

4.1.2.3 Algoritmo Colonia de Hormigas

El ACH tal y como se explica en la sección 4.1 es una metaheurística constructiva inspirada en el comportamiento inteligente y estructurado de las hormigas. En este método de optimización cada hormiga de la colonia explora las opciones para llegar a la fuente de alimento dejando un rastro de feromona que indique a los demás miembros de la colonia la calidad del recorrido. Debido a que existe un proceso natural de evaporación, al final del proceso los caminos más cortos son los que tendrán mayor concentración de feromona y por ende serán más deseables para las hormigas.

En este trabajo de investigación, el ACH no es utilizado para obtener soluciones al problema tratado, sino que es adaptado para fungir como operador de

mutación sobre los individuos candidatos. Este proceso, debido a su naturaleza y comportamiento se denomina *Mutación Cooperativa*.

Debido a las características del ACH, el proceso de mutación cooperativa se implementó con la finalidad de favorecer la explotación del espacio de soluciones de cada uno de los individuos a mutar, lo cual se detalla en la siguiente sección.

4.1.2.3.1 Operador de Mutación Cooperativa

En la evolución de los seres vivos, la mutación es un proceso poco frecuente, ya que corresponde a alteraciones de la estructura genética de un individuo, las cuales suceden de forma aleatoria y constituyen el mecanismo básico para obtener variedad genética.

1. **Función** Mutación($\alpha, \beta, \rho, \tau_0, \gamma, q, Individuo_{Muta}$)
2. Matriz_Feromona(τ_0)
3. $muta = 1 + (rand()\%NUMVEH/2)$ */*Cantidad de cromosomas a mutar*/*
4. Elige_Cromosomas(muta)
5. Copia_Cromosomas(Croms)
6. Colonia = Total de clientes en los cromosomas a mutar
7. **Repetir**

    Para $\forall(hormiga_h)$ con $h \in Colonia$ hacer */*Mutación de soluciones*/*

    Construye _Solucion($Solucion_h$)

    $$\Delta\tau_{ij}^{h} = \begin{cases} \frac{1}{f_h} & Sí\ el\ arco\ (i,j) fue\ recorrido \\ 0 & en\ caso\ contrario \end{cases}$$

    Fin-para

    $\tau_{ij} = (1-\rho)\tau_{ij}(t) + \Delta\tau_{ij}$ */*Evaporación global de feromona*/*

    Sí $hormiga_{mejor}$

    $$\Delta\tau_{ij} = \frac{1}{f_{best}}$$

    Fin-sí

    Hasta alcanzar criterio de paro γ
7. **Regresa** $Solucion_{mejor}$
8. **Fin-función**

Figura 4.13 *Algoritmo de la mutación cooperativa aplicada a los descendientes seleccionados, mismos que son producto del operador de cruzamiento.*

En la mutación cooperativa funciona de manera similar, permitiendo explotar el espacio de soluciones de un individuo, con la finalidad de obtener mejoras que generen diversidad y permitan evitar la convergencia prematura. Para aplicar este proceso al VRPTW, se implementó un ACH, el cual es aplicado a los descendientes del cruzamiento.

Para cada uno de los descendientes se genera un número aleatorio uniformemente distribuido entre 0 y 1, de modo que si el valor obtenido es menor o igual a la Probabilidad de mutación *ProbMuta* definida, el individuo hijo se somete al proceso de mutación, el cual se muestra en la figura 4.13 con la finalidad de obtener mejoras factibles mediante la aplicación del proceso de mutación cooperativa propuesta en esta tesis.

El proceso ilustrado en la figura 4.13 correspondiente al operador de mutación cooperativa es descrito por pasos a continuación:

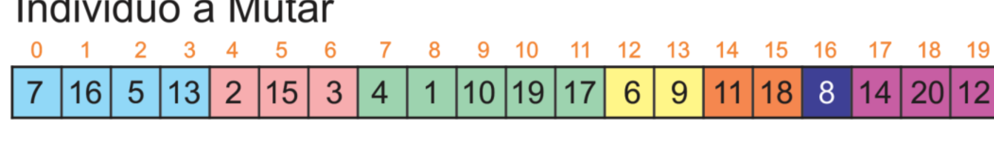
1. La función correspondiente recibe los parámetros (α, β, ρ, τ_0, γ, q, $Individuo_{Muta}$), donde α corresponde a la importancia relativa de los montos de feromona, β indica la importancia relativa de la distancia heurística, ρ es el coeficiente de evaporación de la feromona, τ_0 es el valor inicial de los montos de feromona, γ es el máximo de iteraciones aplicadas en la mutación cooperativa, q coeficiente que indica si se aplicará la fórmula correspondiente a exploración o explotación, $Individuo_{Muta}$ corresponde al individuo a mutar.

2. Inicializa la matriz de feromona con el monto mínimo τ_0 que contendrá cada una de las aristas del grafo $G = (V, A)$. La obtención del valor de τ_0 se explica en el capítulo 5.
3. Selecciona de forma aleatoria la cantidad de cromosomas a mutar, para lo cual se aplica la fórmula 19, donde los límites van de 1 a la mitad de la cantidad de rutas *NUMVEH* utilizadas por el individuo.

$$muta = 1 + (rand()\%NUMVEH/2) \tag{19}$$

4. Se seleccionan de forma aleatoria *muta* cromosomas diferentes, tal y como se muestra en el ejemplo de la figura 4.14. Es importante mencionar que en la figura 4.14 cada uno de los cromosomas del individuo se identifica por un color diferente. Para identificar cada uno de los cromosomas se utiliza un contador que se inicializa en 0.

$muta = 3$



$ruta = rand()\%NUMVEH$

Selección aleatoria de Rutas a mutar

3	5	1

Figura 4.14 *Ejemplo de la selección de cromosomas a mutar, para un valor de "muta = 3".*

5. Se hace uso de un arreglo auxiliar para almacena temporalmente los clientes contenidos en los cromosomas seleccionados (figura 4.14), como se muestra en la figura 4.15.

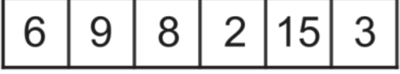


Figura 4.15*. Arreglo que almacena los clientes de las rutas seleccionadas aleatoriamente en la figura 4.14*

6. Define la cantidad de hormigas que conformarán la colonia con base a la sumatoria de clientes contenidos en los cromosomas a mutar. Por lo que el tamaño de la colonia será variable, dependiendo de los cromosomas elegidos para la mutación.

7. Inicia el proceso iterativo donde cada hormiga construye una solución respetando las restricciones del problema. Para esto cada hormiga A se posiciona en uno de los clientes a asignar y realiza el siguiente proceso:
    - Inicializa la lista tabú con los clientes involucrados en la mutación, utilizando 0 para indicar que el cliente no ha sido asignado y 1 de lo contrario.
    - Inicia la evaluación probabilística de los clientes candidatos, para lo cual se realizó una modificación a la fórmula 13 correspondiente al cambio en el parámetro de visibilidad, en donde en lugar de utilizar la distancia entre dos clientes, se utilizaron las restricciones de tiempo del problema, específicamente el cierre de la ventana de tiempo del cliente evaluado, de modo que aquellos con un cierre más próximo tienen mayor probabilidad de ser seleccionados.

      Otra variación en la evaluación probabilística corresponde a generar un número aleatorio uniformemente distribuido entre 0 y 1, de modo que si el valor obtenido es menor al valor de q, se aplica la fórmula 20 que favorece la exploración, de lo contrario favorece la explotación con la aplicación de la fórmula 21.

$$P_{ij}^{h} = [\tau_{ij}(t)]^{\alpha} \; [1/b_i]^{\beta} \tag{20}$$

$$P_{ij}^{h} = \frac{[\tau_{ij}(t)]^{\alpha} \; [1/b_i]^{\beta}}{\sum [\tau_{ij}(t)]^{\alpha} \; [1/b_i]^{\beta}} \tag{21}$$

    - Si el cliente cumple con las restricciones del problema, este se asigna a la estructura Solución de la hormiga h y se actualiza su estado en la lista tabú.
    - Aplica actualización local de la feromona con base en la fórmula 20.

$$\Delta\tau_{ij}^{h} = \begin{cases} \frac{1}{f_h} & Sí\ el\ arco\ (i,j) fue\ recorrido \\ 0 & en\ caso\ contrario \end{cases} \quad (20)$$

- Los pasos del punto 7 se repiten hasta que todos los clientes hayan sido asignados, con lo que se dice que la hormiga h ha construido un individuo factible parcial. Esto debido a que solo constituye una parte del individuo mutado.

El punto 7 se repite hasta que todas las hormigas hayan construido una solución, con lo que se dice ha terminado una iteración.

8. Aplica evaporación global de la feromona utilizando la fórmula 21, lo cual tiene la función de ejercer una especie de olvido en las hormigas, lo que evita una convergencia prematura.

$$\tau_{ij} = (1-\rho)\tau_{ij}(t) + \Delta\tau_{ij} \quad (21)$$

9. Se comparan las soluciones obtenidas por la colonia con base a la función objetivo. La hormiga que construya la mejor solución depositará un extra de feromona en los arcos correspondientes a la mejor solución, aplicando la fórmula 22.

$$\Delta\tau_{ij} = \frac{1}{f_{best}} \quad (22)$$

10. Al término del proceso de mutación, la mejor solución será insertada en el individuo original y actualizan los límites de las rutas, demanda, costos y fitness.

El proceso descrito anteriormente puede ser observado en la figura 4.16 utilizando una representación por un grafo disyuntivo, donde h corresponde a cada una de las hormigas de la Colonia, el rango [*a, b*] indica los límites de la ventana de tiempo del

cliente, τ_{ij} indica el monto actual de feromona en el arco [*i, j*] y P_{ij} indica la probabilidad de cada cliente a ser evaluado.

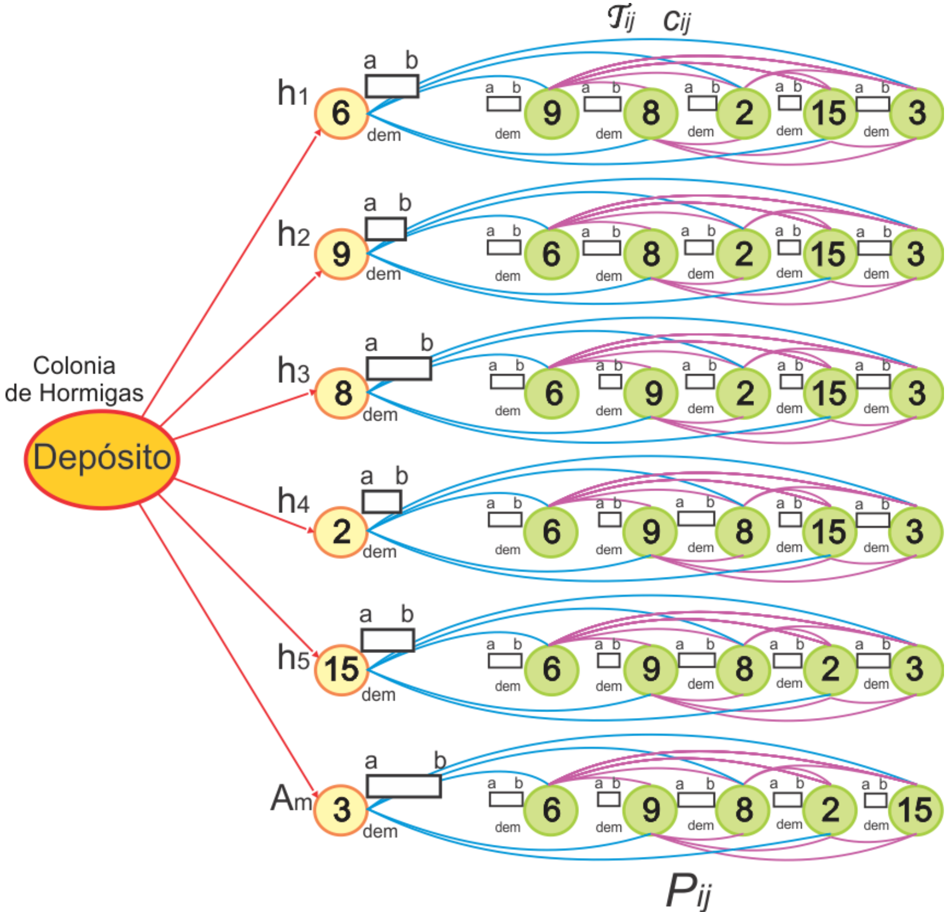


Figura 4.16. *Representación de la Mutación Cooperativa mediante un grafo disyuntivo*

4.1.3 Evaluación de la Distancia Hamming

La distancia Hamming (D_H) es una medida de diversidad basada en el principio de los códigos de Hamming [Hamming, 1950], la cual permite calcular de forma cuantitativa la diferencia existente entre dos secuencias $\{X_i\}$ y $\{X_j\}$ de la misma longitud, como se define en fórmula 23. Dicho valor corresponde al número de elementos que difieren entre ambas secuencias, en el caso de los algoritmos

genéticos, corresponde a la diferencia en el genotipo. Por lo que secuencias similares tendrá una D_H pequeña, contrario a lo que sucede con secuencias muy diferentes entre sí, donde el valor de D_H es cercano a la cantidad total de elementos N que componen la secuencia.

$$D_H = \sum_{n=1}^{N} (X_{in} \neq X_{jn}) \tag{23}$$

De acuerdo a la literatura, la diversidad de la población en un algoritmo genético es punto clave para tener una mejor exploración del espacio de soluciones, evitando la convergencia prematura y por ende quedar atrapado en óptimos locales. Por lo que la evaluación de la diversidad por un método como la distancia Hamming, permite conocer de forma cuantitativa el comportamiento de la diversidad en el algoritmo, por lo que facilita la toma de decisiones con respecto a los tipos de operadores genéticos a implementar, los cuales juegan un papel fundamental en la conservación de dicha diversidad hasta llegar a la convergencia del algoritmo.

Debido a las características de la estructura utilizada para el almacenamiento de los individuos factibles del VRPTW, fue necesario realizar algunas modificaciones al método original de la distancia Hamming para que pudiera ser aplicado a la estructura de los individuos del problema tratado. Para realizar un análisis de la población completa del algoritmo genético fue necesario desarrollar un algoritmo iterativo basado en la metodología propuesta en [Martínez et al., 2012], el cual se muestra en la figura 4.17. Cabe mencionar que dicho algoritmo se desarrolló con la única finalidad de evaluar la diversidad en poblaciones, por lo que es un proceso totalmente independiente del algoritmo AGC-VRPTW.

En el algoritmo 4.17 se lleva a cabo un proceso de comparación exhaustivo representado por la fórmula 13, donde cada uno de los individuos es comparado con el total de individuos que componen la población, lo cual permite, además de calcular la diversidad existente entre los individuos, identificar y eliminar aquellos individuos con información genotípica idéntica, ya que de mantenerlos en la población afectaría

la exploración del espacio de soluciones, debido a que estaría trabajando solo en una pequeña parte del espacio de soluciones, lo que puede causar una convergencia prematura del algoritmo, el proceso implementado para medir la diversidad en poblaciones se detalla en [Martínez et al., 2012].

```
1. INICIO
2. Inicializa ind, Población, aux, hamming, cont, DH
3. Lee_Población(Población)
4. Para ind=0 : ind<Población      /*Proceso iterativo que evalúa la DH*/
     Para aux=ind+1 : aux<Población
         hamming = cont = 0
         Para vhInd1=0 : vhInd1<CromosomasInd1
            Para i=IniCromosomaInd1 : i<=FinCromosomaInd1
                 Bandera=0
                 Para vhInd2=0 : vhInd2<CromosomasInd2
                    Sí i== IniCromosomaInd1
                         cont = 0
                    Fin-sí
                    Sí j<=FinCromosomaInd2 y j<=FinCromosomaInd1
                         Sí genes de ind == genes de aux
                             Bandera=1
                             vhInd2 = CromosomasInd2
                         fin-si
                    fin-si
                 fin-para
                 Sí Bandera == 0
                    hamming++
                 Fin-si
                 cont++
            fin-para
         fin-para
         DH = hamming
     fin-para
   fin-para
5. FIN
```

Figura 4.17. *Algoritmo para cálculo de la distancia hamming de una población de N individuos del VRPTW.*

4.2 *Modelo de Islas Distribuido con Comunicación Colectiva AGCD-VRPTW*

Un modelo distribuido se caracteriza por dividir la carga total de trabajo en múltiples procesos, los cuales serán ejecutados en diferentes procesadores llevando a cabo la distribución y comunicación mediante mensajes. En este trabajo de tesis, se presenta un modelo distribuido con comunicación colectiva, basado en el algoritmo secuencial descrito en la sección 4.1, mediante el uso de la interfaz de paso de mensajes MPI, el cual se denomina en esta tesis como AGC**D**-VRPTW (por sus siglas Algoritmo Genético Cooperativo Distribuido – Vehicle Routing Problem with Time Windows).

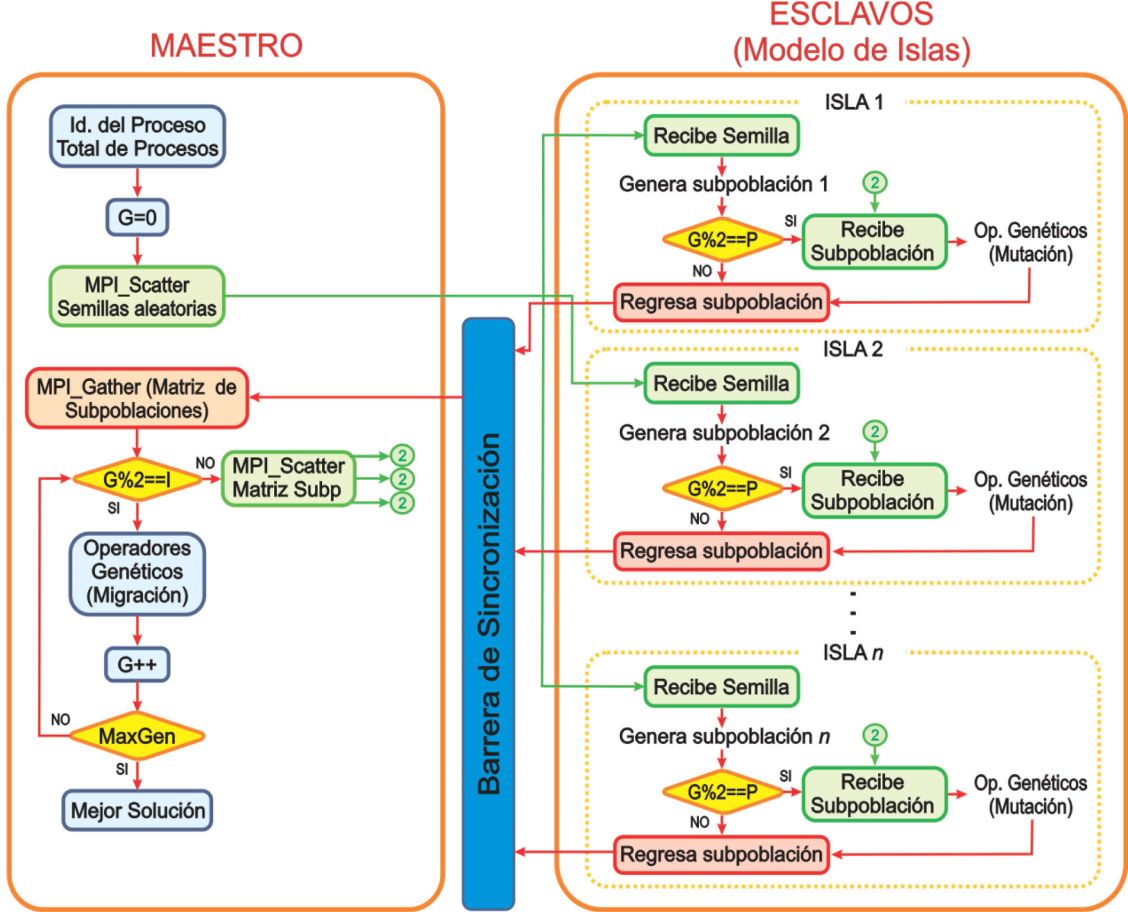


Figura 4.18 *Modelo de Islas Distribuido con Comunicación Colectiva y Migración implícita para el AGC-VRPTW*

El algoritmo AGC**D**-VRPTW a diferencia de la versión secuencial AGC-VRPTW, es un algoritmo que se ejecuta en múltiples núcleos de procesamiento sobre una Grid. El algoritmo lleva a cabo automáticamente la asignación de identificadores de rango para cada proceso, el cual le permitirá tener acceso al segmento de código que le corresponde ejecutar. Por lo que si *rango = 0*, se tendrá acceso al proceso maestro, de lo contrario al correspondiente a los procesos esclavos, como se muestra en la figura 4.18 y cuyas características se detallan en las secciones siguientes.

En esta versión del algoritmo propuesto se implementó un modelo de islas [Pettey et al., 1987] con comunicación colectiva y migración implícita que permita obtener soluciones al VRPTW de forma más eficiente, debido a que permite el manejo de recursos tanto locales como geográficamente dispersos, permitiendo el manejo de diferentes tipos de memoria, como se muestra en la tabla 4.2.

Tabla 4.2 *Recursos de memoria utilizados en el modelo distribuido AGC**D**-VRPTW*

Proceso	Memoria	Comunicación
Maestro	RAM (Local)	No requerida
Esclavos (Islas)	Distribuida	Colectiva

4.2.1 Construcción de la Población Inicial

Como se ha mencionado en secciones anteriores, las características de la población son puntos críticos para el buen desempeño de un algoritmo genético. En este caso se propone la construcción distribuida de la población por medio de un modelo de islas. Para ello, el algoritmo recibe en primera instancia el identificador correspondiente al número de proceso (*rango*), así como la cantidad total de procesos que se están ejecutando (*nprocs*). Posteriormente da inicio al proceso genético inicializando el contador *G* de generaciones.

Tomando en cuenta que el algoritmo distribuido AGC**D**-VRPTW será ejecutado en Grid, la construcción distribuida de la población requiere de tomar en cuenta dos puntos fundamentales:

1. ***Tamaño de la población:*** Esta característica es parte fundamental del comportamiento de un algoritmo genético, ya que el manejo de poblaciones pequeñas no asegura cubrir de forma adecuada el espacio de soluciones. Por otro lado, el manejo de soluciones muy grandes puede afectar considerablemente la eficiencia del algoritmo, debido a que se genera un exceso en el costo computacional.

   Debido a esto es necesario realizar un análisis de sensibilidad que permita obtener el tamaño adecuado de la población que favorezca el comportamiento del algoritmo.
2. ***Generación de números pseudoaleatorios:*** Un clúster es un conjunto de computadoras conectadas mediante una red y configuradas para trabajar como una sola máquina. Por su parte, una Grid es un conjunto de clústeres dispersos geográficamente y conectados mediante una red de alta velocidad. De modo que al ejecutar un algoritmo a nivel clúster o Grid es de suma importancia tomar en cuenta la generación de números pseudoaleatorios, ya que se basan en la hora del sistema para realizar dicho proceso. En un clúster, los nodos de procesamiento se encuentran bajo la misma configuración, de modo que todos se encontrarán sincronizados en fecha y hora, lo que implica que al ejecutar un generador de números pseudoaleatorios en todos los nodos, estos siempre arrojarán los mismos valores, lo cual para un algoritmo genético distribuido resultaría desastroso.

Para evitar la problemática descrita anteriormente fue necesario desarrollar de un método que permita la generación de semillas diferentes para cada uno de los nodos, lo cual se muestra en la figura 4.19, asegurando que el proceso evolutivo no esté centrado en una pequeña parte del espacio de soluciones.

```
1.  INICIO
2.  Inicializa Semillas[islas], SemIni, mult, div
3.  MPI_Init(&argc, &argv)
4.  MPI_Barrier(…)          /*Barrera de sincronización*/
5.  MPI_Comm_rank(…)        /* obtiene id del proceso actual */
6.  MPI_Comm_size(…)        /* obtiene el número total de procesos */
7.  Sí rango == 0
        SemIni = Genera_SemAleat()
        Para i=0 : i<islas
                Mult = Genera_Aleat()
                Div = Genera_Aleat()
                Semillas[i] = (SemIni * mult)/div
                Longitud_Semilla(Semillas[i])
        Fin-para
    Fin-si
8.  MPI_Scatter(…)          /* Envío de semillas a las islas*/
9.  Sí rango != 0
        srand(SemIni)       /* Uso de la semilla recibida por cada nodo */
    fin-si
10. FIN
```

Figura 4.19 *Algoritmo que genera semillas aleatorias y las envía a los nodos de procesamiento utilizando comunicación colectiva.*

El proceso realizado para la generación de semillas aleatorias mostrado en la figura 4.19, se describe a continuación:

1. Línea 2. Inicializa las variables a utilizar.
2. Línea 3. Inicializa el ambiente paralelo MPI.
3. Línea 4. Genera una barrera de sincronización para los procesos.
4. Línea 5 y 6. Obtiene el *id* del proceso y el número total de procesos a ejecutar.
5. Línea 7. El proceso maestro genera una semilla aleatoria la cual servirá de base para la generación de semillas aleatorias.

   En cada iteración del ciclo se generan dos números aleatorios *mult* y *div* que serán utilizados para modificar la semilla base.

   Con los valores generados anteriormente se aplica la fórmula 24 para obtener una modificación de la semilla inicial, la cual es almacenada en el arreglo *Semillas*.

$$Semillas[i] = (SemIni * mult)/div \quad (24)$$

6. Línea 8. Cada posición del arreglo *Semillas* se envía a un nodo de procesamiento utilizando comunicación colectiva con *MPI_Scatter()*
7. Línea 9. Sí corresponde a un nodo esclavo, toma la semilla recibida con la función *srand(SemIni).*

La generación de la población inicial se realiza de forma distribuida bajo un modelo de islas, donde cada nodo de procesamiento se considera como una isla. Cada isla recibe su respectiva semilla de números pseudoaleatorios e inicia la generación de una parte de la población total de individuos factibles conocida como subpoblación, basado en el método de agrupamiento y la heurística de inserción híbrida descritos en la sección 4.1.1. De modo que el tamaño total de la población corresponde al tamaño de la subpoblación por el número de procesos a ser ejecutados. Para definir el tamaño adecuado de la subpoblación fue necesario realizar un análisis de sensibilidad, mismo que se explica en el capítulo 5.

4.2.2 Modelo de Islas con Migración Implícita

El modelo de islas es una implementación paralela propuesta por [Pettey et al., 1987] que ha mostrado ser una de las más populares y eficientes al ser aplicada a algoritmos genéticos [Tanese, 1989; Whitley, Starkweather, 1990; Gorges, 1991; Belding, 1995]. Al igual que sucede con varios de los métodos de optimización, el modelo de islas se encuentra inspirado en la forma en que viven y evolucionan las especies. Al realizar una analogía con los seres humanos, se puede decir que una isla corresponde a una subpoblación ubicada en un espacio geográfico aislado, la cual evoluciona de manera independiente. Pero existe un fenómeno que sucede de forma esporádica y afecta positivamente a los individuos de la subpoblación. La migración de individuos pertenecientes a otra región, al establecerse y relacionarse con los individuos de la subpoblación, favorecen la diversidad genética [Whitley, 1993; Whitley et al., 1998]. La migración permite a las subpoblaciones de las islas compartir material genético,

favoreciendo la competencia entre los individuos y manteniendo una mayor diversidad genética entre las islas. De manera general la aplicación del modelo de islas paralelo ha demostrado tener mejor desempeño que un algoritmo genético secuencial basado en una sola población [Whitley, Starweather, 1990] [Mühlenbein, 1991].

El esquema distribuido presentado en la figura 4.20 se enfoca en las bondades del modelo de islas para ser aplicado al AGC-VRPTW y se explica a continuación.

```
1.  INICIO
2.  Inicializa Semillas[islas], rango, nprocs, G=0, Matriz_Subp[Islas][Pob],
    SubP[Pob]
3.  MPI_Init(&argc, &argv)
4.  MPI_Barrier(…)          /*Barrera de sincronización*/
5.  MPI_Comm_rank(…)        /* obtiene id del proceso actual */
6.  MPI_Comm_size(…)        /* obtiene el número total de procesos */
7.  Sí rango == 0
        Genera_Semillas(Semillas)
    Fin-si
8.  MPI_Scatter(…)                          /* Envío de semillas a las islas*/
9.  Genera_Tipo_Dato()
10. Repetir
        Sí G%2 != 0 && rango == 0           /* Migración implícita*/
            Ind1 = Selección(Matriz_Subp)
            Ind2 = Selección(Matriz_Subp)
            Cruzamiento (Ind1, Ind2)
            Muta_Cooperativa()
        de_lo_contrario                     /* Evolución de subpoblación*/
            MPI_Scatter(…)  /* Envío de subpoblaciones con Comunicación Colectiva*/
            Ind1 = Selección(Subp)
            Ind2 = Selección(Subp)
            Cruzamiento (Ind1, Ind2)
            Muta_Cooperativa()
            MPI_Gather(…)/* Recepción de subpoblaciones con Comunicación Colectiva*/
        fin-si
        G++
    Mientras G<MaxGen
11. Mejor_Individuo()
12. FIN
```

Figura 4.20. *Algoritmo Distribuido AGC**D**-VRPTW con Comunicación Colectiva*

1. Línea 2. Inicializa las estructuras y variables a utilizar.
2. Línea 3. Inicializa el ambiente paralelo MPI
3. Línea 4. Genera una barrera de sincronización
4. Línea 5 y 6. Obtiene el *id* del proceso y el total de procesos a ejecutar
5. Bloque 7. Genera las semillas aleatorias a utilizar en cada una de las islas
6. Línea 8. Envía las semillas a cada uno de los nodos utilizando comunicación colectiva con *MPI_Scatter*.
7. Línea 9. Genera tipo de dato estructura para el envío y recepción de las subpoblaciones por medio de comunicación colectiva *MPI_Scatter* y *MPI_Gather*.
8. Bloque 10. Proceso iterativo que aplica el modelo de islas con migración implícita y comunicación colectiva. La migración implementada en el algoritmo AGC**D**-VRPTW implica dividir las generaciones en pares e impares.
   - Las *generaciones pares* realizan un mapeo y división de la población a una matriz de subpoblaciones, a partir de la cual se envían a cada isla con MPI_Scatter.

     Cada isla aplica los operadores genéticos sobre la subpoblación recibida

     Regresa la subpoblación modificada al proceso maestro.
   - Las *generaciones impares* en la primera generación construyen la subpoblación de individuos factibles utilizando la semilla de números aleatorios recibida. Además de llevar a cabo la migración implícita, ya que el proceso maestro recibe todas las subpoblaciones en una sola población pancmítica y aplica los operadores genéticos sobre el total de la población. Los valores de los parámetros de control correspondientes a los operadores genéticos aplicados en este punto, varían con respecto a aquellos que se aplican en cada isla, y fueron obtenidos mediante un proceso de sintonización, descrito en la sección 5.2. Este proceso permite que lleve a cabo un intercambio de material genético entre individuos de diferentes subpoblaciones, reduciendo la latencia de

migrar individuos entre las poblaciones y favoreciendo la diversidad en la población.

Este proceso se repite hasta que se cumpla el máximo de generaciones requeridas para lograr la convergencia del algoritmo.

9. Línea 11. Devuelve el mejor individuo con base en la función objetivo.

4.2.2.1 Comunicación de Procesos

La comunicación entre procesos para el algoritmo AGC**D**-VRPTW se lleva a cabo de forma distribuida por medio de la interfaz de paso de mensajes MPI, de modo que un envío puede definirse formalmente como el conjunto de procesos $P = \{proceso_0, proceso_1, \ldots proceso_{id}\}$, donde cada proceso es autónomo e independiente.
Cada uno de los procesos $\{proceso_0, proceso_1, \ldots proceso_{id}\}$ se ejecuta en un núcleo de procesamiento de un clúster, generando una relación (proceso ⟶ núcleo). El algoritmo distribuido AGC**D**-VRPTW utiliza comunicación colectiva entre los procesos y el proceso maestro de forma bidireccional.

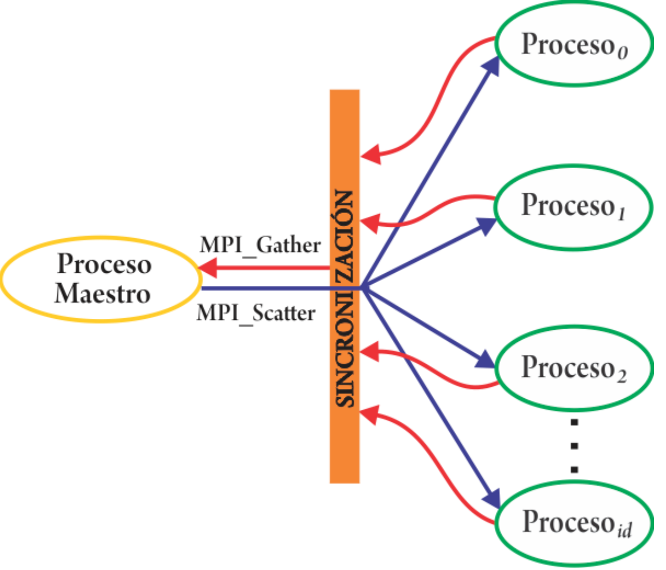


Figura 4.21 *Comunicación de procesos del algoritmo AGC**D**-VRPTW realizada de forma colectiva en ambos sentidos, proceso →maestro y maestro ←proceso.*

En primera instancia, el proceso maestro envía las semillas aleatorias a los procesos en una relación de uno a muchos. El siguiente envío se realiza de los procesos al proceso maestro con una relación de muchos a uno. A la siguiente generación el proceso maestro devuelve la información a los procesos en una relación de uno a muchos, tal como se muestra en la figura 4.21.

El manejo de la información emplea un esquema de paso de mensajes con comunicación colectiva en donde interviene todos los procesos de un comunicador, lo cual se explica a continuación:

- Un *comunicador* se encarga de definir grupos de procesos entre los que puede existir comunicación, asignándoles un *id* del rango.
- Todos los procesos involucrados en el modelo propuesto deben de encontrarse dentro del mismo comunicador para poder establecer la comunicación entre ellos y el proceso maestro.
- La comunicación colectiva en el algoritmo AGCD-VRPTW se lleva a cabo mediante dos funciones: *MPI_Scatter* y *MPI_Gather*.
    - ***MPI_Scatter.*** Esquema de distribución utilizado cuando la información se encuentran en un determinado proceso y el cálculo correspondiente a cada proceso depende de una porción de dicha información. La función *MPI_Scatter* permite tomar toda la información y mediante un solo envío, distribuye la información requerida por cada proceso, como se muestra en la figura 4.22.

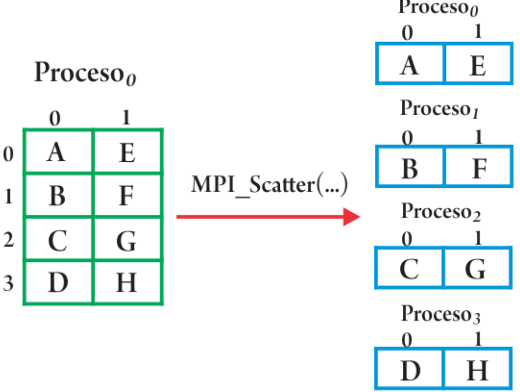


Figura 4.22 *Comunicación colectiva con MPI_Scatter. Ejemplo del envío de la fila de una matriz a cada proceso.*

- ***MPI_Gather.*** Este esquema es implementado cuando los datos requeridos por un proceso se encuentran distribuidos en los procesos del comunicador, por lo que es necesario recopilarlos. *MPI_Gather* realiza la función inversa a *MPI_Scatter*, por lo que se encarga de juntar la información contenida por cada uno de los procesos y enviarla como un paquete al proceso que la requiere, lo cual se puede observar de forma gráfica en la figura 4.23.

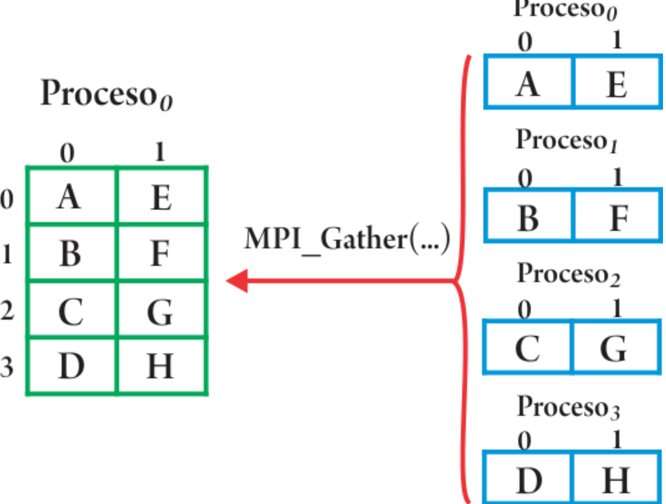


Figura 4.23 *Comunicación colectiva mediante MPI_Gather. Ejemplo de la recolección de información de cada proceso, para ser enviada al proceso$_0$*

Las ventajas de aplicar comunicación colectiva con respecto a la comunicación bloqueante para un escenario similar al mostrado en las figuras 4.22 y 4.23, es que en el caso de la comunicación bloqueante [Pacheco, 1997] es necesario realizar un envío y recepción independiente por proceso, el cual queda bloqueado indefinidamente hasta recibir el mensaje. A diferencia de la comunicación colectiva, donde se genera un solo paquete de datos que distribuye la información requerida a cada proceso. Lo que permite reducir la latencia generada por las comunicaciones y explotar el ancho de banda disponible.

4.2.2.2 Distribución de Procesos

La distribución de los procesos para el algoritmo AGCD-VRPTW se lleva a cabo de manera uniforme sobre los núcleos de procesamiento del clúster. El máximo de procesos a ejecutar sin sobrecarga se encuentra definido por la fórmula 25.

$$P_{max} = \sum_{cs}^{CS} \sum_{nd}^{ND} \sum_{nc}^{NC} t_{c,nd,nc} \tag{25}$$

Donde el máximo número de procesos a ejecutar sin sobrecarga P_{max} se encuentra definido por la suma de los núcleos de procesamiento *nc* disponibles en cada nodo *nd* de cada clúster *cs* utilizado. Para cada proceso a ejecutar, se asigna un identificador único que va de *0* a id_{max}, mediante el cual es posible establecer una comunicación entre los procesos, por lo que el proceso de asignación de identificadores se lleva a cabo mediante una relación uno a uno entre (*proceso→núcleo*).

En caso de tener en ejecución mayor número de procesos que de núcleos disponibles, se dice que se trabaja con sobrecarga [Martin et al., 1997], de modo cada uno de los núcleos de procesamiento recibe más de un proceso para su ejecución, por lo que el tiempo de ejecución del algoritmo se verá afectado de acuerdo al tipo de problema y la diseño del algoritmo.

En esta tesis, el algoritmo AGC**D**-VRPTW es un algoritmo distribuido sobre un clúster de alto rendimiento, donde cada proceso corresponde a una isla del modelo descrito en la sección 4.2. Cada isla trabaja con una subpoblación de individuos para efectuar el proceso de mutación cooperativa. Por lo que la distribución se lleva a cabo de manera uniforme, lo que aunado al uso de comunicación colectiva, reduce los tiempos de espera y los errores de comunicación.

La ubicación de los procesos queda asignada de acuerdo al punto de ejecución, ya que éste siempre corresponderá al proceso maestro, el cual no necesariamente debe ser el nodo maestro del clúster, mientras que los demás procesos

corresponderán a las islas del modelo propuesto, los cuales tendrán la característica de poseer un $id \neq 0$.

Comunicación Colectiva con Tipos de Dato Estructura (*struct*)

La distribución de procesos mediante paso de mensajes, independientemente del tipo de comunicación a ser aplicada, requiere del manejo de tipos de datos primitivos. En el caso del AGCD-VRPTW, la información enviada a los procesos corresponde a estructuras de datos que involucran diferentes tipos de datos, como se muestra en la figura 4.9, por lo que no pueden ser enviadas de forma directa.

Las estructuras de datos contienen características que les impiden clasificarlas dentro de un tipo de dato primitivo, por lo que es necesario generar un nuevo tipo de dato. MPI proporciona funciones que facilitan dicho proceso, lo cual puede ser consultado en el Apéndice A.

Para establecer el envío de las subpoblaciones (conjunto de soluciones almacenadas en elementos de tipo estructura) a cada proceso, fue necesario generar un tipo de dato estructura mediante la función *MPI_Type_struct*, a partir del cual es posible establecer el tipo de dato para el envío, lo cual se describe a continuación.

1. Generar el nuevo tipo de dato a partir de las características de la estructura de datos utilizada.

2. El algoritmo AGCD-VRPTW utiliza comunicación colectiva, de modo que se requiere preparar la población de individuos para ser enviada mediante la función *MPI_Scatter*. Para este propósito, la estructura de la población fue mapeada a una matriz de estructuras, lo cual se muestra en la figura 4.24 con las características requeridas para ser enviada por medio de MPI_Scatter, cuyo funcionamiento se observa en la figura 4.22, donde cada elemento de la matriz tiene la estructura descrita en la figura 4.8.

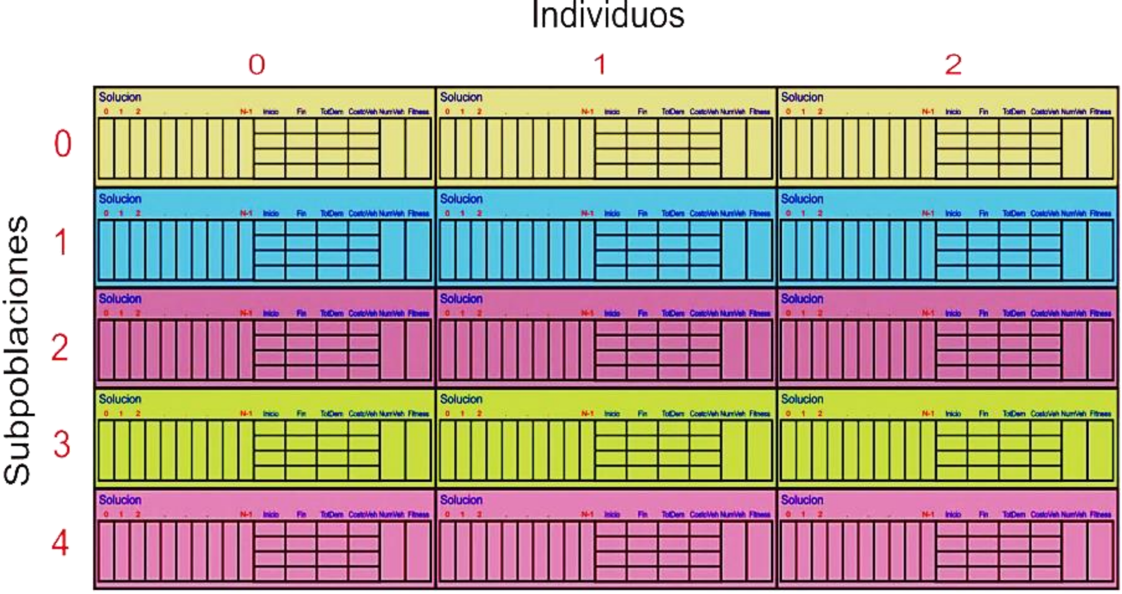


Figura 4.24 *Matriz de estructuras. Cada fila corresponde a la subppoblación que recibe cada proceso y las columnas contienen los individuos de la subpoblación.*

3. La función *MPI_Scatter* toma la población completa distribuyendo una fila de la matriz a cada uno de los procesos ejecutados. Para realizar el proceso correspondiente, la información recibida es almacenada en un arreglo de estructuras.
4. Una vez finalizado el proceso, el arreglo de estructuras correspondiente a cada subpoblación de individuos, es recolectado por la función *MPI_Gather* en la matriz inicial.
5. Los puntos 3 y 4 son repetidos hasta que se cumpla el total de generaciones del algoritmo genético.

4.2.2.3 Sincronización de Procesos

De acuerdo a la distribución de procesos explicada en la sección 4.2.2.2 donde se utiliza el esquema de comunicación descrito en la sección 4.2.2.1, la sincronización es aplicada sobre el conjunto de procesos $P = \{proceso_0, proceso_1, \ldots, proceso_{id}\}$ que se distribuyen en un clúster, asignando un proceso por núcleo de procesamiento.

Cabe mencionar que aun cuando la infraestructura que se está utilizando es homogénea, el tiempo de término de cada proceso llega a ser variable, ya que aunque existe balanceo de carga en la cantidad de información recibida por los procesos y se aplican los mismos métodos de optimización, los procesos aleatorios involucrados hacen que los tiempos de ejecución varíen en mínimas cantidades de tiempo, por lo que los procesos que terminen primero deben esperar hasta que todos los demás hayan finalizado para poder continuar con el proceso de optimización del algoritmo propuesto. La sincronización consiste en definir un punto en donde los procesos que finalicen se mantengan detenidos hasta que todos los procesos del comunicador hayan terminado. Para realizar este proceso, MPI cuenta con la función colectiva *MPI_Barrier*, la cual establece una barrera de sincronización entre los procesos ejecutados. En la figura 4.25 se muestra un ejemplo del funcionamiento de una barrera de sincronización con *MPI_Barrier*.

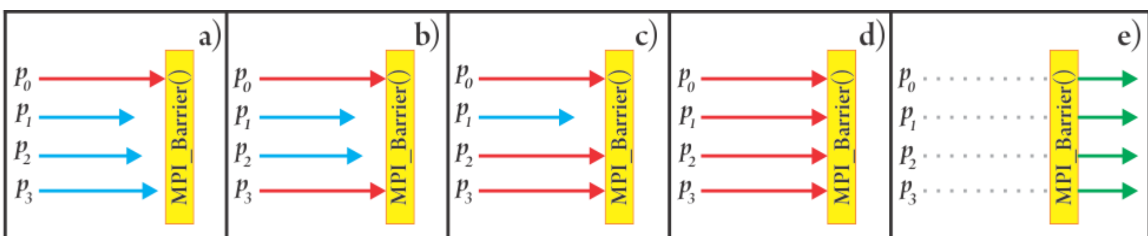


Figura 4.25 *Etapas de sincronización con MPI_Barrier para un ejemplo de 5 procesos. Las imágenes a), b), c), d) muestran la espera de los procesos terminados hasta que todos los procesos del comunicador alcanzan la barrera, e) indica el punto en el la barrera ha sido alcanzada por todos los hilos y es posible continuar con la ejecución del algoritmo.*

Debido a que el algoritmo AGC**D**-VRPTW establece una comunicación colectiva, la barrera de sincronización se activa al momento de aplicar *MPI_Gather* a las subpoblaciones de cada proceso, ya que se requiere que todos hayan finalizado, antes de recopilarlas en la matriz que recibirá el proceso maestro. En el caso de la distribución con *MPI_Scatter*, la sincronización no es necesaria, ya que los datos se encuentran centralizados en un solo proceso, lo cual se puede observar en las figuras 4.18 y 4.21.

4.2.2.4 Cooperación Implícita de Procesos

Como se explica en la sección 4.2.2, el modelo aplicado al algoritmo propuesto corresponde a un modelo de islas, el cual es bien sabido requiere de la implementación de un proceso de migración para evitar la convergencia prematura por pérdida de diversidad entre los individuos de la población, la cual debe aplicarse cuidadosamente para evitar la divergencia del algoritmo o quedar atrapado en óptimos locales [Skolicki, De Jong, 2005].

En el caso del modelo de islas implementado en el algoritmo AGC**D**-VRPTW, se propone una técnica de migración que evita la latencia de las comunicaciones de dicho proceso. El proceso de migración propuesto, corresponde a que cada generación impar el proceso maestro reciba todas las subpoblaciones de las islas y aplique los operadores genéticos correspondientes con un porcentaje de cruzamiento y mutación menores a los establecidos para cada una de las islas. Esto permite generar el efecto de una pequeña migración que no afecte de manera negativa la evolución de los individuos de las subpoblaciones. Por el contrario, favorece la evolución de las mismas hasta alcanzar la convergencia.

Los valores establecidos para los porcentajes de cruzamiento y mutación durante la migración se obtuvieron mediante la aplicación de un análisis de sensibilidad que permitió obtener los valores que favorecen el desempeño del algoritmo. Los resultados correspondientes a la sintonización de dichos parámetros se muestran en el capítulo 6.

4.3 Algoritmo Genético Cooperativo Paralelo Distribuido AGCP-VRPTW

En los últimos años, una de las áreas que más se ha visto favorecida gracias a los continuos avances y mejoras de la computación y la electrónica es el cómputo paralelo, el cual ha tomado gran auge gracias a las ventajas que representa el uso de tarjetas GPU para cálculo científico.

Las tarjetas GPU han incrementado su capacidad de cómputo rápidamente, lo que aunado al desarrollo de librerías que facilitan su programación, como es el caso de OpenCL y CUDA (por sus siglas en inglés *Compute Unified Data Architecture*), han permitido que esta tecnología sea cada vez más frecuentemente aplicada al tratamiento de problemas complejos, como una alternativa eficiente a los procesadores tradicionales [Govindaraju et al, 2006; Owens et al., 2008]. La GPU es un procesador multihilo que permite ejecutar miles de hilos de forma concurrente, por lo que trabaja como un coprocesador de la CPU [Avinash et al., 2013] que favorece la aceleración de aplicaciones paralelas. La mejora en la eficiencia viene dada por la identificación de los procesos que requieren mayor esfuerzo computacional, los cuales son paralelizados para ser ejecutados en la GPU.

Con base en lo anterior, la principal aportación de este trabajo de investigación corresponde a la implementación de programación híbrida CPU-GPU aplicada al AGC**D**-VRPTW, teniendo como objetivo obtener una mejora en la eficiencia. La programación híbrida utilizada en esta tesis permite conjuntar la programación distribuida por medio de paso de mensajes con MPI y la programación paralela mediante hilos con CUDA. Para esto, se realizó un análisis de la versión distribuida del algoritmo AGC-VRPTW presentado en la sección 4.2, llegando a la conclusión de que el proceso que requiere mayor cantidad de tiempo y esfuerzo computacional es la mutación cooperativa, la cual se lleva a cabo mediante la aplicación de un ACH, por lo que cuenta con un alto grado de paralelismo implícito. Esto permitió que este procedimiento fuera paralelizado con CUDA con la finalidad de favorecer la eficiencia del algoritmo.

En esta sección se describe el modelo paralelo-distribuido propuesto para el AGC**D**-VRPTW. En la subsección 4.3.1 se da una introducción a conceptos básicos de programación paralela en CUDA. En la subsección 4.3.3 se muestran las características de la tarjeta gráfica utilizada en la infraestructura de la Grid Morelos (sección 6.1), así como las características del modelo paralelo propuesto para el AGC**D**-VRPTW, descritas en la sección 4.3.4 y 4.3.5.

4.3.1 Conceptos Básicos de Programación en CUDA

CUDA (*Compute Unified Device Architecture* por sus siglas en inglés) es un paradigma de programación desarrollado por NVidia[1]. CUDA facilita el uso de unidades de procesamiento gráficas para mejorar el rendimiento de aplicaciones paralelas por medio de librerías que funcionan como extensiones para el desarrollo de cómputo paralelo en lenguajes de programación como Fortran y C. Lo anterior permite tomar ventaja de las características de las tarjetas gráficas de NVidia para paralelizar aplicaciones en múltiples núcleos de procesamiento mediante la ejecución concurrente de miles de hilos. Además de permitir conjuntar el procesamiento paralelo en múltiples tarjetas gráficas [Jacobsen et al., 2010], no solo en una sola máquina, también sobre plataformas de cómputo de alto rendimiento como clústeres y Grids, a través de un co-procesamiento que involucra tanto la CPU como la GPU.

La paralelización de una aplicación involucra el análisis de un algoritmo con respecto a la arquitectura y paradigma de programación a utilizar. Para lo cual, en el caso de CUDA es necesario conocer algunos conceptos fundamentales que servirán de base para el diseño del algoritmo, los cuales se explican a continuación.

El hablar de un proceso de *paralelización* se enfoca en el cómputo concurrente de instrucciones, donde un problema es dividido en múltiples subproblemas que serán ejecutados en varios procesadores simultáneamente, con lo que pueden ser utilizados tres tipos básicos de arquitectura paralela [Hord, 1990] para reducir el tiempo de cómputo requerido para obtener una solución. La primera arquitectura, *SIMD* (*Single Instruction Multiple Data* por sus siglas en inglés) se centra en el enfoque de que todos los procesadores recibirán la misma instrucción pero la ejecutarán sobre segmentos de datos distintos. Por su parte, la arquitectura *MISD (Multiple Instruction Single Data* por sus siglas en inglés*)* corresponde a una arquitectura teórica donde se envían múltiples instrucciones que serán ejecutadas sobre los mismos datos. Finalmente, la que se podría decir que es una de las

[1] *NVidia. Procesamiento Paralelo CUDA. http://www.nvidia.es/object/cuda-parallel-computing-es.html. Consultada en Octubre 2014.*

arquitecturas más utilizadas es la *MIMD (Multiple Instruction Multiple Data),* donde cada procesador ejecuta su propia instrucción para ser aplicada a conjuntos de datos distintos pero pertenecientes a un mismo problema.

Basado en las arquitecturas clásicas del paralelismo explicadas anteriormente, NVidia acuña su nueva arquitectura *SIMT (Single Instruction, Multiple Threads*) [Cook, 2013; Cheng et al., 2014], el cual se basa en lanzar un conjunto de hilos que ejecutan la misma instrucción sobre diferentes datos de forma concurrente, por lo que se considera que toma características de la arquitectura SIMD en conjunto con la programación vectorial y el manejo de multihilos. Por su parte, la *programación vectorial* muestra todos los datos almacenados como un mismo flujo de datos almacenado en memoria de forma continua (vector), por lo que el manejo de grandes cantidades de información se hace más eficiente debido a que el tiempo de cómputo se reduce considerablemente. Una de las ventajas de la programación vectorial con respecto a la escalar radica en los tiempos de acceso a los datos, los cuales se ven disminuidos debido al tipo de almacenamiento utilizado para los vectores de datos, para lo cual se requiere de procesadores especializados, como los GPUs. Los procesadores gráficos tienen la función de trabajar como aceleradores de aplicaciones paralelas [Tarditi et al., 2006; Silberstein, 2014], que quitan carga a los procesadores CPU, ejecutando aquellos procesos que requieren mayor esfuerzo computacional.

Para obtener el mejor rendimiento posible para un algoritmo paralelo, es importante realizar un análisis que permita explotar el paralelismo inherente, considerando la *granularidad*, basado en el paralelismo de control y el paralelismo de datos [Roosta, 2000; Nickolls et al., 2008]. El *paralelismo de control* también conocido como *paralelismo de grano grueso* es la versión más básica del paralelismo, donde acciones distintas también conocidas como *tareas* que cuentan con cierto grado de independencia, son ejecutadas en núcleos de procesamiento diferentes sobre diferentes conjuntos de datos. La mejora en la eficiencia del algoritmo viene dada por los tipos de dependencias de control existentes, las cuales pueden ser:

- *Dependencia de Control de Secuencia:* Son tareas que de acuerdo a su naturaleza deben realizarse estrictamente de forma secuencial.
- *Dependencia de Control de Comunicación:* Se da cuando una o más tareas requieren de información generada por otras tareas para poder ser culminadas.

A diferencia del paralelismo de control, el *paralelismo de datos* o *paralelismo de grano fino* está dado por las características del problema y su representación, la cual generalmente se da mediante la aplicación de ciclos a elementos almacenados en vectores o matrices. Bajo este esquema, una instrucción es repetida sobre cada uno de los elementos de la estructura, por lo que el total de elementos puede ser repartido entre el total de núcleos de procesamiento disponibles, aplicando la misma instrucción a cada uno de ellos. Algunas veces, explotar el paralelismo de un algoritmo mediante la aplicación de alguno de los tipos de paralelismo detallado anteriormente resulta complicado, por lo que es necesario aplicar una mezcla de ambos.

En el caso de los GPUs, el paralelismo de datos ha demostrado obtener una mayor escalabilidad que el paralelismo de tareas para aplicaciones que requieren el manejo de grandes cantidades de datos [Kirk, Hwu, 2013]. Otro factor fundamental para la eficiencia de los algoritmos paralelos, además del tipo de paralelismo, es el acceso a los datos, ya que la arquitectura de la GPU involucra varios niveles de memoria, mismos que serán explicados a detalle en la sección 4.3.5.1. Cada nivel de memoria cuenta con un tiempo diferente para el acceso a los datos, lo cual se conoce como *latencia*, que es la suma del tiempo requerido por las comunicaciones y acceso a los datos. Para reducir la latencia de un algoritmo paralelo en GPUs se requiere analizar el modelo de memoria de la tarjeta utilizada, además del patrón de acceso a memoria de los hilos, lo cual es crucial para el rendimiento del algoritmo.

4.3.2 Programación Híbrida MPI-CUDA

El surgimiento y constante evolución de las tecnologías de cómputo de alto rendimiento, han venido a proponer nuevos retos para el desarrollo de algoritmos cada vez más eficientes, que permitan explotar estas nuevas infraestructuras que proporcionan un gran poder de cómputo y escalabilidad, lo cual se ha logrado gracias a la incorporación de procesadores CPU y GPU. La arquitectura de la CPU y la GPU es muy diferente, por lo que requiere del uso de paradigmas de programación distintos.

Como producto del surgimiento y constante evolución de las tecnologías clúster y Grid, consideradas de alto rendimiento debido a la capacidad de cómputo que proporcionan a los usuarios, superando por mucho la de las supercomputadoras debido a que incorporan arquitecturas CPU y GPU. El manejo de arquitecturas diferentes dentro de una misma infraestructura requiere del uso de paradigmas de programación diferentes. El término de *programación híbrida* se acuña como resultado de la combinación de dos o más paradigmas de programación, producto del surgimiento de las tecnologías clúster y Grid, así como de la incorporación de tarjetas gráficas como unidades de procesamiento para cálculo numérico.

De forma general, el esquema híbrido de programación paralela más frecuentemente utilizado es MPI/OpenMP [Siegfried, Brandes, 2001; Rabenseifner, 2003; Rabenseifner et al., 2009; Hongzhong, 2011]. Aunque cabe mencionar que el esquema de programación a implementar depende de la infraestructura que se tenga disponible. De modo que tomando en cuenta las características de la Grid Morelos, descritas en la sección 6.1, en este trabajo de investigación se implementó un esquema de programación híbrida que involucra el uso de programación paralelo-distribuida utilizando paso de mensajes con MPI y CUDA, que permite el desarrollo de algoritmos para ser ejecutados sobre tarjetas gráficas de NVidia [Karunadasa, Ranasinghe, 2009]. Algunas de las ventajas de trabajar con este tipo de programación híbrida se enlistan a continuación:

- Mayor escalabilidad
- Permite combinar paralelismo de grano grueso y grano fino
- Mayor flexibilidad
- Permite eliminar conflictos en las dependencias de datos

Con la finalidad de mejorar la eficiencia del algoritmo AGC**D**-VRPTW, descrito en la sección 4.2, se desarrolló la versión paralelo-distribuida, nombrada en esta tesis AGC**P**-VRPTW (por sus siglas Algoritmo Genético Cooperativo Paralelo – Vehicle Routing Problem with Time Windows), la cual toma ventaja de estas nuevas tecnologías y busca la explotación de las características de la Grid Morelos (sección 6.1). En esta versión se identifica la parte del algoritmo que requiere mayor tiempo de procesamiento y es paralelizada y ejecutada en las GPUs. La versión del algoritmo genético AGC**D**-VRPTW fue analizada tomando el tiempo requerido para la generación de la población inicial, así como de cada uno de los operadores genéticos utilizados de forma independiente. El análisis mostró que el operador de mutación cooperativa es el proceso que consume la mayor parte del tiempo requerido por el algoritmo, por lo que fue paralelizado con CUDA. Para realizar una paralelización eficiente, es fundamental conocer a detalle la arquitectura de la tarjeta a utilizar. En la siguiente sección se describen las características de la tarjeta NVidia Tesla C2070 utilizada en este trabajo de tesis.

4.3.3 Tarjeta NVidia Tesla C2070

En la actualidad, la tecnología GPU ha sido ampliamente utilizada en aplicaciones de cómputo científico con la finalidad de maximizar el rendimiento de los algoritmos. De acuerdo a las características de la infraestructura de la Grid Morelos, se utilizó la tarjeta Tesla C2070, la cual se muestra en la figura 4.26.

Figura 4.26 *Tarjeta NVidia Tesla C2070*

La tarjeta NVidia Tesla C2070 es una unidad de procesamiento gráfico multinúcleo con arquitectura Fermi que funciona como un acelerador para aplicaciones paralelas utilizando infraestructura de cómputo de alto rendimiento (*High Performance Computing, HPC* por sus siglas en inglés). Las especificaciones y características de esta tarjeta se muestran en la tabla 4.3.

Tabla 4.3 *Especificaciones de la Tarjeta NVidia Tesla C2070*[2]

Características	**Arquitectura**
Total de SMs	14
SPs por SM	32
Total de Cores	448
Memoria Compartida/SP	48 KB
Registros/SM	215
Tamaño Warp	32
MaxHilos/SP	210
Warps/SM	48
MáxHilos/SM	1536
Memoria Compartida/Bloque	48KB
Bloques/SM	8
Ancho de Banda de Memoria	144 GB/seg.
Memoria Dedicada	6 GB GDDR5 Velocidad: 1.5 GHz.

[2] NVidia. Procesador de GPU Computing Tesla C2050 / C2070.
http://www.nvidia.es/object/product_tesla_C2050_C2070_es.html

A nivel de hardware y de acuerdo a las características mostradas en la tabla 4.2, cada tarjeta Tesla C2070 se compone de 14 *streaming multiprocesors* (SMs), donde cada uno contiene 32 *streaming processors* (SP) también conocidos como *cuda cores* o núcleos de procesamiento GPU, sumando un total de 448 cores por tarjeta. Cada SM tiene acceso a 4 unidades de funciones especiales (*SFU*), calendarizador de hilos, unidades de carga y almacenamiento (*LD/ST*), las memorias de solo lectura Constante y de Textura, así como la memoria de compartida, tal y como se muestra en la figura 4.27.

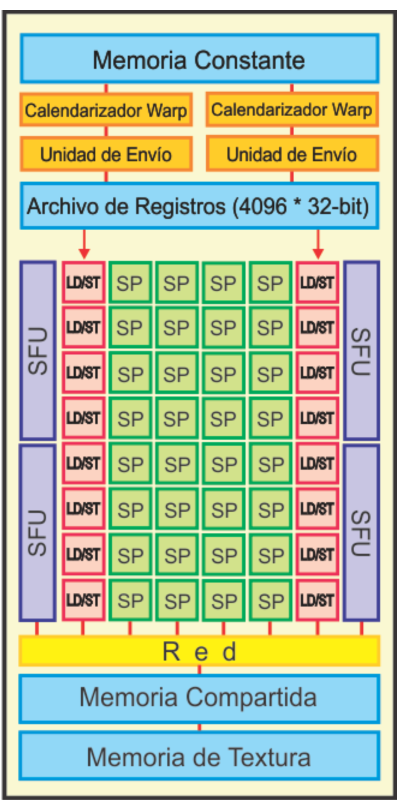


Figura 4.27 *Arquitectura de cada SM de la tarjeta NVidia Tesla C2070*

La GPU, también conocida como *device,* ejecuta porciones paralelas de una aplicación, denominadas *kernel,* los cuales se ejecutan de forma independiente en el *device*, por lo que cada *kernel* debe completar su ejecución antes de que inicie el siguiente. Un *kernel* se ejecuta por medio de un arreglo de hilos (*threads*) denominado *grid*. Los *hilos* de una *grid* se encuentran distribuidos en *bloques,* donde

cada *hilo* cuenta con un *id* local continuo que permite identificarlo, además de facilitar el control de acceso a las direcciones de memoria. De forma similar, los *bloques* pertenecientes a una *grid* también cuentan con un identificador único. Cabe mencionar que cada *hilo* ejecuta un mismo segmento de código de forma paralela sobre datos diferentes, a lo que se le denomina *contexto*, por lo que si existen diferencias entre el contexto ejecutado por *hilos* pertenecientes a una misma *grid*, se dice que existe *divergencia*[3].

La ejecución de un *bloque* se lleva a cabo en un solo SM por grupos de *hilos* que comparten datos y requieren ser sincronizados, los cuales se conocen como *warps*, por lo que bloques diferentes pueden ser asignados a SMs diferentes para ser ejecutados de forma paralela. El tamaño de un warp se toma preferentemente en múltiplos de 8, con un máximo de 32 *hilos*, por lo que la cantidad de *warps* por *bloque* se calcula con la fórmula 26.

$$Warps_{Bloque} = \frac{Threads_{Bloque}}{Threads_{Warp}} \qquad (26)$$

Cada *SM* puede ejecutar máximo 48 *warps* repartidos a lo más en 8 *bloques*, de modo que el total de *hilos* que se ejecutarán por *SM* se calcula con la fórmula 27.

$$Hilos_{SM} = bloques_{Total} * Warps_{Bloque} * Warp_{Tam} \qquad (27)$$

De acuerdo a lo anterior, la ejecución de un *kernel* se lleva a cabo asignando sus elementos de la siguiente manera:

- Un *hilo* se ejecuta en un core.
- Un *bloque* se asigna a un *SM* y no puede ser migrado.
- Varios *bloques* pueden residir en un mismo *SM*, lo cual dependerá de los requerimientos de memoria de cada *bloque*.

[3] Nvidia Research. Parallel Computing in CUDA.
http://www.gpgpu.org/oldsite/asplos2008/ASPLOS08-3-CUDA-model-and-language.pdf

- El conjunto total de *bloques de hilos* se denomina *grid* y es ejecutada como un *kernel* en un *device*.

El modelo de ejecución de un *kernel* consiste en obtener una paralelización eficiente mediante la distribución uniforme de los *hilos*, la cual consiste en determinar los valores adecuados para la cantidad *hilos* por *bloque* ($Threads_{Bloque}$) y tamaño del *warp* ($Warp_{Tam}$), para mantener la máxima ocupación posible entre *SMs* y *SPs*. Otro punto fundamental para el desarrollo de aplicaciones paralelas, consiste en tener un completo entendimiento de la jerarquía y modelos de memoria manejados en el *device*, los cuales se explican a detalle en la sección 4.3.5.1. Esto debido a que el *device* trabaja como un coprocesador de la CPU (*host)*, la cual se encarga del lanzamiento del *kernel* a ejecutar, tal como se muestra en la figura 4.28.

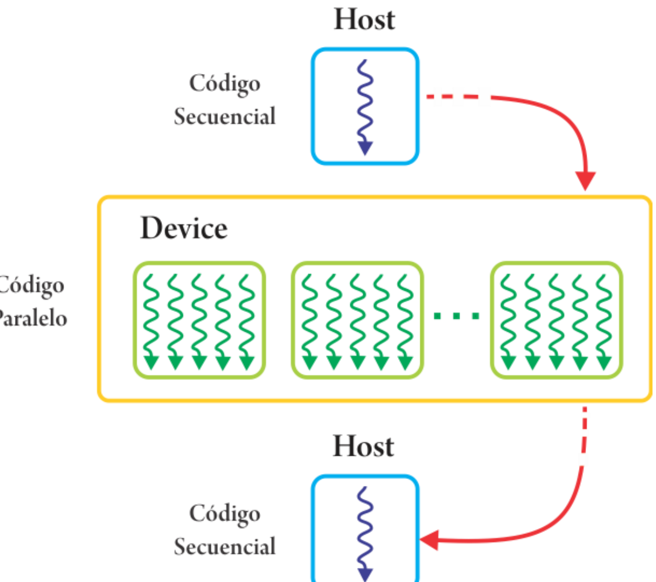


Figura 4.28 *Lanzamiento de un kernel a partir de un proceso secuencial ejecutado en el host y enviado al device para ser paralelizado*

Con base a lo descrito anteriormente, la adecuada distribución de bloques e hilos, así como el manejo de los *kernels* tiene como objetivo mejorar la eficiencia de las aplicaciones paralelas con base en las siguientes características:

- Adecuar el grado de paralelismo para utilizar en su totalidad la infraestructura disponible, explotando el uso del mutihilado (*multithreading*) para mantener los *cores* de procesamiento ocupados.
- Optimizar el acceso a los diferentes niveles de memoria del *device.*
- Almacenar los datos de modo que facilite el acceso a los *hilos*, reduciendo la latencia.

4.3.4 Operadores Genéticos Distribuidos

El algoritmo genético cooperativo paralelo-distribuido denominado en esta tesis como AGC**P**-VRPTW (por sus siglas Algoritmo Genético Cooperativo Paralelo – Vehicle Routing Problem with Time Windows), corresponde a una modificación del AGC**D**-VRPTW explicado en la sección 4.2, el cual se enfoca en mejorar la eficiencia del algoritmo mediante la implementación de programación híbrida, la cual comprende el uso de programación distribuida con paso de mensajes MPI y la programación paralela con CUDA, para ser ejecutado en la infraestructura Grid Morelos (sección 6.1).

A diferencia del método secuencial AGC-VRPTW y del distribuido AGC**D**-VRPTW, el método propuesto ejecuta los procesos que requieren mayor esfuerzo computacional de forma paralela en la GPU, enviando los demás procesos a la CPU de forma distribuida. Lo anterior, basado en que tanto la GPU como la CPU son unidades de procesamiento que deben complementarse para obtener mejores resultados, ya que aunque la GPU tiene mayor capacidad de procesamiento y es más rápida que la CPU, está diseñada para trabajar como un coprocesador, por lo que la CPU se encarga de controlar y enviar los procesos que serán ejecutados en la GPU.

El modelo mostrado en la figura 4.26 ilustra la distribución de procesos realizada para el AGCP-VRPTW donde intervienen procesos distribuidos en los nodos de procesamiento CPU y el proceso paralelo de mutación cooperativa enviado a las tarjetas gráficas NVidia.

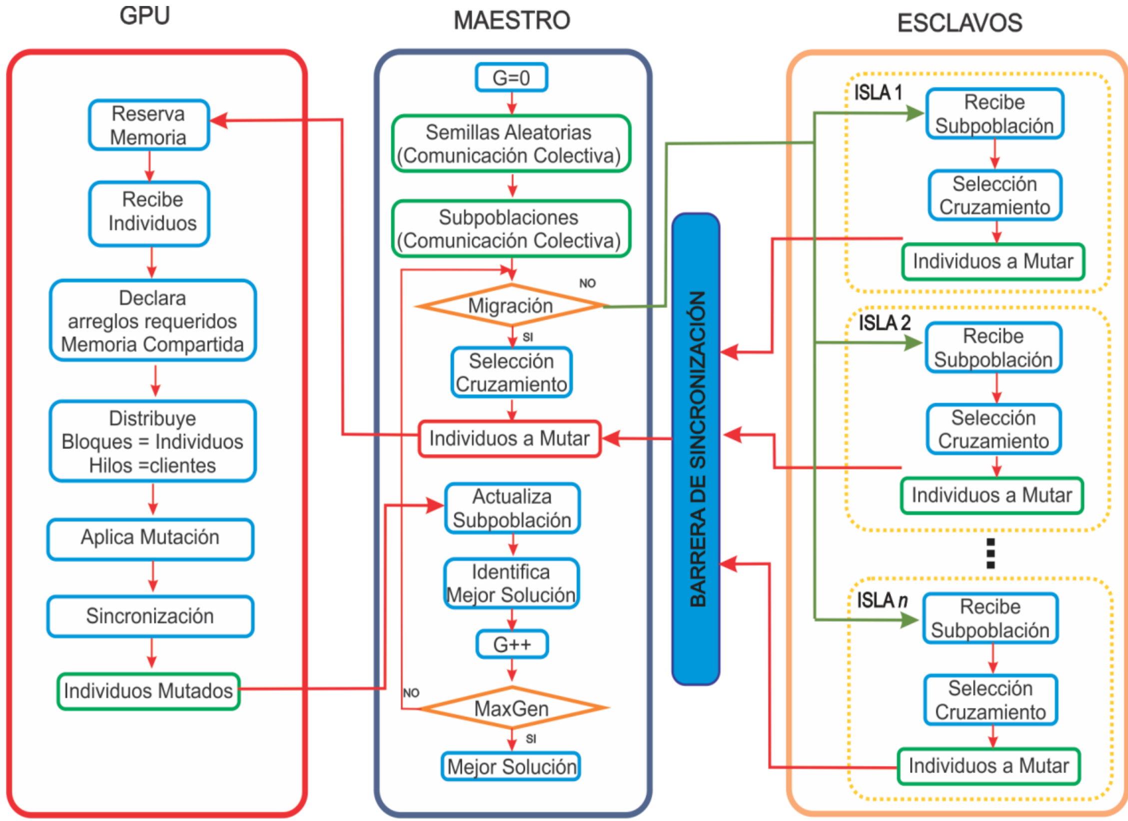


Figura 4.29 *Modelo paralelo-distribuido AGCP-VRPTW aplicado al VRPTW*

El modelo presentado en la figura 4.29 representa un algoritmo genético cooperativo paralelizado mediante un modelo de islas con migración implícita, donde se hace uso de dos tecnologías de gran relevancia en el cómputo científico, la programación distribuida con paso de mensajes MPI y la programación paralela en GPUs con CUDA. El modelo propuesto se compone de tres partes, la primera parte constituye las funciones del proceso maestro que se encarga de controlar los procesos que serán enviados a los nodos esclavos de forma distribuida y los procesos que serán paralelizados para ser ejecutados en el *device*. La segunda parte se enfoca a recibir los datos enviados por el proceso maestro y trabajar de como una isla en la evolución de su subpoblación, aplicando únicamente los operadores de selección y cruzamiento explicados en la sección 4.1. La tercera parte se enfoca en paralelizar el operador de mutación cooperativa para ser ejecutado sobre los device. Debido a las características

del modelo, uno de los puntos fundamentales para su desarrollo es el manejo de la programación hibrida, lo cual se detalla en la sección 4.3.2.

Para llevar a cabo el desarrollo del algoritmo paralelo-distribuido propuesto en esta tesis, fue necesario identificar el o los procesos que consumen mayor tiempo computacional en el algoritmo AGC**D**-VRPTW. Con base al análisis explicado en la sección 4.2, el proceso de mutación cooperativa implementado por medio de un ACH es el operador genético que requiere mayor cantidad de esfuerzo computacional, por lo que fue analizado para ser paralelizado con CUDA.

De acuerdo al modelo presentado en la figura 4.29, el proceso maestro en el *host* se encarga de controlar el proceso general del algoritmo, por lo que lleva a cabo la generación de semillas aleatorias (sección 4.2.1) para posteriormente enviarlas a cada una de las islas utilizando comunicación colectiva por medio de MPI. Cada una de las islas recibe la semilla correspondiente y genera su propia subpoblación de individuos factibles basado en el método explicado en la sección 4.2.1. Posteriormente, sí el número de generación del algoritmo es par, cada una de las islas aplica los operadores genéticos de selección por torneo aleatorio (sección 4.1.2.1) y cruzamiento variable (sección 4.1.2.2). Subsecuentemente, aplican el operador de selección con base al parámetro de *ProbMuta*, lo cual se muestra en la tabla 5.9, para indicar los individuos que serán sometidos al operador de mutación cooperativa, para finalmente regresar las subpoblaciones al proceso maestro. En caso de que generación sea impar, la aplicación de los operadores genéticos de selección y cruzamiento, así como la selección de los individuos a mutar, se lleva a cabo en el proceso maestro, con base a los valores de los parámetros de control mostrados en la tabla 5.16, tomando los individuos de todas las subpoblaciones como una población pancmítica, lo cual simula un proceso de migración.

Previo al término de cada generación, los individuos seleccionados para ser mutados son enviados al *device*, donde son almacenados y sometidos al proceso de mutación cooperativa paralelo, mismo que se describe a detalle en las secciones siguientes.

4.3.5 Mutación Cooperativa Paralela con GPUs

Como se mencionó anteriormente, el *device* se compone de un conjunto de núcleos de procesamiento gráfico que permiten ejecutar instrucciones en paralelo, donde algunas de las características más importantes a evaluar es la distribución de bloques e hilos, así como el manejo del modelo de memoria, mismo que se explica a detalle en la sección 4.3.5.1.

Los *hilos* ejecutados corresponden a procesos muy ligeros que deben ser sincronizados por medio de barreras que garantizan la integridad de los resultados. Por lo que basado en lo anterior, el proceso de mutación cooperativa paralela requirió de un análisis profundo del método aplicado, así como de la infraestructura descrita en la sección 6.1.

El operador de mutación cooperativa implementa un algoritmo colonia de hormigas que es bien conocido por tener características altamente paralelizables, por lo que es importante tomar en cuenta el proceso realizado y su distribución en hilos y bloques que definan la granularidad que favorezca la eficiencia y minimice la comunicación entre procesos [Cecilia et al., 2013; Dawson, Iain, 2013; Uchida et al., 2013]. El proceso de paralelización del operador de mutación cooperativa consiste en tres partes: la mutación de la solución, actualización de la feromona y actualización global, donde cada una de estas partes corresponde a un *kernel*. A continuación, en la figura 4.30 se muestra el modelo de ejecución del operador paralelo para cada uno de los *bloques de hilos*.

La figura 4.30 muestra una pepresentación de los procesos ejecutados por cada uno de los bloques de hilos, donde cada *hilo* corresponde a un cliente $i \in N$, de modo que cada *bloque de hilos* corresponde a una hormiga, que representa a un individuo mutado.

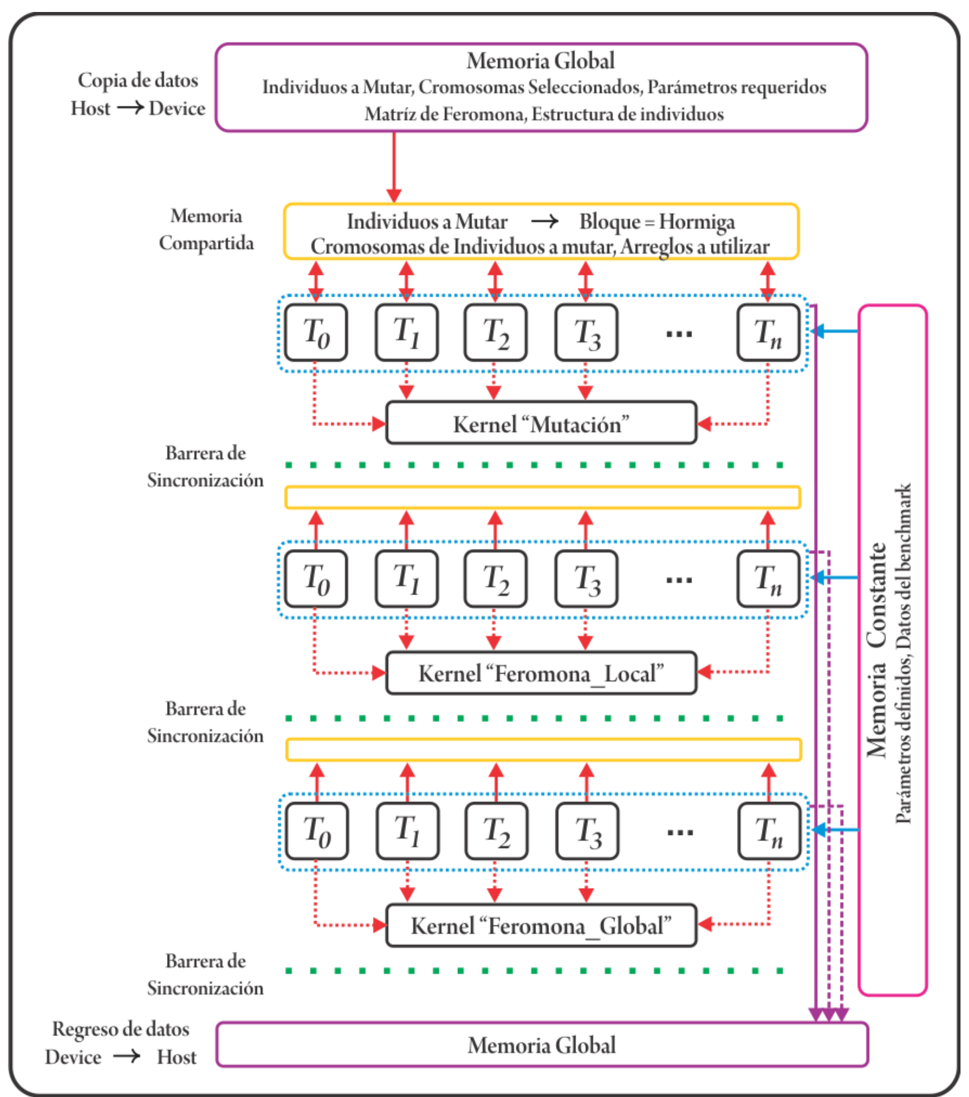
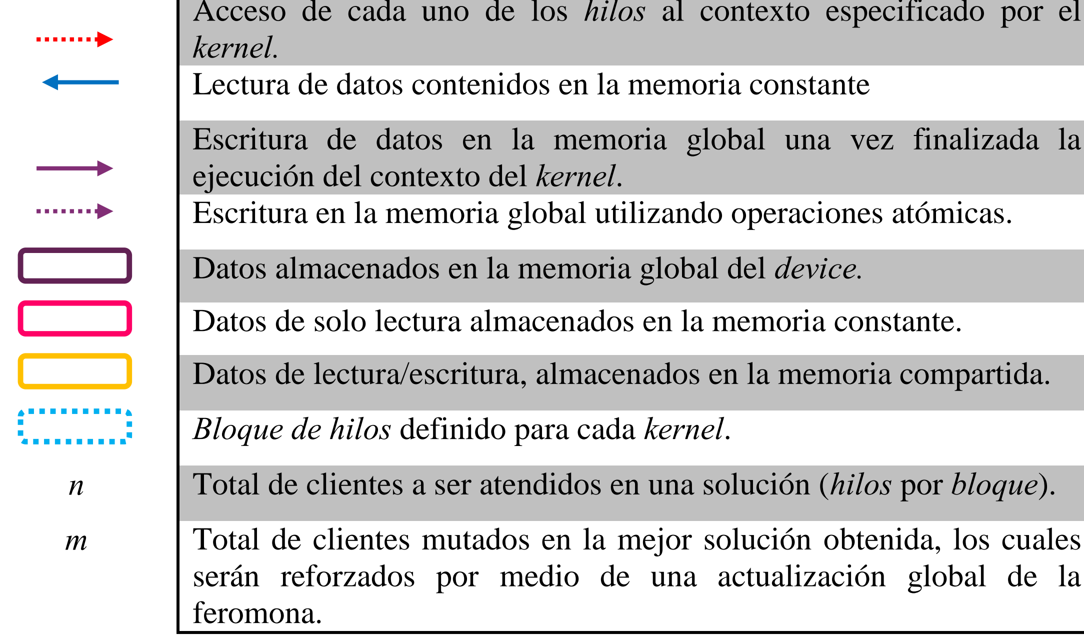


Figura 4.30 *Mapeo del proceso de mutación cooperativa a un modelo paralelo en CUDA.*

A continuación en la tabla 4.3, se explica la simbología utilizada en el diagrama.

Tabla 4.4 *Simbología utilizada en la representación del mapeo del proceso de mutación cooperativa para ser paralelizado con GPUs.*

Símbolo	Significado
→ (flecha roja)	Acceso hacia o desde alguno de los niveles de memoria del *device* en un solo sentido, lectura o escritura.
↔ (flecha roja doble)	Acceso de lectura/escritura a la memoria global o memoria compartida, según corresponda.



	Acceso de cada uno de los *hilos* al contexto especificado por el *kernel.*
	Lectura de datos contenidos en la memoria constante
	Escritura de datos en la memoria global una vez finalizada la ejecución del contexto del *kernel*.
	Escritura en la memoria global utilizando operaciones atómicas.
	Datos almacenados en la memoria global del *device.*
	Datos de solo lectura almacenados en la memoria constante.
	Datos de lectura/escritura, almacenados en la memoria compartida.
	Bloque de hilos definido para cada *kernel*.
n	Total de clientes a ser atendidos en una solución (*hilos* por *bloque*).
m	Total de clientes mutados en la mejor solución obtenida, los cuales serán reforzados por medio de una actualización global de la feromona.

Bajo este esquema, de manera general, el proceso enviado por el *host* copia la información requerida por la mutación cooperativa a la memoria global del *device* (sección 4.3.5.1), en seguida el host lanza el *kernel* "*Mutación*", el cual se encarga de aplicar el proceso de mutación cooperativa, mismo que se detalla en la sección 4.3.5.2. En primera instancia los datos que requieren acceso constante, así como la estructura para las soluciones de las hormigas son almacenadas en la memoria compartida. Por su parte los datos que se mantienen constantes durante la ejecución del *kernel* y que a su vez son accedidos de forma recurrente, son almacenados en la memoria constante. Ambas niveles de memoria son accedidas por cada uno de los *hilos* para llevar a cabo el proceso de construcción de soluciones. Al término del proceso se aplica una barrera de sincronización que garantiza que todos los *hilos* hayan terminado su respectivo contexto.

Una vez que la solución ha sido construida, se lanza el *kernel* "*Feromona_local*" que permite incrementar los montos de feromona en los trayectos que hayan sido recorridos por cada una de las hormigas aplicando operaciones atómicas (sección 4.3.5.3). Para esto, cada uno de los *hilos* accede a la información de las soluciones almacenadas en la memoria compartida y los datos de la memoria

constantes, aplicando una barrera de sincronización que asegure que todos los arcos recorridos sean actualizados.

Finalmente, se ejecuta el *kernel* "*Feromona_Global*", detallado en la sección 4.3.6.5, donde cada uno de los *hilos* aplica una evaporación global en el arco correspondiente, para posteriormente aplicar un incremento sobre los arcos recorridos en la mejor solución mutada encontrada.

Un punto fundamental para la ejecución del modelo paralelo, es el manejo eficiente de la memoria del *device*, para esto es necesario tener un amplio conocimiento de los diferentes niveles de memoria y sus características, los cuales se detallan en la sección 4.3.5.1.

4.3.5.1 Modelo de Memoria

Durante la ejecución de un *kernel*, los *hilos* pueden requerir acceso a datos almacenados en diferentes espacios de memoria, por lo cual es importante tomar en cuenta que la arquitectura del *device* involucra diferentes tipos de memoria, mismos que poseen características específicas que tienen la función que brindar los recursos de almacenamiento necesarios para la ejecución de una aplicación paralela. Una parte fundamental para lograr una paralelización adecuada consiste en el análisis de dichas características de memoria, para obtener el uso adecuado de los recursos.

De forma general, los niveles de memoria disponibles para el *device* se encuentran divididos en dos niveles [Kirk, Hwu, 2013]: la memoria principal (fuera de la GPU) y la memoria en el chip (dentro de la GPU). La *memoria principal* se caracteriza por tener una gran capacidad de almacenamiento que además de permitir el alojamiento de información, funge como un puente de conexión que permite la entrada y salida de datos entre el *host* y el *device*. Aunque cuenta con una desventaja, la alta latencia que conllevan los accesos frecuentes, que lejos de favorecer el tiempo de ejecución, afectan considerablemente la eficiencia. Por el contrario, la *memoria* existente *dentro de la GPU* es muy rápida, pero a diferencia de la memoria global, su tamaño es pequeño y limitado, además de manejar una sub-clasificación que la divide

en varios niveles con características distintas, las cuales se muestran en la figura 4.31. Debido a dicha sub-clasificación es fundamental identificar los datos que requieran acceso frecuente y que cumplan con las características especificadas para ser almacenados en cada tipo de memoria.

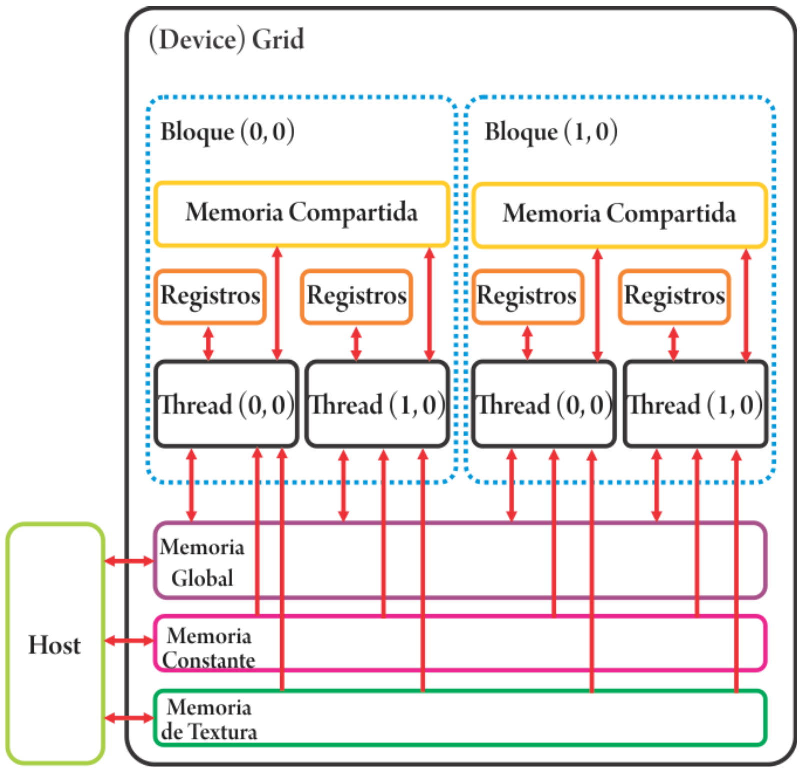


Figura 4.31 *Modelo de memoria del device [Kirk, Hwu, 2013]*

En la figura 4.31 se muestra la ubicación de los niveles de memoria del *device,* donde de acuerdo al modelo de ejecución, las instrucciones que requieren acceso a los datos se manejan de forma diferente, dependiendo del nivel de memorial al que requieran acceder. El manejo adecuado del modelo de memoria del *device* permite el mapeo eficiente de un algoritmo a un modelo paralelo enfocado en la aceleración basada en el paralelismo masivo.

Los tipos de memoria disponibles en el *device* de acuerdo a su jerarquía se muestran en la tabla 4.5, donde se presentan sus características básicas de almacenamiento y duración.

Tabla 4.5 Modelo de memoria del device de acuerdo a su alcance

Tipo de Memoria	En chip	Caché	Acceso	Alcance	Tiempo de Vida
Registros	Si	No	Lectura/Escritura	hilo	hilo
Compartida	Si	Residente	Lectura/Escritura	hilos (bloque)	bloque
Constante	No	Si	Lectura	hilos y host	aplicación
Textura	No	Si	Lectura	hilos y host	aplicación
Local	No	No	Lectura/Escritura	hilo	hilo
Global	No	No	Lectura/Escritura	hilos y host	aplicación

Además de las características mostradas en la tabla 4.5, otro punto fundamental para el desarrollo de una aplicación paralela eficiente son los tiempos de acceso a memoria y el tiempo de vida de los datos almacenados, como se muestra en la tabla 4.5. Los tiempos de acceso, mejor conocidos como latencia, varían considerablemente dependiendo del patrón de acceso aplicado [NVidia, 2008], así como del nivel de memoria al que se acceda [Sanders, Kandrot, 2010], lo cual se muestra en la tabla 4.6.

Tabla 4.6 Modelo de memoria del device de acuerdo a su jerarquía [Sanders, Kandrot, 2010][Kirk, Hwu, 2013]

Tipo de Memoria	Tamaño	Ciclos de Reloj	Velocidad
Registros	4 bytes c/u	1	Rápida
Compartida	Pequeña (48 KB)	4	Rápida
Constante	Pequeña (64 KB)	Variable	Rápida
Textura	Pequeña (64 KB)	Variable	Rápida
Local	Arbitrario	400-600	Lenta
Global	Grande (6 GB)	400-600	Lenta

De acuerdo a la jerarquía basada en la latencia de acceso, los *registros* son un tipo de memoria de lectura/escritura que permiten almacenar las variables privadas de cada *hilo* que reducen la latencia, comparada con los demás niveles de memoria, requiriendo un solo ciclo de reloj por acceso. Una de las características fundamentales

de este tipo de memoria es que cada *hilo* cuenta con una cantidad limitada de registros (acorde al modelo de la GPU), además de que no permite el almacenamiento de arreglos o matrices, los cuales deberán ser almacenados en el siguiente nivel de memoria.

La *memoria compartida* es una memoria de lectura/escritura residente en la GPU, a partir de la cual puede existir cooperación entre los *hilos* de un mismo *bloque*. Por lo que corresponde a una memoria de acceso rápido divida en bancos, como se muestra en la figura 4.32.

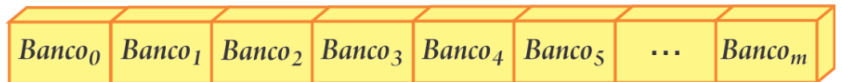


Figura 4.32 *Bancos de la Memoria Compartida de la tarjeta C2070*

Si múltiples *hilos* de un mismo *warp* hacen peticiones a un mismo *banco*, aunque se trate de accesos a direcciones distintas, se dice que existe un *conflicto de banco,* lo que ocasiona que las operaciones sean serializadas, como se observa en la figura 4.33.

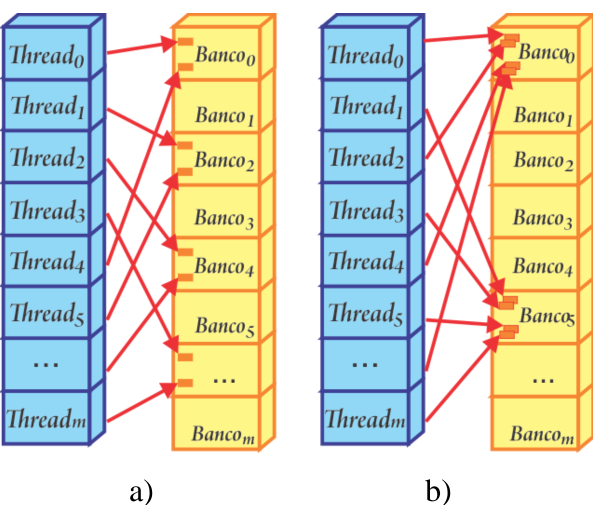


Figura 4.33 *Conflicto de bancos en la memoria compartida. a) Acceso a datos con un desplazamiento de dos datos. b) Acceso a datos con un desplazamiento de ocho datos [Sanders, Kandrot, 2010][Kirk, Hwu, 2013]*

Sí *r* operaciones son ejecutadas de forma secuencial ocasionan un decremento del paralelismo, lo cual se ve reflejado en la eficiencia del algoritmo.

El efecto de la serialización se refleja en la disminución del ancho de banda efectivo. Por lo que, para obtener el máximo rendimiento se requiere minimizar los conflictos de banco, basado en que los datos se almacenan por palabras de 32 bits, de modo que para evitar los conflictos de banco, cada *hilo* de un mismo bloque debe acceder a bancos de memoria distintos, lo que hace más eficiente los accesos a memoria, como se muestra en las figuras 4.34.

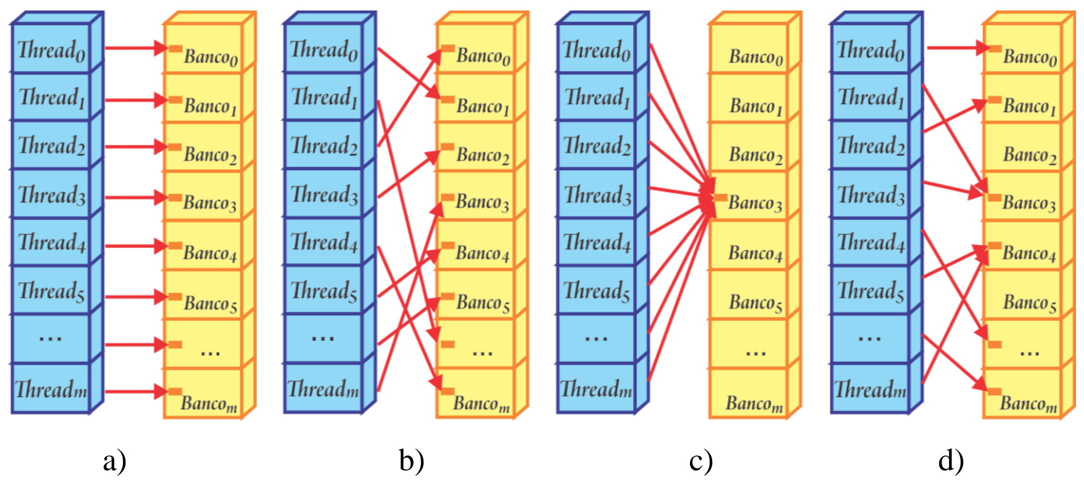


Figura 4.34 *Acceso de un bloque de hilos a memoria compartida sin conflicto de bancos. a) Acceso ordenado, b) Acceso aleatorio, c) y d) Un dato es accedido por varios hilos simultáneamente [Sanders, Kandrot, 2010][Kirk, Hwu, 2013]*

Las figuras 4.34 a y b, representan *accesos coherentes*, lo que implica que todos los *hilos* de un bloque pueden ser ejecutados de forma paralela sin que exista conflicto de ningún tipo, lo que representa una aceleración en la ejecución del algoritmo.

El siguiente nivel de acuerdo a la latencia de acceso corresponde a dos tipos de memoria de solo-lectura que pueden ser accedidas por todos los hilos del kernel, lo que permite reducir los tiempos de acceso a valores constantes. Los dos tipos de memorias son: Constante y de Textura [Sanders, Kandrot, 2010; Kirk, Hwu, 2013], donde ambas solo pueden ser escritas por el host. La *memoria constante* es una memoria caché de solo lectura, donde los tiempos de acceso son variables dependiendo de la cantidad de accesos concurrentes, de modo que mientras mayor sea la cantidad de *hilos* que acceden a una dirección , la latencia será más cercana al de

acceso a un registro. En el caso de la *memoria de textura*, al igual que la memoria constante, es una memoria caché de solo lectura que se encuentra optimizada para aprovechar el acceso a dos dimensiones, ofreciendo diferentes modos de direccionamiento, como filtrado de datos y algunos formatos específicos de datos [Sanders, Kandrot, 2010; Kirk, Hwu, 2013]. Por lo que el uso adecuado de la memoria de textura permite mejorar el rendimiento del algoritmo, debido a que permite ocultar la latencia de acceso a direcciones aleatorias.

Debido a que los tipos de memoria explicados anteriormente cuentan con un límite de almacenamiento, mostrado en la tabla 4.5, los datos que no puedan ser almacenados en alguno de los niveles anteriores, serán enviados automáticamente por el compilador a la *memoria local*, la cual se encuentra ubicada dentro del espacio de la memoria global, por lo que el costo de acceso, mostrado en la tabla 4.5. Por su parte, la *memoria global* corresponde a la memoria principal, la cual al igual que la memoria local, cuenta con un costo de acceso muy alto debido a que se encuentra fuera del chip, por lo que el *patrón de acceso* utilizado juega un papel fundamental en la eficiencia del algoritmo. Basado en esto, un punto fundamental para ocultar la latencia de las peticiones es aplicar un *patrón de accesos coherentes*, de preferencia mediante la ejecución de *medio warp*, ya que la ejecución de *warps* completos involucra realizar dos peticiones, una por cada medio *warp*. Los *accesos coherentes* (figura 4.35) de los datos se dan cuando cada *hilo* accede a un dato de forma consecutiva, de lo contrario, se acceden a segmentos de memoria distintos, ejecutando *t* operaciones.

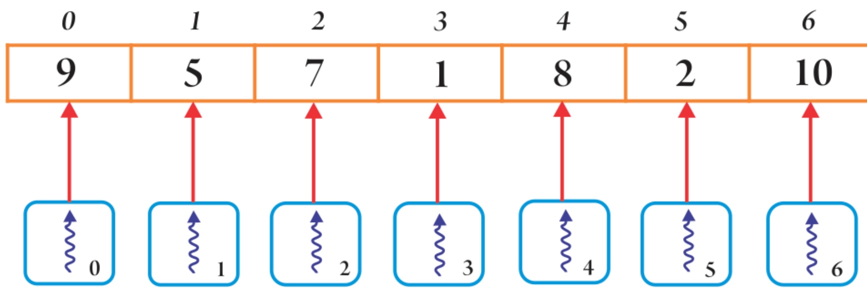


Figura 4.35 *Ejemplo de un patrón de acceso coherente a la memoria global [Sanders, Kandrot, 2010; Kirk, Hwu, 2013].*

Tomando en cuenta las características de cada uno de los niveles de memoria explicados anteriormente, el almacenamiento y acceso a los datos juega un papel fundamental en el desempeño del algoritmo, específicamente en la eficiencia. Por lo que, para almacenar variables en alguno de los niveles de memoria, se cuentan con calificadores definidos que facilitan dicha tarea, los cuales se muestran en la tabla 4.7.

Tabla 4.7 *Tipos de calificadores para el almacenamiento de variables en CUDA [Sanders, Kandrot, 2010; Kirk, Hwu, 2013]*

Declaración de Variables	Memoria	Alcance	Duración
Almacenamiento automático de variables diferentes a arreglos	Registros	Hilo	Kernel
Almacenamiento automático de arreglos	Local	Hilo	Kernel
`__device__, __shared__, int SharedVar`	Compartida	Bloque	Kernel
`__device__, int GlobalVar`	Global	Grid	Aplicación
`__device__, __constant__, int ConstVar`	Constant	Grid	Aplicación

Con base al análisis del modelo de memoria descrito anteriormente, se llevó a cabo el mapeo de la versión secuencial del operador de mutación cooperativa para ser ejecutado en paralelo con GPUs, proceso que se detalla en las siguientes secciones.

4.3.5.2 kernel "Mutación"

De acuerdo a implementaciones paralelas desarrolladas con base al algoritmo colonia de hormigas y sus variantes, se ha observado que el grado de paralelismo que maneja este método es muy alto, por lo cual puede ser manejado tanto con paralelismo de *grano fino* como de *grano grueso* [Dawson, Stewart, 2013; Cecilia et al., 2014].

La granularidad es la característica fundamental de aplicaciones paralelas que permite medir la cantidad de procesamiento requerido por un proceso, lo cual corresponde al número de instrucciones en un elemento de procesamiento (*grano*). En un enfoque de *grano grueso,* ACH maneja dos esquemas principalmente, basado en paralelismo de tareas. El primero corresponde al mapeo de múltiples colonias de hormigas como elementos de procesamiento, donde una de sus características

principales es la comunicación manejada entre las colonias [Dorigo, Stützle, 2004]. El segundo esquema mapea cada una de las hormigas como elementos de procesamiento, de modo que la cooperación se da a partir de múltiples colonias [Bai et al., 2009]. Por otro lado, el paralelismo de *grano fino* permite obtener aplicaciones altamente paralelas. En el caso del paralelismo con GPUs, el enfoque que ha obtenido mejores resultados de acuerdo a lo reportado en la literatura, es el mapeo de un bloque de *hilos* como una hormiga, lo que permite incrementar considerablemente el grado de paralelismo, favoreciendo la eficiencia del algoritmo.

En este trabajo de investigación se propone un mapeo del proceso de mutación cooperativa explicado en la sección 4.1.2.3.1 a un modelo paralelo en GPUs con la finalidad de mejorar la eficiencia del algoritmo, para lo cual se utiliza un enfoque basado en paralelismo de grano fino.

El primer paso para incorporar la hibridación de la mutación cooperativa paralela en CUDA con el algoritmo genético distribuido con paso de mensajes, es necesario generar un código intermedio que permita tener acceso a la memoria del *device*. Para esto se desarrolla una función conocida como *wrapper*, el cual se encarga de realizar las reservaciones en memoria para cada uno de los arreglos requeridos por el proceso de mutación y asegurarse de que no existan problemas en su alojamiento, además de lanzar el o los *kernel*, para finalmente realizar la liberación de memoria correspondiente.

La declaración de una función *wrapper* se realiza como una función externa de C, a la cual se envían los datos a utilizar en el proceso que se ejecuta de forma paralela, tal como se muestra en el siguiente ejemplo.

```
extern "C" void FuncionWrapper(int TamMuta, float
(*matriz)[N], float *arreglo)
```

El pseudocódigo del *wrapper* desarrollado para el algoritmo AGC**P**-VRPTW se muestra en la figura 4.36, donde se reserva memoria tanto en la memoria global como en la memoria constante.

```
1.  __constant__ float g_matrizD[N*N]
2.  extern "C" void wrapperMuta(int TamMuta, InitialPop *PopulationCross,
    benchmark *instance, float (*MAT_DISTANCES)[N], float (*Tij)[N], int
    (*CHROMS)[CHROMUTA])
3.  Declara *g_matrizD,*g_matrizF,*g_Croms,
    grid=POPULATION*NPROCS*MUTATION
4.  InitialPop *g_Muta, *g_Best
5.  Benchmark *g_instance
6.  cudaError_t err
7.  size_t pitch_F, pitch_C
8.  size = ((POPULATION*NPROCS*MUTATION)/100)*sizeof(InitialPop)
9.  err = cudaMalloc(… &g_muta)
10. err = cudaMalloc(… &g_Best)
11. err = cudaMalloc(… &g_instance)
12. err = cudaMallocPitch(&g_matrizF, &pitch_F,…)
13. err = cudaMallocPitch(&g_Croms, &pitch_C,…)
14. err = cudaMemcpy(g_Muta, PopulationCross, …,cudaMemcpyHostToDevice)
15. err = cudaMemcpy(g_instance, instance, …,cudaMemcpyHostToDevice)
16. err = cudaMemcpyToSymbol(g_matrizD, MAT_DISTANCES, sizeof(float)*(n*n))
17. err = cudaMemcpy2D(g_matrizF, pitch_F, Tij, …,cudaMemcpyHostToDevice)
18. err = cudaMemcpy2D(g_Croms, pitch_C, CHROMS,
    …,cudaMemcpyHostToDevice)
19. Mutacion<<<grid, BLOCK>>>(g_matrizF, pitch_F/sizeof(float), g_Chroms,
    pitch_C/sizeof(int), g_Muta, g_instance)
20. err = cudaThreadSynchronize();
21. Feromona_Local<<<grid, BLOQUE>>>(g_Muta)
22. err = cudaThreadSynchronize();
23. Feromona_Global<<<grid, BLOQUE>>>(g_Muta, g_Best)
24. err = cudaThreadSynchronize();
25. err = cudaMemcpy(PopulationCross, g_Muta, …, cudaMemcpyDeviceToHost)
26. Libera_Memoria(g_instance, g_Croms, g_matrizD, g_Muta, g_Best, g_matrizF)
27. Fin-wrapper
```

Figura 4.36 *Pseudocódigo del wrapper utilizado para reservar memoria, copiado de datos, lanzamiento de kernels y liberación de memoria.*

1. Declara la matriz de distancias en la memoria constante, utilizando el calificador `__constant__`.

2. Declaración del *wrapper* como una función externa de C, la cual recibe los datos requeridos por el proceso de mutación cooperativa: total de cromosomas

a mutar, individuos a mutar, datos del benchmark, matriz de distancias, matriz de feromonas y cromosomas a mutar.

2-4. Declara los arreglos que serán utilizados para almacenar los datos que serán requeridos los kernel, además de definir el tamaño de la grid.

5. Variable que bloquea el *host* en caso de que ocurra un error en alguna de las instrucciones de cuda evaluadas.

6. Declara las variables de acolchamiento para mapear las matrices 2D a arreglos, donde pitch almacena el tamaño en bytes por filas mediante la instrucción:

```
size_t pitch_D, pitch_F, pitch_C;
```

7. Asigna a la variable `size`, el tamaño en bytes de los individuos a mutar, correspondiente al tamaño de la subpoblación por el número de procesos a ejecutar por la probabilidad de mutación, el resultado se divide entre 100 y se multiplica por el tamaño en bytes de cada elemento de la estructura.

8. Calcula el tamaño en bytes de la población de individuos a mutar.

9-11. Reserva memoria para los individuos a mutar, la mejor solución y los datos del benchmark, mediante la sintaxis de la siguiente instrucción.

```
err = cudaMalloc((void**)&g_arreglo, tamaño_bytes);
```

En la instrucción anterior, la variable `err`, se evalúa en cada uno de los alojamientos de memoria para identificar un posible error, mediante el siguiente código.

```
if(err != 0)
{
      printf("Error allocating g_arreglo: %s\n",
      cudaGetErrorString(err));
      exit(1);
}
else printf("\n g_arreglo successful");
```

El proceso descrito anteriormente, se repite para el alojamiento de memoria de cada arreglo.

12-13. Mediante la función `cudaMallocPitch`, se lleva a cabo un mapeo de una matriz 2D a un arreglo, donde la variable pitch, permite completar los bloques de memoria por fila para evitar el desfasamiento con respecto a las columnas. Esto se lleva a cabo mediante la sintaxis de la siguiente instrucción.

```
err=cudaMallocPitch(&g_arreglo,&pitch,elementos *
sizeof(tipo_dato),elementos);
```

El proceso anterior se repite para la reservación de memoria de la matriz de distancias, matriz de feromona y cromosomas a mutar.

14-15. Se lleva a cabo el copiado de datos a los espacios reservados para los individuos a mutar y para los datos del benchmark a tratar, para lo cual se utiliza la función `cudaMemcpy`, aplicando la siguiente sintaxis:

```
err = cudaMemcpy(arreglo_device, arreglo,
sizeof(tipo_dato), cudaMemcpyHostToDevice);
```

La instrucción `cudaMemcpy` indica al final, el tipo de copiado a realizar, en este caso corresponde a un copiado de datos del *host* al *device*.

16. Copia el contenido de la matriz de distancias al arreglo reservado en la memoria constante, mediante la siguiente función.

```
err = cudaMemcpyToSymbol(arreglo_constante, matriz,
sizeof(tipo_dato)*(elementos))
```

17-18. Realiza el copiado de datos de una matriz bidimensional a un arreglo previamente alojado en memoria. Para el caso particular de un mapeo de 2D a 1D, se utiliza la función `cudaMemcpy2D`, donde se requiere el uso de la variable *pitch* para mantener el alineamiento por filas en la memoria, para lo cual se utiliza la siguiente sintaxis.

```
err = cudaMemcpy2D(arreglo_device, pitch, matriz,
elementos*sizeof(tipo), elementos*sizeof(tipo),
elementos, cudaMemcpyHostToDevice);
```

Esta función se aplica para copiar los datos de la matriz de distancias, la matriz de feromona y la matriz que contiene los cromosomas a mutar.

19-24. Lanzamiento de cada uno de los kernel, utilizando la siguiente sintaxis:

```
kernel<<<grid, bloque>>>(parámetros);
```

donde *grid* indica el número de bloques a ejecutar y bloque la cantidad de *hilos* en un bloque.

25. Al igual que en el punto 14-15, se utiliza la función `cudaMemcpy` para copiar los resultados obtenidos, solo que en esta ocasión se copia del *device* al *host*, indicando que termina una generación del AGCP-VRPTW.

26. Se libera la memoria almacenada de todos los arreglos utilizando la función `cudaFree`, tal y como se muestra en el siguiente ejemplo.

```
cudaFree(nom_arreglo);
```

De acuerdo al pseudocódigo mostrado en la figura 4.36, una vez que la función `wrapper` se ejecuta, se procede a la reservación de memoria y al copiado de la información requerida por el algoritmo, de modo que el modelo propuesto se divide en tres partes (*kernels*): Mutación, Feromona_Local y Feromona_Global.

En esta sección se aborda el proceso del *kernel Mutación*, el cual corresponde al operador de mutación cooperativa, enfocado en la construcción de soluciones, mediante paralelismo de grano fino (paralelismo de datos).

Bajo el enfoque de paralelismo de datos, cada hormiga *h* corresponde a un bloque de *hilos* que representa un individuo, donde cada *hilo* se considera como un

cliente que debe ser atendido por un solo vehículo, este proceso se explica a detalle en el pseudocódigo de la figura 4.37.

```
1.  __global__ void Mutación(….)
2.  int il = threadIdx.x, sum, index
3.  __shared__ individuo_mutado[BLOQUE]
4.  __shared__ individuo_mutar[BLOQUE]
5.  __shared__ lista_Tabu [BLOQUE]
6.  __shared__ Pij [BLOQUE]
7.  individuo_mutar[il]← g_Muta[blockIdx.x].SOLUCION[il]
8.  Inicializa_tabú(il, lista_Tabu)  /*1 – asignado, de lo contrario 0*/
9.  Pij[il] ← Probabilidad(il, BLOQUE)
10. __syncthreads()
11. Sí Evalua_Restricciones(il) == 1
      Individuo_mutado[index] ← individuo_mutar[il]
      Actualiza_costo(costoVh)
12. Fin-sí
13. __syncthreads()
14. g_Muta[blockIdx.x].SOLUCION[il] ← individuo_mutar[il]
```

Figura 4.37. *Pseudocódigo del kernel "Mutación"*

1. Declaración del *kernel* "Mutación" mediante el calificador `__global__`, el cual indica que el *kernel* será llamado desde el *host* para lanzar una *grid de hilos* en el *device*, para lo cual se utiliza la siguiente instrucción.

   ```
   __global__ void Mutacion(float *g_matrizF, int pitchF,
   int *g_Chroms, int pitch_C, InitialPop *g_Muta, benchmark
   *g_instance)
   ```

   Los datos que recibe el *kernel* corresponden a la matriz de feromona, los cromosomas e individuos a mutar y la información del benchmark.

2. Declara la variable índice *il*, que permite acceder a las posiciones compartidas del arreglo, *sum* almacena el resultado de la sumatoria requerida en la fórmula 8 e *index* indica la posición en donde se debe almacenar el cliente.

3-6. Declara en la memoria compartida los arreglos a utilizar durante el proceso de mutación cooperativa. Para ello se utiliza el calificador `__shared__`, bajo la siguiente sintaxis.

```
__shared__ arreglo[longitud]
```

Este proceso se repite hasta que todos los arreglos a utilizar de forma independiente por bloque, hayan sido declarados en la memoria compartida. Las variables almacenadas bajo este nivel de memoria son: individuos a mutar, individuo mutado, lista tabú y el arreglo de probabilidad *Pij*.

7. Copia el contenido del arreglo `g_Muta` en su posición `blockIdx.x` almacenado en la memoria global al arreglo *individuo_mutar* alojado en la memoria compartida, lo cual aunado a un patrón de acceso adecuado, permite reducir la latencia de acceso a memoria.
8. Realiza la inicialización del arreglo que fungirá como la memoria de las hormigas, llamado *lista_Tabu*. Para este proceso, cada uno de los *hilos* accede a una posición del individuo a mutar, bajo un esquema de accesos coherentes, como se muestra en la figura 4.38. Una vez localizado su respectivo cliente, se verifica si se encuentra calendarizado en alguno de los cromosomas seleccionados para ser mutados.

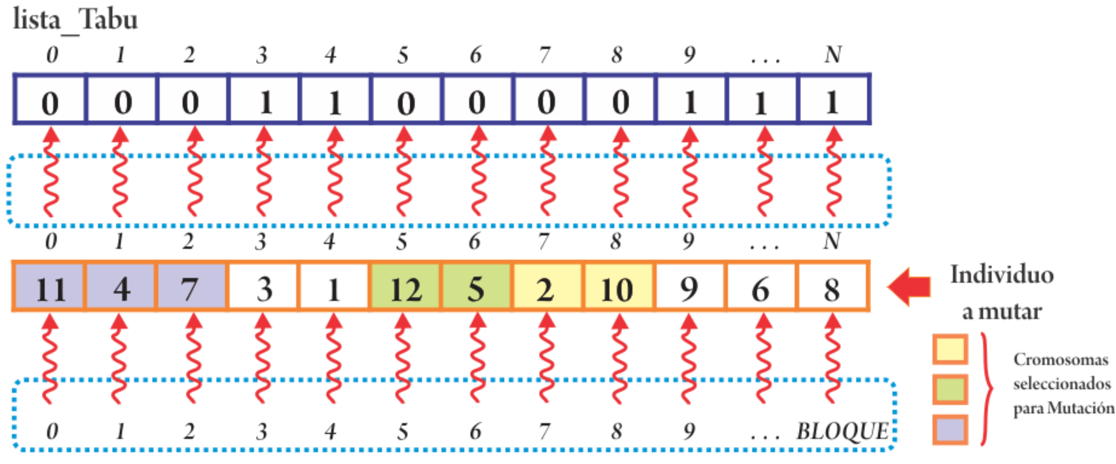


Figura 4.38 *Patrón de acceso coherente al individuo a mutar para el llenado de la lista tabú*

9. Cada uno de los hilos calcula la probabilidad de transición de su respectivo cliente con base a las fórmulas 7 y 8, las cuales son seleccionadas de forma aleatoria. En el caso de la fórmula 7, se implementó el siguiente código.

```
if(lista_Tabu[individuo_mutar.SOLUCION[threadIdx.x]==0)
{
    Pij[threadIdx.x]  =(float)pow(Tij[g_Muta[blockIdx.x].
    SOLUCION   [il]+g_Muta[blockIdx.x].SOLUCION[il-1]*N],
    alpha)  *  (float)pow  (g_matrizD[g_Muta[blockIdx.x].
    SOLUCION[il]+  g_Muta[blockIdx.x].SOLUCION[il-1]*N]],
    betha);
}
```

Lo cual en el caso de la fórmula 8, se complementa con el siguiente código, aplicando la división del resultado de *Pij* entre el valor de *sum*.

```
    sum   +=   (float)pow(Tij[g_Muta[blockIdx.x].SOLUCION
    [il]+g_Muta[blockIdx.x]. SOLUCION[il-1]*N], alpha) *
    (float)pow     (g_matrizD[g_Muta[BlockIdx.x].SOLUCION
    [il]+ g_Muta[blockIdx.x].SOLUCION[il-1]* N]],betha);
    __syncthreads();
```

10. Establece una barrera de sincronización que asegure que todos los *hilos* del bloque hayan calculado su respectiva probabilidad. Esto se realiza mediante la siguiente instrucción.

```
__syncthreads()
```

11. Se toma el cliente que cuente con la mayor probabilidad y se evalúan las restricciones de capacidad y tiempo descritas en la sección 2.2. Sí las restricciones se cumplen, se asigna el cliente a el arreglo *individuo_mutado* (figura 4.39) y se actualiza su estado en la *lista_Tabu*, de lo contrario, se toma el cliente que cuenta con la siguiente probabilidad más alta y se repite el proceso desde el punto 8, hasta que todos los clientes hayan sido asignados.

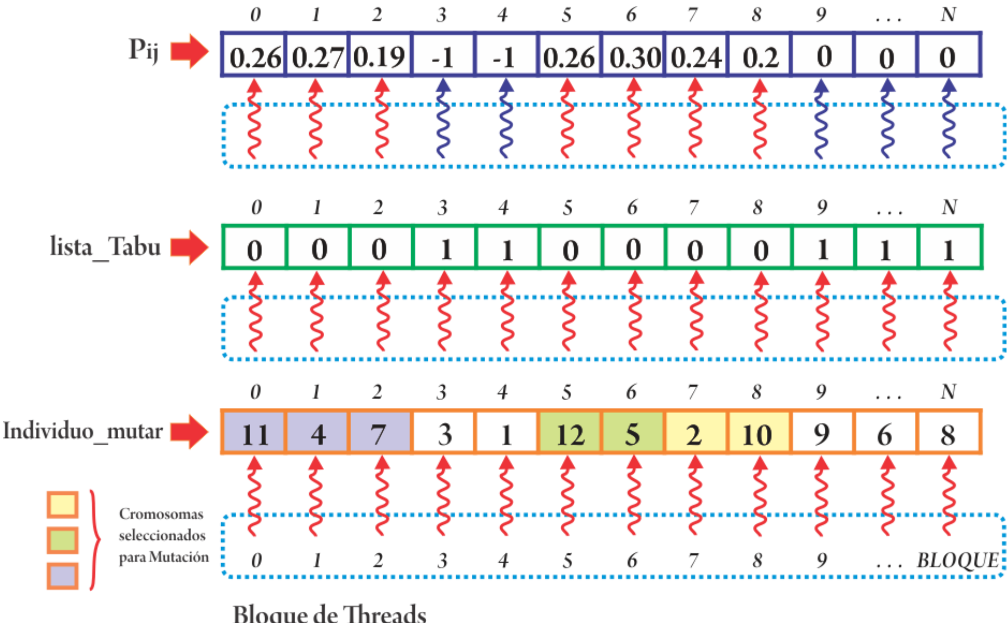


Figura 4.39 *Calculo paralelo de la probabilidad de los clientes contenidos en los cromosomas a mutar*

Una vez que el cliente fue asignado al arreglo *individuo_mutado*, se actualiza el costo del individuo en *costoVh*.

En caso de que ninguno de los clientes disponibles pueda ser asignado a la ruta actual, se crea una nueva ruta y se repite

13. Establece una barrera de sincronización que asegura que todos los *hilos* han terminado
14. Copia la solución mutada a la memoria global, actualizando el contenido de `g_Muta`.

Al término de construcción de soluciones por cada uno de los *hilos*, se ejecuta `cudaThreadSyncronize()`, la cual garantiza que se han concluido las funciones del *kernel*.

4.3.5.3 kernel "Feromona_Local"

Una vez terminada la mutación cooperativa de individuos, es importante llevar a cabo la actualización de feromona. De acuerdo a lo explicado en la sección 4.1.2.3.1, cada una de las hormigas recorre una de las soluciones construidas, donde cada *hilo* realiza un incremento $\Delta\tau_{ij}^{h}$ a los montos de feromona en los arcos que fueron recorridos, de modo que el monto a incrementar es proporcional a la calidad de la solución $\frac{1}{f^h}$.

La matriz de feromona utilizada en el ACH funciona como una comunicación indirecta entre todos los *hilos* de la *grid,* lo que de acuerdo al funcionamiento y manejo de los *bloques* en CUDA, corresponde a la única forma de comunicación posible entre ellos. Debido a las características del proceso de actualización de la feromona, donde se requiere forzosamente que todas las hormigas modifiquen los montos de los arcos recorridos, además de que un mismo arco puede ser actualizado por más de una hormigas, fue necesario utilizar *operaciones atómicas*.

Las operaciones atómicas son funciones específicas frecuentemente utilizadas para la coordinación de *hilos* que requieren modificar o actualizar una misma dirección de memoria sin interrupción, ya sea en la memoria compartida o en la memoria global. Esto asegura que cada *hilo* pueda aplicar la actualización correspondiente y los nuevos montos actualizados puedan ser accedidos por todos los *hilos* de la *grid*. Esto se logra debido a que las operaciones atómicas permiten serializar las instrucciones que requieran acceso concurrente a un mismo dato, eliminando el riesgo de colisión.

Existen varios tipos de operaciones atómicas, de acuerdo a la operación que se desee emplear, las más destacadas se muestran en la tabla 4.8.

Tabla 4.8 *Listado de operaciones atómicas disponibles en CUDA [Wilt, 2013]*

Operación	**Descripción**
atomicAdd	Suma
atomicSub	Resta

atomicExch	Intercambio
atomicMin	Mínimo
atomicMax	Máximo
atomicInc	Incremento en 1
atomicDec	Decremento en 1
atomicCAS	Compara e intercambia
atomicAnd	AND lógico
atomicOr	OR lógico
atomicXOR	XOR lógico

El proceso de actualización de la feromona corresponde un incremento en los montos de feromona de un arco que haya sido recorrido por una hormiga, de modo que se hizo uso de la función *atomicAdd*, mostrada en la tabla 4.8. El proceso completo de actualización de la feromona se muestra en la figura 4.40 y se detalla a continuación.

1. **__global__ void Feromona_local(InitialPop *g_Muta)**
2. int il = threadIdx.x, costo
3. __shared__ Ind_Actualiza[BLOQUE]
4. Ind_Actualiza[il] ← g_Muta[BlockIdx.x].SOLUCION[il]
5. costo ← g_Muta[blockIdx.x].FITNESS
6. int clienteActual = Ind_Actualiza[il]
7. int clienteAnterior=Ind_Actualiza[il-1]
8. **atomicAdd(Tij[clienteActual+clienteAnterior*N],**
   (float) $\Delta T_{ij}^{h} = \begin{cases} \frac{1}{f_h} & si\ \ se\ recorri\acute{o} \\ 0\ de\ lo\ contrario \end{cases}$)
9. __syncthreads()
10. **Fin-kernel**

Figura 4.40. *Pseudocódigo del proceso de actualización de feromona local aplicando operaciones atómicas en la memoria global*

1. Declara el *kernel "Feromona_local"* con el calificador **__global__**, mediante la siguiente instrucción,

```
__global__ void Feromona_local(InitialPop *g_Muta)
```

la cual recibe como parámetro los individuos mutados en el *kernel* *"Mutacion"*.

2. La variable *il* almacena el *id* del *hilo* actualmente ejecutado y que servirá como índice para el acceso a los arreglos.

3. Declara el arreglo *Ind_Actualiza* con una longitud *BLOQUE* en la memoria compartida. Este arreglo almacena el individuo sobre el cual se llevará a cabo la actualización de feromona.

4. Copia el contenido de *g_Muta.SOLUCION[il]* a el arreglo *Ind_Actualiza*, almacenado en la memoria compartida del *device*, lo que permite reducir la latencia de acceso a cada uno de los elementos de la solución.

5. Copia el fitness del individuo que modificará los montos de feromona a la variable "costo", almacenada en los registros del *hilo*.

6-7. Calcula los índices que corresponden al cliente asignado a cada *hilo*, así como al cliente que fue asignado en la posición anterior, esto con la finalidad de establecer las coordenadas de acceso al arco a modificar en la matriz de feromona.

8. Realiza los incrementos de feromona mediante el uso de operaciones atómicas de adición, para lo cual se utiliza la función *atomicAdd* bajo la siguiente sintaxis.

```
atomicAdd(&dirección, incremento)
```

donde en este caso, &dirección corresponde a la posición a modificar en la matriz de feromona e incremento corresponde al monto a agregar, lo cual se especifica en la fórmula 9. El comportamiento de las operaciones atómicas se muestra en la figura 4.41.

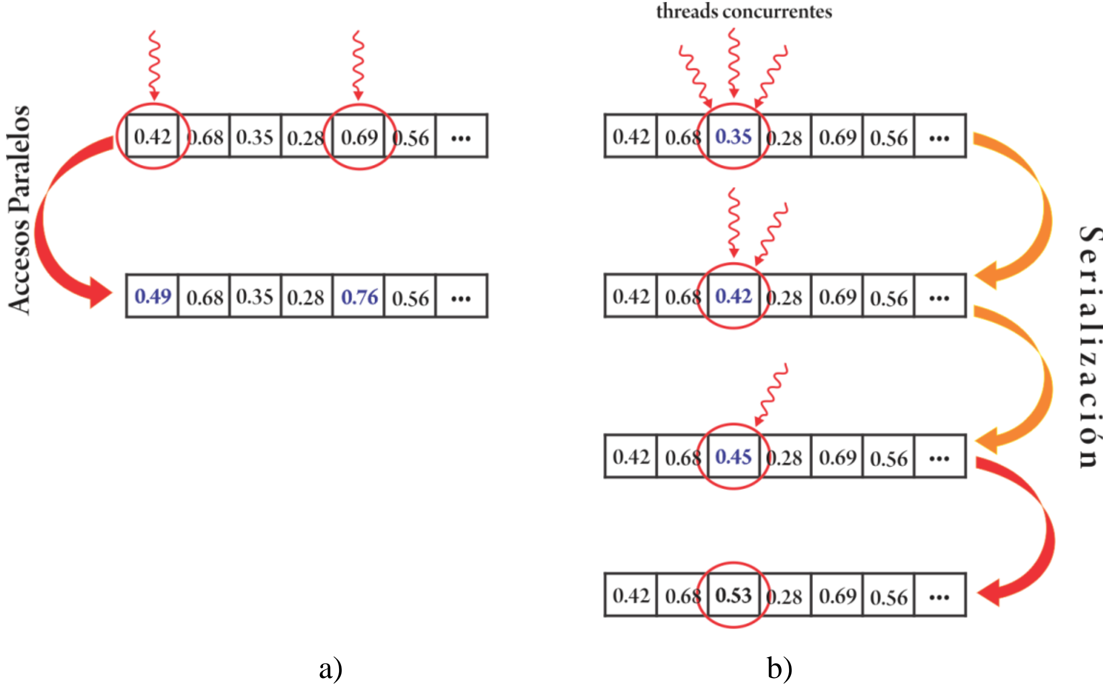


Figura 4.41 *Comportamiento de la función atomicAdd(...). a) Incrementos paralelos cuando un hilo modifica una posición. b) Múltiples hilos accediendo a una misma posición producen incrementos serializados.*

9. Establece una barrera de sincronización que asegura que todos los hilos han realizado el incremento correspondiente.

4.3.6.4 kernel "Actualización_Global" de la Feromona

La intensidad de los montos de feromona juega un papel crucial en el proceso de optimización, ya que el depósito de feromona incrementa la probabilidad de selección de los clientes, por lo cual, además de aplicar dicho incremento en los arcos recorridos, al igual que en el caso de las hormigas reales, se lleva a cabo un proceso de evaporación cada cierto tiempo (sección 4.1.2.3.1), lo que en el caso de las hormigas artificiales tiene la función de aplicar una especie de olvido que favorece la exploración de nuevos caminos, evitando la rápida convergencia del algoritmo, cuyo resultado quedaría atrapado en un sub-óptimo local.

La adecuada interacción de la actualización local y global permite mantener una mejora continua basada en la exploración y explotación del espacio de soluciones, mediante el incremento local y la evaporación global por un factor constante $\rho \epsilon (0, 1]$ que decrementa los montos de feromona de todos los arcos. Posterior al proceso de evaporación, se aplica un depósito de feromona $\Delta\tau_{ij}^{Best}$ sobre los arcos que correspondan a la mejor solución de la iteración, la cual se aplica para mantener una pequeña ventaja sobre los mejores arcos.

En el proceso de actualización global requiere de la aplicación de operaciones atómicas en la memoria global que permitan evaporar la feromona de todos los arcos y actualizar los pertenecientes a la mejor solución obtenida. El proceso explicado anteriormente se muestra en el pseudocódigo de la figura 4.42 y se detalla a continuación.

```
1.  __global__ void Feromona_Global(…)
2.   int il = threadIdx.x, mejor
3.  __shared__  Mejor_Sol[BLOQUE]
4.  __shared__ Ind_Actualiza[BLOQUE]
5.  Ind_Actualiza[il]  ← g_Muta[blockIdx.x].SOLUCION[il]
6.  int clienteActual = Ind_Actualiza[il]
7.  int clienteAnterior=Ind_Actualiza[il-1]
8.  atomicExch(Tij[clienteActual+clienteAnterior*N],
         (float)( τij = (1 − ρ)τij(t) + Δτij )
9.  __syncthreads()
10. mejor=Mejor_Solucion(g_Muta)
11. Mejor_Sol[il] ← g_Muta[mejor].SOLUCION[il]
12. Sí blockIdx.x == mejor
        int i=Mejor_Sol[il]
        int j=Mejor_Sol[il-1]
        atomicAdd(Tij[i + j *N], (float) 1/f_Best )
    fin-si
13. __syncthreads
14. Fin-kernel
```

Figura 4.42 *Pseudocódigo del proceso de evaporación y actualización de feromona del mejor individuo, mediante el uso de operaciones atómicas en la memoria global*

1. Declara el *kernel “Feromona_global”* con el calificador __global__, mediante la siguiente instrucción

```
__global__ void Feromona_global(InitialPop *g_Muta,
                InitialPop *g_Best)
```

   Donde `g_Best` conrresponde a la estructura que almacenará el mejor individuo de la iteración.

2. La variable *il* almacena el *id* del *hilo* actualmente ejecutado y que servirá como índice para el acceso a los arreglos.

3-4 Declara los arreglos *Mejor_Sol* e *Ind_Actualiza,* ambos con una longitud *BLOQUE* en la memoria compartida. Estos arreglos almacenan el mejor individuo de la iteración, así como el individuo sobre el cual se llevará a cabo la evaporación de la feromona.

5. Copia un individuo de la población mutada a la memoria compartida del SM donde fue asignado el bloque correspondiente que aplicará la evaporación a dicho individuo.

6-7 Identifica los índices que permitirán el acceso a los datos de la matriz de feromona, almacenada en la memoria global.

8. Aplica operaciones atómicas que permiten cambiar el valor almacenado en la matriz de feromona, en el arco correspondiente. Para esto se utiliza la sintaxis de la función *atomicExch*.

```
atomicExch(&dirección, valor_cambio)
```

   donde `&dirección` corresponde a la dirección de memoria donde se encuentra el valor a cambiar y `valor_cambio` indica el valor que reemplazará el valor original almacenado en `&dirección`, el cual corresponde al resultado de aplicar la fórmula 10 al monto actual de la dirección accedida.

9. Ejecuta una barrera de sincronización que asegura que todos los *hilos* del *bloque* hayan completado su proceso de evaporación.

10. Al término de cada iteración se aplica un proceso de reducción en `g_Muta` que permite identificar el individuo mutado con el mejor fitness, para lo cual se copia el fitness de cada uno de los individuos a un arreglo en la memoria compartida, mediante la instrucción

```
Costos[blockIdx.x] = g_Muta[blockIdx.x].FITNESS
```

Posteriormente, se procede a la aplicación del método paralelo de reducción comparando los fitness del arreglo por pares, para encontrar el mejor, de acuerdo a la función objetivo de minimización explicada en la sección 2.2. Un ejemplo del funcionamiento del método aplicado se muestra en la figura 4.43.

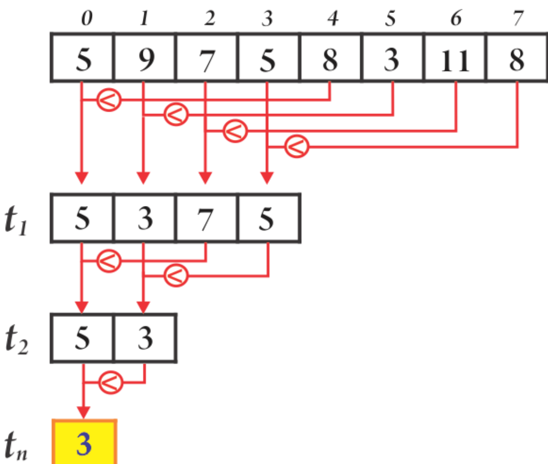


Figura 4.43 *Representación del método de reducción aplicado para la identificación del individuo con el mejor fitness, de acuerdo a la función objetivo*

Al aplicar cada una de las iteraciones que permite comparar los fitness en un factor de *Población/2*, donde se van almacenando los mejores fitness, guardando el *id* de las hormigas con los mejores fitness, correspondientes a *blockIdx.x*.

11. Copia el mejor individuo al arreglo *Mejor_Sol*, para aplicar la actualización de la feromona en los arcos correspondientes. El $\Delta\tau_{ij}^{Best}$ corresponde a la inversa de la calidad de la solución $\frac{1}{f_{Best}}$.

12. Sincroniza los hilos del bloque, previo al término del *kernel*.

A este punto se considera que se ha terminado una iteración de la mutación cooperativa. El número de iteraciones corresponde al valor de la probabilidad de mutación *ProbMuta* (capítulo 5).

4.3.6 Uso de Múltiples GPUs

En años recientes, uno de los retos que está tomando auge en el desarrollo del cómputo científico, es la programación paralela sobre múltiples GPUs, donde el ancho de banda, así como la distribución, calendarización y cooperación entre *hilos* de un mismo *bloque*, juegan un papel fundamental para el buen desempeño de los algoritmos. Aunado a esto, la creciente necesidad de tratar problemas con características cada vez más cercanas a la realidad y con instancias cada vez más grandes, requiere de mayor cantidad de recursos. En la actualidad, las GPUs cuentan con cientos e incluso de miles de cores de procesamiento por tarjeta, pero sus recursos en cuanto a memoria y ancho de banda se encuentran limitados, por lo que el uso de una arquitectura *multiGPU* proporciona una oportunidad para el procesamiento de mayor cantidad de información y para la mejora en la eficiencia de los algoritmos.

Las ventajas de una arquitectura multiGPU vienen dadas principalmente por su gran capacidad de procesamiento [Jacobsen et al., 2010], por lo que se ha venido utilizando para el desarrollo de proyectos de supercómputo de alto rendimiento, donde se han podido observar mejoras considerables en la *aceleración*, además de una mejor distribución de la información, ya que al manejar conjuntos de grandes cantidades de datos, los niveles de memoria disponibles en la GPU muchas veces no son suficientes, quedando saturados y afectando el rendimiento del algoritmo, por lo que al manejar más de una tarjeta, los conjuntos de datos (*tiles*) pueden ser manejados con una mejor distribución que permita incluso, la ejecución de mayor cantidad de hilos que favorezcan el desempeño del algoritmo.

En este trabajo de investigación se utiliza una arquitectura multiGPU en Grid, descrita en la sección 6.1, donde de manera general se trabaja con dos clústeres homogéneos que cuentan con dos tarjetas gráficas tesla C2070 cada uno, ambos conectados mediante una red de alta velocidad, donde cada uno de los *device*s cuenta con *id* lógico (inician en 0) diferente que permite identificarlos.

Para adecuar el algoritmo paralelo AGC**P**-VRPTW descrito en la sección 4.3.4 para su ejecución en una arquitectura *multiGPU*, es necesario tomar en cuenta que en esta tesis se está trabajando con un arreglo de estructuras que permite

```
/*Instrucciones a agregar en el wrapper*/
1. int num_gpu, dev
2. cudaGetDeviceCount(num_gpu)
3. int grid=(POPULATION*NPROCS*MUTATION)
4. int tam = grid/num_gpu
5. Se divide la g_Muta en num_gpu estructuras
6. Para dev=0 : dev<num_gpu
     cudaSetDevice(dev)
     cudaMalloc(…)              /*Alojamientos de memoria requeridos*/
     …
     cudaMallocPitch(…)
     …
     cudaMemcpy(…)              /* Copiado de datos */
     …
     cudaMemcpyToSymbol(…)
     cudaMemcpy2D(…)
     …
     /*Se evalua el segmento tam de g_Muta a enviar*/
     Mutacion<<<grid, BLOCK>>>(g_matrizF, pitch_F/sizeof(float),
     g_Chroms, pitch_C/sizeof(int), g_Muta, g_instance)
     err = cudaThreadSynchronize();
     Feromona_Local<<<grid, BLOQUE>>>(g_Muta)
     err = cudaThreadSynchronize();
     Feromona_Global<<<grid, BLOQUE>>>(g_Muta, g_Best)
     err = cudaThreadSynchronize();
     err = cudaMemcpy(PopulationCross, g_Muta, …, cudaMemcpyDeviceToHost)
     err = cudaMemcpy(PopulationCross, g_Muta, …, cudaMemcpyDeviceToHost)
     Libera_Memoria(g_instance,g_Croms,g_matrizD,g_Muta, g_Best, g_matrizF)
   Fin-para
7. Fin-wrapper
```

Figura 4.44 *Pseudocódigo del manejo de multiGPUs para el AGCP-VRPTW*

almacenar la información de los individuos a mutar, como se observa en la figura 4.8, conocida como *Población*, la cual debe ser dividida en grupos de individuos que serán repartidos entre la cantidad de *GPUs* disponibles. La estructura general de las modificaciones realizadas, se muestra en la figura 4.44.

1. Declara las variables que almacenan el número de gpus disponibles y el contador de dispositivos.
2. Se identifica la cantidad de GPUs disponibles mediante la función `cudaGetDeviceCount(…)`[4], cuyo funcionamiento se muestra en el siguiente ejemplo

   ```
   int num_gpu;
   cudaGetDeviceCount(&num_gpu)
   ```

   donde `num_gpu` corresponde a la variable que almacenará el número de dispositivos que se encuentran disponibles para ser utilizados en la ejecución.
3. Calcula el tamaño de la *grid* a ejecutar, que corresponde al total de *hilos* ejecutados en el *kernel.*
4. Divide `g_Muta` en cuatro estructuras parciales, debido a que la infraestructura Grid Morelos cuenta con un total de 4 tarjetas tesla C2070.
5. Genera un ciclo que en primera instancia selecciona el *device* con:

   ```
   cudaSetDevice(dev)
   ```

   donde *dev* corresponde al número de *device* en el que se realiza el proceso correspondiente de alojamiento de memoria, copiado de datos de la estructura correspondiente, ejecución de los *kernels* requeridos para la mutación cooperativa y liberación de memoria. Basado en lo anterior, se presenta el diagrama de la topología utilizada (figura 4.45) enfocada la infraestructura de la Grid Morelos.

[4] NVidia. Device Management. http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/group__CUDART__DEVICE_g665468e8cb33be42434f11bee2684ec9.html

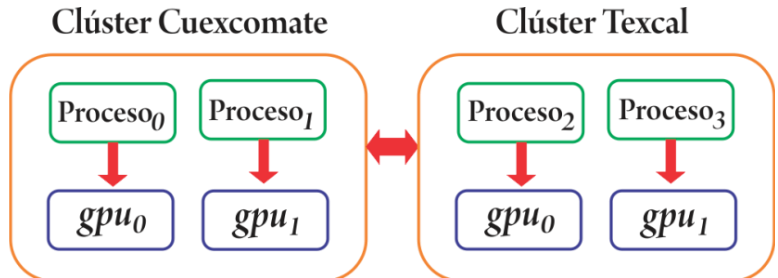


Figura 4.45 *Diagrama de ejecución de la mutación cooperativa en Grid, sobre las cuatro tarjetas tesla C2070.*

De forma independiente y de acuerdo al diagrama mostrado en la figura 4.5, el funcionamiento *multi-GPU* dentro de cada clúster se muestra en la figura 4.6, donde se observa claramente que no existe ningún tipo de comunicación entre los *devices*, permitiendo reducir los tiempos de procesamiento

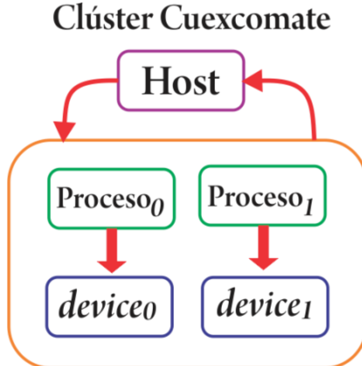


Figura 4.46 *Diagrama de ejecución multiGPU en el clúster Cuexcomate, donde muestra el lanzamiento de un kernel y su distribución en los device.*

4.3.7 Ejecución MPI-CUDA

Previo a la ejecución de un algoritmo con programación híbrida, es importante conocer el comportamiento y ejecución de cada uno de los enfoques utilizados. En este trabajo de investigación se utilizó una hibridación entre MPI y CUDA utilizando lenguaje C, donde de forma independiente, el manejo de ambos enfoques es muy distinto. En el caso de MPI, es una interfaz de paso de mensajes para el manejo de

programación distribuida y CUDA es una arquitectura de cómputo paralelo que favorece la eficiencia de los algoritmos mediante el uso de unidades gráficas.

La infraestructura de la Grid Morelos trabaja sobre el Sistema Operativo Linux Centos 2.6, donde MPI utiliza el compilador OpenMPI 1.8, donde de manera general, un algoritmo distribuido con MPI se compila mediante la instrucción:

```
mpicc –o archivo.o archivo.c
```

en caso de que no existan errores de sintaxis, el algoritmo puede ser ejecutado utilizando el comando `mpiexec`, en el cual es posible controlar el balanceo de carga de acuerdo al número de *cores* de procesamiento con que cuenta cada nodo. Su sintaxis se observa en la siguiente instrucción.

```
mpiexec –n <No.procesos> -host <nodo> ./<archivo_ejercutar> :
–n <No.procesos> -host <nodo> ./<archivo_ejercutar> …
```

Para CUDA se utiliza el compilador nvcc 5.5, donde, a diferencia de los archivos fuente de C y MPI, donde se guardan con extensión .c, en CUDA, los archivos tienen la extensión .cu y la sintaxis para su compilación se muestra a continuación.

```
nvcc –c <archivo_fuente>
```

Si el archivo no presenta errores de sintaxis y como en el caso anterior, no se especificó el nombre del ejecutable, se ejecuta de la siguiente forma.

```
./a.out
```

Tomando en cuenta que la ejecución de CUDA comprende instrucciones tanto del *host* como del *device*, el proceso de ejecución de un programa cuenta con las etapas mostradas en la figura 4.47.

Figura 4.47 *Proceso de ejecución de un programa en la GPU*

En este trabajo de tesis se presenta una hibridación MPI – CUDA para resolver el VRPTW. El origen de esta hibridación vienen dada por el hecho de que MPI es 100% compatible con CUDA, por lo cual es utilizada para el cómputo científico de alto rendimiento, debido a que permite explotar las características de las nuevas herramientas de supercómputo que involucran tanto CPUs como GPUs. Algunas de las ventajas que surgen como resultado de dicha hibridación son:

- Permite el tratamiento de problemas de *big-data.*
- Mejora la eficiencia de problemas que requieren gran esfuerzo computacional.
- Aceleran las aplicaciones distribuidas con MPI mediante la aplicación de programación paralela en GPUs
- Permite el manejo de multiGPU, lo que incrementa el poder de cómputo disponible en una infraestructura CPU-GPU.

Como se mencionó en capítulos anteriores, la GPU tiene la función de trabajar como un coprocesador que mejora la aceleración de aplicaciones paralelas, por lo que el *host* es el encargado de controlar la carga de trabajo y los procesos que serán ejecutados en la GPU, para lo cual, aunque se encuentran en la misma infraestructura y son compatibles, el manejo y representación de los datos es diferente, ya que a diferencia del *host*, la GPU trabaja con programación vectorial, por lo que para

combinar estos dos enfoques es necesario trabajar con dos archivos fuente, uno correspondiente a las instrucciones del *host* (procesos que serán ejecutados de forma secuencial o distribuida con MPI) y el otro contiene un código intermedio denominado *wrapper*, el cual prepara los espacios de memoria y lleva a cabo el copiado de la información requerida por la GPU, además de realizar el lanzamiento del o los *kernel*s, los cuales se encuentran contenidos en el mismo archivo, lo cual se muestra en la figura 4.48.

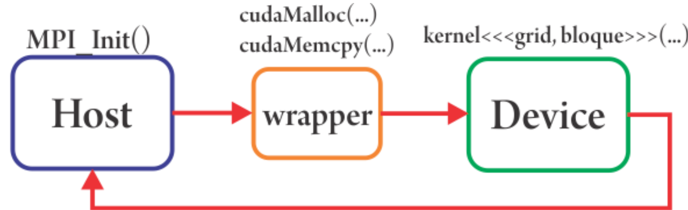


Figura 4.48 *Diagrama de ejecución de un programa MPI-CUDA*

Debido a las diferencias en la estructura de los archivos fuente, así como de los modelos de programación implementados, es necesario realizar algunos cambios en la compilación y ejecución del algoritmo. Bajo este esquema, el proceso de compilación se encarga de compilar los archivos MPI y CUDA de forma independiente.

```
mpicc -c main.c
nvcc -c cuda.cu
```

Debido a que el *host* se encarga de controlar los procesos de la GPU, el compilador de mpi realiza una segunda compilación que integra los dos archivos objeto producto de la compilación previa, como se muestra a continuación.

```
mpicc cuda.o main.o -lcudart-L/opt/cuda-5.5/lib64
```

Una vez integrados los archivos a ejecutar, se procede al enlazado, para el cual se utiliza la siguiente instrucción.

```
mpirun -np <procesos> ./a.out
```

Para realizar este proceso es importante considerar que se requiere forzosamente del compilador de OpenMPI, debido a que el compilador de Intel no es compatible con el compilador *nvcc*, por lo que en caso de que se esté utilizando el compilador de Intel es necesario cambiarlo. Para ello se modifica el archivo *.bashrc* para agregar la variable de entorno correspondiente, tal como se muestra en la figura 4.49.

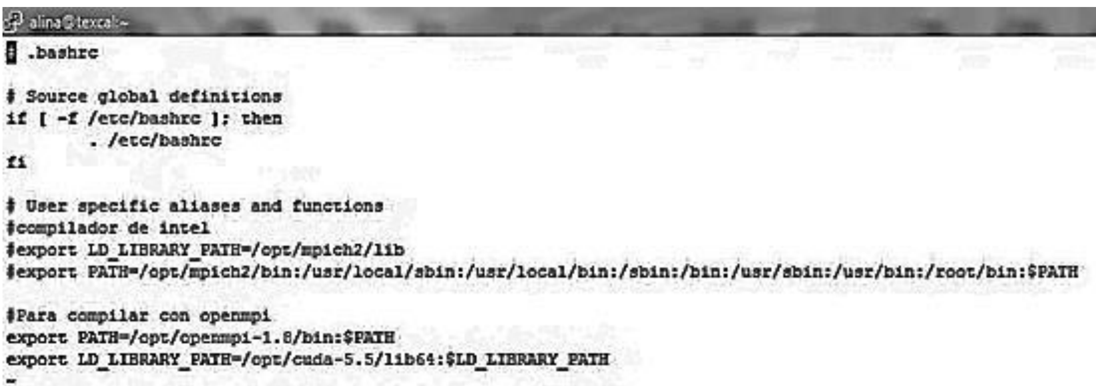


Figura 4.49 *Variable de entorno para el uso del compilador OpenMPI.*

Para tener un panorama más claro sobre las diferencias existentes entre cada uno de los algoritmos propuestos en este trabajo de investigación para abordar el VRPTW, se presentan en la tabla 4.9 las características de cada algoritmo, enfocadas al uso de los recursos computacionales.

Tabla 4.9 *Tabla comparativa de las características de los algoritmos AGC-VRPTW, AGC**D**-VRPTW y AGC**P**-VRPTW*

Algoritmo	Procesamiento	Tipo de Memoria	Granularidad	Tipo de Comunicación
AGC-VRPTW	Secuencial	RAM (local)	No aplica	No aplica
AGC*D*-VRPTW	Distribuido	Distribuida	Grano grueso	Comunicación Colectiva
Modelo de Islas	Distribuido	Distribuida	Grano Grueso	Comunicación Colectiva
Mutación Cooperativa	Paralelo	Global Compartida Constante Registros	Grano Fino	operaciones atómicas

Capítulo 5

Análisis de Sensibilidad

El análisis de sensibilidad es un estudio experimental que se realiza sobre los parámetros de control de un algoritmo, para identificar los valores que estadísticamente favorecen su comportamiento tanto en términos de eficacia como de eficiencia. Para llevar a cabo este análisis, es necesario realizar una búsqueda en la literatura para establecer rangos de valores entre los cuales se han registrado mejores resultados para el problema tratado, tomando como punto de comparación algoritmos que involucren metaheurísticas similares.

Un análisis de sensibilidad es un estudio experimental que permite identificar los valores de los parámetros de control que favorecen el comportamiento de un algoritmo, a lo que se conoce como *Sintonización,* lo cual a su vez permite saber que tan sensible es el algoritmo a cambios en los valores de dichos parámetros. Debido a las características de las instancias (sección 2.4) utilizadas para el problema tratado, fue necesario tomar un conjunto de instancias representativas que permitan abarcar un panorama general del problema, por lo que se evaluó una instancia correspondiente a cada distribución y una a cada tipo. Lo anterior, tanto para los benchmarks de Solomon como para los de Gehring y Homberger.

La implementación de un análisis de sensibilidad distribuido se propone con la finalidad de reducir los tiempos para obtener los valores sintonizados de los parámetros de control del algoritmo propuesto en esta tesis. Esto debido a que su aplicación de forma secuencial requiere demasiado tiempo dada la cantidad de parámetros y repeticiones que se tienen que hacer para cada uno. De acuerdo a la metodología de aplicación del análisis de sensibilidad, explicada en las secciones 5.1 y 5.2, el tiempo requerido puede ser reducido considerablemente si en lugar de utilizar una sola máquina, se utiliza un conjunto de procesadores que permitan procesar múltiples ejecuciones a la vez.

La metodología propuesta para el análisis de sensibilidad distribuido se explica de forma detallada en la sección 5.1 para el algoritmo secuencial AGC-VRPTW. Finalmente, la sección 5.2 se enfoca en la sintonización de los parámetros requeridos para la versión distribuida del algoritmo AGCD-VRPTW.

5.1 Algoritmo Secuencial AGC-VRPTW

Dado un conjunto de parámetros $P = \{c_1, c_2, c_3, ..., c_p\}$ que requieren ser sintonizados, donde cada parámetro $c \in P$ está compuesto por un conjunto de valores $V= \{p_1, p_2, ..., p_v\}$, se propone un análisis de sensibilidad distribuido que acorte el tiempo de cómputo requerido.

Para obtener la sintonización de los parámetros de control del algoritmo secuencial propuesto en este trabajo de investigación se describe una metodología que permite realizar un análisis de sensibilidad de forma distribuida sobre la infraestructura de la Grid Morelos (sección 6.1). Es importante considerar que para poder realizar el análisis de sensibilidad, se tomaron en cuenta los parámetros de las dos metaheurísticas hibridadas en el AGC-VRPTW, que son el AG y el ACH. Asimismo, la metodología explicada a continuación, muestra como una sintonización distribuida permite reducir los tiempos y abarcar rangos más grandes para cada parámetro, lo cual favorece los resultados obtenidos por el análisis de sensibilidad.

1. ***Identificar los Parámetros de Control***. Los parámetros de control de un algoritmo, corresponden a aquellos donde el cambio en su valor involucra una consecuencia positiva o negativa sobre los resultados obtenidos. Para el algoritmo AGC-VRPTW, se identificaron los parámetros del AG y del ACH que lo componen, los cuales se muestran en la tabla 5.1.

Tabla 5.1 *Parámetros de Control a sintonizar para el AGC-VRPTW*

Metaheurística	Parámetros					
Algoritmo Colonia de Hormigas	A	β	ρ	*Colonia*	q_0	τ_0
Algoritmo Genético	*Tam_Pob*	$T_{muestra}$	*ProbCruce*	*CromMuta*	*ProbMuta*	*MaxGen*

Donde para el algoritmo colonia de hormigas:

- α → Importancia relativa de la feromona
- β → Importancia relativa de la distancia heurística
- ρ → Coeficiente de evaporación de la feromona
- *Colonia* → Cantidad total de hormigas
- q_0 → Coeficiente de equilibrio entre exploración y explotación.
- τ_0 → Monto de feromona Inicial

Para el algoritmo genético:

- *Tam_Pop* → Tamaño de la población
- $T_{muestra}$ → Tamaño de la muestra para el torneo
- *ProbCruce* → Probabilidad de cruzamiento
- *CromMuta* → Cantidad de cromosomas a mutar
- *ProbMuta* → Probabilidad de Mutación
- *MaxGen* → Máximo de generaciones en el AG

2. ***Establecer Rangos.*** Los rangos de valores asignados a cada parámetro de control se definieron tomando los valores con los cuales los resultados del algoritmo siguieron siendo buenos. Para el algoritmo AGC-VRPTW, los rangos definidos para los parámetros de control en tabla 5.1, se muestran en la tabla 5.2 para el ACH y en la 5.3 para el AG.

Tabla 5.2 *Rangos de los parámetros de control del algoritmo colonia de hormigas*

Parámetro	Min.	Max.
α	0.2	2
β	0.2	2
ρ	0.1	0.9
Colonia	10	100 y Var.
q_0	0.2	0.9
τ_0	$1/(n*L_{nm})$	10

Donde:

- *Var* → Tamaño de colonia variable
- *$1/(n*L_{nm})$* → Inversa de multiplicar los *n* clientes por la longitud de la solución L_{nm} encontrada por una heurística del vecino más cercano [Gilmour, Dras, 2005].

Tabla 5.3 *Rangos de los parámetros de control del algoritmo genético*

Parámetro	Min.	Max.
TamPop	100	1000
$T_{muestra}$	2	10
ProbCruce	0.4	0.95
CromMuta	3	Rutas/2
ProbMuta	0.01	0.6
MaxGen	100	5000

3. ***Tamaño de las Muestras.*** Con base en los rangos establecidos en el punto 2, se establece un número de muestras para cada rango, como se observa en la tabla 5.4 y 5.5. Dichas muestras serán utilizadas para evaluar el comportamiento del algoritmo.

Tabla 5.4 *Tamaño de las muestras para los parámetros de control del algoritmo colonia de hormigas*

Parámetro	Incremento	No. de muestras
α	0.1	19
β	0.1	19
ρ	0.05	17
Colonia	10	12
q_0	0.05	15
τ_0	$1/(n*L_{nm})$ y 10	2

Tabla 5.5 *Tamaño de las muestras para los parámetros de control del algoritmo genético*

Parámetro	Incremento	No. de Muestras
TamPop	50	19
$T_{muestra}$	1	9
ProbCruce	0.05	12
CromMuta	1	10
ProbMuta	0.05	13
MaxGen	100	50

De modo que el conjunto de valores *p* correspondientes a cada parámetro *c* a utilizar en el análisis de sensibilidad se muestran en la tabla 5.6.

Tabla 5.6. *Muestras definidas para cada uno de los parámetros de control del algoritmo genético cooperativo*

Algoritmo Colonia de Hormigas	
α	0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0
β	0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0
ρ	0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9
Colonia	10, 20, 30, 40, 50, 60, 70, 80, 90, 100, Var.
q_0	0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9
τ_0	$1/(n*L_{nm})$, 10
Algoritmo Genético	
TamPop	100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750, 800, 850, 900, 950, 1000
$T_{muestra}$	2, 3, 4, 5, 6, 7, 8, 9, 10
ProbCruce	0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95
CromMuta	3, 4, 5, 6, 7, 8, 9, 10, 11, 12
ProbMuta	0.01, 0.06, 0.11, 0.16, 0.21, 0.26, 0.31, 0.36, 0.41, 0.46, 0.51, 0.56, 0.61
MaxGen	100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000, 2100, 2200, 2300, 2400,

2500, 2600, 2700, 2800, 2900, 3000, 3100, 3200, 3300, 3400, 3500, 3600, 3700, 3800, 3900, 4000, 4100, 4200, 4300, 4400, 4500, 4600, 4700, 4800, 4900, 5000

4. ***Proceso de Sintonización de Parámetros***. El objetivo del análisis de sensibilidad se enfoca en la evaluación de los valores *p* definidos para cada uno de los parámetros *c* con base a la función objetivo, la cual se enfoca en fijar el valor de la muestra que permite obtener los mejores resultados.

   El procedimiento general del análisis de sensibilidad consiste en evaluar las permutaciones sobre los valores de un parámetro *c*, mientras se mantiene fijo el primer valor de los parámetros restantes, como se muestra en la tabla 5.7. Para cada permutación se realizan 30 ejecuciones hasta cubrir el total de permutaciones de un parámetro *c*. El valor que permita obtener el mejor resultado sobre la media de las 30 pruebas, permitirá determinar el valor del parámetro que será fijado.

Tabla 5.7 *Ejemplo 1. Análisis de sensibilidad de* ***α***, *manteniendo fijos los valores de los demás parámetros.*

Análisis de Sensibilidad para α

Permutación Evaluada	**1**	**2**	**3**	**…**	*v*
α	**0.2**	**0.3**	**0.4**	**…**	**2**
β	0.2	0.2	0.2	0.2	0.2
ρ	0.1	0.1	0.1	0.1	0.1
Colonia	10	10	10	10	10
q0	0.2	0.2	0.2	0.2	0.2
τ0	$1/(n*L_{nm})$	$1/(n*L_{nm})$	$1/(n*L_{nm})$	$1/(n*L_{nm})$	$1/(n*L_{nm})$
TamPop	100	100	100	100	100
$T_{muestra}$	2	2	2	2	2
ProbCruce	0.4	0.4	0.4	0.4	0.4
CromMuta	3	3	3	3	3
ProbMuta	0.01	0.01	0.01	0.01	0.01
MaxGen	100	100	100	100	100

En la tabla 5.7 cada una de las permutaciones de α se ejecuta 30 veces. El valor de la permutación que obtenga el mejor resultado como media de las

30 ejecuciones con respecto a la función objetivo del problema queda fijado para repetir el proceso con el siguiente parámetro.

Una vez definido el valor del primer parámetro c_1, se procede a aplicar el mismo procedimiento al segundo parámetro c_2, mientras se mantienen fijos los valores de los demás parámetros, ejecutando 30 veces cada permutación, como se muestra en la tabla 5.8.

Tabla 5.8 *Ejemplo 2. Análisis de sensibilidad de **β**, manteniendo fijos los valores de los demás parámetros.*

Análisis de Sensibilidad para α

Permutación Evaluada	1	2	3	…	v
α	**1.8**	**1.8**	**1.8**	**1.8**	**1.8**
β	**0.2**	**0.3**	**0.4**	**…**	**2.0**
ρ	0.1	0.1	0.1	0.1	0.1
Colonia	10	10	10	10	10
q0	0.2	0.2	0.2	0.2	0.2
τ0	$1/(n*L_{nm})$	$1/(n*L_{nm})$	$1/(n*L_{nm})$	$1/(n*L_{nm})$	$1/(n*L_{nm})$
TamPop	100	100	100	100	100
$T_{muestra}$	2	2	2	2	2
ProbCruce	0.4	0.4	0.4	0.4	0.4
CromMuta	3	3	3	3	3
ProbMuta	0.01	0.01	0.01	0.01	0.01
MaxGen	100	100	100	100	100

El procedimiento explicado anteriormente se repite hasta que todos los parámetros hayan fijado su valor, lo que al final corresponde a la configuración que estadísticamente produce los mejores resultados. El número de permutaciones P_{total} requeridas para determinar la mejor combinación, se encuentra dada por la fórmula 28.

$$P_{total} = V \in c_1 + V \in c_2 + \cdots + V \in c_p \quad (28)$$

Una vez encontrada la combinación que favorece el desempeño del algoritmo, se procedió a realizar un proceso de *refinamiento*, donde se toman los valores fijados para definir rangos más pequeños que permitan el manejo de valores con una menor dispersión. Para la aplicación de este procedimiento, se aplica la misma metodología explicada anteriormente, a partir del paso 2. La diferencia radica en que los rangos y tamaño de las muestras deben recalcularse, esta vez tomando valores más pequeños, que permitan obtener una sintonización de los parámetros más precisa.

Análisis de Sensibilidad Distribuido

La sintonización distribuida aplicada al AGC-VRPTW consiste en realizar una distribución de procesos de forma uniforme sobre la infraestructura de la Grid Morelos, donde los procesos corresponden a las permutaciones de un conjunto de valores $V = \{p_1, p_2, p_3, \ldots, p_v\}$ a evaluar, correspondientes a cada uno de los parámetros $c \in P$. Para llevar a cabo una distribución uniforme es importante tomar en cuenta las características de la infraestructura, la cual se encuentra descrita en la sección 6.1, tomando en cuenta el número de nodos en cada clúster, así como la cantidad de núcleos en cada nodo.

Uno de los problemas al realizar una sintonización en un clúster o Grid es la distribución de los procesos. Bajo este esquema, se cuenta con tres tipos básicos de distribución con respecto a la ocupación de la infraestructura. La *distribución ideal* que corresponde a ejecutar un número de procesos igual al total de núcleos disponibles en la Grid, por lo que cada núcleo de procesamiento deberá ejecutar una sola permutación a la vez. La *distribución con sobrecarga* se presenta cuando se lanza un mayor número de procesos que núcleos disponibles, por lo que cada núcleo deberá ejecutar un conjunto de procesos. En este caso, cada núcleo de procesamiento espera a terminar el proceso actual para que un nuevo proceso le sea asignado. Finalmente, la *distribución insuficiente*, donde se tiene mayor número de núcleos de

procesamiento disponibles que de procesos a distribuir, por lo que algunos de los núcleos permanecerán ociosos.

Otro problema a considerar es la generación de números aleatorios, ya que una de las características de un clúster o Grid es que la configuración de todos los nodos que la componen es la misma, por lo que la fecha y hora es idéntica. En el algoritmo AGC-VRPTW, la generación de números aleatorios se realiza a partir de la hora del sistema, por lo que al lanzar los procesos, los resultados obtenidos como producto de las 30 ejecuciones para una permutación, siempre será los mismos. Para evitar este problema, se utilizó el proceso generador de semillas aleatorias explicado en la sección 4.2.1.

Para el análisis de sensibilidad del AGC-VRPTW, en la mayoría de los casos se trabaja con una distribución con sobrecarga. La cantidad de procesos que pueden ejecutarse de forma simultanea se calcula tomando el número de muestras a evaluar por parámetro, como se muestra en las tablas 5.4 y 5.5, multiplicado por el número de repeticiones requeridas para considerarse una prueba estadísticamente aceptable [Ross, 1999; Pérez et al., 2005].

La distribución de los procesos se lleva a cabo mediante la ejecución de un *script* que se encarga de enviar los procesos a cada uno de los núcleos de procesamiento de forma homogénea, como se muestra en la figura 5.1

El esquema mostrado en la figura 5.1 representa la distribución de procesos realizada por medio de un *script* para el análisis de sensibilidad del AGC-VRPTW sobre un clúster o Grid, donde cada uno de los procesos se encarga de ejecutar una versión secuencial del algoritmo con una permutación diferente correspondiente al parámetro a sintonizar. Bajo este esquema, el proceso maestro ejecuta el script de distribución, el cual manda cada uno de los procesos a un núcleo de procesamiento para ser ejecutado. El núcleo asignado recibe el proceso y ejecuta el algoritmo de

forma secuencial con la configuración correspondiente. Una vez que el proceso es

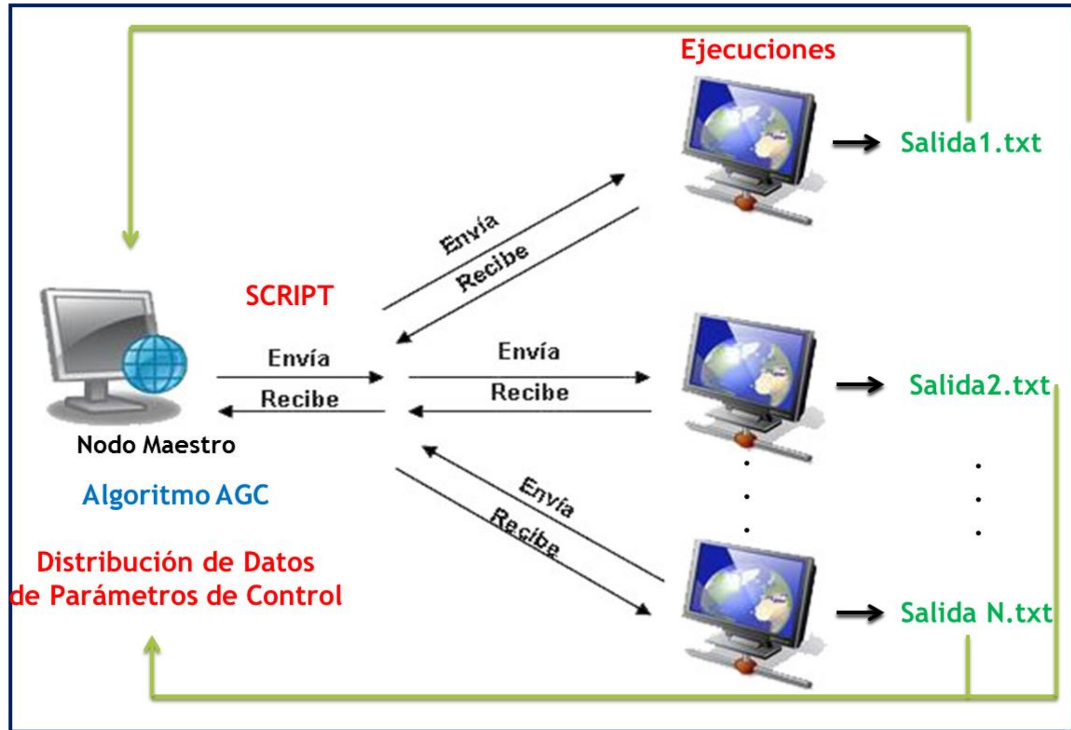


Figura 5.1 *Esquema de distribución de procesos en la Grid utilizado para llevar a cabo el análisis de sensibilidad del AGC-VRPTW.*

terminado, el tiempo de procesamiento, la mejor solución así como la configuración utilizada, se envían a un archivo de texto que indica el parámetro que está siendo evaluado. Cabe mencionar que los resultados de las 30 ejecuciones correspondientes a una misma permutación de un parámetro, se almacenan en un solo archivo con un número consecutivo en el nombre. Esto con la finalidad de facilitar el análisis de los resultados. Una vez que todas las permutaciones V de un parámetro han sido evaluadas, el archivo contiene un listado de los resultados obtenidos con cada permutación, por lo que se procede a identificar el valor que favorece el desempeño del algoritmo y se fija, para continuar con la distribución de procesos para la sintonización del siguiente parámetro.

Cada archivo de texto contiene la configuración correspondiente a la permutación utilizada, así como el resultado de las 30 ejecuciones de cada

permutación, donde se almacena: número de prueba, tiempo de procesamiento, tiempo de ejecución, semilla utilizada y valor de la función objetivo.

Sintonización de Parámetros de Control

El análisis realizado a los resultados del análisis de sensibilidad distribuido explicado anteriormente para el AGC-VRPTW para cada uno de los benchmarks de Solomon evaluados, se muestra en la tabla 5.9.

Tabla 5.9 *Resultados del análisis de sensibilidad del AGC-VRPTW para los benchmarks de Solomon de 100 clientes*

Parámetro	C101	R101	RC101	C201	R201	RC201
α	1.8	1.8	1.8	1.75	1.75	1.75
β	0.2	0.18	0.2	0.22	0.21	0.22
ρ	0.86	0.87	0.87	0.84	0.86	0.86
A	Var	Var	Var	Var	Var	Var
q_0	0.35	0.35	0.35	0.31	0.32	0.32
τ_0	10	10	10	10	10	10
Población	350	350	350	350	350	350
Tmuestra	2	2	2	2	2	2
ProbCruce	0.75	0.78	0.78	0.72	0.75	0.75
CromMuta	3	3	3	3	3	3
ProbMuta	0.26	0.3	0.28	0.25	0.28	0.27
MaxGen	2500	2500	2500	2000	2100	2100

De acuerdo a los resultados mostrados en la tabla 5.9, el valor de los parámetros de control llega a variar de acuerdo al tipo y distribución manejada en cada una de las instancias.

Un comportamiento similar se presenta en el caso de los benchmarks de Gehring y Homberger de 1000 clientes, donde para el proceso de sintonización se

seleccionaron las instancias a evaluar bajo las mismas condiciones que en el caso de los benchmarks de Solomon. Los valores sintonizados se muestran en la tabla 5.10.

Tabla 5.10*. Resultados del análisis de sensibilidad del algoritmo secuencial AGC-VRPTW para los benchmarks de Gehring y Homberger de 1000 clientes*

Parámetro	C1_10_1	R1_10_1	RC1_10_1	C2_10_1	R2_10_1	RC2_10_1
α	1.9	1.88	1.9	1.8	1.82	1.8
β	0.2	0.2	0.2	0.25	0.25	0.25
ρ	0.86	0.87	0.86	0.85	0.85	0.85
A	Var	Var	Var	Var	Var	Var
q_0	0.4	0.42	0.4	0.39	0.42	0.4
τ_0	10	10	10	10	10	10
Población	1000	1000	1000	1000	1000	1000
Tmuestra	2	2	2	2	2	2
ProbCruce	0.68	0.7	0.7	0.67	0.69	0.69
CromMuta	8	8	8	7	7	7
ProbMuta	0.25	0.3	0.32	0.28	0.32	0.3
MaxGen	4500	4500	4500	4400	4400	4400

Los valores sintonizados de los parámetros de control mostrados en las tablas 5.9 y 5.10, fueron utilizados en las pruebas experimentales para evaluar el desempeño del algoritmo, debido a que fueron los valores que permitieron obtener mejores resultados para las instancias utilizadas.

5.2 Algoritmo Genético Cooperativo Distribuido AGCD-VRPTW

El modelo desarrollado para la versión distribuida del algoritmo propuesto AGCD-VRPTW utiliza la misma sintonización que el algoritmo secuencial, debido a que en esencia se trata del mismo algoritmo. La diferencia radica en que el algoritmo AGCD-VRPTW implementa el modelo de islas con migración implícita explicado en la sección 4.2.3, donde el proceso de migración juega un papel fundamental en el proceso evolutivo. En la literatura, de forma general la migración se lleva a cabo cada

cierto tiempo o cada cierto número de generaciones, donde en cada intervalo, cada una de las islas hace un envío sincronizado de un porcentaje de su subpoblación a otra isla [Whitley et al., 1998], la cual se define de acuerdo a la topología implementada. Esto con la finalidad de evitar la pérdida de diversidad en las subpoblaciones. En este trabajo, se propone una modificación a este esquema, la cual se detalla en la sección 4.2.

Utilizando la metodología explicada en la sección 5.1 se realizó un análisis sobre la importancia de los parámetros requeridos para el modelo de islas con migración implícita propuesto en este trabajo de investigación, utilizando las instancias representativas de Gehring y Homberger previamente evaluadas.

1. ***Identificar Parámetros de Control***. De acuerdo al modelo planteado, la parte que difiere del modelo secuencial corresponde a la migración propuesta para el modelo de islas, donde las subpoblaciones se juntan en una población pancmítica para aplicar los operadores genéticos descritos en la sección 4.2.2 en generaciones alternadas.

   Para evitar afectar la convergencia del algoritmo debido a una migración implícita frecuente, fue necesario identificar los parámetros de control fundamentales para la aplicación de dicho de proceso, los cuales se muestran en la tabla 5.11.

Tabla 5.11. *Parámetros de control a sintonizar para el modelo de islas con migración implícita*

Procedimiento	Parámetros de Control
Migración Implícita	$ProbCruce_{Migra}$, $ProbMuta_{Migra}$

   Debido a que se aplican los mismos operadores genéticos en cada una de las islas y en la migración implícita, se observó que era necesario cambiar dos parámetros fundamentales, que son la *probabilidad de cruce* y la *probabilidad de mutación* aplicadas durante el proceso de migración, para lo cual se realizó un análisis de sensibilidad.

2. ***Establecer Rangos***. Una vez identificados los parámetros de control a sintonizar, se procedió a definir los rangos correspondientes, para lo cual se realizó una reducción de los rangos iniciales, tal y como se muestra en la tabla 5.12.

Tabla 5.12. *Rangos establecidos para los parámetros de control utilizados durante la migración implícita.*

Parámetro	Min.	Max.
$\mathbf{\mathit{ProbCruce}}_{\mathit{Migra}}$	0.05	0.5
$\mathbf{\mathit{ProbMuta}}_{\mathit{Migra}}$	0.01	0.4

3. ***Tamaño de Muestras***. Con respecto a los rangos establecidos en el punto 2, se definió el tamaño de las muestras a evaluar en el AGC**D**-VRPTW. Los tamaños, así como el número de muestras evaluadas por cada parámetro se observan en la tabla 5.13.

Tabla 5.13. *Tamaño y cantidad de muestras a evaluar en el análisis de sensibilidad de la migración implícita*

Parámetro	Incremento	No. de Muestras
$\mathbf{\mathit{ProbCruce}}_{\mathit{Migra}}$	0.05	10
$\mathbf{\mathit{ProbMuta}}_{\mathit{Migra}}$	0.03	14

De modo que los valores utilizados en el análisis de sensibilidad correspondiente a cada uno de los parámetros, se muestran en la tabla 5.14.

Tabla 5.14. *Valores utilizados para cada uno de los parámetros de control de la migración implícita*

Parámetro	Incremento
$\mathbf{\mathit{ProbCruce}}_{\mathit{Migra}}$	0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5
$\mathbf{\mathit{ProbMuta}}_{\mathit{Migra}}$	0.01, 0.04, 0.07, 0.1, 0.13, 0.16, 0.19, 0.22, 0.25, 0.28, 0.31, 0.34, 0.37, 0.4

4. ***Proceso de Sintonización de Parámetros***. De la misma forma como se explica este proceso en el punto 4 de la sección 5.1, se evalúa cada una de las

permutaciones *V* con los valores definidos en la tabla 5.14, como se muestra en la tabla 5.15.

Tabla 5.15. *Ejemplo del análisis de sensibilidad de **ProbCruce**, manteniendo fijo el valor de ProbMuta.*

Análisis de Sensibilidad para *ProbCruce*

Permutación Evaluada	**1**	**2**	**3**	**…**	*v*
***ProbCruce**$_{Migra}$*	**0.05**	**0.1**	**0.15**	**…**	**0.5**
***ProbMuta**$_{Migra}$*	0.015	0.015	0.015	…	0.01

Cada permutación se ejecuta 30 veces y este procedimiento se repite hasta que todas las permutaciones *V* del parámetro hayan sido evaluadas, de modo que ambos parámetros hayan sido fijados, lo que indica que se han obtenido los valores de sintonización, mismos que se muestran en la tabla 5.16.

Tabla 5.16 *Valores sintonizados para los parámetros mostrados en la tabla 5.11 para las instancias representativas de los benchmarks de Gehring y Homberger.*

Parámetro	**C1_10_1**	**R1_10_1**	**RC1_10_1**	**C2_10_1**	**R2_10_1**	**RC2_10_1**
***ProbCruce**$_{Migra}$*	**0.2**	**0.2**	**0.2**	**0.15**	**0.15**	**0.15**
***ProbMuta**$_{Migra}$*	**0.015**	**0.015**	**0.015**	**0.01**	**0.01**	**0.01**

Con base a lo observado en la sintonización de parámetros de la tabla 5.16, se puede decir que para una migración implícita aplicada en generaciones intercaladas, es necesario reducir considerablemente tanto la probabilidad de cruce como la de mutación. Esto permite que exista un intercambio mínimo de genes entre los individuos de la población pancmítica, favoreciendo gradualmente la diversidad entre los individuos de cada isla, así como la convergencia de las subpoblaciones.

Capítulo 6

Pruebas Experimentales y Análisis de Resultados

En este capítulo se explica el proceso de experimentación y análisis de resultados de cada una de las tres versiones del algoritmo propuesto explicadas en el capítulo 4, esto con la finalidad de mostrar el desempeño de cada una de las versiones tanto en términos de eficiencia como de eficacia.

La distribución de este capítulo inicia con la sección 6.1, la cual se encuentra enfocada a la descripción de la infraestructura utilizada, correspondiente a la Grid Morelos. La sección 6.2 explica la estructura de las instancias utilizadas en las pruebas experimentales. La sección 6.3 presenta la sintonización del algoritmo, así como el análisis de resultados del algoritmo secuencial AGC-VRPTW. Posteriormente, se presenta la sintonización y resultado de las pruebas experimentales de eficacia y eficiencia en Grid para la versión distribuida AGC**D**-VRPTW del algoritmo propuesto. Finalmente, se presentan los resultados el algoritmo paralelo-distribuido AGCP-VRPTW en Grid, haciendo énfasis en el análisis del efecto de las comunicaciones y accesos a memoria de la GPU.

6.1 Infraestructura Grid Morelos

Los continuos avances tecnológicos han favorecido considerablemente la capacidad de cómputo, enfocándose en el desarrollo de procesadores cada vez más rápidos, así como en el uso de las tarjetas gráficas como unidades de procesamiento. Pero otro factor primordial en el manejo de infraestructura de alto rendimiento es la conectividad. El tiempo requerido para el transporte de datos es pieza fundamental, ya que tanto el diseño del algoritmo como la velocidad de la red influyen en el cálculo de la latencia y por ende en la eficiencia del algoritmo.

Debido a la necesidad de resolver instancias cada vez más grandes para diversos problemas considerados difíciles de resolver, los requerimientos de cómputo se hacen cada vez mayores. Los clústeres de computadores corresponden a una infraestructura de supercómputo de alto rendimiento que surge como una alternativa robusta y escalable de menor costo comparado con las supercomputadoras. Con el tiempo, esta tecnología ha ido evolucionando hasta incluir el manejo de GPUs y de la computación Grid, mejor conocida como *Grid Computing*, con lo que dio paso al manejo de la programación paralela y paralelo-distribuida, lo cual favorece el manejo y procesamiento de mayor cantidad de información.

Una *Grid multi-clúster* se puede definir formalmente como el conjunto *CS* de clústeres de alto rendimiento $CS=\{clúster_1, clúster_2, ..., clúster_s\}$, los cuales se encuentran dispersos geográficamente y conectados mediante una red de alta velocidad que permita reducir el tiempo requerido por las comunicaciones. Por su parte, cada uno de los clústeres $cs \in CS$ está compuesto por un conjunto de nodos *ND* que a su vez cuentan con uno o más núcleos de procesamiento. Los nodos *ND* que se conforman un clúster pueden ser homogéneos o heterogéneos. En caso de que los nodos cuenten con las mismas características en cuanto a arquitectura, velocidad y número de núcleos se dice que se trata de un clúster homogéneo, de lo contrario se considera heterogéneo. De modo que algunas de las características que presenta una Grid son:

Algunas de sus características de use enlistan a continuación:

- Integra recursos dispersos geográficamente
- Escalabilidad
- Flexibilidad
- Permite la ejecución de algoritmos distribuidos y paralelos de acuerdo al tipo de procesadores en su infraestructura.
- De acuerdo al diseño, permite obtener algoritmos más eficientes

De acuerdo a la literatura, este tipo de infraestructura se considera como una de las mejores opciones para el tratamiento de problemas considerados difíciles de resolver, ya que han demostrado obtener mejoras significativas en la eficiencia [Aida, 2005; Melab et al., 2006; Kouki et al., 2011] para el tratamiento de instancias consideradas grandes.

De acuerdo a las características de una plataforma Grid, el número óptimo de procesos a ejecutar de forma simultánea es directamente proporcional al número de núcleos de procesamiento disponibles.

La Grid Morelos[5] es una infraestructura de alto rendimiento que surgió como resultado del auge que han tomado las tecnologías Grid en los últimos años, las cuales se han vuelto parte fundamental del tratamiento de problemas tanto científicos como industriales. En la actualidad, la Grid Morelos es una infraestructura en crecimiento para proyectos de investigación de e-Ciencia, donde se conjuntan instituciones educativas como la UAEM (Universidad Autónoma del Estado de Morelos), la UPEMor (Universidad Politécnica del Estado de Morelos) y el ITVer (Instituto Tecnológico de Veracruz).

Debido a las características de cada uno de los clústeres integrados a la Grid y tomando en cuenta las características del algoritmo propuesto en este trabajo de investigación, se decidió utilizar únicamente el clúster *cuexcomate*, ubicado en la UAEM y el *texcal*, localizado en la UPEMor, ya que son los únicos que cuentan con unidades gráficas de procesamiento (GPUs). Cabe mencionar que el clúster *cuexcomate* y el *texcal* cuentan con las mismas características (tabla 6.1), por lo que se puede decir que en esta tesis se está trabajando con clústeres homogéneos en Grid.

Tabla 6.1 *Infraestructura de Hardware de los clústeres Cuexcomate y Texcal*

Elemento	Hardware
Comunicaciones	Switch 3COM 24/10/100/1000 Switch Infiniband Mellanox de 18 puertos de 40 Gb/s QDR
Nodo maestro *CPU*	1 Motherboard: • 2 procesadores Intel Xeon Six Core a 3.06 GHz, 12 MB cache.

[5] *Grid Morelos (s.f). http://www.gridmorelos.uaem.mx/index.php?option=com_content&view=article&id=25&Itemid=28*

Total 12 cores Total 24 GB RAM Total 12 TB HD	• 2 HD Enterprise, 7200 RPM de 500GB (para S.O.). • 6 HD Enterprise, 7200 RPM, 12 TB en total. • 6 módulos RAM de 4GB 1333MHZ DDR3. Total de 24 GB RAM. • 1 tarjeta Infiniband 40Gb/s
Nodos de procesamiento CPU 01 al 04 Total 48 cores Total 96 GB RAM Total 2 TB HD	1 Motherboard: • 2 procesadores Intel Xeon Six Core a 3.06 GHz, 12 MB cache. • 1 HD Enterprise, 7200 RPM de 500GB. • 6 módulos RAM de 4GB 1333MHZ DDR3. Total de 24 GB RAM. 1 tarjeta Infiniband 40Gb/s
Nodo de procesamiento GPU 05 Total 896 cores Total 36 GB RAM Total 1 TB HD	1 Motherboard: • 1 procesador Intel Xeon Six Core a 3.06 GHz, 12 MB cache. • 2 HD Enterprise, 7200 RPM de 500GB. • 9 módulos RAM de 4GB 1333MHZ DDR3. Total de 36 GB RAM. • 2 tarjetas NVIDIA TESLA C2070, arquitectura Fermi, con 6 GB RAM DDR5 c/u. 448 cores c/u. • DVD/RW • Lector de memoria 1 tarjeta Infiniband 40Gb/

En resumen, los clústeres utilizados en Grid cuentan con 66 núcleos de procesamiento CPU cada uno, haciendo un total de 132, con las características mostradas en la tabla 6.1. En el caso de los GPUs, cada clúster cuenta con dos tarjetas con un total de 896 cores en cada clúster, teniendo un total de 1792 núcleos de procesamiento gráfico. Los recursos contenidos en la Grid Morelos se encuentran distribuidos como se muestra en la tabla 6.2.

Tabla 6.2 *Distribución de recursos en la Grid Morelos por clúster.*

Nodo	Cuexcomate (cores)	Nodo	Texcal (cores)
cuexcomate	12	**texcal**	12
ciicap01	12	**node01**	12
ciicap02	12	**node02**	12
ciicap03	12	**node03**	12
ciicap04	12	**node04**	12

ciicap-gpu01	6	gpu01	6
TOTAL	**66**	**TOTAL**	**66**

6.2 Instancias de Prueba

Las instancias de prueba utilizadas para realizar las pruebas experimentales de las tres versiones del algoritmo propuesto son los bien conocidos benchmarks de Solomon y los bechmarks de Gehring y Homberger, que son una extensión de los benchmarks de Solomon para la inclusión de instancias más grandes. Las características y estructura de dichas instancias se presentan en la sección 2.4.

La selección de instancias utilizadas en las pruebas experimentales, se llevó a cabo con base en su tamaño y complejidad. Para las pruebas experimentales del algoritmo secuencial AGC-VRPTW se utilizaron los 56 benchmarks de Solomon de 100 clientes y los 60 benchmaks de Gehring y Homberger de 1000 clientes.

En el caso de la versión distribuida AGC**D**-VRPTW y de la versión paralelo-distribuida AGC**P**-VRPTW, se trabajó únicamente con los benchmarks de Gehring y Homberger de 1000 clientes debido a que se busca la explotación de los recursos de la Grid, lo cual no sería posible con el uso de instancias más pequeñas, como es el caso de los de Solomon de 100 clientes.

Como ya se mencionó anteriormente, los benchmarks de Gehring y Homberger corresponden a una extensión de los benchmarks de Solomon, por lo que cuentan con las mismas clasificaciones, lo cual se muestra en la tabla 6.3.

Tabla 6.3 *Clasificación de las instancias utilizadas (Solomon y Gehring y Homberger).*

Tipo 1 **(Restricciones más duras)**	**Tipo 2** **(Restricciones con menor dureza)**
C (por sus siglas en inglés *Clustered*)	**C** (por sus siglas en inglés *Clustered*)
R (por sus siglas en inglés *Random*)	**R** (por sus siglas en inglés *Random*)
RC (por sus siglas en inglés *Random-Clustered*)	**RC** (por sus siglas en inglés *Random-Clustered*)

6.3 *Algoritmo AGC-VRPTW*

En esta sección se presentan los resultados obtenidos para el algoritmo secuencial AGC-VRPTW, utilizando los bechmarks de Solomon de 100 clientes y los de Gehring y Homberger de 1000 clientes. El análisis presentado inicia con la evaluación de la convergencia del algoritmo propuesto en la sección 6.3.1. Posteriormente, la sección 6.3.2 presenta los resultados obtenidos en las pruebas de eficacia comparados con las mejores cotas reportadas en la literatura. Finalmente, se presenta el análisis de eficiencia, donde se evalúa el tiempo requerido para obtener una solución al VRPTW.

6.3.1 Convergencia del Algoritmo

La *convergencia* es una propiedad que permite identificar en qué punto un algoritmo alcanza la estabilidad mediante una progresión hacia la uniformidad en los resultados obtenidos. Para el algoritmo AGC-VRPTW se utilizaron las instancias descritas en la sección 2.4 y 6.2. Las diferencias existentes entre las instancias debido a la dureza de sus restricciones y a su distribución provocan que se comporten de forma diferente durante el proceso de optimización, de modo que para la evaluación de la convergencia se utilizaron instancias representativas. Para el caso de los benchmarks de Solomon, se utilizaron las instancias C101, R101, RC101, C201, R201 y RC201.

Las pruebas de convergencia se realizaron aplicando los valores sintonizados de los parámetros de control mostrados en la sección 5.1. Para cada una de las instancias se realizaron 30 ejecuciones, la cual es considerada como la muestra mínima aceptable para este tipo de análisis estadístico [Ross, 1999; Pérez et al., 2005]. Los parámetros relevantes para la evaluación de la convergencia son el fitness y el número de generaciones, el cual funge como criterio de paro del algoritmo. El número de generaciones corresponde al parámetro crítico que más afecta el tiempo de procesamiento.

Los resultados de la convergencia para las instancias representativas de los benchmarks de Solomon muestran la mejora continua de la solución hasta llegar a un máximo de generaciones dependiente del tipo de instancia, donde el valor de la función objetivo alcanza la convergencia, manteniéndose constante en las siguientes generaciones, tal y como se muestran en las figuras 6.1.

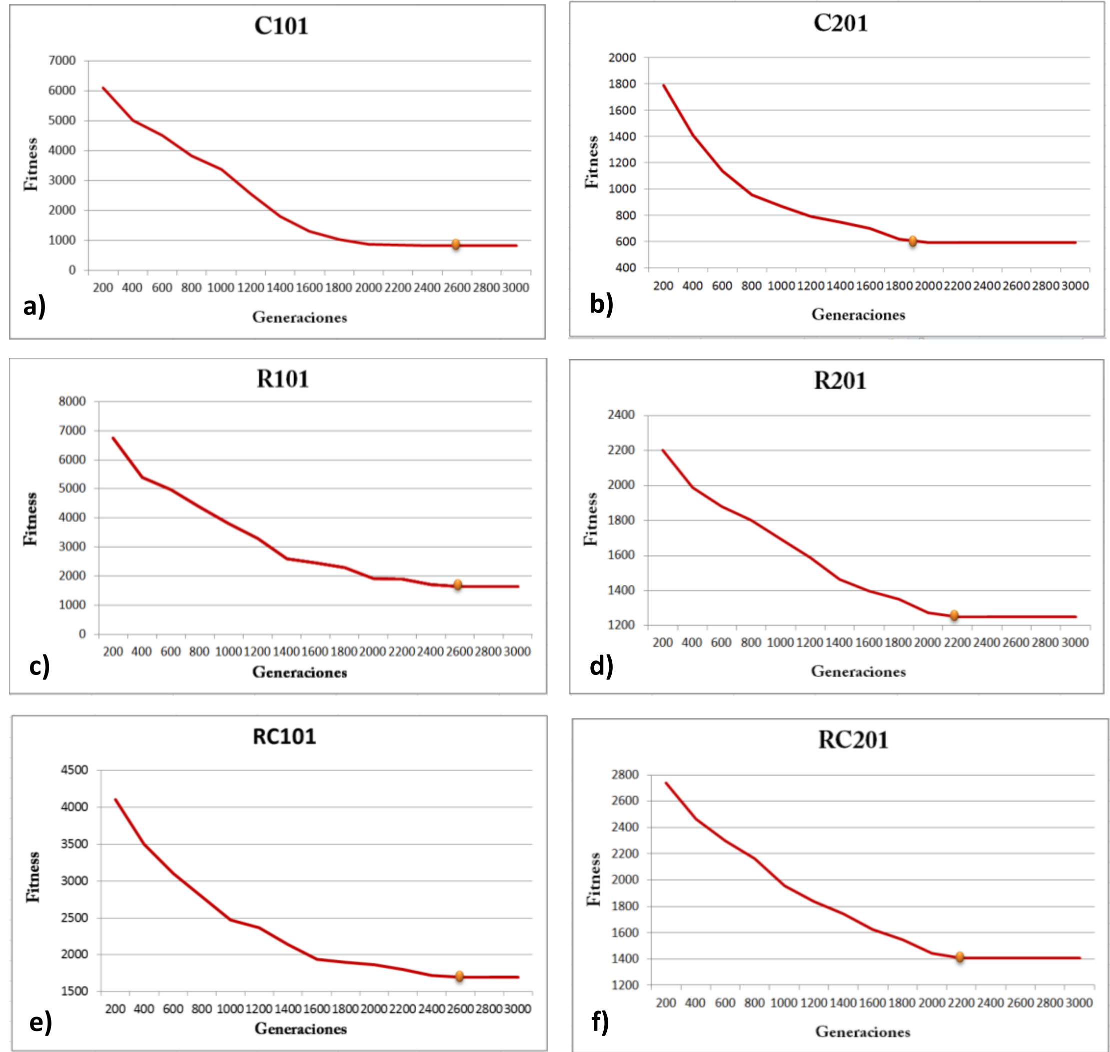


Figura 6.1. *Gráficas de convergencia de las instancias representativas de los benchmark de Solomon.*

En las figuras 6.1 se muestra el comportamiento promedio de la calidad de las soluciones de acuerdo a la función objetivo, con respecto al número de generaciones de las instancias a) C101, b) C201, c) R101, d) R201, e) RC101 y f) RC201 de los

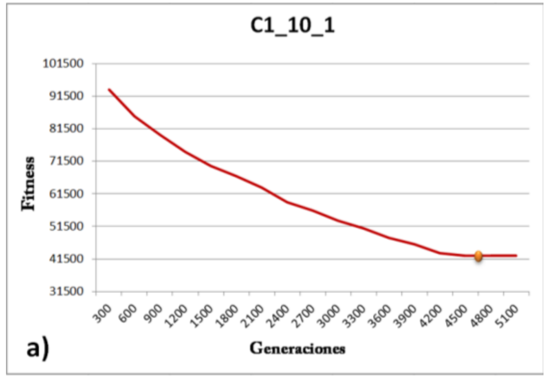
benchmarks de Solomon, donde cada valor corresponde a un promedio de 30 ejecuciones. El punto de convergencia (punto en la gráfica), tomado como el número de generaciones, llega a variar de acuerdo a la dureza de las restricciones y al tipo de distribución, como se muestra en la tabla 6.4.

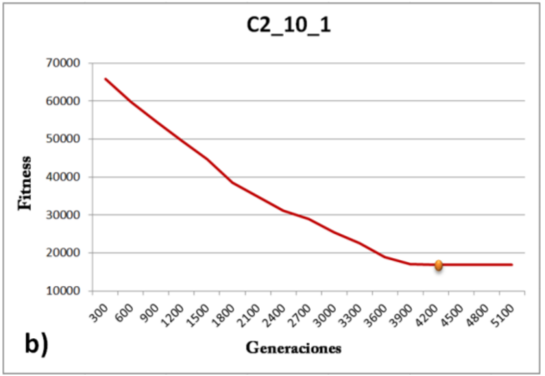
Tabla 6.4 *Punto de convergencia de las instancias representativas de los benchmarks de Solomon, con base en lo mostrado en las figuras 6.1.*

Clasificación	**C**	**R**	**RC**
	Número de Generaciones		
Tipo 1	2500	2500	2500
Tipo 2	2000	2100	2100

Mediante el mismo método de selección, para los benchmarks de Gehring y Homberger se tomaron las instancias C1_10_1, R1_10_1, RC1_10_1, C2_10_1, R2_10_1, RC2_10_1 y se realizó el mismo procedimiento para identificar el punto de convergencia de cada una de las instancias de acuerdo a su tipo y distribución, tomando los parámetros sintonizados en la sección 5.1 con un rango de 300 a 5100 generaciones.

Los resultados del análisis de convergencia para las instancias correspondientes a los benchmarks de Gehring y Homberger de 1000 clientes muestran un comportamiento similar al presentado con los benchmarks de Solomon, como se muestran en las figuras 6.2.





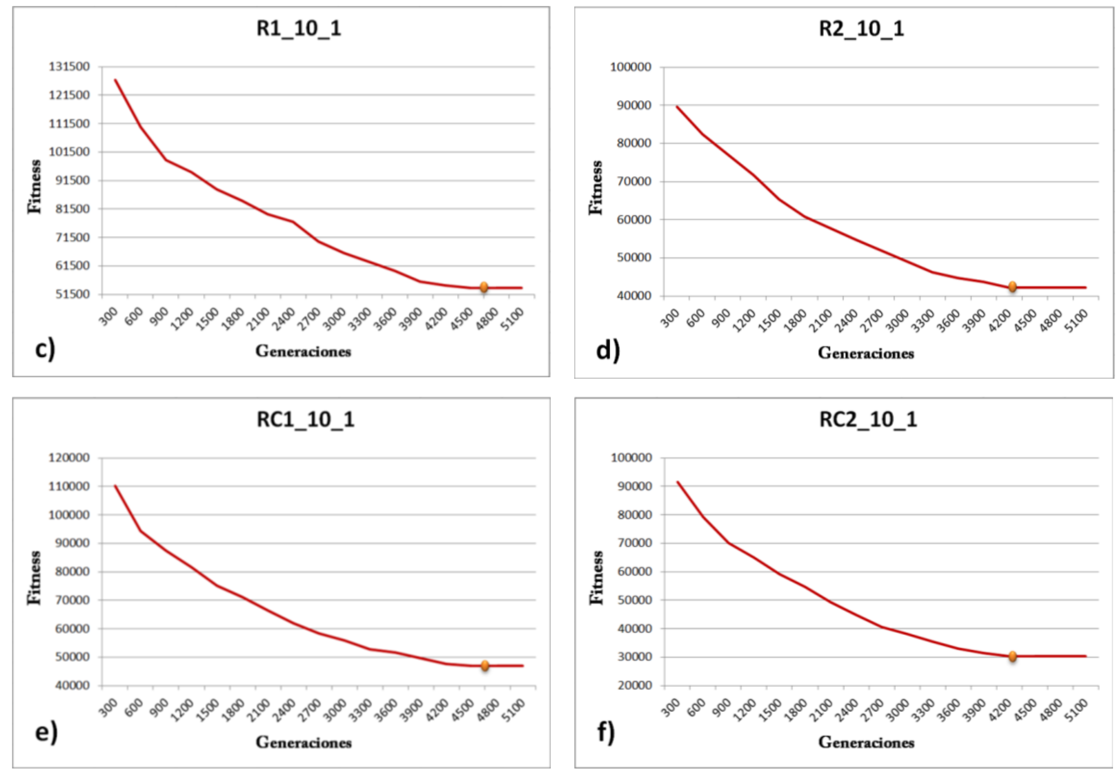


Figura 6.2. *Gráficas de convergencia de las instancias representativas de los benchmark de Gehring y Homberger.*

El conjunto de figuras 6.2 muestran el comportamiento del valor (promedio de 30 ejecuciones) de las soluciones de acuerdo a la función objetivo con respecto al número de generaciones de las instancias: a) C1_10_1, b) C2_10_1, c) R1_10_1, d) R2_10_1, e) RC1_10_1 y f) RC2_10_1 de los benchmarks de Gehring y Homberger, donde se puede observar que en algunos casos la reducción en cuanto al valor de la función objetivo se presenta en menor escala, como es el caso de la figura 6.2c correspondiente a la instancia R1_10_1, donde con 2400 generaciones se presenta un comportamiento diferente con respecto al comportamiento general de la gráfica. Esto se debe al grado de aleatoriedad manejado por los operadores (sección 4.1), donde el resultado siempre corresponde a una mejora por mínima que esta sea. De acuerdo a los resultados obtenidos, el punto de convergencia de acuerdo al número de generaciones se muestra en la tabla 6.5.

Tabla 6.5 *Punto de convergencia de las instancias representativas de los benchmarks de Gehring y Homberger, con base en lo mostrado en las gráficas de la figura 6.2.*

Clasificación	**C**	**R**	**RC**
	Número de Generaciones		
Tipo 1	4500	4500	4500
Tipo 2	4400	4400	4400

6.3.2 Análisis de Eficacia

En problemas de optimización combinatoria, el realizar un análisis estadístico es fundamental para observar el comportamiento del algoritmo con respecto a las instancias evaluadas, ya que de esta forma es posible identificar puntos críticos que afectan o benefician el desempeño del algoritmo. Por lo que una vez terminado el proceso de sintonización presentado en la sección 5.1, los valores sintonizados son utilizados para realizar las pruebas de eficacia del algoritmo AGC-VRPTW. Para este proceso de evaluación, se utilizaron los 56 benchmarks de Solomon y los 60 de Gehring y Homberger.

Aunque el algoritmo AGC-VRPTW es secuencial, las pruebas experimentales fueron realizadas de forma distribuida en la Grid Morelos, la cual se describe en la sección 6.1, donde en cada núcleo de procesamiento se llevó a cabo una ejecución del algoritmo totalmente independiente a las realizadas en los demás núcleos. Esto con la finalidad de reducir el tiempo requerido para la realización de las pruebas. Es importante mencionar que para cada uno de los benchmarks utilizados se realizaron 30 pruebas, ya que esta es la muestra mínima aceptable para este tipo de análisis [Ross, 1999; Pérez et al., 2005].

Para el caso de los benchmarks de Solomon, los resultados obtenidos por el algoritmo secuencial se muestran a continuación en la tabla 6.6.

Tabla 6.6 *Análisis estadístico de los resultados obtenidos por el algoritmo secuencial AGC-VRPTW para los benchmarks de Solomon de 100 clientes*

Instancia	Veh	Mejor	Peor	Prom	σ	Veh	Cota	ER (%)
C101	10	**828.94**	828.94	828.94	0	10	**828.94**	0
C102	10	**828.94**	828.94	828.94	0	10	**828.94**	0
C103	10	**828.06**	828.06	828.06	0	10	**828.06**	0
C104	10	**824.78**	824.78	824.78	0	10	**824.78**	0
C105	10	829.12	829.43	828.28	0.895	10	**828.94**	0.0217
C106	10	**828.94**	828.94	828.94	0	10	**828.94**	0
C107	10	**828.94**	828.94	828.94	0	10	**828.94**	0
C108	10	**828.94**	828.94	828.94	0	10	**828.94**	0
C109	10	**828.94**	828.94	828.94	0	10	**828.94**	0
C201	3	**591.56**	591.56	591.56	0	3	**591.56**	0
C202	3	**591.56**	591.56	591.56	0	3	**591.56**	0
C203	3	**591.17**	591.17	591.17	0	3	**591.17**	0
C204	3	**590.6**	590.6	590.6	0	3	**590.6**	0
C205	3	**588.88**	588.88	588.88	0	3	**588.88**	0
C206	3	**588.49**	588.49	588.49	0	3	**588.49**	0
C207	3	**588.29**	588.29	588.29	0	3	**588.29**	0
C208	3	**588.32**	588.32	588.32	0	3	**588.32**	0
R101	19	**1650.8**	1650.8	1650.8	0	19	**1650.8**	0
R102	17	**1486.12**	1486.12	1486.12	0	17	**1486.12**	0
R103	**13**	**1292.66**	**1293.12**	**1292.86**	*	**13**	**1292.68**	**-0.0015**
R104	9	1007.37	1007.93	1007.68	0.640	9	**1007.31**	0.0060
R105	14	**1377.11**	1377.11	1377.11	0	14	**1377.11**	0
R106	12	**1252.03**	1252.03	1252.03	0	12	**1252.03**	0
R107	10	1105.03	1105.36	1105.15	0.351	10	**1104.66**	0.0335
R108	9	**960.88**	960.88	960.88	0	9	**960.88**	0
R109	1	**1194.73**	1194.73	1194.73	0	11	**1194.73**	0
R110	**10**	**1118.81**	**1119.2**	**1118.95**	*	**10**	**1118.84**	**-0.0026**
R111	10	1096.89	1097.16	1096.99	0.288	10	**1096.72**	0.0155
R112	9	**982.14**	982.14	982.14	0	9	**982.14**	0
R201	4	**1252.37**	1252.37	1252.37	0	4	**1252.37**	0
R202	4	**1191.7**	1191.7	1191.7	0	3	**1191.7**	0
R203	3	**939.5**	939.5	939.5	0	3	**939.5**	0
R204	2	**825.52**	825.52	825.52	0	2	**825.52**	0
R205	3	**994.42**	994.42	994.42	0	3	**994.42**	0

R206	3	906.21	906.37	906.26	0.168	3	**906.14**	0.0077
R207	2	**890.61**	890.61	890.61	0	2	**890.61**	0
R208	2	**726.82**	726.82	726.82	0	2	**726.82**	0
R209	3	**909.16**	909.16	909.16	0	3	**909.16**	0
R210	3	**939.37**	939.37	939.37	0	3	**939.37**	0
R211	2	**885.71**	885.71	885.71	0	2	**885.71**	0
RC101	14	**1696.94**	1696.94	1696.94	0	14	**1696.94**	0
RC102	13	**1554.75**	1554.75	1554.75	0	12	**1554.75**	0
RC103	11	**1261.67**	1261.67	1261.67	0	11	**1261.67**	0
RC104	10	**1135.48**	1135.48	1135.48	0	10	**1135.48**	0
RC105	13	**1629.41**	**1629.44**	**1629.42**	*	**13**	**1629.44**	**-0.0018**
RC106	11	**1424.73**	1424.73	1424.73	0	11	**1424.73**	0
RC107	12	1232.04	1232.28	1232.12	0.253	11	**1230.48**	0.1268
RC108	10	**1139.82**	1139.82	1139.82	0	10	**1139.82**	0
RC201	4	**1406.94**	1406.94	1406.94	0	4	**1406.94**	0
RC202	4	**1365.65**	1365.65	1365.65	0	3	**1365.65**	0
RC203	3	**1049.62**	1049.62	1049.62	0	3	**1049.62**	0
RC204	3	**798.46**	798.46	798.46	0	3	**798.46**	0
RC205	4	**1297.65**	1297.65	1297.65	0	4	**1297.65**	0
RC206	4	1146.92	1147.28	1147.12	0.412	3	**1146.32**	0.0523
RC207	4	**1061.14**	1061.14	1061.14	0	3	**1061.14**	0
RC208	3	**828.14**	828.14	828.14	0	3	**828.14**	0

El análisis presentado en la tabla 6.6 muestra que se obtuvieron 3 mejoras a las cotas reportadas en la literatura correspondientes a las instancias R103, R110 y RC105. Por otro lado, el análisis muestra las mejores y peores soluciones obtenidas por el algoritmo secuencial AGC-VRPTW para cada una de las instancias, donde cada valor corresponde al promedio de las 30 ejecuciones de cada instancia, lo cual permite calcular la desviación estándar que corresponde al grado de dispersión entre las soluciones obtenidas el cual indica una dispersión mínima de 0.168 y una máxima de 0.895, lo que indica que los valores obtenidos se encuentran cercanos al promedio de las soluciones, el cual a su vez es cercano a la mejor solución obtenida.

En el análisis se incluye el número de vehículos y costo de las mejores cotas reportadas en la literatura[6]. Finalmente, se calcula el error relativo con base a la mejor solución obtenida con respecto a la mejor cota reportada, aplicando la fórmula 29.

$$ER = \frac{mejorSol - mejorCota}{mejorCota} \qquad (29)$$

De acuerdo al análisis estadístico, se observa que en la mayoría de las instancias, se alcanza la mejor cota conocida, en 7 ocasiones se obtuvieron resultados peores a los reportados, con un error relativo máximo de 0.1268% y mínimo del 0.006%, indicando que los resultados obtenidos son muy cercanos a la mejor cota reportada.

Para probar la escalabilidad del algoritmo, se realizaron pruebas experimentales con instancias más grandes, para lo cual se utilizaron los benchmarks propuestos por Gehring y Homberger de 1000 clientes. Las pruebas experimentales se realizaron bajo las mismas condiciones que en el caso de los benchmarks de Solomon, con la única diferencia que en este caso se tomaron los valores de los parámetros de control sintonizados en la sección 5.1. El análisis estadístico obtenido se muestra en la tabla 6.7.

Tabla 6.7 *Análisis estadístico de los resultados obtenidos por el algoritmo secuencial AGC-VRPTW para los benchmarks de Gehring y Homberger de 1000 clientes.*

Instancia	Veh	Mejor	Peor	Prom	σ	Veh	Cota	ER (%)
C1_10_1	100	**42478.95**	42479.99	42479.49	0.736	100	**42478.95**	0
C1_10_2	90	42280.11	42281.49	42280.45	1.094	90	**42278.45**	0.0039
C1_10_3	90	**40207.71**	40208.14	40207.96	0.308	90	**40207.71**	0
C1_10_4	90	**39468.6**	39469.83	39468.9	0.977	90	**39468.6**	0
C1_10_5	100	**42469.18**	42470.43	42469.65	0.911	100	**42469.18**	0
C1_10_6	100	43834.98	43836.48	43835.26	1.252	99	**43830.21**	0.0109
C1_10_7	97	**43453.92**	43453.92	43453.92	0	97	**43453.92**	0
C1_10_8	**94**	**41853.36**	**42149.58**	**41912.58**	*	**93**	**42149.58**	**-0.07**

[6] TOP-VRPTW. http://www.sintef.no/Projectweb/TOP/VRPTW/Solomon-benchmark/100-customers/. Actualizada en Noviembre, 2014

C1_10_9	91	40571.2	40574.24	40573.19	2.250	90	**40570.6**	0.0015
C1_10_10	90	**39933.06**	39934.21	39933.85	0.868	90	**39933.06**	0
C2_10_1	30	**16879.24**	16882.47	16880.2	2.465	30	**16879.24**	0
C2_10_2	29	**17126.39**	17126.85	17126.64	0.326	29	**17126.39**	0
C2_10_3	28	**16884.08**	16884.58	16884.34	0.354	28	**16884.08**	0
C2_10_4	28	**15656.75**	15657.96	15657.31	0.858	28	**15656.75**	0
C2_10_5	30	**16561.29**	16562.91	16561.79	1.227	30	**16561.29**	0
C2_10_6	30	16921.43	16923.03	16922.09	1.149	29	**16920.33**	0.0065
C2_10_7	29	**17882.42**	17882.42	17882.42	0	29	**17882.42**	0
C2_10_8	29	16579.93	16580.82	16580.23	0.662	28	**16577.32**	0.0157
C2_10_9	29	**16370.44**	16371.87	16370.59	1.289	29	**16370.44**	0
C2_10_10	28	**15944.72**	15946.02	15945.32	0.922	28	**15944.72**	0
R1_10_1	100	**53560.85**	53561.15	53560.99	0.213	100	**53560.85**	0
R1_10_2	92	49107.96	49109.14	49108.69	0.858	91	**49105.21**	0.0056
R1_10_3	91	**45237.29**	45239.48	45238.01	1.637	91	**45237.29**	0
R1_10_4	91	42788.83	42789.39	42788.98	0.437	91	**42787.19**	0.0038
R1_10_5	92	51832.47	51832.96	51832.69	0.348	91	**51830.36**	0.0041
R1_10_6	91	**47849.05**	47849.95	47849.49	0.637	91	**47849.05**	0
R1_10_7	91	**44435.5**	44437.24	44436.12	1.280	91	**44435.5**	0
R1_10_8	91	42487.25	42487.98	42487.55	0.524	91	**42485.38**	0.0044
R1_10_9	91	**50490.49**	50492.9	50491.87	1.722	91	**50490.49**	0
R1_10_10	91	**48294.71**	48295.88	48294.89	1.006	91	**48294.71**	0
R2_10_1	19	**42188.86**	42188.86	42188.86	0	19	**42188.86**	0
R2_10_2	19	33514.01	33515.12	33514.79	0.847	19	**33512.83**	0.0035
R2_10_3	19	**24940.32**	24940.32	24940.32	0	19	**24940.32**	0
R2_10_4	19	**17926.45**	17926.45	17926.45	0	19	**17926.45**	0
R2_10_5	19	36232.97	36233.29	36233.11	0.228	19	**36232.18**	0.0022
R2_10_6	19	**30091.93**	30091.93	30091.93	0	19	**30091.93**	0
R2_10_7	19	**23257.36**	23257.36	23257.36	0	19	**23257.36**	0
R2_10_8	19	17498.55	17499.89	17498.92	1.038	19	**17495.51**	0.0174
R2_10_9	19	33009.45	33010.25	33009.98	0.595	19	**33002.36**	0.0215
R2_10_10	19	**30215.24**	30216.94	30215.64	1.360	19	**30215.24**	0
RC1_10_1	90	**46272.07**	46272.89	46272.46	0.581	90	**46272.07**	0
RC1_10_2	90	**44129.42**	44129.92	44129.67	0.354	90	**44129.42**	0
RC1_10_3	91	42490.21	42491.32	42490.68	0.794	90	**42487.54**	0.0063
RC1_10_4	91	41615.48	41616.05	41615.87	0.430	90	**41613.58**	0.0046
RC1_10_5	90	**45564.81**	45566.09	45566.39	1.608	90	**45564.81**	0

RC1_10_6	91	**45303.67**	45304.21	45303.88	0.391	90	**45303.67**	0
RC1_10_7	90	**44903.8**	44905.1	44904.39	0.923	90	**44903.8**	0
RC1_10_8	90	44368.65	44369.48	44369.1	0.589	90	**44366.01**	0.0060
RC1_10_9	90	**44280.84**	44282.03	44281.76	0.959	90	**44280.84**	0
RC1_10_10	90	43898.45	43898.85	43898.69	0.288	90	**43896.78**	0.0038
RC2_10_1	**20**	**30278.48**	**30279.34**	**30278.8**	*****	**20**	**30278.5**	**-0.000066**
RC2_10_2	18	**26327.92**	26327.92	26327.92	0	18	**26327.92**	0
RC2_10_3	18	**20050.71**	20050.99	20050.83	0.200	18	**20050.71**	0
RC2_10_4	19	15750.98	15751.88	15751.48	0.640	18	**15747.13**	0.0244
RC2_10_5	19	**27237.68**	27239.6	27238.64	1.358	18	**27237.68**	0
RC2_10_6	19	26799.1	26799.93	26799.65	0.617	18	**26797.76**	0.0050
RC2_10_7	18	**25112.77**	25114.01	25113.49	0.888	18	**25112.77**	0
RC2_10_8	18	**23709.29**	23709.99	23709.43	0.577	18	**23709.29**	0
RC2_10_9	18	**23028.1**	23028.7	23028.41	0.424	18	**23028.1**	0
RC2_10_10	18	21967.99	21969.15	21968.75	0.859	18	**21965.94**	0.0093

Al igual que el caso de los benchmarks de Solomon, las mejores cotas se encuentran reportadas en el TOP.VRPTW[7], donde de manera continua se integran los mejores resultados reportados en la literatura una vez que han sido validados. Los resultados comparados fueron obtenidos con heurísticas diferentes (cuyas referencias pueden ser consultadas en la misma página) pero todas enfocadas al mismo problema, lo que permite que se pueda realizar una comparación con los resultados obtenidos por el algoritmo secuencial AGC-VRPTW.

De acuerdo a los resultados obtenidos en el análisis estadístico, para las instancias grandes de los benchmarks de Gehring y Homberger se obtuvieron dos mejoras a las cotas reportas, las cuales corresponden a las instancias C1_10_8 y RC2_10_1. En el caso de las demás instancias, en la mayoría de los casos se alcanzaron las mejores cotas reportadas en la literatura, con excepción de 20 resultados de 60 donde se obtienen resultados ligeramente peores a las mejores cotas reportadas, mostrando un error relativo máximo de 0.024% y un mínimo de 0.0015%, demostrando una gran cercanía con los mejores resultados conocidos. Por otro lado,

[7] TOP-VRPTW. TOP-VRPTW. http://www.sintef.no/Projectweb/TOP/VRPTW/Gehring-Homberger/ benchmark /1000-customers/.
Actualizada en Enero, 2015

la dispersión mostrada en los resultados obtenidos muestra que el máximo valor es de 1.722 y el mínimo de 0.200, indicando que para instancias grandes existe una dispersión ligeramente mayor que para instancias pequeñas, lo que muestra un comportamiento similar del algoritmo tanto para instancias pequeñas como para instancias grandes, demostrando la eficacia, robustez y escalabilidad del mismo.

6.3.3 Análisis de Eficiencia

Benchmarks de Solomon

El análisis de eficiencia de un algoritmo implica identificar la cantidad de recursos requeridos para su ejecución, lo cual se encuentra en función del tamaño de la entrada de datos, por lo que se tienen dos tipos de recursos:

- ***Espaciales:*** Se enfoca en la cantidad de recursos necesarios para almacenar los datos requeridos por el algoritmo.
- ***Temporales:*** Se centra en el tiempo requerido por el algoritmo para ejecutar todas las instrucciones que lo componen.

Este tipo de análisis permite tener un acercamiento a la cantidad de recursos mínimos con los que se tiene que contar para la ejecución del algoritmo. Por lo que si un algoritmo requiere gran cantidad de recursos, se dice que no es eficiente y por lo tanto no es apto para ser aplicado al problema tratado, por el contrario, un algoritmo eficiente permite obtener soluciones rápidamente o en un tiempo computacional razonable.

Con base en los conjuntos de instancias evaluados en la sección anterior, se realizaron pruebas experimentales que permitan medir los tiempos de procesamiento requeridos por cada una de ellas. Las pruebas experimentales fueron ejecutadas en la infraestructura de la Grid Morelos (sección 6.1) tomando cada uno de los nodos de procesamiento como un proceso independiente que ejecuta el algoritmo secuencial

AGC-VRPTW con instancias distintas. Esto se realizó con la finalidad de explotar las características de la grid para reducir los tiempos requeridos para la ejecución total de las instancias, ya que al igual que en las pruebas de eficacia, para cada una de las instancias se realizaron 30 ejecuciones.

El tiempo de ejecución de un algoritmo para un problema específico define su eficiencia, es decir, que tan rápido es el algoritmo para encontrar una solución para el problema tratado. La eficiencia es la medida que permite conocer la cantidad de recursos computacionales requeridos por un algoritmo, en función del tamaño de la entrada, donde los recursos más preciados son el tiempo y los recursos computacionales. El tiempo computacional juega un papel fundamental en el diseño de algoritmo, ya que de ello depende que un algoritmo sea aplicable o no a cierto problema.

Para llevar a cabo el análisis de eficiencia, en primera instancia se realizó un análisis de los tiempos de ejecución del algoritmo AGC-VRPTW con respecto a las instancias representativas de los benchmarks de Solomon utilizadas durante el análisis de sensibilidad (sección 5.1). Cabe mencionar que los resultados obtenidos en la tabla 6.8 corresponden al promedio de 30 ejecuciones por instancia.

Tabla 6.8 *Promedio de los tiempos de ejecución del AGC-VRPTW, aplicado a las instancias representativas de los benchmarks de Solomon.*

Instancia	Tiempo de ejecución (segs.)
C101	168
R101	191
RC101	172
C201	150
R201	169
RC201	158

Para el caso de los benchmarks de Solomon, la tabla 6.8 muestra los tiempos de ejecución requeridos para las instancias representativas de los benchmarks de

Solomon. Dichos tiempos se comparan con algunos de los algoritmos que han reportado las mejores cotas. Es importante mencionar que el contenido de la tabla 6.9 es únicamente de carácter informativo, ya que debido a que los algoritmos fueron ejecutados bajo diferentes condiciones, estos no pueden ser comparados en términos de eficiencia.

Tabla 6.9 *Tabla comparativa de tiempos de ejecución para las instancias representativas de Solomon.*

Instancia	Tiempo (segs.)					
	BBB[8]	**BVH**[9]	**GTA**[10]	**RGP**[11]	**RT**[12]	**AGC-VRPTW**
C101	120-180	1800-7200	100-1800	11000	540	**168**
R101	120-180	1800-7200	100-1800	11000	450	**191**
RC101	120-180	1800-7200	100-1800	11000	430	**172**
C201	120-180	1800-7200	100-1800	11000	1600	**150**
R201	120-180	1800-7200	100-1800	11000	1200	**169**
RC201	120-180	1800-7200	100-1800	11000	1300	**158**

El algoritmo BBB corresponde a un algoritmo genético híbrido que fue ejecutado en una Pentium a 400 MHz y 128 MB en RAM. Por su parte, el algoritmo BVH es un algoritmo híbrido de dos-fases que fue ejecutado en una máquina SunUltra10 a 440MHz. En el caso del algoritmo GTA correspondiente a un algoritmo de optimización por colonia de hormigas, este fue ejecutado en una máquina SunUltraSparc1 a l67 MHz. Para las pruebas experimentales del algoritmo RGP se utilizó una máquina SunUltra, al igual que en el caso del algoritmo BVH, se utilizó una estación de trabajo SunUntra10. Finalmente, el algoritmo RT ejecuta una búsqueda local en en una m´quina Silicon Graphics Indigo a 100 MHz. Por lo que

[8] [Berger et al., 2001]
[9] [Bent, Hentenryck, 2001]
[10] [Gambardella et al., 1999]
[11] [Rousseau et al., 2002]
[12] [Rochat, Taillard, 1995]

con respecto a dichos trabajos, el algoritmo presenta un excelente aprovechamiento de los recursos de la Grid Morelos.

Benchmarks de Gehring y Homberger

El análisis de eficiencia se realiza principalmente para evaluar un algoritmo con base a los recursos computacionales que requiere para su ejecución, enfocándose principalmente en el tiempo para llegar a una solución de buena calidad, el cual de forma general depende del tamaño de la entrada de datos.

Para llevar a cabo un análisis que refleje el comportamiento de un algoritmo lo más cercano a la realidad, es necesario llevar a cabo pruebas de eficiencia con instancias de tamaños diferentes, esto con la finalidad de identificar que tan susceptible es el algoritmo a los incrementos en los datos de entrada. De acuerdo a esto, además de las pruebas de eficiencia llevada a cabo con los benchmarks de Solomon de 100 clientes, se evaluó la eficiencia del algoritmo AGC-VRPTW con instancias de mayor tamaño. Los benchmarks de Gehring y Homberger de 1000 clientes fueron utilizados para la evaluación del algoritmo propuesto, debido a que corresponden a una extensión de los benchmarks de Solomon, los cuales cuentan con las mismas características y clasificación, de modo que el análisis permitirá observar la escalabilidad y robustez del algoritmo.

Al igual que en el caso de los benchmarks de Solomon, para el análisis de eficiencia se utilizaron las instancias representativas de los bechmarks de Gehring y Homberger con base en los parámetros sintonizados en la sección 5.2, realizando 30 ejecuciones por cada instancia, de modo que los promedios de ejecución se muestran en la tabla 6.10.

Tabla 6.10 *Promedio de los tiempos de ejecución del algoritmo secuencia AGC-VRPTW, aplicado a las instancias representativas de los benchmarks de Gehring y Homberger.*

Instancia	Tiempo de ejecución (segs.)
C1_10_1	1986
R1_10_1	2314
RC1_10_1	2120
C2_10_1	1890
R2_10_1	2076
RC2_10_1	2111

6.4 Cálculo de Complejidad del Algoritmo

Para calcular la complejidad del algoritmo genético cooperativo propuesto en este trabajo de investigación AGC-VRPTW en el pero de los casos, es necesario realizar un análisis que permita conocer su comportamiento y eficiencia, de acuerdo a la cantidad de recursos que requiere para su ejecución. Par esto se avalúa la cantidad de instrucciones que evalua el algoritmo con base en lo explicado en la sección 1.1.

De modo que un algoritmo muestra una complejidad $T(n)$, si para cualquier instancia de tamaño n, requiere ejecutar $T(n)$ instrucciones para resolverlo, de modo que $T(n)$ corresponde al tiempo requerido para resolver un problema de tamaño n de forma secuencial [Martínez, 2010]. En el caso del VRPTW, n corresponde a la cantidad de clientes.

Al tratarse de un algoritmo híbrido, la complejidad temporal del algoritmo AGC-VRPTW se ecuentra dada por la suma de la complejidad del algoritmo genético tomando en cuenta únicamente los operadores de selección y cruzamiento mas la complejidad temporal del operador de mutación cooperativa, el cual se basa en un algoritmo colonia de hormigas.

De acuerdo a lo anterior, la ecuación temporal que representa al algortimo AGC-VRPTW se muestra representada por la ecuación 21.

$$T_{sec}(n) = (n^3 + n^2) + \left(\sum_{i=1}^{p} n^2 + v^2 + vh\right) + n\,log(n) + \sum_{i=1}^{c} n^3vh^2 + \sum_{i=0}^{H} n^2 \quad vh + vh \qquad (21)$$

Donde:

n	→	Tamaño de la instancia con respecto al número de clientes.
v	→	Tamaño de la flota vehicular.
vh	→	Total de rutas en una solución.
c	→	Número de generaciones.
p	→	Tamaño de la población.

Por lo que se puede decir que la complejidad temporal del algoritmo secuencial AGC-VRPTW en el peor de los casos corresponde a la ecuación 22.

$$T(n) \in O(n^3) \qquad (22)$$

6.4 Algoritmo AGCD-VRPTW

Para el desarrollo de un algoritmo distribuido eficiente, es importante buscar el máximo paralelismo al momento de realizar la distribución, sincronización y sobre todo, la cooperación entre procesos, ya que mientras mayor sea la cooperación, mayor será latencia generada, la cual se refleja directamente en el deterioro de la eficiencia del algoritmo. Las ventajas del desarrollo de algoritmos distribuidos eficientes son

muchas, siendo la principal el poder resolver instancias cada vez más grandes de problemas considerados difíciles de resolver en un tiempo mucho menor al requerido de forma secuencial por una sola computadora. Por otro lado, una desventaja se da cuando, para mantener su eficiencia es necesario contar con una infraestructura similar a la utilizada para su desarrollo.

Otro de los puntos fundamentales para el buen desempeño de un algoritmo, es la distribución uniforme de los procesos sobre la infraestructura a utilizar, para lo cual es necesario indicar explícitamente la carga de procesos para cada núcleo desde el momento de la ejecución.

Basado en lo anterior, en esta sección se describe el análisis aplicado al algoritmo distribuido AGC**D**-VRPTW para los benchmarks de Gehring y Homberger, el cual se divide en las pruebas de convergencia, descritas en la sección 6.4.1, el análisis de eficacia (sección 6.4.2) y el análisis de eficiencia (sección 6.4.3), donde se analiza el comportamiento del algoritmo con base a la distribución, cooperación y sincronización de procesos.

6.4.1 Convergencia del Algoritmo

Para evaluar la convergencia del algoritmo genético distribuido AGC**D**-VRPTW, además del número de generaciones y el valor de la función objetivo, es necesario tomar en cuenta la cantidad de núcleos de procesamiento que serán requeridos durante la ejecución. Para el análisis de convergencia se utilizaron los benchmarks de Gehring y Homberger de 1000 clientes, ya que lo que se busca es explotar la infraestructura de la Grid Morelos, lo cual no sería posible con instancias pequeñas como las de Solomon.

Para las pruebas de convergencia se tomaron las instancias representativas de Gehring y Homberger, de la misma forma que en el caso del proceso de sintonización de parámetros (sección 5.1). Se realizaron 30 ejecuciones para cada una de las instancias, utilizando la sintonización de parámetros descrita en el capítulo 5 e incrementando de 2 en 2 el número de procesos lanzados hasta completar los 132

núcleos CPU de la Grid. Para todas las pruebas se utilizaron los valores de los parámetros de control sintonizados en la sección 5.1, la diferencia radica en el tamaño de la población, la cual debe ser repartida entre el total de procesos a ejecutar. Cabe mencionar que para efectos de legibilidad, la escala de procesos en las gráficas se manera de 10 en 10. Los resultados experimentales se muestran en las gráficas de la figura 6.3, donde se evalúa el fitness con respecto al incremento en el número de procesadores.

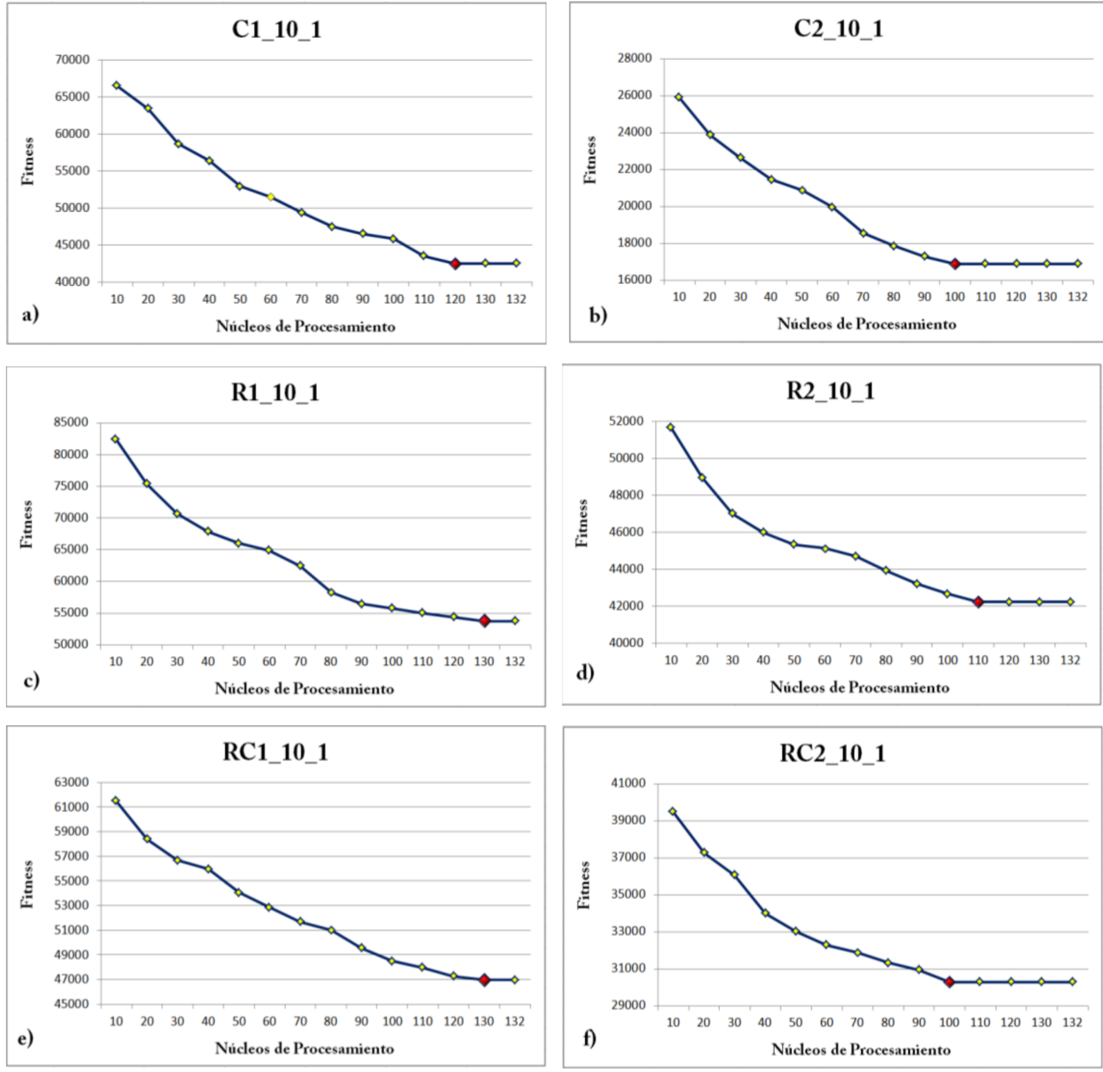


Figura 6.3 *Gráficas de convergencia del algoritmo distribuido AGCD-VRPTW para los benchmarks de Gehring y Homberger a) C1_10_1, b) R1_10_1, c) RC1_10_1, d) C2_10_1, e) R2_10_1, f) RC2_10_1*

De acuerdo a las gráficas mostradas en la figura 6.3, las instancias con mayor dureza en sus restricciones (tipo 1), requieren mayor número de núcleos de procesamiento para alcanzar la convergencia, contrario a lo que sucede con las instancias de tipo 2. Además, se puede observar en las gráficas 6.3c y 6.3d correspondientes a las instancias con distribución aleatoria, que existe un punto alrededor de los 70 núcleos de procesamiento donde disminuye la velocidad de la convergencia, ya que aunque existe una mejora en todo momento, no corresponde al comportamiento de las demás instancias. Esto se debe principalmente a las características de la instancia, así como a la aleatoriedad manejada en la aplicación de los operadores. En el caso de las instancias representadas en las gráficas 6.3a, 6.3b, 6.3e y 6.3f, su convergencia mantiene un comportamiento uniforme.

De acuerdo a los resultados obtenidos, el valor del punto de convergencia observado en las gráficas como un punto, para cada una de las instancias se muestra a continuación en la tabla 6.11.

Tabla 6.11 *Punto de convergencia del algoritmo AGCD-VRPTW para las instancias representativas de los benchmarks de Gehring y Homberger*

Instancia	Núcleos de Procesamiento
C1_10_1	120
R1_10_1	130
RC1_10_1	130
C2_10_1	100
R2_10_1	110
RC2_10_1	100

6.4.2 Análisis de Eficacia

La eficacia de un algoritmo viene dada por la calidad de las soluciones obtenidas, en el caso del VRPTW donde para instancias grandes como las de Gehring y Homberger no se conoce la solución óptima, la eficacia se mide como la cercanía entre las soluciones obtenidas con las mejores cotas reportadas en la literatura.

En el caso de un algoritmo paralelo existen factores que juegan un papel fundamental en el desempeño del algoritmo, como es el caso de la distribución, cooperación y sincronización de procesos, los cuales manejados de forma adecuada permiten el desarrollo de algoritmos sumamente eficientes y eficaces, capaces de tratar problemas complejos que de forma secuencial se consideran intratables.

Planeación de la Distribución de Procesos

La distribución de procesos consiste en asignar un conjunto de procesos a uno o más nodos de la Grid de forma uniforme en tiempo y espacio. Es decir, que en caso de que uno o más núcleos de procesamiento terminen antes, no tengan que esperar largo tiempo a que terminen los demás núcleos de procesamiento, de modo que la eficiencia se vea comprometida.

En el caso del AGCD-VRPTW, la distribución de procesos es una tarea que requiere ser realizada de forma explícita, debido a que lleva de forma implícita el *balaceo de carga* de los núcleos de procesamiento. Para el algoritmo distribuido AGCD-VRPTW esta tarea fue definida desde el momento de la ejecución del algoritmo, utilizando el enlazador de MPI como se muestra a continuación.

```
mpiexec –n <No.procesos> -host <nodo> ./<archivo_ejercutar> :
–n <No.procesos> -host <nodo> ./<archivo_ejercutar> …
```

Con esta instrucción es posible distribuir gran cantidad de *procesos* en un clúster o Grid*,* asegurando que exista balanceo de carga de acuerdo a lo especificado por el programador. Ejemplo: Se quieren ejecutar 12 procesos en 3 nodos con 4 núcleos de procesamiento cada uno. De forma general se tienen 12 núcleos de procesamiento (3 nodos * 4 núcleos), por lo que para realizar una distribución equitativa es necesario enviar un proceso a cada uno de los núcleos, los cuales se encuentran contenidos en grupos de 4 dentro de cada nodo, de modo que para tener una adecuada distribución, la ejecución debe realizarse de la siguiente manera.

```
mpiexec -n 4 -host nodo1 ./programa : -n 4 -host nodo2
./programa : -n 4 -host nodo3 ./programa
```

donde a cada nodo se envían 4 procesos que ejecutarán el archivo `programa`, lo cual se repite para cada uno de los 3 nodos disponibles, garantizando la distribución uniforme de procesos.

De acuerdo al modelo distribuido descrito en la sección 4.2 para el AGCD-VRPTW, cada núcleo de procesamiento se considera como una *isla*, por lo que se puede decir que *No. Núcleos =No. Islas*.

Con base en las características de la Grid Morelos, se tiene que $Grid = \{cluster_1, cluster_2\}$, donde $cluster_1$ corresponde al *clúster Cuexcomate* y $cluster_2$ al *clúster Texcal*. Cada uno de los nodos que componen cada *clúster* contiene un número fijo de núcleos de procesamiento, por lo que para el total de núcleos de la Grid, la distribución de los procesos se realiza tomando en cuenta lo siguiente:

Para determinar la carga a utilizar en la Grid para la ejecución del algoritmo distribuido AGCD-VRPTW, se tomaron en cuenta los resultados de convergencia, por lo que se decidió utilizar la Grid Morelos al 100% de su capacidad, lo cual puede ser observado desde el monitor de ganglia, como se muestra en la figura 6.4.

En la figura 6.4 se muestran los 6 nodos correspondientes al clúster Cuexcomate, donde el nodo maestro indica su nombre en la parte superior y los nodos esclavos se identifican con el nombre *ciicap* seguido de su *id.* Para el caso de los 6 nodos del clúster Texcal, estos se identifican con base en su dirección IP, donde el nodo maestro se identifica con la dirección 192.168.100.250. Para identificar el porcentaje de ocupación de la Grid, es importante tomar en cuenta los ejes correspondientes a cada una de las gráficas de la figura 6.3, así como la cantidad de procesos en ejecución (parte inferior de la gráfica) comparado con la cantidad de núcleos de procesamiento disponibles por nodo. De acuerdo a las gráficas, el eje de las x indica la hora en que están siendo ejecutados los procesos y el eje de las y

Figura 6.4. *AGCD-VRPTW trabaja con 132 islas, por lo que se muestra la ocupación del de la Grid Morelos al 100%*

muestra la cantidad de procesos que se están ejecutando de forma concurrente. El porcentaje de ocupación se encuentra dado por la comparación del número de procesos ejecutados por nodo, con respecto a los núcleos de procesamiento disponibles, los cuales de muestran en la tabla 6.2. De modo que para la figura 6.3, se muestra que los nodos maestro (Cuexcomate y Texcal) cuentan con una sobrecarga de 0.19 y 0.10, respectivamente, lo cual es debido a los procesos que requiere tener activos cada nodo maestro para que la Grid se mantenga funcionando.

Para la ejecución del AGC**D**-VRPTW en Grid se utilizaron los valores de los parámetros de control obtenidos mediante el análisis de sensibilidad presentado en el capítulo 5. Bajo estas condiciones, cada una de las *islas* trabaja con una sub-población definida de 150 clientes, por lo que el tamaño total de la población se encuentra dado por la fórmula 30.

$$Población_{total} = islas * tamPop \tag{30}$$

Cabe mencionar que al igual que en el caso de las pruebas experimentales realizadas con el algoritmo secuencial, para el AGC**D**-VRPTW también se realizaron 30 ejecuciones por cada una de las instancias.

***Tabla 6.12** Análisis estadístico de los resultados obtenidos por el algoritmo distribuidoAGC**D**-VRPTW para los benchmarks de Gehring y Homberger de 1000 clientes*

Instancia	Veh	Mejor	Peor	Prom	σ	Veh	Cota	ER (%)
C1_10_1	100	**42478.95**	42479.23	42479.08	0.198	100	**42478.95**	0
C1_10_2	90	**42278.45**	**42278.88**	**42278.59**	**0.322**	90	**42278.45**	**0**
C1_10_3	90	**40207.71**	40207.71	40207.71	0	90	**40207.71**	0
C1_10_4	90	**39468.6**	39468.82	39468.69	0.158	90	**39468.6**	0
C1_10_5	100	**42469.18**	42470.43	42469.65	0.911	100	**42469.18**	0
C1_10_6	100	**43834.12**	**43835.05**	**43834.68**	**0.671**	99	**43830.21**	**0.0089**
C1_10_7	97	**43453.92**	43453.92	43453.92	0	97	**43453.92**	0
C1_10_8	**94**	**41853.36**	**42149.58**	**41912.58**	*	**93**	**42149.58**	**-0.7**
C1_10_9	91	40571.2	40572.14	40571.59	0.674	90	**40570.6**	0.0015
C1_10_10	90	**39933.06**	39933.71	39933.45	0.469	90	**39933.06**	0
C2_10_1	30	**16879.24**	16879.24	16879.24	0	30	**16879.24**	0
C2_10_2	29	**17126.39**	17126.39	17126.39	0	29	**17126.39**	0
C2_10_3	28	**16884.08**	16884.08	16884.08	0	28	**16884.08**	0
C2_10_4	28	**15656.75**	15656.75	15656.75	0	28	**15656.75**	0
C2_10_5	30	**16561.29**	16561.51	16561.39	0.156	30	**16561.29**	0
C2_10_6	30	16921.43	16921.93	16921.69	0.354	29	**16920.33**	0.0065
C2_10_7	29	**17882.42**	17882.42	17882.42	0	29	**17882.42**	0
C2_10_8	29	**16578.21**	**16578.85**	**16578.63**	**0.474**	28	**16577.32**	**0.0054**
C2_10_9	29	**16370.44**	16370.44	16370.44	0	29	**16370.44**	0
C2_10_10	28	**15944.72**	15944.72	15944.72	0	28	**15944.72**	0
R1_10_1	100	**53560.85**	53560.85	53560.85	0	100	**53560.85**	0
R1_10_2	92	49107.96	49108.99	49108.52	0.731	91	**49105.21**	0.0056
R1_10_3	91	**45237.29**	45237.29	45237.29	0	91	**45237.29**	0
R1_10_4	91	**42788.36**	**42789.65**	**42788.85**	**0.938**	91	**42787.19**	**0.0027**
R1_10_5	92	51832.47	51832.96	51832.69	0.348	91	**51830.36**	0.0041
R1_10_6	91	**47849.05**	47849.05	47849.05	0	91	**47849.05**	0
R1_10_7	91	**44435.5**	44436.12	44435.89	0.453	91	**44435.5**	0
R1_10_8	91	42487.25	42487.97	42487.65	0.512	91	**42485.38**	0.0044
R1_10_9	91	**50490.49**	50490.49	50490.49	0	91	**50490.49**	0
R1_10_10	91	**48294.71**	48295.38	48294.99	0.480	91	**48294.71**	0
R2_10_1	19	**42188.86**	42188.86	42188.86	0	19	**42188.86**	0
R2_10_2	19	33514.01	33514.78	33514.49	0.561	19	**33512.83**	0.0035
R2_10_3	19	**24940.32**	24940.32	24940.32	0	19	**24940.32**	0
R2_10_4	19	**17926.45**	17926.45	17926.45	0	19	**17926.45**	0
R2_10_5	19	**36232.18**	**36232.89**	**36232.61**	**0.513**	19	**36232.18**	**0**

R2_10_6	19	**30091.93**	30091.93	30091.93	0	19	**30091.93**	0
R2_10_7	19	**23257.36**	23257.36	23257.36	0	19	**23257.36**	0
R2_10_8	19	17498.55	17499.89	17498.92	1.038	19	**17495.51**	0.0174
R2_10_9	19	33009.45	33010.25	33009.98	0.595	19	**33002.36**	0.0215
R2_10_10	19	**30215.24**	30215.24	30215.24	0	19	**30215.24**	0
RC1_10_1	90	**46272.07**	46272.07	46272.07	0	90	**46272.07**	0
RC1_10_2	90	**44129.42**	44129.42	44129.42	0	90	**44129.42**	0
RC1_10_3	91	42490.21	42491.32	42490.68	0.794	90	**42487.54**	0.0063
RC1_10_4	91	41615.48	41615.48	41615.48	0	90	**41613.58**	0.0046
RC1_10_5	90	**45564.81**	45564.81	45564.81	0	90	**45564.81**	0
RC1_10_6	91	**45303.67**	45303.67	45303.67	0	90	**45303.67**	0
RC1_10_7	90	**44903.8**	44903.8	44903.8	0	90	**44903.8**	0
RC1_10_8	90	44368.65	44368.65	44368.65	0	90	**44366.01**	0.0060
RC1_10_9	90	**44280.84**	44280.84	44280.84	0	90	**44280.84**	0
RC1_10_10	90	43898.45	43899.85	43898.89	1.056	90	**43896.78**	0.0038
RC2_10_1	**20**	**30278.48**	**30279.46**	**30278.86**	*****	**20**	**30278.5**	**-0.000066**
RC2_10_2	18	**26327.92**	26327.92	26327.92	0	18	**26327.92**	0
RC2_10_3	18	**20050.71**	20050.71	20050.71	0	18	**20050.71**	0
RC2_10_4	19	15750.98	15750.98	15750.98	0	18	**15747.13**	0.0244
RC2_10_5	19	**27237.68**	27239.86	27238.64	1.552	18	**27237.68**	0
RC2_10_6	19	26799.1	26799.93	26799.65	0.617	18	**26797.76**	0.0050
RC2_10_7	18	**25112.77**	25112.77	25112.77	0	18	**25112.77**	0
RC2_10_8	18	**23709.29**	23709.29	23709.29	0	18	**23709.29**	0
RC2_10_9	18	**23028.1**	23028.1	23028.1	0	18	**23028.1**	0
RC2_10_10	18	21967.99	21967.99	21967.99	0	18	**21965.94**	0.0093

El análisis de resultados mostrado en la tabla 6.12 mostró que la calidad de las soluciones obtenidas por AGC**D**-VRPTW es similar a la obtenida por el algoritmo secuencial en la mayoría de los casos, con la diferencia de que se obtuvieron algunas mejoras en soluciones que no habían alcanzado la mejor cota conocida, como se muestra en la tabla 6.12, donde de 20 soluciones que no alcanzaron las mejores cotas con el algoritmo secuencial, con el distribuido AGC**D**-VRPTW se reduce a 18 instancias, donde el error relativo también se reduce, teniendo un máximo de 0.0244% y el mínimo de 0.0015%, lo que indica que el algoritmo propuesto obtuvo resultados muy cercanos a la mejor cota conocida. Por otro lado, la dispersión de las soluciones se reduce ligeramente con respecto al secuencial, teniendo un máximo de

1.552 y un mínimo de 0.158, esto debido a que existe mayor diversidad entre las soluciones, ya que las poblaciones evolucionan de forma cuasi-independiente, ya que el proceso de migración genera una cooperación entre ellas, manteniendo la diversidad en las subpoblaciones, lo que favorece la exploración y explotación.

De acuerdo a los resultados mostrados en la tabla 6.10, se obtuvieron 5 mejoras a las soluciones obtenidas por el AGC-VRPTW correspondientes a las instancias C1_10_2, C1_10_6, C2_10_8, R1_10_4 y R2_10_5, donde C1_10_6, C2_10_8 y R1_10_4 lograron reducir su error relativo con respecto a las mejores cotas y para R2_10_5, se alcanzó la mejor cota.

Análisis de Eficacia en la Cooperación de Procesos

De acuerdo al modelo distribuido AGC**D**-VRPTW presentado en la figura 4.19, la cooperación que se realiza entre los procesos se lleva a cabo de forma implícita, de modo que durante el proceso de evolución de los individuos no existe una comunicación directa entre procesos, sino que la migración de individuos se realiza a través del proceso maestro, el cual se encarga de juntar todas las subpoblaciones en una sola población pancmítica y aplicar los operadores de selección, cruzamiento y mutación con los valores sintonizados en la sección 5.2. Los parámetros correspondientes a *probabilidad de cruzamiento* y *probabilidad de mutación* aplicados a la migración implícita de individuos, son diferentes a los utilizados durante el proceso de evolución en cada una de las islas, debido a que la migración entre poblaciones debe realizarse de forma moderada, ya que una migración deficiente se refleja en la pérdida de diversidad en las poblaciones y por ende, en la convergencia prematura del algoritmo. Por el contrario, un exceso en la migración evita que el algoritmo converja en una buena solución.

Para evaluar el comportamiento del algoritmo AGC**D**-VRPTW con base en la cooperación de procesos, se utilizaron las instancias representativas de los benchmarks de Gehring y Homberger de 1000 clientes (C1_10_1, R1_10_1, RC1_10_1, C2_10_1, R2_10_1, RC2_10_1). Dichas instancias fueron evaluadas en

grupos de núcleos de procesamiento de tamaño creciente, es decir, cada una de las instancias fue ejecutada 30 veces sobre {5, 10, 30, 60, 90, 132, 264, 396, 528} núcleos, donde los últimos 3 grupos de núcleos presenta una sobrecarga al 200%, 300% y 400%, respectivamente. La importancia de variar el número de núcleos de procesamiento a utilizar radica en la evaluación de la calidad de las soluciones con respecto a la cantidad de islas utilizadas en el modelo, lo cual implica un incremento en el tamaño de la población total.

La distribución de procesos en ocasiones puede requerir de la ejecución de un mayor número de procesos que los núcleos de procesamiento disponibles en la Grid, lo que se le conoce como *sobrecarga*. La sobrecarga permite distribuir más de un proceso por núcleo, por ejemplo, la asignación de 264 procesos a 120 núcleos, o bien un conjunto de procesos que serán ejecutados en cada uno de los núcleos. Los resultados obtenidos para cada uno de los casos se muestran en la tabla 6.13.

Tabla 6.13. *Análisis del comportamiento de la eficacia al variar la cantidad de procesos a ejecutar*

Procs.	**Instancias**					
	C1_10_1	**R1_10_1**	**RC1_10_1**	**C2_10_1**	**R2_10_1**	**RC2_10_1**
5	51138.78	62844.18	55562.16	20003.5	50597.4	36824.02
10	50621.45	62624.26	54854.12	19235.87	48994.48	35495.98
30	48454.4	61416.46	51465.22	18152.15	46548.89	33852.1
60	46287.58	58899.41	49721.64	17652.51	44021.96	31887.35
90	43529.58	54714.58	47098.65	17121.25	42952.87	30939.87
132	***42478.95***	***53560.85***	***46272.07***	***16879.24***	***42188.86***	***30278.48***
264	42479.57	53561.03	46272.89	16879.24	42189.15	30279.23
396	42479.57	53561.03	46272.89	16879.24	42189.15	30279.23
528	42479.99	53562.15	46272.89	16880.05	42190.12	30279.92

De acuerdo a los resultados mostrados en la tabla 6.13, los seis tipos de instancias utilizadas fueron ejecutadas en 9 niveles de carga de procesos, donde los primeros 5 corresponden a una ocupación reducida de la Grid, el nivel 6 indica una distribución al 100% y a partir del nivel 7 se maneja una sobrecarga del 200% al 400%, respectivamente. Los resultados obtenidos con respecto a la calidad de la solución al incrementar la carga de la Grid muestran que con una ocupación menor al 100% se

favorece la convergencia del algoritmo, misma que se mantiene al incrementar la sobrecarga de procesos hasta el 300%. Este comportamiento cambia al sobrecargar la Grid a un 400%, donde se obtiene que las instancias C1_10_1, R1_10_1, C2_10_1, R2_10_1 y RC2_10_1 presenten una reducción en la eficacia. Esto se debe a que al incrementar la cantidad de procesos implica incrementar drásticamente el tamaño de la población total utilizada, de modo que surge un desbalance entre la exploración y la explotación.

La distribución de procesos con sobrecarga requiere de identificar la cantidad de procesos a ejecutar y dividirlos entre el factor de sobrecarga, de modo que si por ejemplo, se desean ejecutar 396 procesos en la Grid Morelos, el total se divide entre el número de procesadores disponibles, quedando la distribución de la siguiente manera *{n1:3, n2:3, n3:3, …,* n_m*:3}*, como se muestra en la figura 6.5.

Figura 6.5. *Sobrecarga de procesos del AGCD-VRPTW en la Grid Morelos al 300%*

La figura 6.5 muestra el monitor ganglia de grid, donde se observan los nodos en rojo, indicando una sobrecarga de procesos. Para identificar cual es el porcentaje de sobrecarga con el que se está ejecutando el algoritmo, es necesario conocer la capacidad y distribución de la infraestructura utilizada. Por ejemplo, la gráfica correspondiente al *cuexcomate* presenta una carga de 36.34 procesos (parte baja de la gráfica), lo que indica que esta cantidad se debe dividir entre el número de núcleos de procesamiento con que cuenta el nodo, que en este caso es 12, por lo que se tiene que

cada núcleo de procesamiento está ejecutando 3 procesos, lo que indica una sobrecarga al 300%. En el caso de los nodos *gpu*, estos solo contienen 6 núcleos de procesamiento, por lo que su sobrecarga es de 18 procesos, como se muestra en la gráfica correspondiente al *ciicap-gpu01* y al 192.168.100.105.

6.4.3 Análisis de Eficiencia

El análisis de eficiencia de un algoritmo distribuido en ambiente Grid no es una tarea trivial, ya que es necesario analizar el desempeño del mismo con base a la influencia de la distribución y cooperación entre procesos de acuerdo al tipo de comunicación, sincronización e incluso tomando en cuenta la infraestructura en la que es ejecutado, ya que algunos de los puntos fundamentales que definen la eficiencia de un algoritmo en el desarrollo de nuevos modelos es el esquema utilizado y la cantidad de comunicación requerida entre los procesos.

En el caso del algoritmo AGC**D**-VRPTW, descrito en la sección 4.2, se propone un modelo de islas con comunicación implícita que evita la comunicación directa entre procesos durante la migración, esto debido a que a medida que aumenta el número de procesos, la latencia de las comunicaciones tiende a incrementarse afectando negativamente la eficiencia del algoritmo. El modelo propuesto se encarga de llevar a cabo una migración implícita al recolectar todas las subpoblaciones en el proceso maestro y aplicar los operadores genéticos con base a la sintonización de parámetros mostrada en la sección 5.2 tomando el total de individuos como una población pancmítica.

Las pruebas de eficiencia se realizaron con base a las instancias representativas de los benchmarks de Gehring y Homberger. Cada una de las instancias se evaluó con 30, 60, 90, 120 y 240 procesos, respectivamente. Los parámetros de control utilizados son los obtenidos a partir del análisis de sensibilidad en la sección 5.1, a excepción del tamaño de la población, el cual se define con base a la fórmula 6.2, tomando un tamaño de subpoblación de 150 individuos.

6.4.3.1 Análisis de la Distribución de Procesos

Los resultados del análisis se muestran a continuación, donde *P* corresponde al total de procesos lanzados, *NP* al número de núcleos utilizados durante la ejecución, t_{dist} es la media del tiempo requerido para llegar a la mejor solución con base a las 30 ejecuciones, $Best_{dist}$ es la mejor solución encontrada por el AGCD-VRPTW, $Best_{dist}$ muestra el fitness de la mejor solución encontrada. En el caso del tiempo secuencial t_{sec}, se realizaron pruebas experimentales con los tamaños de población correspondientes a 5 y 10 procesos, pero debido al tiempo requerido para trabajar con poblaciones más grandes, a partir de las poblaciones de 30 procesos se calcula el $t_{sec,}$ de forma teórica, con base a las tiempos obtenidos en pruebas con 30 procesos de 5 y 10 procesos. En las tablas 6.14, 6.15, 6.16, 6.17, 6.18 y 6.19 se presentan los resultados obtenidos como el promedio de 30 ejecuciones para cada cantidad de procesos 30, 60, 90, 120 y 240, utilizando subpoblaciones de 150 individuos.

Tabla 6.14 *Resultados del análisis de eficiencia realizado a la instancia C1_10_1*

C1_10_1

Procesos	*Núcleos$_{Proc}$*	t_{dist}	$Best_{dist}$	t_{sec}	$Best_{sec}$
30	30	380	49131.26	**13104**	49668.41
60	60	385	46831.26	**27208**	47257.24
90	90	388	43859.68	**43812**	43864.59
120	**120**	**390**	**42478.95**	**63416**	**42478.95**
240	120	527.59	42478.95	**112983**	42478.95

Tabla 6.15 *Resultados del análisis de eficiencia realizado a la instancia R1_10_1*

R1_10_1

Procesos	*Núcleos$_{Proc}$*	t_{dist}	$Best_{dist}$	t_{sec}	$Best_{sec}$
30	30	392	61416.46	**13843**	61410.01
60	60	399	58899.41	**27973**	58910.23
90	90	403	54714.58	**44215**	54802.66
120	120	407	55435.11	**63987**	55235.49
130	**130**	**539**	**46272.07**	**69319**	**46272.07**
240	120	546	42472.07	**113342**	46272.26

Tabla 6.16 *Resultados del análisis de eficiencia realizado a la instancia RC1_10_1*

RC1_10_1

Procesos	***Núcleos**$_{Proc}$*	t_{dist}	***Best**$_{dist}$*	t_{sec}	*Best$_{sec}$*
30	30	391	61416.46	**13516**	61410.01
60	60	392	58899.41	**27704**	58910.23
90	90	395	54714.58	**44120**	54802.66
120	**120**	**399**	**51435.11**	**63712**	**55235.49**
150	120	523	46272.07	**79640**	46272.07
180	120	527	46272.89	**95568**	46272.89
240	120	534	46472.89	**113124**	46272.89

Tabla 6.17 *Resultados del análisis de eficiencia realizado a la instancia C2_10_1*

C2_10_1

Procesos	***Núcleos**$_{Proc}$*	t_{dist}	***Best**$_{dist}$*	t_{sec}	*Best$_{sec}$*
30	30	375	18152.15	**12965**	18196.18
60	60	381	17652.51	**26818**	17632.21
90	90	386	17121.25	**43564**	17184.45
120	**120**	**387**	**16879.24**	**63104**	**16879.24**
150	120	510	16879.24	**79862**	16879.24
180	120	515	16879.24	**95547**	16879.24
240	120	521	16879.24	**112642**	16879.24

Tabla 6.18 *Resultados del análisis de eficiencia realizado a la instancia R2_10_1*

R2_10_1

Procesos	***Núcleos**$_{Proc}$*	t_{dist}	***Best**$_{dist}$*	t_{sec}	*Best$_{sec}$*
30	30	371	46548.89	**13521**	46521.15
60	60	376	44021.96	**27612**	44132.15
90	90	381	42952.87	**43985**	42935.68
120	**120**	**385**	**42188.86**	**63345**	**42188.86**
150	120	503	42188.86	**79983**	42188.86
180	120	507	42188.86	**95981**	42188.86
240	120	511	42188.86	**112965**	42188.86

Tabla 6.19 *Resultados del análisis realizado a la instancia RC2_10_1*

RC2_10_1

Procesos	***Núcleos**$_{Proc}$*	t_{dist}	***Best**$_{dist}$*	t_{sec}	*Best$_{sec}$*
30	30	370	33852.1	**13118**	33957.45
60	60	374	31887.35	**27235**	32014.12

90	90	378	30939.87	**43714**	30978.54
120	**120**	**381**	**30278.48**	**63201**	**30278.48**
150	120	492	30278.48	**79893**	30278.48
180	120	496	30278.48	**95744**	30278.48
240	120	501	30278.48	**112801**	30278.48

Las tablas 6.12 a 6.17 muestran la comparación entre el comportamiento del algoritmo distribuido AGC**D**_VRPTW y el secuencial AGC-VRPTW para las instancias representativas de los benchmarks de Gehring y Homberger, tanto en tiempo como en la calidad de las soluciones, donde se presenta un promedio de 30 ejecuciones de una instancia variando la cantidad de procesos. Los resultados muestran el punto de convergencia (datos en rojo) con base al número de procesos ejecutados, además de mostrar que la cooperación entre procesos permite obtener mejores resultados conforme aumenta la cantidad de procesos utilizados. Por su parte, el tiempo de ejecución presenta una reducción considerable con respecto a los del algoritmo secuencial, esto debido a que la latencia en las comunicaciones es mínima, dado que los clústeres que incorporan la Grid se encuentran conectados mediante una red de alta velocidad. Otro factor que favorece el rendimiento del algoritmo es el esquema de comunicación colectiva implementado, ya que permite mover los datos con un solo envío que reparte los datos requeridos a cada uno de los procesos, contrario a lo que sucede con la comunicación bloqueante, donde se debe realizar un envío independiente a cada proceso.

El comportamiento del tiempo de ejecución del algoritmo secuencial (AGC-VRPTW) con respecto al del algoritmo distribuido (AGC**D**-VRPTW) permite visualizar los efectos de la comunicación entre procesos, así como la sincronización. Lo explicado anteriormente se muestra en las figuras 6.6 - 6.11, donde se grafica el tiempo requerido por el algoritmo secuencial AGC-VRPTW versus el algoritmo distribuido AGC**D**-VRPTW para obtener la mejor solución a cada uno de las instancias representativas de los benchmarks de Gehring y Homberger de 1000 clientes. En dichas gráficas también es posible observar que los tiempos de ejecución también dependen del tipo y distribución de cada instancia.

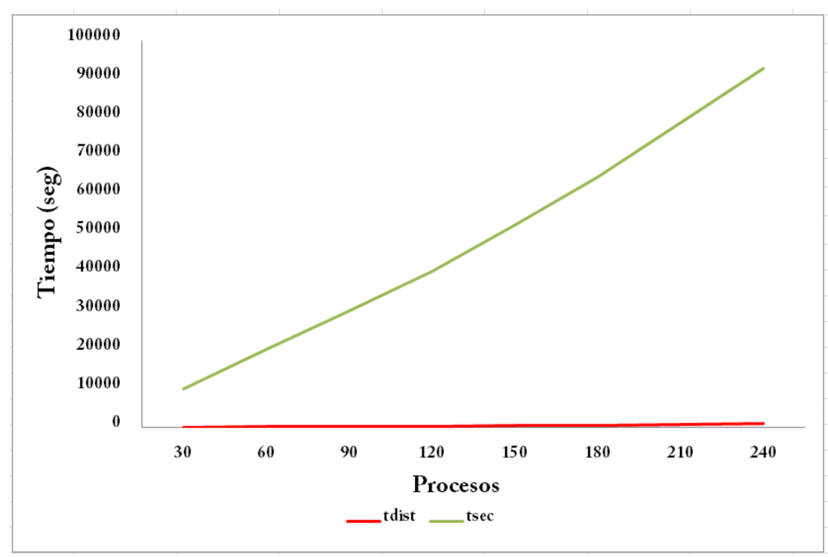


Figura 6.6 *Resultados del análisis de eficiencia realizado a la instancia C1_10_1, con los datos de la tabla 6.13*

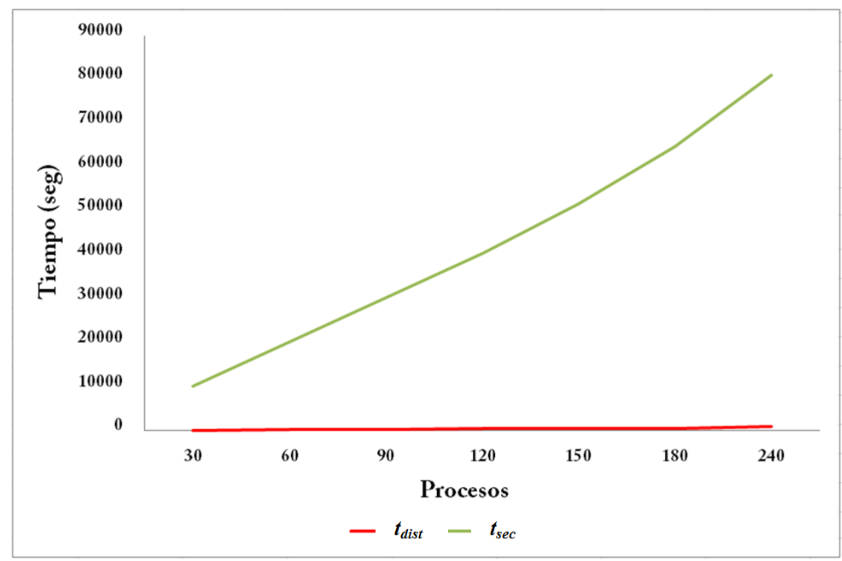


Figura 6.7 *Resultados del análisis de eficiencia realizado a la instancia R1_10_1, con los datos de la tabla 6.14*

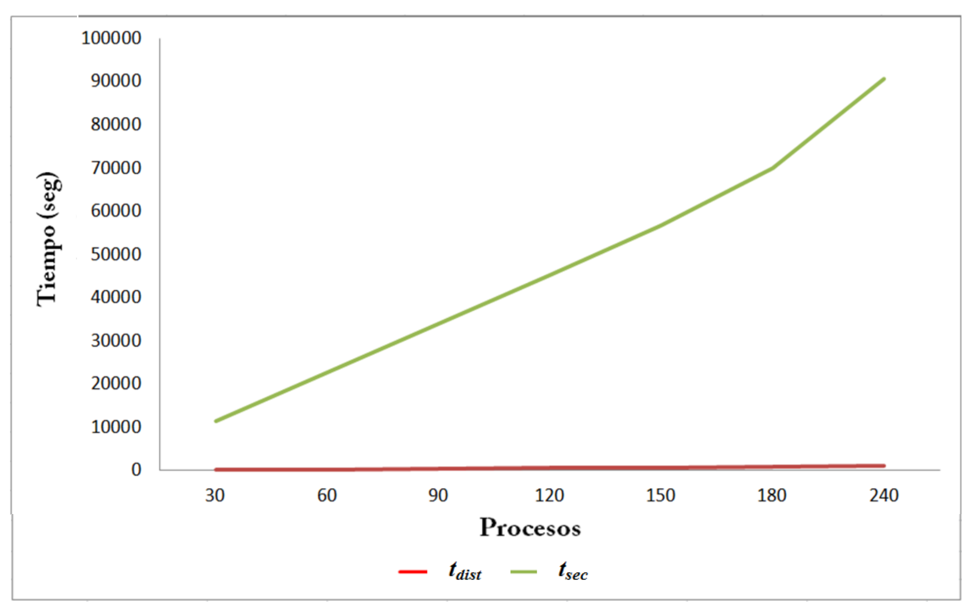


Figura 6.8 *Resultados del análisis de eficiencia realizado a la instancia RC1_10_1, con los datos de la tabla 6.15*

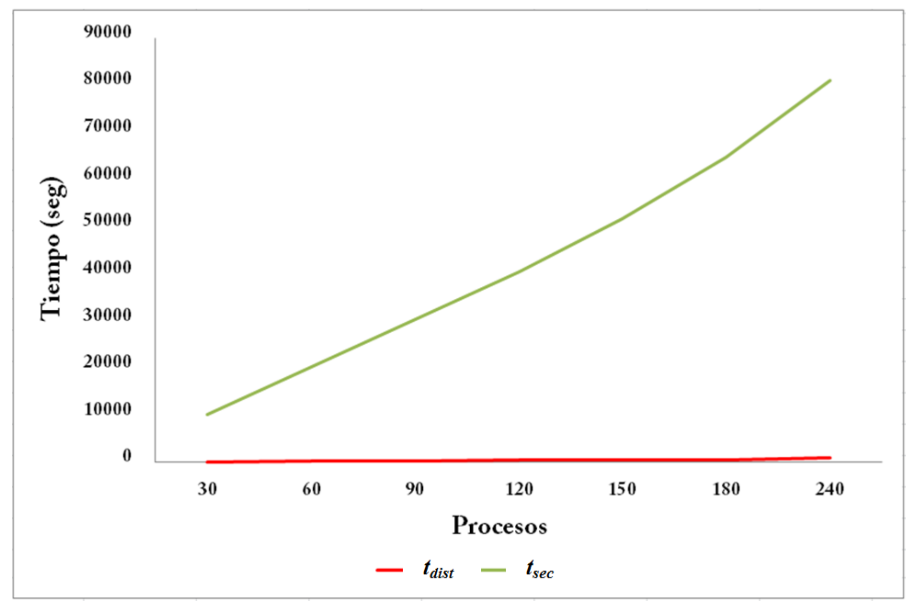


Figura 6.9 *Resultados del análisis de eficiencia realizado a la instancia C2_10_1, con los datos de la tabla 6.16*

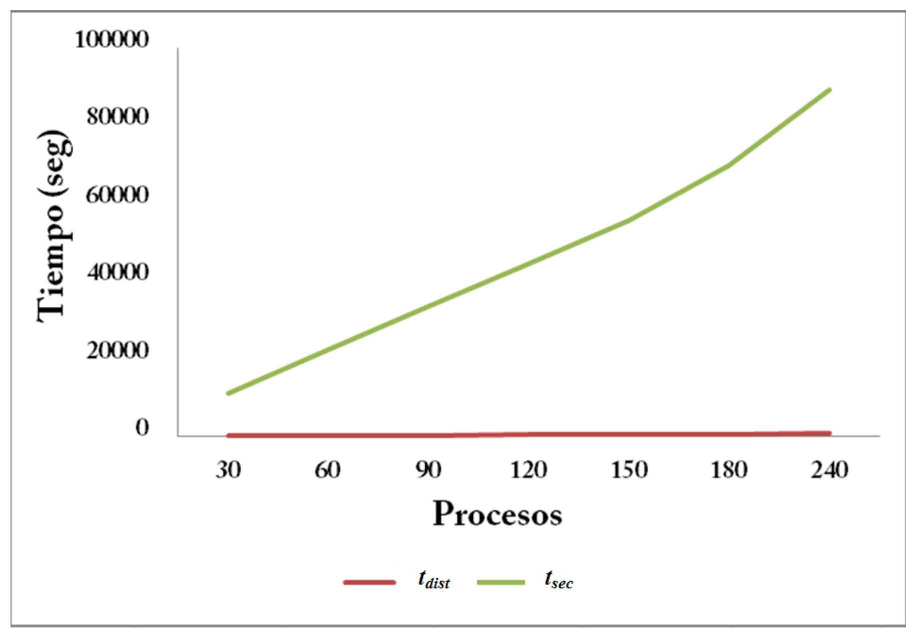


Figura 6.10 *Resultados del análisis de eficiencia realizado a la instancia R2_10_1, con los datos de la tabla 6.17*

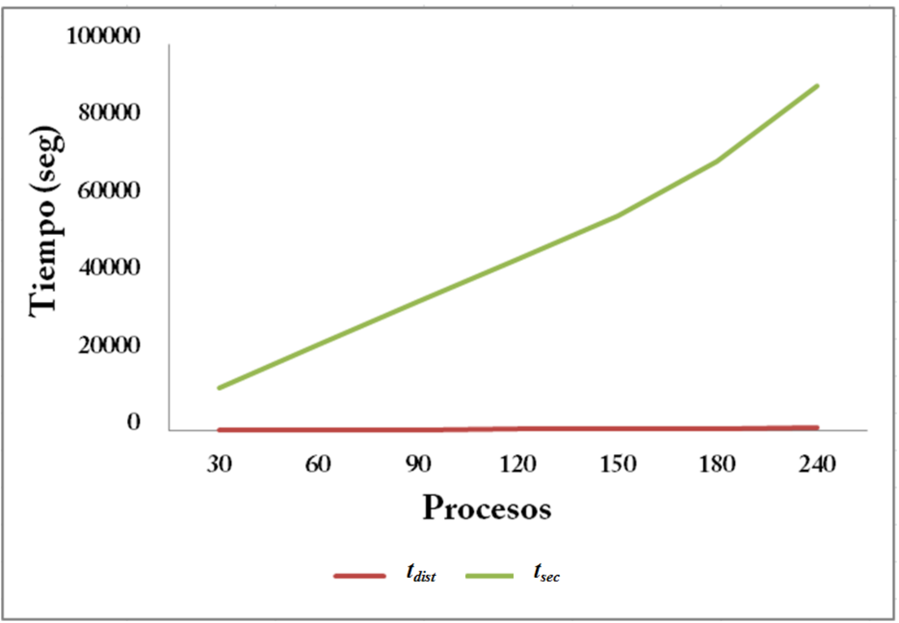


Figura 6.11 *Resultados del análisis de eficiencia realizado a la instancia RC2_10_1, con los datos de la tabla 6.18*

Las gráficas de la figura 6.10 muestran la comparación del comportamiento del algoritmo distribuido AGC**D**-VRPTW versus su versión secuencial AGC-VRPTW, para lo cual se evalúa el tiempo requerido para obtener una solución al VRPTW

variando el número de procesos ejecutados, lo cual en el caso del algoritmo secuencial, se evalúa el tamaño de población correspondiente al resultado de multiplicar el número de procesos por el tamaño de la sub-población.

De acuerdo a los resultados promedio mostrados en las gráficas 6.6 a 6.11, los seis tipos de instancias muestran un comportamiento similar, donde los tiempos de cómputo distribuido se mantienen en balance comparado con los requeridos por su ejecución en un solo núcleo de procesamiento, ya que en el caso del algoritmo paralelo se muestra un incremento mínimo en el tiempo al incrementar el número de procesos, esto debido a las comunicaciones y al proceso de migración definido para el modelo de islas en la sección 4.2, contrario a lo que sucede con el algoritmo secuencial AGC-VRPTW, el cual incrementa su tiempo de cómputo considerablemente conforme se incrementa el tamaño de la entrada.

Por otro lado, en el caso de sobrecarga de la Grid a 240 procesos, como se observa en las tablas 6.13 a 6.18, el tiempo se duplica, ya que cada núcleo de procesamiento se encarga de trabajar con dos procesos. Además de las características explicadas anteriormente, existen factores externos que deben ser cuidados para evitar que se incrementen los tiempos de procesamiento de forma inusual, afectando los resultados del análisis de eficiencia, dichos factores se pueden clasificar en tres puntos:

1. *Procesos en ejecución simultánea*. Cuando se realizan pruebas, principalmente de eficiencia, es importante que no existan otros procesos en ejecución. Independientemente de los muchos o pocos recursos computacionales que lleguen a utilizar, se está compartiendo tiempo de cómputo, lo cual afecta al desempeño del algoritmo.
2. *Balanceo de carga incorrecto*. Previo a la ejecución del algoritmo, es importante realizar una planeación correspondiente a la distribución de procesos entre los núcleos de procesamiento disponibles, la cual se debe llevar a cabo de la forma más uniformemente posible, para evitar que algunos procesos terminen antes y permanezcan ociosos hasta que los demás alcancen la barrera de sincronización.

3. *Comunicaciones.* La velocidad y tipo de infraestructura utilizada para establecer la comunicación entre clústeres dispersos geográficamente juega un papel fundamental en el desempeño de algoritmos ejecutados en grid.

Para evitar o reducir la problemática explicada anteriormente, el algoritmo AGC**D**-VRPTW fue ejecutado de forma exclusiva en la Grid Morelos, es decir, ningún otro usuario estaba conectado a la Grid ni se estaban ejecutando procesos externos al algoritmo en análisis. En cuanto al balanceo de carga, este se realizó mediante un *script* indicando el número de procesos que fueron asignados a cada uno de los núcleos de procesamiento, lo cual se detalla en la sección 4.2.2.2. Por otro lado, los esquemas de comunicación por parte del modelo permiten reducir la latencia debido a que incorporan el uso de comunicación colectiva (sección 4.2.2.1).

Para esta versión del algoritmo se utilizaron instancias grandes donde se observa cómo influye la dureza de las restricciones y el tamaño de la entrada en la eficiencia del algoritmo propuesto. Donde la cantidad de procesadores utilizados juega un papel fundamental en el desempeño del algoritmo, como se observa en las figuras 6.13 a 6.18.

6.4.3.2 Análisis de Aceleración (Speedup)

El uso de una infraestructura de alto rendimiento como la utilizada en este trabajo de investigación (sección 6.1) provee muchas ventajas a un algoritmo distribuido, ya que permite mejorar la eficiencia del mismo, así como llevar a cabo el tratamiento de instancias más grandes en comparación con un algoritmo secuencial.

Para las pruebas experimentales se hizo uso de las instancias representativas de los benchmarks de Gehring y Homberger, tanto en la versión distribuida como en la secuencial, ejecutando 30 veces cada una de las instancias para el número de procesos definido, donde cada proceso trabaja con una sub-población de 150 individuos, de modo que el promedio de los resultados obtenidos se muestra en la tabla 6.20.

Tabla 6.20 *Tabla comparativa entre los resultados obtenidos por el algoritmo secuencial AGC-VRPTW y la versión distribuida AGCD-VRPTW.*

Procesos	***Tiempos de Ejecución (segs.)***					
	C1_10_1-CPU	***C1_10_1-Dist.***	***R1_10_1-CPU***	***R1_10_1-Dist.***	***RC1_10_1-CPU***	***RC1_10_1-Dist.***
5	2184.15	371.59	2307.25	372.65	2252.78	372.17
10	4371.89	375.41	4614.96	381.14	4505.45	377.54
30	13112.15	380.29	13843.88	392.48	13516.74	391.02
60	27205.96	385.98	27947.69	399.05	27704.9	392.64
90	43809.45	388.12	44215.9	403.14	44120.5	395.44
120	63421.19	390.48	63987.21	407.25	63712.9	399.17
240	112991.55	527.59	113342.07	546.95	113124.41	534.88
Procesos	***Tiempos de Ejecución (segs.)***					
	C2_10_1-CPU	***C2_10_1-Dist.***	***R2_10_1-CPU***	***R2_10_1-Dist.***	***RC2_10_1-CPU***	***RC2_10_1-Dist.***
5	2160.14	362.47	2253.45	367.14	1921.75	364.48
10	4321.47	366.54	4507.29	369.45	2851.69	368.06
30	12965.58	375.45	13521.47	371.25	13118.32	370.59
60	26818.1	381.14	27612.01	376.48	27235.88	374.15
90	43564.1	386.12	43985.06	381.96	43714.11	378.59
120	63104.87	387.55	63345.98	385.47	63201.94	381.09
240	112642.54	521.89	112965.35	511.69	112801.56	501.99

De acuerdo a los resultados mostrados en la tabla 6.20, la cantidad de procesos influye en el tamaño de la población utilizada, por lo que para calcular el tiempo del algoritmo secuencial se tomó el número de procesos del algoritmo distribuido por el tamaño de la subpoblación como el tamaño total de la población del algoritmo secuencial, con lo que se obtuvieron los resultados mostrados en las tabla 6.20.

Para evaluar el desempeño del algoritmo uno de los puntos fundamentales a evaluar en un algoritmo distribuido es su rendimiento, que permite medir el tiempo de ejecución del algoritmo basado en su comportamiento al incrementar la cantidad de procesos involucrados en la ejecución, ya que la finalidad del diseño de un algoritmo distribuido es mejorar la eficiencia de su análogo secuencial. En este sentido, el *Speedup* corresponde al rango en el cual la versión distribuida de un algoritmo se

ejecuta más rápido que la versión secuencial, por lo que su aceleración se calcula con base en la fórmula 31.

$$Speedup = \frac{Tiempo_{secuencial}}{Tiempo_{distribuido}} \tag{31}$$

De modo que de forma teórica el *speedup ideal* corresponde a un algoritmo distribuido que es dos veces más rápido si se ejecuta en dos procesadores, tres veces más rápido en tres procesadores, etc., de modo que el *speedup ideal* es igual a *nprocs.* Por el contrario, en el caso de aplicaciones reales, el rango de aceleración se encuentra en función de las siguientes tres características:

1. *Operaciones secuenciales* ($\delta(n)$): Son aquellas instrucciones que de acuerdo al modelo distribuido implementado, requieren ser ejecutadas de forma secuencial.
2. *Procesos distribuidos* ($\varphi(n)$): Son segmentos de código que pueden ser ejecutados de forma paralela.
3. *Latencia (L(n,nprocs))*: Tiempo requerido para establecer la distribución, comunicación y cooperación entre los procesos.

Con base a estos tres puntos se tiene que la aceleración de algoritmos distribuidos puede variar con respecto al *speedup* ideal. Por lo que el término de *eficiencia* se enfoca en comparar que tan cercana se encuentra la medida de aceleración del algoritmo con el speedup ideal, obteniendo las siguientes clasificaciones:

- *Speedup lineal:* corresponde al *speedup* ideal
- *Speedup Sublineal:* es una aceleración menor al ideal
- *Speedup Superlineal:* aceleración que supera el *speedup* ideal

Para evaluar el *speedup* del algoritmo AGC**D**-VRPTW se utilizó una sola instancia de los benchmarks de Gehring y Homberger de 1000 clientes. La instancia R1_10_1 fue seleccionada debido a que es la que requiere más tiempo en encontrar una solución. Las pruebas se realizaron en la Grid Morelos lanzando 20, 40, 60, 80, 100 y 120 procesos respectivamente, utilizando subpoblaciones de 150 individuos,

donde cada una de las pruebas fue ejecutada 30 veces. Los resultados promedio del análisis de aceleración, obtenido a partir de la fórmula 6.3 se muestra en la tabla 6.21.

Tabla 6.21 *Cálculo del Speedup para la instancia R1_10_1 de 1000 clientes*

No. Procesos	$\text{Tiempo}_{paralelo}$	$\text{Tiempo}_{secuencial}$	Speedup
20	432	7736	17.9074074
40	502	17772	35.4023904
60	516	27208	52.7286822
80	545	37944	69.6220183
100	565	48940	86.619469
120	610	63416	103.960656

La figura 6.12 muestra la aceleración correspondiente al speedup ideal y al real, obtenido a partir de la tabla 6.20, donde de acuerdo a los resultados obtenidos, el algoritmo muestra una aceleración sub-lineal y un buen desempeño para la ejecución de procesos sin sobrecarga.

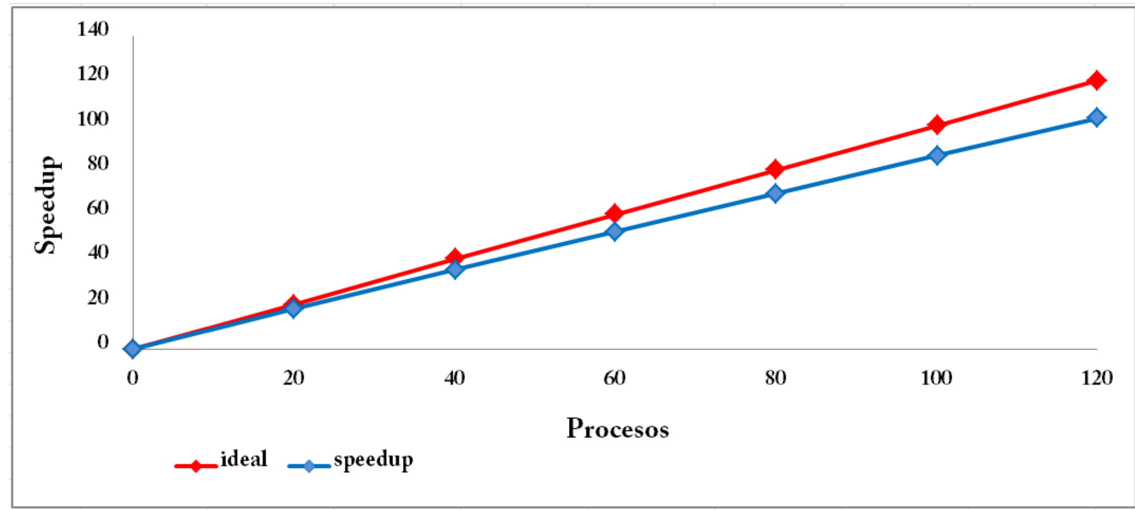


Figura 6.12 *Calculo del Speedup, correspondiente a la aceleración del algoritmo comparado con el speedup ideal. En la gráfica se observa un comportamiento sub-lineal comparado con el speedup ideal.*

En el caso de existir sobrecarga en los procesos, el algoritmo muestra un buen desempeño, obteniendo solo un mínimo incremento en las comunicaciones. Este comportamiento se debe a la limitante del ancho de banda disponible, el cual varía si

se utiliza en el clúster de forma local o si se hace uso de la Grid, donde se tiene un ancho de banda de 30 Mpbs.

El comportamiento del algoritmo con base a las comunicaciones se puede observar directamente en el monitor ganglia durante el tiempo de ejecución. Ganglia es un sistema de monitoreo para infraestructuras de cómputo de alto rendimiento, como clústeres y Grids, el cual permite visualizar el comportamiento de la infraestructura en cuestiones del uso de los recursos, lo cuál permitió observar el uso y comportamiento de la red durante la ejecución del algoritmo distribuido, lo cual se muestra en la figura 6.13.

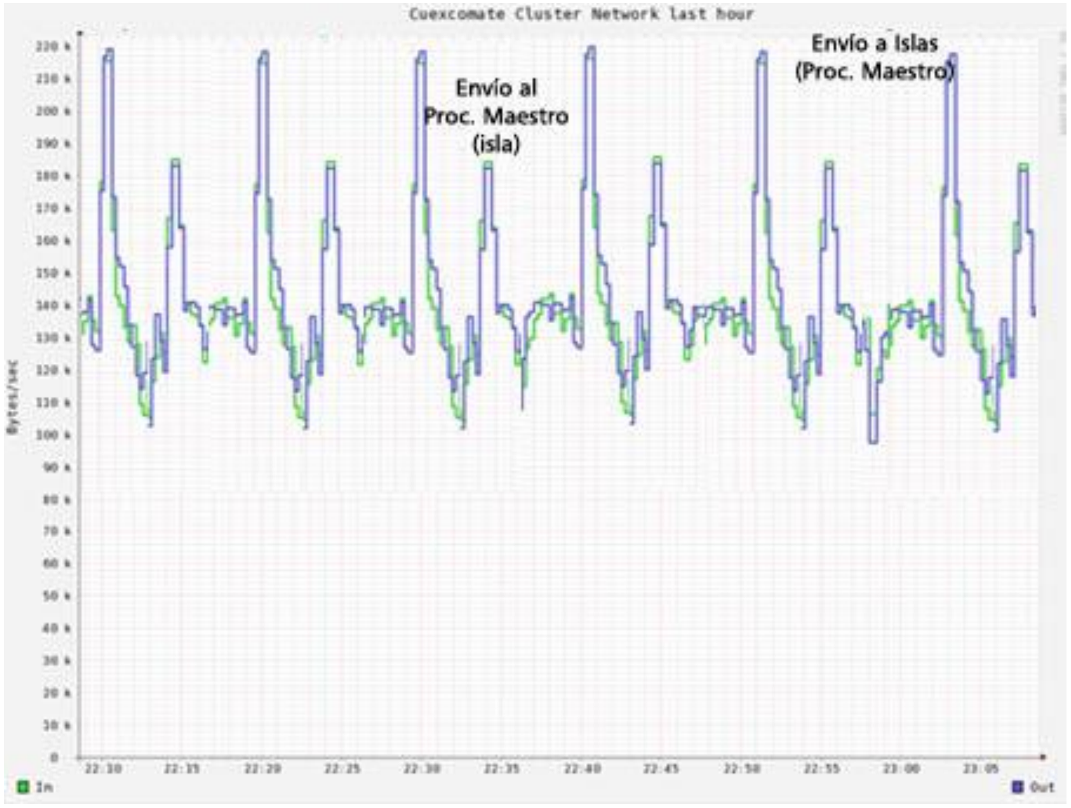


***Figura 6.13**. Comportamiento de las comunicaciones durante la ejecución del algoritmo distribuido AGC**D**-VRPTW utilizando el 100% de los núcleos de la Grid.*

En la figura 6.13 se puede observar el comportamiento de red durante la ejecución del algoritmo distribuido, donde el eje de las *x* indica la el tiempo durante el que el algoritmo se ha estado ejecutando y el eje de las *y* indica la cantidad de bytes/seg. que se están enviando. Con base en lo anterior, se puede observar un comportamiento uniforme que permite identificar los envíos realizados por el proceso maestro (los picos más altos), correspondientes a las subpoblaciones, las cuales se reparten por medio de comunicación colectiva, en seguida se muestra un pico más pequeño el cual corresponde al proceso donde las islas regresan la subpoblación al proceso maestro, con lo que se obtiene un desempeño eficiente con respecto al uso de las comunicaciones y a los tiempos de ejecución obtenidos.

6.5 Algoritmo AGCP-VRPTW

El algoritmo AGCP-VRPTW, descrito en la sección 4.3, corresponde a un algoritmo desarrollado con programación híbrida, lo que lo define como un algoritmo paralelo-distribuido. Para el manejo de este enfoque se utilizó MPI-CUDA, lo que incrementa la dificultad inherente en la paralelización de un algoritmo, debido a que MPI trabaja con programación distribuida con paso de mensajes y CUDA con *hilos* paralelos.

Para el desarrollo de un algoritmo paralelo-distribuido eficaz y eficiente, es importante identificar las funciones que requieren mayor tiempo computacional. Esto con la finalidad de identificar las funciones que serán paralelizarlas con CUDA y cuáles serán manejadas por la CPU mediante paso de mensajes. Otro de los puntos esenciales para el buen desempeño de un algoritmo, es la distribución uniforme de los procesos sobre la infraestructura a utilizar, así como el manejo de la comunicación entre procesos.

Con base en lo anterior, en esta sección se describe el análisis aplicado al algoritmo paralelo-distribuido AGC**P**-VRPTW. En la sección 6.5.1 se describen las pruebas experimentales llevadas a cabo para evaluar la eficacia del algoritmo utilizando los benchmarks de Gehring y Homberger. Posteriormente, en la sección 6.5.2, se presenta el análisis de eficiencia en la Grid, donde se hace énfasis en la importancia de la distribución de *hilos* y *bloques.* Posteriormente se explican los resultados de las pruebas experimentales con sobrecarga realizadas sobre la CPU-GPU, donde se presenta un análisis del efecto de las comunicaciones, así como del efecto de los accesos a memoria para la obtención de los cálculos de aceleración del algoritmo.

6.5.1 Análisis de Eficacia

En esta sección se presentan los resultados del análisis de eficacia llevado a cabo a la versión paralelo-distribuida AGCP-VRPTW del algoritmo propuesto en este trabajo de investigación, sobre la infraestructura de la Grid Morelos (sección 6.1) utilizando

los benchmarks de Gehring y Homberger de 1000 clientes y la sintonización de parámetros mostrada en el capítulo 5.

A diferencia del algoritmo distribuido AGC**D**-VRPTW analizado en la sección 6.4, para la versión paralelo-distribuida AGC**P**-VRPTW el análisis se enfoca en el desempeño de la hibridación MPI-CUDA, haciendo énfasis en la sección paralelizada para GPUs, correspondiente al operador de mutación cooperativa mediante un ACH. Los detalles de los resultados obtenidos se enfocan en los efectos de los parámetros utilizados, el uso de la jerarquía de memoria, la sincronización y la cooperación entre *hilos* para lo cual se realizaron pruebas experimentales utilizando subpoblaciones de 50, 100, 150 y 200 individuos respectivamente, obteniendo los mejores resultados con subpoblaciones de 150 individuos y ejecutando una *grid* de 1792 *bloques*. Es importante recordar que el tamaño de la *grid* es igual al tamaño de la *Colonia*. Los resultados obtenidos correspondientes a los 60 benchmarks de Gehring y Homberger se muestran en la tabla 6.22.

Tabla 6.22 *Resultados obtenidos por el algoritmo paralelo-distribuido AGCP-VRPTW para los benchmarks de Gehring y Homberger.*

Instancia	Veh	Mejor	Peor	Prom	σ	Veh	Cota	ER (%)
C1_10_1	100	**42478.95**	**42479.14**	**42479.02**	**0.139**	100	**42478.95**	**0**
C1_10_2	90	**42278.45**	42279.26	42278.85	0.573	90	**42278.45**	0
C1_10_3	90	**40207.71**	40207.71	40207.71	0	90	**40207.71**	0
C1_10_4	90	**39468.6**	39468.75	39468.65	0.112	90	**39468.6**	0
C1_10_5	100	**42469.18**	42469.72	42469.32	0.424	100	**42469.18**	0
C1_10_6	100	**43832.08**	**43832.79**	**43832.39**	**0.506**	99	**43830.21**	**0.0043**
C1_10_7	97	**43453.92**	43453.92	43453.92	0	97	**43453.92**	0
C1_10_8	**94**	**41853.36**	**42149.58**	**41982.36**	*	**93**	**42149.58**	**-0.7**
C1_10_9	91	**40570.98**	**40571.96**	**40571.42**	**0.697**	90	**40570.6**	**0.0009**
C1_10_10	90	**39933.06**	39933.71	39933.28	0.483	90	**39933.06**	0
C2_10_1	30	**16879.24**	16879.24	16879.24	0	30	**16879.24**	0
C2_10_2	29	**17126.39**	17126.39	17126.39	0	29	**17126.39**	0
C2_10_3	28	**16884.08**	16884.08	16884.08	0	28	**16884.08**	0
C2_10_4	28	**15656.75**	15656.75	15656.75	0	28	**15656.75**	0
C2_10_5	30	**16561.29**	16561.63	16561.41	0.251	30	**16561.29**	0
C2_10_6	30	**16921.04**	**16921.55**	**16921.3**	**0.361**	29	**16920.33**	**0.0042**
C2_10_7	29	**17882.42**	17882.42	17882.42	0	29	**17882.42**	0

C2_10_8	29	**16577.43**	**16578.02**	**16577.91**	**0.492**	28	**16577.32**	**0.0007**
C2_10_9	29	**16370.44**	16370.44	16370.44	0	29	**16370.44**	0
C2_10_10	28	**15944.72**	15944.72	15944.72	0	28	**15944.72**	0
R1_10_1	100	**53560.85**	53560.85	53560.85	0	100	**53560.85**	0
R1_10_2	92	**49106.58**	**49107.75**	**49107.29**	**0.846**	91	**49105.21**	**0.0028**
R1_10_3	91	**45237.29**	45237.29	45237.29	0	91	**45237.29**	0
R1_10_4	91	42788.36	42788.99	42788.51	0.503	91	42787.19	0.0027
R1_10_5	92	51832.47	51832.88	51832.72	0.297	91	**51830.36**	0.0041
R1_10_6	91	**47849.05**	47849.05	47849.05	0	91	**47849.05**	0
R1_10_7	91	**44435.5**	44435.5	44435.5	0	91	**44435.5**	0
R1_10_8	91	**42486.12**	**42487.02**	**42486.56**	**0.637**	91	**42485.38**	**0.0017**
R1_10_9	91	**50490.49**	50490.49	50490.49	0	91	**50490.49**	0
R1_10_10	91	**48294.71**	48295.03	48294.87	0.226	91	**48294.71**	0
R2_10_1	19	**42188.86**	42188.86	42188.86	0	19	**42188.86**	0
R2_10_2	19	**33512.83**	**33512.83**	**33512.83**	**0**	19	**33512.83**	**0**
R2_10_3	19	**24940.32**	24940.32	24940.32	0	19	**24940.32**	0
R2_10_4	19	**17926.45**	17926.45	17926.45	0	19	**17926.45**	0
R2_10_5	19	**36232.18**	**36232.18**	**36232.18**	**0**	19	**36232.18**	**0**
R2_10_6	19	**30091.93**	30091.93	30091.93	0	19	**30091.93**	0
R2_10_7	19	**23257.36**	23257.36	23257.36	0	19	**23257.36**	0
R2_10_8	19	**17497.16**	**17497.9**	**17497.48**	**0.528**	19	**17495.51**	**0.0094**
R2_10_9	19	**33008.56**	**33009.48**	**33008.93**	**0.663**	19	**33002.36**	**0.0188**
R2_10_10	19	**30215.24**	30215.24	30215.24	0	19	**30215.24**	0
RC1_10_1	90	**46272.07**	46272.07	46272.07	0	90	**46272.07**	0
RC1_10_2	90	**44129.42**	44129.42	44129.42	0	90	**44129.42**	0
RC1_10_3	91	**42488.64**	**42489.16**	**42488.86**	**0.372**	90	**42487.54**	**0.0026**
RC1_10_4	91	**41613.58**	**41613.58**	**41613.58**	**0**	90	**41613.58**	**0**
RC1_10_5	90	**45564.81**	45564.81	45564.81	0	90	**45564.81**	0
RC1_10_6	91	**45303.67**	45303.67	45303.67	0	90	**45303.67**	0
RC1_10_7	90	**44903.8**	44903.8	44903.8	0	90	**44903.8**	0
RC1_10_8	90	**44367.12**	**44367.93**	**44367.41**	**0.595**	90	**44366.01**	**0.0025**
RC1_10_9	90	**44280.84**	44280.84	44280.84	0	90	**44280.84**	0
RC1_10_10	90	**43897.17**	**43898.02**	**43897.78**	**0.656**	90	**43896.78**	**0.0009**
RC2_10_1	**20**	**30278.48**	**30278.99**	**30278.67**	*	**20**	**30278.5**	**-0.000066**
RC2_10_2	18	**26327.92**	26327.92	26327.92	0	18	**26327.92**	0
RC2_10_3	18	**20050.71**	20050.71	20050.71	0	18	**20050.71**	0
RC2_10_4	19	**15747.13**	15747.13	15747.13	0	18	**15747.13**	0
RC2_10_5	19	**27237.68**	27238.85	27238.14	0.846	18	**27237.68**	0
RC2_10_6	19	**26798.24**	**26798.86**	**26798.53**	**0.439**	18	**26797.76**	**0.0018**

RC2_10_7	18	**25112.77**	25112.77	25112.77	0	18	**25112.77**	0
RC2_10_8	18	**23709.29**	23709.29	23709.29	0	18	**23709.29**	0
RC2_10_9	18	**23028.1**	23028.1	23028.1	0	18	**23028.1**	0
RC2_10_10	18	21967.99	21967.99	21967.99	0	18	**21965.94**	0.0093

De acuerdo al análisis realizado a los resultados obtenidos por el algoritmo AGCP-VRPTW para los benchmarks de Gehring y Homberger, mostrados en la tabla 6.21, se observa una mejora en la calidad de las soluciones para las instancias C1_10_6, C1_10_9, C2_10_6, C2_10_8, R1_10_2, R1_10_8, R2_10_8, R2_10_9, RC1_10_3, RC1_10_8, RC1_10_10, RC2_10_6, las cuales a pesar de no llegar a la mejor cota, logran reducir su error relativo, obteniendo valores muy cercanos a las mejores cotas reportadas, donde el error relativo máximo es de 0.0094 y el mínimo de 0.0007, los cuales comparados con los obtenidos por el algoritmo distribuido presentan una mejora considerable.

Por su parte, 4 instancias más alcanzaron la mejor cota conocida, donde R2_10_2, R2_10_5 y RC1_10_4 obtuvieron el mejor resultado reportado en cada una de las 30 ejecuciones del algoritmo. Por el contrario, la instancia C1_10_1, alcanzó la mejor cota reportada, aunque no en todas las ejecuciones, obteniendo una desviación estándar de 0.139.

En el caso de las mejoras obtenidas a las cotas reportadas en la literatura, estas se mantienen con el mismo valor de la función objetivo obtenido por el algoritmo distribuido AGCD-VRPTW, con la diferencia de una reducción en el valor de la desviación estándar, lo que indica una menor dispersión entre las soluciones de la muestra.

Es importante mencionar que al tratarse de un método de solución híbrido, es necesario evaluar cada una de las partes que lo componen. En la sección 6.4 se muestra el análisis realizado a la versión distribuida del algoritmo con respecto al número de procesos enviados a cada uno de los núcleos de procesamiento de la CPU, obteniendo el punto de convergencia del algoritmo. Para el caso del algoritmo AGCP-VRPTW, además de contener los procesos distribuidos analizados en la sección anterior, incluye una sección paralelizada en CUDA, correspondiente al operador de

mutación cooperativa aplicando un ACH. Un algoritmo paralelizado con CUDA contiene una característica fundamental que define su eficacia, la descomposición y distribución de datos en *bloques* e *hilos*, mejor conocido como *granularidad*. Por lo que una baja granularidad (grano grueso), indica la implementación de paralelismo de tareas, por el contrario, el manejo de paralelismo de grano fino favorece la ejecución simultánea de un mismo proceso sobre diferentes datos en múltiples núcleos de procesamiento.

Efecto de la cantidad de Bloques e Hilos

En el caso del algoritmo paralelo-distribuido AGC**P**-VRPTW, se implementó un diseño basado en paralelismo de grano fino (paralelismo de datos) donde una hormiga $h \in Colonia$ corresponde a un *bloque* y cada cliente $i \in N$ a visitar por la hormiga, se maneja como un *hilo*. Por lo que, el tamaño de la *Colonia* está definido por el tamaño de la *grid*, correspondiente a la cantidad de *bloques* ejecutados por el *kernel* y el tamaño del *bloque* se encuentra en función del tamaño de la instancia. De acuerdo a esto, el tamaño de la *grid* es parte crucial para el desempeño del algoritmo, ya que de ello depende la cantidad de agentes (hormigas) encargados del proceso de optimización, además del grado de ocupación de la infraestructura.

Para realizar el análisis correspondiente a la evaluación del efecto del tamaño de la *grid* sobre la eficacia de las soluciones, se llevaron a cabo pruebas experimentales donde se realizan 30 ejecuciones para cada una de las instancias representativas de los benchmarks de Gehring y Homberger de 1000 clientes, donde se evalúa el promedio de las soluciones obtenidas con respecto a la cantidad de *bloques* lanzados durante la ejecución. Los resultados obtenidos se muestran en las figuras 6.14 a y b.

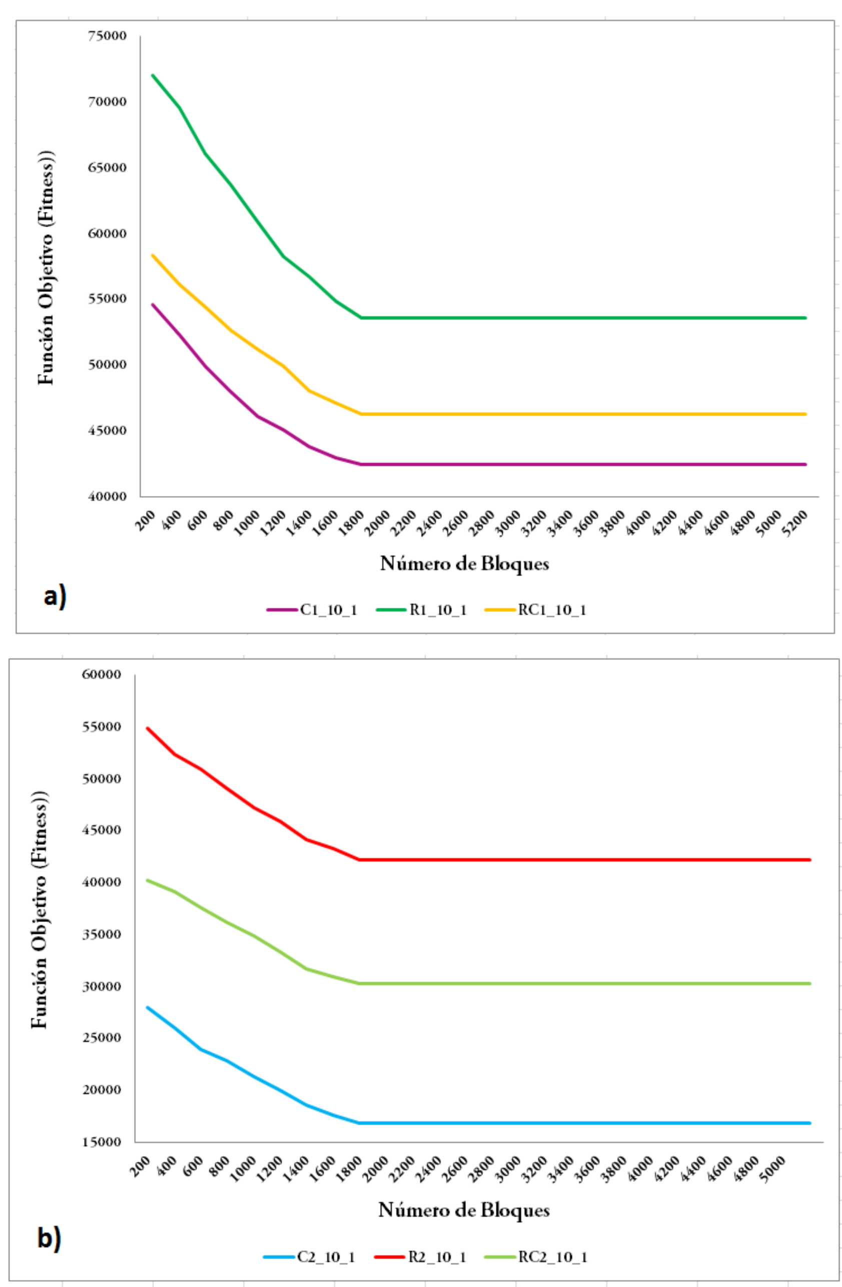


Figura 6.14 *Efecto del número de bloques sobre el valor de la función objetivo para los benchmarks de Gehring y Homberger de 1000 clientes. a) benchmarks tipo 1, b) benchmarks tipo 2.*

Las figuras 6.14 muestran el comportamiento de la función objetivo conforme se incrementa el número de bloques utilizados, con lo que se observa claramente el punto de convergencia del algoritmo, el cual tiene una variación mínima entre cada tipo de instancia.

La implementación del ACH como operador de mutación, muestra que el número de bloques constituye parte fundamental del proceso de exploración y explotación realizado mediante el método de optimización para la aplicación del operador de mutación cooperativa. Lo anterior se justifica en el hecho de que cada bloque constituye la estructura de una hormiga con *N hilos* cada una, donde cada *hilo* representa un cliente a visitar. Bajo este enfoque, el tamaño de los *bloques* se mantiene constante, por lo cual las pruebas de eficacia se realizaron con base al tamaño de la *grid* a ejecutar.

De acuerdo a los resultados de las pruebas experimentales graficadas en la figura 6.14*a*, donde se evalúa el valor de la función objetivo con respecto al número de bloques, se concluye que las instancias de tipo 1 requieren un mayor número de bloques para alcanzar las mejores cotas, lo que implica el manejo de una colonia más grande con respecto a los resultados obtenidos para las instancias tipo 2, mostrado en la figura 6.14b, que involucran restricciones más relajadas. Aunado a esto, cabe mencionar que los resultados experimentales mostraron que el número de bloques a ejecutar es sensible a las características de la instancia a tratar, como se muestra en la tabla 6.23, donde se presentan los puntos de convergencia de cada instancia con respecto al total de bloques en ejecución requeridos para la convergencia del algoritmo.

Tabla 6.23 *Puntos de convergencia para cada una de las instancias representativas de los benchmarks de Gehring y Homberger de 1000 clientes.*

Instancia	Total de Bloques
C1_10_1	1792
R1_10_1	1792
RC1_10_1	1792
C2_10_1	1680
R2_10_1	1736
RC2_10_1	1680

6.5.2 Análisis de Eficiencia

La evaluación de la eficiencia de un algoritmo paralelizado en GPUs es una tarea compleja, ya que requiere de la evaluación de diversos factores que juegan un papel fundamental en el desempeño de este tipo de algoritmos, como es el caso de impacto del número de *hilos* y *bloques* utilizados para la ejecución del algoritmo, así como el efecto del modelo de memoria implementado, la ocupación del ancho de banda y la latencia.

Previo al análisis de eficiencia, es importante tomar en cuenta que aunque el modelo MPI-CUDA propuesto en el algoritmo AGC**P**-VRPTW para el operador de mutación cooperativa involucra la ejecución de tres *kernels*, su comportamiento está en función del *kernel "Mutación"*, ya que el número y distribución de bloques utilizado en la mutación de individuos, permitirá que los *kernels* correspondientes a la actualización local y global de la feromona utilicen el mismo esquema de acceso.

Con base en lo anterior, para las pruebas de eficiencia se utilizaron las instancias representativas de los benchmarks de Gehring y Homberger de 1000 clientes sobre la infraestructura de la Grid Morelos, la cual incluye 2 tarjetas tesla C2070 en cada uno de clústeres Cuexcomate y Texcal, teniendo un total de 4 tarjetas con 448 núcleos cada una, con un total de 1792 cuda cores. Debido a que el algoritmo AGC**P**-VRPTW involucra programación MPI-CUDA, para las pruebas experimentales se tomó el número de procesos distribuidos en CPUs igual al requerido para obtener la convergencia como se muestra en la tabla 6.8. Por su parte, para la evaluación de la parte paralela del algoritmo, se lanzaron *grids* con cantidades de bloques variables, lo que indica que el tamaño de la colonia de hormigas cambia en cada ejecución, de modo que el total de *hilos* se calcula con base en la fórmula 32.

$$No.Threads = No.Bloques * TamBloque \quad (32)$$

Cabe mencionar que la el tamaño de las muestras evaluadas se definió con base al total de cuda cores disponibles en la *grid,* el cual se calcula tomando en cuenta que

cada clúster de la grid cuenta con dos tarjetas tesla C2070 con 448 cuda cores cada una, haciendo un total de 1792 cuda cores disponibles en la grid.

Para las pruebas experimentales se utilizaron las instancias representativas de los benchmarks de Gehring y Homberger, realizando 30 ejecuciones por cada instancia. Los resultados obtenidos se compararon con los obtenidos por la versión distribuida del algoritmo, mismos que se muestran en la tabla 6.24.

Tabla 6.24 *Relación entre la cantidad de bloques ejecutados por el AGCP-VRPTW con respecto al tiempo de ejecución para las instancias representativas de los benchmarks de Gehring y Homberger*

Bloques	***Tiempos de Ejecución (segs.)***					
	C1_10_1-GPU	***C1_10_1-Dist.***	***R1_10_1-GPU***	***R1_10_1-Dist.***	***RC1_10_1-GPU***	***RC1_10_1-Dist.***
224	459	811.3	461	856.1	460	834.53
448	460	1122.75	461	1167.65	460	1145.95
672	463	1734.13	463	1780.45	463	1758.85
896	463	2135.4	464	2182.6	463	2160.9
1120	467	2698.44	467	2742.15	468	2720.35
1344	469	3254.07	469	3298.97	470	3277.27
1568	470	3854.24	470	3899.14	471	3877.34
1792	470	4321.48	471	4365.98	472	4344.28
Bloques	***Tiempos de Ejecución (segs.)***					
	C2_10_1-GPU	***C2_10_1-Dist.***	***R2_10_1-GPU***	***R2_10_1-Dist.***	***RC2_10_1-GPU***	***RC2_10_1-Dist.***
224	458	801.35	462	836.19	460	812.96
448	458	1112.8	462	1147.64	460	1123.45
672	460	1724.18	465	1759.03	462	1734.82
896	461	2125.44	465	2160.29	462	2136.1
1120	463	2688.47	468	2723.29	464	2699.09
1344	464	3244.12	468	3278.98	465	3254.79
1568	463	3844.27	470	3879.09	465	3854.91
1792	465	4311.49	469	4346.33	464	4322.14

De acuerdo a los resultados mostrados en la tabla 6.23, la cantidad de bloques influye en el tamaño de la población utilizada, por lo que se realizaron pruebas

experimentales con tamaños de bloque de 224, 448, 672, 896, 1120, 1344, 1568 y 1792 respectivamente, por lo que para las pruebas con el algoritmo distribuido se tomó el número de bloques del algoritmo paralelo-distribuido como el tamaño de la población del algoritmo secuencial, con lo que se obtuvieron los resultados mostrados en la tabla 6.23 y su gráfica de comportamiento se muestra en las figuras 6.15 y 6.16, correspondientes a las instancias representativas de los benchmarks de Gehring y Homberger separados por tipo para una mejor legibilidad.

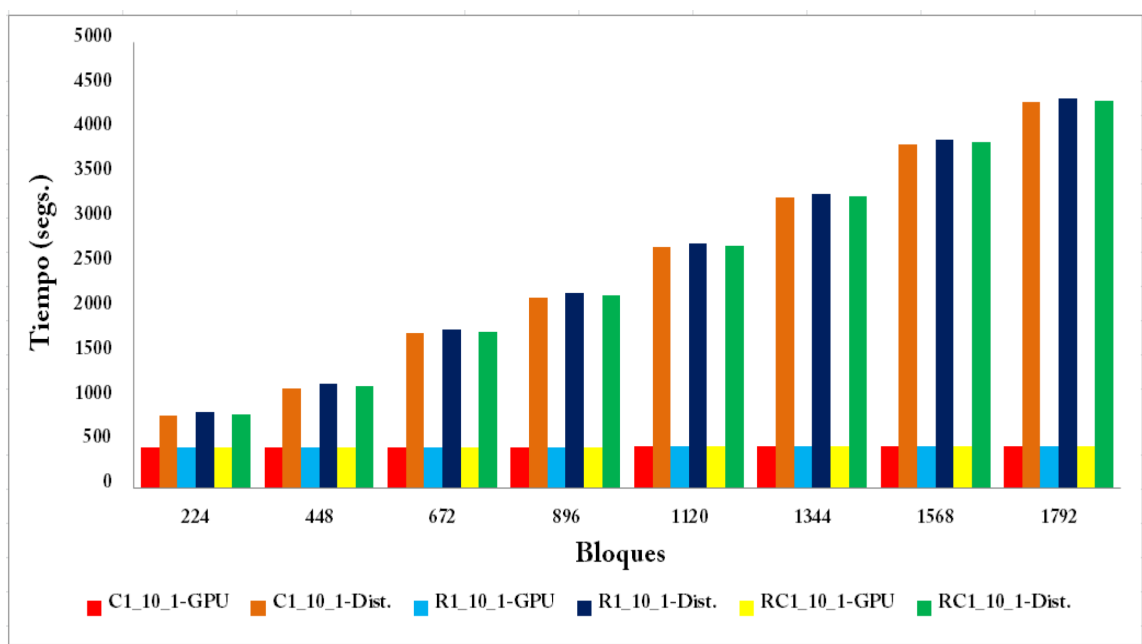


Figura 6.15. *Gráfica comparativa de los tiempos obtenidos por el algoritmo paralelo-distribuido AGC**P**-VRPTW versus la versión distribuida AGC**D**-VRPTW para instancias de Gehring y Homberger tipo 1*

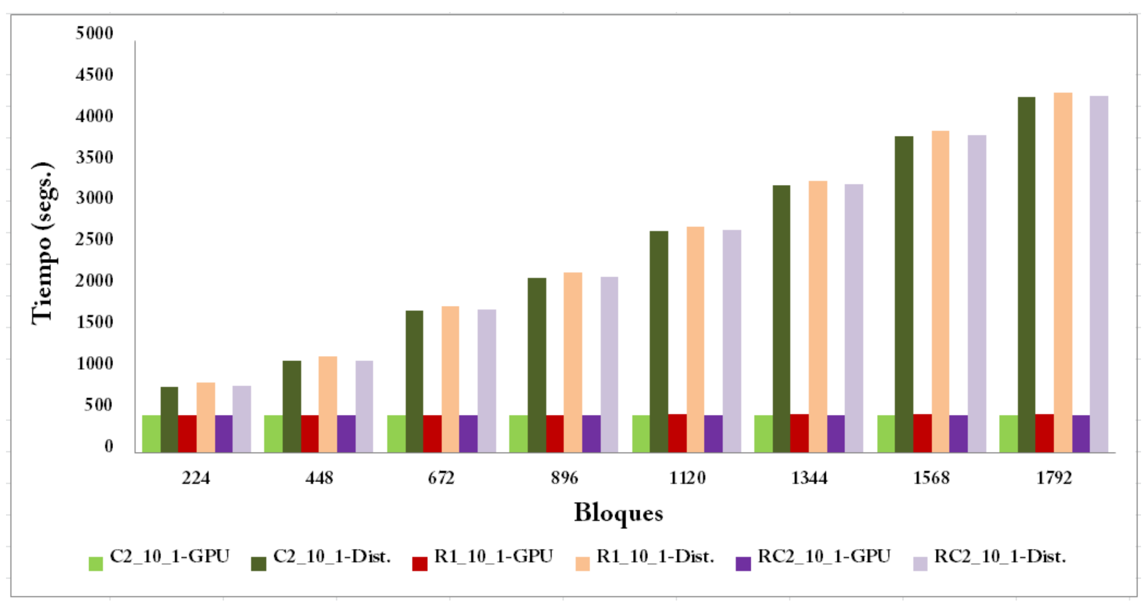


Figura 6.16. *Gráfica comparativa de los tiempos obtenidos por el algoritmo paralelo-distribuido AGC**P**-VRPTW versus la versión distribuida AGC**D**-VRPTW para instancias de Gehring y Homberger tipo 2*

De acuerdo a los resultados mostrados en las gráficas 6.15 y 6.16 se observa claramente que algoritmo paralelo-distribuido AGCP-VRPTW es en promedio 4.3x más rápido para instancias tipo 1 y 4.1x para instancias tipo 2 que el algoritmo distribuido para un tamaño de población donde el número de bloques corresponde al tamaño de la colonia, pero su eficiencia aumenta conforme se incrementa el tamaño de la población (cantidad de bloques) hasta alcanzar un promedio de 9.2x más eficiente para instancias tipo 1 y 5.7x para instancias tipo 2 que el distribuido para poblaciones que varían de acuerdo a la cantidad de procesos ejecutados, donde el algoritmo distribuido tarda en promedio 45 min. para instancias de tipo 1 y 45.38 min. para instancias de tipo 2, en comparación con 7.78 min. que tarda el algoritmo AGCP-VRPTW para instancias tipo 1 y 7.8 min. para instancias tipo 2.

6.5.2.2 Sobrecarga de Núcleos en la Grid

Para la evaluación del algoritmo paralelo-distribuido AGC**P**-VRPTW con sobrecarga, se debe tomar en cuenta el ancho de banda máximo definido por las comunicaciones

entre los clústeres de la Grid, donde la infraestructura Grid Morelos proporciona una comunicación bidireccional de 30 Mbps, la cual es una característica fundamental en el desempeño del algoritmo.

Con base en las pruebas realizadas en la sección anterior, se realizó la evaluación del algoritmo paralelo-distribuido AGC**P**-VRPTW aplicando sobrecarga únicamente en los *device*, de modo que la parte distribuida del algoritmo permanece sin sobrecarga, para lo cual se utilizaron las muestras presentadas en la tabla 6.25.

Tabla 6.25 *Muestras a utilizadas para evaluar el desempeño con sobrecarga del algoritmo paralelo-distribuido AGC**P**-VRPTW.*

Instancia	Total de Bloques	Total de Hilos
R1_10_1	672	672,000
	1344	1,344,000
	2016	2,016,000
	2688	2,688,000
	3360	3,360,000
	4032	4,032,000
	4704	4,704,000
	5376	5,376,000

Para cada uno de los rangos mostrados en la tabla 6.24 se realizaron 30 ejecuciones, de modo que los resultados promedio correspondientes al tiempo de ejecución se muestran en la figura 6.17. Cabe mencionar que el número de hilos por bloque no se modifica debido a que cada *bloque* corresponde a la solución de una hormiga, por lo tanto es un valor que permanece fijo durante todas las ejecuciones.

En la figura 6.17 se puede observar que el tiempo de ejecución aumenta de forma similar en las seis instancias al incrementar el número de bloques utilizados, esto se debe a que existe una sobrecarga de procesos sobre la grid, lo cual implica que cada SP tendrá que ejecutar un conjunto de bloques, lo cual no se puede realizar de forma concurrente.

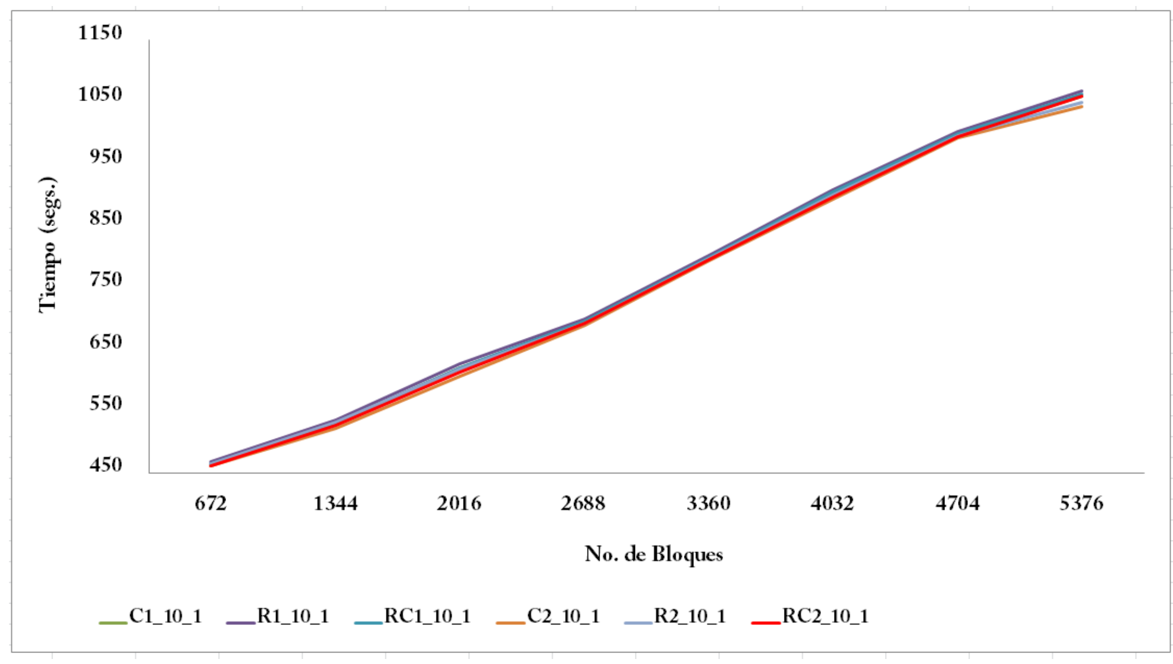


Figura 6.17 *Comportamiento del promedio de los tiempos de ejecución conforme aumenta la cantidad de bloques*

De acuerdo a los resultados correspondientes al tiempo de ejecución, mostrados en la figura 6.17 y su relación directa con respecto a la calidad de la solución, se observa que el máximo rendimiento se da al utilizar un promedio de 1792 bloques.

6.5.3 Análisis de Ocupación de la GPU

La ocupación de la GPU se encuentra en función del uso de los recursos del device, debido a que tanto los registros como la memoria compartida se reservan por cada bloque activo, lo que mantiene los recursos reservados hasta que todos los hilos del bloque hayan finalizado. Bajo este enfoque es importante tomar en cuenta algunas de las principales limitantes del rendimiento de un algoritmo en CUDA.

- Uso de los registros
- Almacenamiento en la memoria compartida
- Tamaño del bloque

Es importante tomar en cuenta estas tres características al momento de diseñar un algoritmo paralelo. En el caso del uso de los registros es importante tomar en cuenta la cantidad de recursos que requiere cada hilo ya que utilizar demasiados recursos por hilo puede limitar tanto la cantidad de hilos en un bloque, como la cantidad de bloques ejecutados concurrentemente, cayendo en una ejecución que lejos de favorecer al algoritmo, afecta su eficiencia debido a que no alcanza a ocultar la latencia de las comunicaciones. El cálculo de la ocupación de los hilos se realiza aplicando la fórmula 33

$$Ocupación_{hilos} = \left(\frac{Hilos_{Activos}}{MaxHilos_{SM}}\right)100 \qquad (33)$$

Donde el $MaxHilos_{SM}$ se encuentra definido por la tarjeta utilizada, en el caso de la Tesla C2070 cuenta con un máximo de 1536 hilos por SM. Para el caso de los $Hilos_{Activos}$ es necesario realizar la evaluación correspondiente al uso de los registros, lo cual se lleva a cabo tomando en cuenta que la tarjeta utilizada permite hasta 32768 registros por SM, de modo que la cantidad de hilos que podrán ejecutarse concurrentemente está en función de la fórmula 34.

$$Hilos_{Activos} = \frac{MaxRegistros_{SM}}{Registros_{Hilo} + 1} \qquad (34)$$

El siguiente punto a tomar en cuenta corresponde al uso de la memoria compartida, ya que al igual que los registros, ésta se reserva por bloque, por lo que ejecutar procesos que demanden mucha memoria limita la cantidad de bloques que pueden ser ejecutados de forma concurrente. En el caso de la tarjeta utilizada en este trabajo de investigación, la memoria compartida por bloque es de 48 KB, por lo que la cantidad de $hilos_{Activos}$ está en función de la demanda de memoria de cada uno, como se muestra en la fórmula 35.

$$Hilos_{Activos} = \frac{MemoriaComp_{SM}}{MemoriaComp_{Hilo}} \qquad (35)$$

De modo que la ocupación con respecto al aprovechamiento de la memoria compartida se encuentra en función de la fórmula 36.

$$Ocupación_{MemComp} = \frac{Hilos_{Activos}}{MaxHilos_{SM}} \quad (36)$$

Con base en lo anterior es importante tomar en cuenta que cada SM puede tener hasta 8 bloques activos, pero generar bloques muy pequeños limitarán el total de hilos que pueden ser ejecutados, por otro lado, al generar bloques grandes es necesario tomar en cuenta los recursos que requiere cada hilo, de lo contrario también se limitará la cantidad de hilos que podrán ser ejecutados de forma concurrente. En el caso del algoritmo AGC**P**-VRPTW, se utilizan bloques grandes de 1000 hilos, pero la demanda de recursos por parte de los hilos es mínima, debido a que el modelo se basa en paralelismo de datos (paralelismo de grano fino), el cual permite obtener un mejor escalamiento y a su vez una mayor ocupación del *device*.

Otro punto fundamental correspondiente a la eficiencia del algoritmo paralelo-distribuido AGC**P**-VRPTW consiste en el volumen de tráfico de datos existente para el acceso a los registros (fórmula 37), debido a que la información contenida en los registros es de carácter privado, por lo que cada hilo tendrá que acceder a sus respectivos registros.

$$VDatos_{SM} = Hilos_{Activos}\left(RD_{MG} + ST_{Registros}\right) * sizeof(type) \quad (37)$$

donde RD_{MG} corresponde a la cantidad de accesos a la memoria global que hace un hilo para cargar datos a sus registros, y $ST_{Registros}$ indica la cantidad de datos almacenados por un hilo.

El resultado del análisis de ocupación aplicado al algoritmo paralelo-distribuido AGCP-VRPTW con base en las fórmulas 33, 34, 35, 36 y 37 se muestra en la tabla 6.26.

Tabla 6.26 *Calculo de Ocupación del algoritmo Paralelo-Distribuido AGCP-VRPTW*

Fórmula	Aplicación	Resultado
$Hilos_{Activos} = \frac{MaxRegistros_{SM}}{Registros_{Hilo} + 1}$	$Hilos_{Activos} = \frac{32768}{23}$	**1424 hilos**
$Ocupación_{hilos} = \left(\frac{Hilos_{Activos}}{MaxHilos_{SM}}\right)100$	$Ocupación_{hilos} = \left(\frac{1424}{1536}\right)100$	**92.7%**
$Hilos_{Activos} = \frac{MemoriaComp_{SM}}{MemoriaComp_{Hilo}}$	$Hilos_{Activos} = \frac{49152\ bytes}{28\ bytes}$	1755
$Ocupación_{MemComp} = \left(\frac{Hilos_{Activos}}{MaxHilos_{SM}}\right)100$	$Ocupación_{MemComp} = \left(\frac{1755}{1536}\right)100$	114%
$VDatos_{SM} = Hilos_{Activos}(RD_{MG} + ST_{Registros}) * sizeof(type)$	$VDatos_{SM} = 1424(3+2) * sizeof(float)$	$VDatos_{SM} =$ 28480 bytes

De acuerdo a los resultados obtenidos en la tabla 6.25 se muestra que el algoritmo puede tener hasta 1424 hilos ejecutados concurrentemente, esto se obtiene tomando el rango menor entre la ocupación de registros y la ocupación de la memoria compartida. De modo que el total de $Warps_{Activos}$ de forma concurrente en cada SM se calcula con base en la fórmula 38.

$$Warps_{Activos} = \left(\frac{Hilos_{Activos}}{32}\right) / MaxWarps_{SM} \tag{38}$$

Complementando el análisis presentado, se muestran los resultados de ocupación obtenidos mediante el Occupancy Calculator, el cual es una hoja de cálculo proporcionada por Nvidia que indica la ocupación del device con base al tamaño de bloque, cantidad de registros y cantidad de memoria compartida requerida por cada hilo. El Occupacy Calculator divide los resultados en el impacto de variar el tamaño de bloque, como se muestra en la figura 6.18.

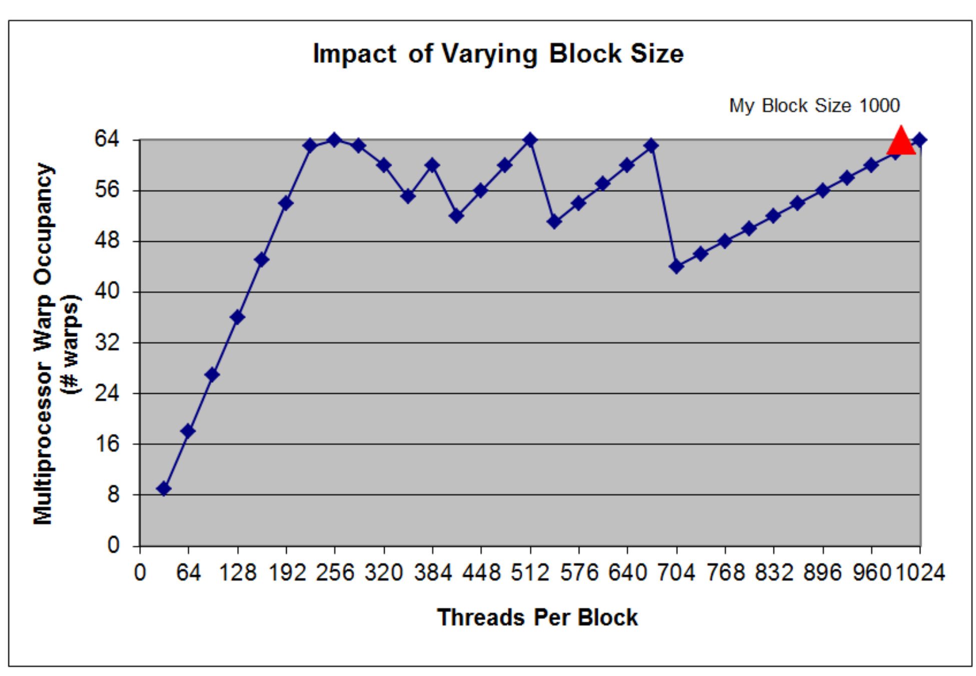


Figura 6.18. *Impacto de variar el tamaño de bloque con respecto a la ocupación del SM*

La figura 6.18 muestra la tendencia de cambiar la cantidad de hilos por bloque con respecto a la ocupación del SM, con lo que se observa que con un tamaño de bloque de 1000 elementos de obtiene una ocupación de 92.7% con base en las características de uso de los registros y la memoria compartida para cada uno de los hilos. Un comportamiento similar se muestra al considerar el impacto de variar la cantidad de registros utilizados por hilo, donde es importante lograr un equilibrio, ya que hilos con demasiados registros limitan la ocupación, contrario a lo que sucede en hilos con muy pocos registros, donde se maximiza el acceso a otros niveles de memoria, lo que afecta la eficiencia del algoritmo. Para el caso del algoritmo AGCP-VRPTW, se obtuvo un buen equilibrio, permitiendo la ejecución de 64 warps sin problemas en cuanto al límite de memoria, lo cual se observa en la figura 6.19.

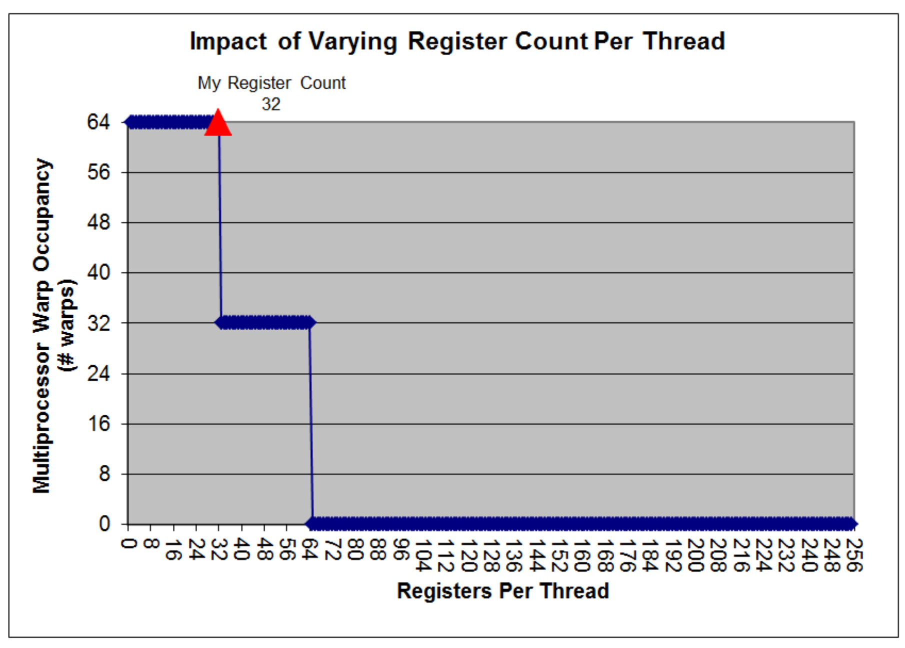


Figura 6.19. *Impacto de variar la cantidad de registros por hilo.*

De acuerdo a lo mostrado en la figura 6.19 al incrementar la cantidad de registros utilizados por hilo, se reduce la cantidad de warps a 32, lo que implica que pueden ser ejecutados un menor número de hilos, lo cual también se ve influenciado con base al uso de la memoria compartida, cuyo comportamiento se muestra en la figura 6.20.

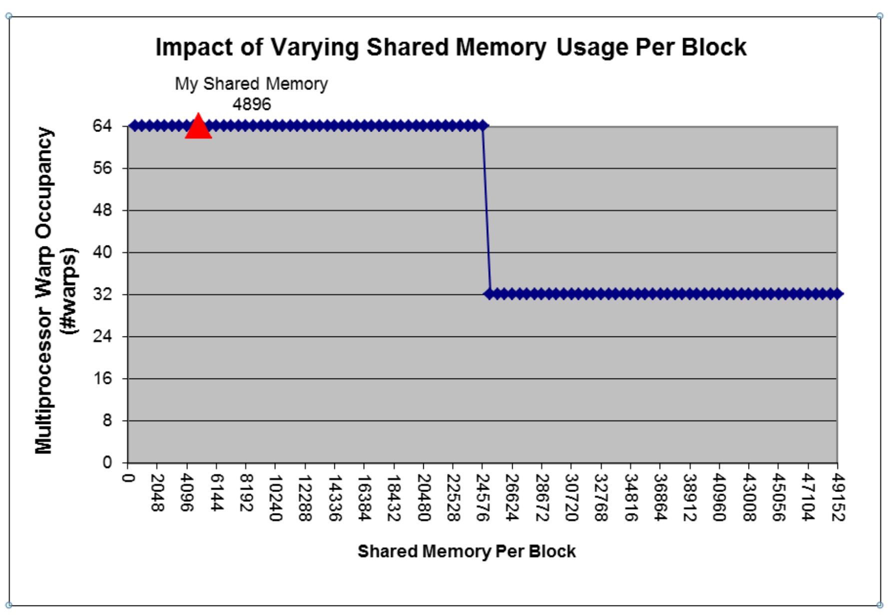


Figura 6.20. *Impacto de variar la memoria compartida utilizada por bloque.*

A figura 6.20 muestra como se reduce la cantidad de warps que pueden ser ejecutados conforme se incrementa el uso de la memoria compartida, lo que también implica la ejecución de una menor cantidad de hilos. Para el caso del algoritmo paralelo-distribuido propuesto se observa que el uso de paralelismo de grano fino favorece la ocupación de la infraestructura GPU, permitiendo obtener una ocupación del 92.7%.

Resultados Comparativos de los Algoritmos AGC-VRPTW, AGC*D*-VRPTW y AGC*P*-VRPTW

Para obtener un análisis comparativo de los tres algoritmos propuestos en este trabajo de investigación, a continuación se muestra en la tabla 6.26, una comparación de los tiempos requeridos por cada uno de los algoritmos para obtener una solución a la instancia C1_10_1.

Tabla 6.26. *Tabla comparativa de los tiempos de ejecución requeridos por cada uno de los algoritmos propuestos en esta tesis*

Bloques	***Tiempos de Ejecución (segs.)***					
	C1_10_1-GPU	***C1_10_1-Dist.***	***C1_10_1-CPU***	***C2_10_1-GPU***	***C2_10_1-Dist.***	***C2_10_1-CPU***
224	**459**	811.3	2609.58	**458**	801.35	2581.84
448	**460**	1122.75	5218.56	**458**	1112.8	5162.45
672	**463**	1734.13	7827.54	**460**	1724.18	7744.02
896	**463**	2135.4	10436.10	**461**	2125.44	10325.21
1120	**467**	2698.44	13045.15	**463**	2688.47	12907.48
1344	**469**	3254.07	15654.02	**464**	3244.12	15488.15
1568	**470**	3854.24	18264.63	**463**	3844.27	18070.10
1792	**470**	4321.48	20873.25	**465**	4311.49	20651.15

Capítulo 7

Conclusiones y Trabajos Futuros

En este capítulo se presentan las conclusiones y trabajos futuros derivados de esta tesis doctoral, haciendo especial énfasis en el trabajo con programación híbrida y programación paralela con GPUs.

7.1 Conclusiones

Esta tesis doctoral constituye un amplio trabajo de investigación que involucra el desarrollo de tres versiones del algoritmo propuesto, el primero es el secuencial que trabaja con una hribridación de un algoritmo genético con un algoritmo colonia de hormigas. La segunda versión corresponde a un algoritmo distribuido ejecutado en grid, el cual trabaja con paralelismo de tareas (paralelismo de grano grueso) y finalmente se hace un refinamiento de dicho algoritmo para obtener la versión AGCP-VRPTW la cual corresponde a un algoritmo MPI-CUDA paralelo-distribuido diseñado para la explotación de la infraestructura de la Grid Morelos tomando en cuenta tanto procesadores CPU como GPU.

Algoritmo Secuencial AGC-VRPTW

Se utilizó una metodología que al ser aplicada al VRPTW permitió mejorar tres de las mejores cotas reportadas en la literatura para los benchmarks de Solomon (R103, R110, RC105) y dos correspondientes a los benchmarks de Gehring y Homberger (C1_10_8, RC2_10_1).

La hibridación algoritmo genético – algorimo colonia de hormigas favoreció la exploración y explotación del espacio de soluciones, permitiendo alcanzar en su mayoría las mejores cotas reportadas en la literatura, tanto para los benchmarks de Solomon como para los Gehring y Homberger.

Para los casos donde no se alcanzan las mejores cotas, el porcentaje de error relativo obtenido es muy pequeño, lo que indica que se encuentran soluciones muy cercanas a las mejores reportadas.

Se comprobó experimentalmente que el uso de una infraestructura de alto rendimiento favorece la eficiencia del algoritmo AGC-VRPTW, permitiendo que fuera ejecutado de forma secuencial sobre cada uno de los núcleos de la Grid para llevar a cabo el análisis de sensibilidad distribuido y las pruebas experimentales, lo cual permitió reducir considerablemente el tiempo requerido para dichos procesos, ya que utilizando una sola computadora este proceso requiere de varios meses de experimentación.

Algoritmo Distribuido AGCD-VRPTW

El diseño y ejecución de algoritmos en Grid, permite hacer uso de recursos remotos que se encuentran dispersos geográficamente, lo cual incrementa el poder de cómputo disponible, favoreciendo la eficiencia de los algoritmos. Con lo que se comprobó que el uso masivo de recursos computacionales favorece la eficiencia y eficacia del algoritmo AGCD-VRPTW para el tratamiento del problema de ruteo vehicular con ventanas de tiempo, el cual por su naturaleza e inherente complejidad se considera como intratable.

El uso adecuado de la tecnología Grid enfocada principalmente en la infraestructura de la Grid Morelos permitió obtener mejoras en la eficiencia debido al modelo utilizado para cada una de las versiones que fueron ejecutadas en dicha infraestructura. El modelo de islas con migración implícita aplicado para la versión distribuida fue adaptado para evitar al máximo la comunicación entre procesos, para

lo cual se utiliza un esquema de migración que realiza el proceso maestro, donde se aplican los operadores genéticos de una forma relajada, con la finalidad de aplicar una pequeña migración cada dos generaciones.

Con base al modelo de islas con migración implícita, la latencia de las comunicaciones se da únicamente durante la distribución de procesos, lo cual también reduce el costo de las comunicaciones aplicando un esquema de comunicación colectiva, donde se envía un solo paquete que se encarga de distribuir la información requerida a cada proceso del comunicador, contrario a lo que sucede cuando se aplica comunicación bloqueante, la cual exige que se realice un envío independiente por cada núcleo de procesamiento.

El speedup obtenido por el algoritmo distribuido AGC**D**-VRPTW ejecutado en la Grid Morelos, muestra un comportamiento sub-lineal, ligeramente debajo del ideal, lo que indica que se trata de un algoritmo eficiente.

Algoritmo Paralelo-Distribuido AGCP-VRPTW

El desarrollo de un algoritmo paralelo en CUDA requiere que el método aplicado sea adaptado a las caracteristicas de la infraestructura para obtener el máximo rendimiento.

Para lograr un algoritmo paralelo eficiente, es necesario realizar un análisis y planeación detallada que permita el manejo de una distribución uniforme sobre los núcleos de procesamiento, así como considerar el compilador que será utilizado de acuerdo a las librerías utilizadas, ya que para el manejo de MPI, los compiladores de Intel han mostrado tener un mejor desempeño, pero desafortunadamente, para el manejo de programación híbrida no es aplicable, ya que no son compatibles con *nvcc*, por lo que la compilación MPI-CUDA hace uso del compilador de OpenMPI.

Se comprobó que la cooperación de procesos realizada tanto en la versión distribuida AGC**D**-VRPTW como paralela AGC**P**-VRPTW permite encontrar

soluciones de buena calidad, ya que al realizar la migración implícita se lleva a cabo una cooperación mutua entre todas las subpoblaciones, lo que favorece la diversidad en las subpoblaciones.

La comparación de las tres versiones de los algoritmos en cuanto a eficiencia demuestra que un algoritmo paralelizado para GPUs y ejecutado en Grid es más eficiente que uno distribuido en una Grid. Esto debido a que MPI trabaja con paralelismo de grano grueso sobre CPUs, a diferencia de CUDA que permite trabajar con paralelismo de grano fino (paralelismo de datos), lo que mejora la eficiencia de las aplicaciones y más aún al ser ejecutadas sobre procesadores gráficos en una Grid de alto rendimiento.

De acuerdo a los resultados obtenidos, se obtuvo una mejora en la aceleración del algoritmo de 4.3x para instancias de tipo 1 y 4.1x para instancias de tipo 2 comparado con el algoritmo distribuido. Cabe mencionar que la eficiencia del algoritmo AGC**P**-VRPTW se incrementa al aumentar el tamaño de la entrada, llegando a obtener una aceleración promedio de 9.0x para colonias de 5400 individuos, por lo que se puede decir que se trata de un algoritmo que a mayor cantidad de individuos más oculta la latencia de las comunicaciones y la divergencia que pueda existir entre los hilos durante el proceso de mutación.

Como resultado de las pruebas experimentales, se obtuvo que la versión AGC**P**-VRPTW demostró ser eficaz y eficiente. En el caso de la eficacia, el algoritmo AGC**P**-VRPTW obtiene resultados similares a los obtenidos por el algoritmo secuencial AGC-VRPTW, solo permitiendo que algunas de las instancias que no había alcanzado la mejor cota, lleguen a ella. En cuanto a la eficiencia, el algoritmo demostró ser en promedio 7 veces más rápido que su versión distribuida, el AGC**D**-VRPTW, lo cual se obtuvo con el uso de la grid a un 90% debido al tamaño de las instancias utilizadas.

7.2 Trabajos Futuros

El presente trabajo de investigación permitió involucrarse en una todavía nueva área correspondiente al cómputo paralelo en GPUs, la cual se encuentra en auge y constante innovación debido a que frecuentemente se presentan mejoras a dicho hardware. El trabajar con tecnología innovadoras crea nuevos nichos de oportunidad para el desarrollo de algoritmos que permitan obtener soluciones a problemas que con un solo núcleo de procesamiento sería imposible tratar. Por lo cual, uno de los muchos trabajos a seguir es el desarrollo de algoritmos tanto páralo-distribuidos como paralelos enfocados en el tratamiento de problemas cercanos a entornos reales.

Continuar con la investigación y aplicación de métodos que permitan mejorar la eficiencia de los algoritmos paralelos, enfocados en la reducción de latencia generada por accesos no coherentes a memoria y por la divergencia de los *hilos*.

Aplicar nuevos métodos de reducción al algoritmo AGCP-VRPTW con la finalidad de incrementar la eficiencia del algoritmo.

Buscar nuevos métodos de balancear los procesos de exploración y explotación al trabajar con poblaciones muy grandes, ya que de esta forma existe la posibilidad de poder lograr una mejora en la eficacia y eficiencia del algoritmo paralelo-distribuido.

Seguir trabajando con el desarrollo de algoritmos paralelos en GPUs, ya que constituye un campo muy amplio que requiere de mucho tiempo para lograr perfeccionar las técnicas de paralelización.

Buscar métodos que permitan medir el speedup para algoritmos paralelizados en GPUs.

Referencias

Aarts E., Korst J. (1989). *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. John Wiley & Sons, Gran Bretaña. pp. 272.

Aida K. (2005). *A Case Study in Running a Parallel Branch and Bound Application on the Grid.* IEEE/IPSJ. Symposium on Applications and the Internet. pp. 164 – 173.

Arbelaitz O., Rodríguez C. (2000). *A High Efficiency Parallel Algorithm for the VRPTW based on Simulated Annealing.* Proceedings of the JCIS, Vol 1. Pp. 411 – 416.

Avinash K., Pandey Monu, Deshpande Ajinkya, Babu Rajasekhara. (2013). *Improved GPU Co-processor Sorting Algorithm with Barrier Synchronization.* International Journal of Applied Engineering Research. ISSN. 0973-4562. Vol. 8. No. 19.

Bai Hongtao, Ouyang Dantong, Li Ximing, He Lili, Yu Haihong. (2009). *MAX-MIN Ant System on GPU with CUDA.* Fourth International Conference on Innovative Computing. Information and Control.

Balseiro S. (2007). *Un Algoritmo de Colonia de Hormigas para el Problema de Ruteo con Costos Dependientes del Tiempo y con Ventanas de tiempo (TDVRPTW).*

Beasley J.E. (1997). *Route-first Cluster-second Methods for the Multi-trip Vehicle Routing and Scheduling Problem.* European Journal of Operational Research, 100:180-191.

Belding Theodore C. (1995). *The Distributed Genetic Algorithm Revisited*. In L. Eshelman Ed. Proceedings of the 6th. International Conference on GAs. pp. 114 – 121. Morgan Kaufmann.

Blocho Miroslaw, Czech Zbigniew. (2013). *A Parallel Memetic Algorithm form the Vehicle Routing Problem with Time Windows.* Eight International Conferences on P2P, Parallel, Grid, Cloud and Internet Computing.

Bent R., Van Hentenryck P. (2001). *A Two-Stage Hybrid Local Search for the Vehicle Routing Problem with Time Windows.* Technical Report CS-01-06. Department of Computer Science. Brown University.

Berger J., Barkaoui M., Bräysy O. (2001). *A Parallel Hybrid Genetic Algorithm for the Vehicle Routing Problem with Time Windows*. Working paper, Defense Reasearch Establishment Valcartier, Canada.

Brandão de Oliveira Humberto César, Vasconcelos Germano Crispim. (2008). *A Hybrid Search Method for the Vehicle Routing Problem with Time Windows.* Springer Science+Bussiness Media.

Bräysy Olli (2001). *Genetic Algorithms for the Vehicle Routing Problem with Time Windows.* Special Issue on Bioinformatics and Genetic Algorithms.

Bräysy O., Gendreau M., Hasle G., Lokketangen A. (2002). *A Survey of Rich Vehicle Routing Models and Heuristic Solution Techniques.* Technical Report. SINTEF.

Campbell Ann Melissa, Savelsbergh Martin. (2004). *Efficient Insertion Heuristics for Vehicle Routing and Scheduling Problems.* INFORMS. Transportation Science. Vol. 38 issue 3

Cecilia José M., García José M., Nisbet Andy, Amos Martyn, Ugaldón Manuel. (2013). *Enhancing Data Parallelism for Ant Colony Optimization on GPUs.* Journal of Parallel and Distributed Computing. Elsevier. pp. 42-51.

Cecilia José M., Llanes Antonio, Chang Li-Wen, García José M., Navarro Nacho, Hwu Wen-Mei. (2014). V-ACO: A Vectorization Approach for High-Performance Ant Colony Optimization. Proceedings of the InternationalConference on Metaheuristics and Nature Inspired Computing. Francia.

Cheng John, Grossman Max, McKercher Ty. (2014). *Professional Cuda C Programming.* Wiley Ed. ISBN. 1-1187-3927-2, 978-111-873-927_3.

Clarke G., Wright J. W. (1964). *Scheduling of Vehicles from a Central Depot to a Number of Delivery Points.* Operations Research. Vol. 12, Issue 4, pp. 568 – 581.

Cook Shane. (2013). *Programming: A Developer's Guide to Parallel Computing with GPUs.* Morgan Kaufmann. Elsevier. ISBN. 978-0-12-415933-4. pp. 203-205.

Cordeau J.F., Desaulniers G., Desrosiers J., Solomon M. M., Soumis F. (2001). *The VRP with Time Windows*. The Vehicle Routing Problem, SIAM Monographs on Discrete Mathematics and Applications. P. Toth and D. Vigo (eds). pp. 157−194, SIAM, Philadelphia.

Cortéz A. (2004). *Teoría de la Complejidad Computacional y Teoría de la Computabilidad.* Revista Investigación y Sistemas de Información. pp. 102-105.

Cruz-Chávez Marco Antonio, Martínez-Oropeza Alina, Zavala-Díaz José Crispín, Martínez-Rangel Martín G. (2009). *Relajación del Problema de Calendarización de Trabajos en un Taller de Manufactura utilizando un Grafo Bipartita*. CICos 2009, 7to Congreso Internacional de Cómputo en Optimización y Software, ISBN(e) 978-607-00-1970-8, pp 178-188, 17-20 Noviembre, México, 2009.

Cruz-Chávez Marco Antonio, Martínez-Oropeza Alina, Sergio Serna Barquera A. (2010). *Neighborhood Hybrid Structure for Discrete Optimization Problems*. Electronics, Robotics and Automotive Mechanics Conference, CERMA2010, IEEE-Computer Society, ISBN 978-0-7695-4204-1, pp 108 - 113, September 28 - October 1, México, 2010.

Cruz-Chávez Marco Antonio, Martínez-Oropeza Alina, Ávila-Melgar E. Y., Rivera-López Rafael. (2010b). *Relaxation of Job Shop Scheduling Problem using a Bipartite Graph.* Electronics, Robotics and Automotive Mechanics Conference, CERMA2010, IEEE-Computer Society, ISBN 978-0-7695-4204-1, pp 132 - 136, September 28 - October 1, México.

Cruz-Chávez Marco Antonio, Rodríguez-León Abelardo, Rivera-López Rafael, Juárez-Pérez Fredy, Peralta-Abarca Carmen, Martínez-Oropeza Alina. (2012). *Grid Platform Applied to the Vehicle Routing Problem with Time Windows for the Distribution of Products.* Chapter 3. Logistics Management and Optimization through Hybrid Artificial Intelligence Systems. IGI Global. pp. 52–83.

Cruz-Chávez Marco Antonio, Martínez-Oropeza Alina. (2013). *B-Tree Algorithm Complexity Analysis to Evaluate the Feasibility of its Application in the University Course Timetabling Problem*. Journal of Artificial Intelligence and Soft Computing Research, Polish Neural Network Society, ISSN: 2083-2567, vol. 3, No. 4, pp. 251 - 263, 2013.

Cruz-Chávez Marco Antonio, Martínez Oropeza Alina, Peralta-Abarca J. del C., Cruz-Rosales Martín H., Martínez-Rangel Martín. (2014). *Variable Neighborhood Search for Non-deterministic Problems*. Lecture Notes in Computer Science, Springer Verlag Pub., Switzerland, ISSN: 0302-9743, Vol.8468, No. 2, pp. 468-478, 2014.

Cruz-Chávez Marco Antonio, Martínez-Oropeza Alina, Martínez-Rangel Martín, Moreno-Bernal Pedro, Pecina Federico Alonso, Juárez-Chavez Jazmín Yanel, Flores-Pichardo Mireya. (2014b). *Experimental Analysis with Variable Neighborhood Search for Discrete Optimization Problems.* Enclyclopedia of Information Science and Technology. Third Edition. Information Resources Management Association. ISBN. 978-1-4666-5888-2, e-ISBN. 978-1-4666-5889-9. USA.

Dantzig G. B., Ramser J. H., (1959). *The Truck Dispatching Problem.* Institute for Operations Research and Management Sciences (INFORMS). Management Science, Vol. 6, No. 1. pp. 80–91.

Dawson Laurence, Stewart Iain. (2013). *Improving Ant Colony Optimization Performance on the GPU using CUDA.* IEEE Congress on Evolutionary Computation (CEC). E-ISBN. 978-1-4799-0452-5. Print-ISBN. 978-1-4799-0453-2. pp. 1901-1908.

De Donno Danilo, Esposito Alexandra, Torricone Luciano, Catarinucci Luca. (2010). *Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD*. IEEE Antennas and Propagation Magazine Vol. 52, No. 3. Italia, pp. 1804-1808.

Díaz Parra Ocotlán. "*Algoritmo Evolutivo Paralelizado para una Cadena de Suministro con Ventanas de Tiempo*". Tesis de Doctorado. Universidad Autónoma del Estado de Morelos. México

Diego Francisco Javier, Gómez Eva María, Ortega-Mier Miguel, García-Sánchez Alvaro. (2012). *Parallel CUDA Architecture for Solving VRP with ACO.* Industrial Engineering: Innovative Networks. Chapter 43. Springer-Verlag London.

Dorigo Marco (1992). *Optimization, Learning and Natural Algorithms.* Ph.D. Thesis, Politecnico di Milano, Italy. DT.01-POLIMI92.

Dorigo Marco, Maniezzo Vittorio, Colorni Alberto. (1996). *Systems, Man, and Cybernetics.* Part B: Cybernetics, IEEE Transactions on. ISSN: 1083-4419. DOI: 10.1109/3477.484436. pp. 29-41.

Dorigo Marco (1992). *Optimization, Learning and Natural Algorithms.* Ph.D. Thesis, Politecnico di Milano, Italy. DT.01-POLIMI92.

Dorigo M., Birattari M., Stutzle T. (2006). *Ant Colony Optimization Artificial Ants as a Computational Intellicente Technique.* IEEE Computational Intelligence Magazine.

Dorigo M., Stützle T. (2004). *Ant Colony Optimization.* MIT Press.

Fu Jie, Lei Lin, Zhou Guohua. (2010). *A Parallel Ant Colony Optimization Algorithm with GPU-Acceleration Based on All-in-Roulette Selection*. Third International Workshop on Advanced Computational Intelligence. China.

Gács P., Lovász L. (1999) *Complexity of Algorithms*. Lecture Notes for Spring 1999. http://www.cs.yale.edu/homes/lovasz/notes.html.

Gambardella Luca Maria, Taillard Éric, Giovanni Agazzi (1999). *MACS-VRPTW: A Multiple Ant Colony System for Vehicle Routing Problems with Time Windows.* New

Ideas in Optimization. Capítulo 5, pp. 63 – 67. Editores: In D. Corne, M. Dorigo, F. Glover. Mc. Graw-Hill, UK.

Garey M. R. and Johnson D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, New York. ISBN. 0-7167-1044-7.

Gehring, H. and Homberger J. (1999). "*A parallel Hybrid Evolutionary Metaheuristic for the Vehicle Routing Problem with Time Windows"*. In: K. Miettinen, M. Makela, and J. Toivanen (eds.), Proceeding of EUROGEN99 - Short Course on Evolutionary Algorithms in Engineering and Computer Science, pages 57-64. University of Jyväskylä.

Gehring H. and Homberger J. (2001). *Parallelization of a Two-phase Metaheuristic for Routing Problems with Time Windows.* Asia-Pacific. Journal of Operations and Reasearch. 18. pp. 35-42.

Gilmour Stephen, Dras Mark. (2005). *Understanding the Pheromone System within Ant Colony Optimization.* Proceedings of the 18th Australian Joint Conference on Advances in Artificial Intelligence. Springer-Verlag Berlin Heidelberg. ISBN. 3-540-30462-2. pp. 786 – 789.

Golden Bruce, Raghavan S., Wasil Edward. (2008). *The Vehicle Routing Problem. Latest Advances and New Challenges.* Springer Science + Business Media. ISBN. 978-0-387-77777-1, e-ISBN. 978-0-387-77778-8. DOI:10.1007/978-0-387-77778-8. USA.

Gorges-Schleuter Martina. *Explicit Parallelism of Genetic Algorithms through Population Structures*. In H.P. Schwefel and Reinhard Männer Ed. Parallel Problem Solving from Nature. pp. 150 – 159. Springer-Verlag.

Govindaraju N.K., Larsen S., Gray J., Manocha D. (2006). *A Memory Model for Scientific Algorithms on Graphics Processors.* IEEEXplore. Proceedings of the ACM/IEEE. E-ISBN. 0-7695-2700-0. pp. 6.

Grassé P. (1959). *La Reconstruction du nid et les Coordinations Inter-individuelles Chez Bellicostitermes Natalensis et cubitermes.* sp. La Theorie de la Stigmergie: Essai

d'interpretation du Comportement des Termites Constructeurs. Inserctes Soc. 61:41 – 81.

Gupta Deepti, Ghafir Shabina. (2012). *An Overview of Mathods ManteiningDiversity in Genetic Algorithm*. International Journal of Emerging Technology and Advanced Engineering. Vol. 2. Issue. 5. ISSN. 22502459.

Hamming Richard W.. (1950). *Error Detecting and Error Correcting Codes*. The Bell System Technical Journal; Vol. XXVI, No. 2, pp. 147-160.

Hartmanis J., Hopcroft J.E. (1971). *An Overview of the Theory of Computational Complexity.* Journal of the ACM (JACM) archiveVolume 18, Issue 3. pp. 444 – 475. ISSN. 0004-5411. Publisher ACM.

Holland J. H. (1962). *Outline for a Logical Theory of Adaptive Systems.* Journal of the Association for Computing Machinery, 9:297-314.

Holland J. H. (1962). *Concerning Efficient Adaptive System.* In M.C. Yovits, G. T.Jacobi, G.D. Goldstein. pp. 215-230. Spartan Books. USA.

Holland J. H. (1967). *A Universal Computer Capable of Executing an Arbitrary Number of Subprograms Simultaneously.* In Proceedings of East Joint Computer Conference. pp. 8-113.

Holland J. H. (1975). *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor MI.

Homberger Jörg, Gehring Hermann. (1999). *Two Evolutionary Metaheuristics for the Vehicle Routing Problem with Time Windows.* INFOR 37. pp. 297-318.

Homberger Jörg, Gehring Hermann. (2005). *A Two-phase Hybrid Metaheuristic for the Vehicle Routing Problem with Time Windows.* European Journal of Operational Research. Elsevier. Volumen 162, Issue 1. pp. 220-238.

Hongzhong Shan. (2011). *Hybrid Programming for Multicore Processors.* IEEEXplore. e-ISBN. 978-0-7695-4335-2. Fourth International Joint Conference on Computational Sciences and Optimization. pp. 261 – 262.

Hopcroft J., Ullman J. (1993). *Introducción a la Teoría de Autómatas.* Ed. CECSA.

Hord R. Michael. (1990). *Parallel Supercomputing in SIMD Architectures*. ISBN. 0-8493-4271-6. International Standard Book Number 0-8493-4271-6. pp. 5 – 6.

Huang Zhe Xue. (1998). *Extensions to the k-means Algorithm for Clustering Large Data Sets with Categorical Values*. Data Mining and Knowledge Discovery. pp. 283 –304. Netherlands.

Jacobsen Dana A., Thibault Julien C., Senocak Inanc. (2010). *An MPI-CUDA Implementations for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters.* 48th. Aerospace Science Meeting. AIAA, USA

Jian-Ming Li, Hong-Song Tan, Xu Li, Lin-Lin Liu. (2010). *A Parallel Simulated Annealing Solution for VRPTW based on GPU Acceleration.* Advances in Intelligent Decision Technologies. G. Phillips-Wren et al. (Eds). Springer-Verlag Berlin Heidelberg. pp. 201-208.

Karunadasa N.P., Ranasinghe D.N. (2009). *Accelerating High Performance Applications with CUDA and MPI.* IEEE Xplore. International Conference on Industrial and Information Systems. ISBN. 978-1-4244-4837-1. pp. 331-336.

Karypis George, Hong Eui, Kumar Vipin. (1999). *CHAMALEON: A Hierarchical Clustering Algorithm using Dynamic Modeling*. IEEE Computer. Vol. 32. pp. 68–75. ISBN. 0018-9162.

Kirk David B., Hwu Wen-mei W. (2013). *Programming Massively Parallel Processors. A Hands-on Approach.* Second edition. Morgan Kaufmann, Elsevier. NVidia. ISBN: 978-0-12-415992.1. pp. 42. USA.

Knight K., J. Hofer. (1968). *Vehicle Scheduling with Timed and Connected Calls: A Case Study*. Operational Research Quarterly, 19:299-310.

Kouki S., Jemni M., Ladhari T. (2011). *Solving the Permutation Flow Shop Problem with Makespan Criterion using Grids.* International Journal of Grid and Distributed Computing. Vol 4. No. 2. pp. 53 – 64.

Le Bouthiller Alexandre, Crainic Teodor Gabriel. (2005). *A Cooperative Parallel Meta-heuristic for the Vehicle Routing Problem with Time Windows.* Computers & Operations Research. Elsevier. Volume 32, Issue 7. pp 1685-1708.

Lenstra J.K., Rinnoy H. G.K. (1981). *Complexity of Vehicle Routing and Scheduling Problems.* Networks. Vol 11. No. 2. pp. 221 - 227.

Lozano Manuel, Herrera Francisco, Cano José Ramón. (2008). *Replacement Strategies to Preserve Useful Diversity in Genetic Algorithms*. Elsevier. Information Sciences. Pp. 4421 – 4433. DOI: 10.1013. 2008.

Madsen O.B.G.. (1976). *Optimal Scheduling of Trucks – A Routing Problem with Tight Due Times for Delivery*. In H. Strobel, R. Genser, and M. Etschmaier, editors, Optimization Applied to Transportation Systems, IIASA. International Institute for Applied System Analysis, Laxenburgh, Austria. pp. 126-136.

Magoulès Frédéric. (2010). *Fundamentals of Grid Computing. Theory, Algorithms and Technologies.* CRC Press. Taylor & Francis Group. ISBN. 978-1-4398-0367-7.

Martí, R. (2003). *Procedimientos Metaheurísticos en Optimización Combinatoria. Matemátiques.* Vol.1. No. 1. Pp. 3 – 62.

Martin Richard P., Vahdat Amin M., Culler David E., Anderson Thomas E. (1997). *Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture.* ISCA '97. Proceedings of the 24th Annual International Symposium on Computer Architecture. pp. 85-97.

Martínez-Oropeza Alina. (2010). *Solución al Problema de Máquinas en Paralelo mediante un Algoritmo de Colonia de Hormigas.* Tesis de Maestría. Universidad Autónoma del Estado de Morelos. pp. 3,6,34,77,97.

Martínez-Oropeza Alina, Cruz-Chávez Marco Antonio. (2011). *Método de Agrupamiento no Supervisado para el Problema de Ruteo Vehicular con Restricciones de Capacidad en Vehículos*. Memorias del 8vo. Congreso Internacional de Cómputo en Optimización y Software. México.

Martínez-Oropeza Alina, Cruz-Chávez Marco Antonio, Moreno-Bernal Pedro. (2012). *Distancia Hamming: Método y Desarrollo de un Algoritmo Computacional para el Problema de Ruteo Vehicular con Ventanas de Tiempo*. 9no. Congreso Internacional de Cómputo en Optimización y Software. México.

Martínez-Oropeza Alina, Cruz-Chávez Marco Antonio, Cruz-Rosales Martín H., Moreno-Bernal Pedro, Peralta-Abarca Jesús Del Carmen. (2012b). *Unsupervised Clustering Method for the Capacited Vehicle Routing Problem, Electronics, Robotics and Automotive Mechanics Conference*. CERMA2012, IEEE-Computer Society, ISBN: 978-0-7695-4878-4, pp 211-216, November 20-23, México, 2012.

Melab N., Mezmaz M., Talbi E.G. (2006). *Parallel Cooperative Meta-heuristic on the Computational Grid. A Case Study: The Bi-objective Flow Shop Problem,* Elsevier Parallel Computing. DOI:10.1016/j.parco.2006.01.003. pp. 643 – 659.

Memarsadegui Nargess, Mount David M., Netanyahu Nathan S., Moigne Jacqueline Le. *A Fast Implementation of the Isodata Clustering Algorithm. International Journal of Computational Geometry & Applications*. pp. 71–103.

Morales-Navarro R. E., Cruz-Chavez M. A., Rivera-López R., Rodríguez-León A., Martínez-Oropeza A., Moreno-Bernal P.. (2011). *Paralelización de un Algoritmo de Búsqueda Local Iterada para el Problema del Agente Viajero.* Congreso Internacional de Cómputo en Optimización y Software, CICOS2011, ISBN: 978-607-00-5091, pp 16-25, Cuernavaca Morelos, 2011

Moreno-Bernal Pedro, Cruz-Chávez Marco Antonio, López Otoniel, Malumbres M. P., Martínez-Oropeza Alina, Flores-Pichardo Mireya, Peralta-Abarca J. del Carmen. (2013). *Tuning an Iterated Local Search Algorithm for Wavelet Sign Coding for 2D Image Compression*. ICMEAE2013, IEEE-Computer Society, ISBN: 978-1-4799-2252-9, pp 72-77, November 19-22, México, 2013.

Mount David M. (2005). *Kmlocal: A Testbed for k-means Clustering Algorithms.* University of Maryland and David Mount. Partially Supported by the National Science Foundation.

Mühlenbein H. Evolution in Tine and Sapce: The Parallel Genetic Algorithm. In G. Rawlins Ed., FOGA -1. pp. 316 -337. Morgan Kaufmann.

Nagata Yuichi. (2007). *Efficient Evolutionary Algorithm for the Vehicle Routing Problem with Time Windows: Edge Assembly Crossover for the VRPTW.* IEEE Xplore. Congress on Evolutionary Computation (CEC 2007). Pp. 1175-1182. e-ISBN: 978-1-4244-1340-9, p-ISBN: 978-1-4244-1339-3.

Neapolitan Richard E. (2014). *Foundations of Algorithms.* Jones & Bartlett Learning. 5th. Edition. pp. 26-41. ISBN. 978-1-284-04919-0. Northwestern University. USA.

Nickolls John, Buck Ian, Garland Michael. (2008). *NVidia. Magazine of GPU Computing on Scalable Parallel Programming with CUDA.* Vol 6. Issue. 2. pp. 40-53.

NVidia. (2008). *Nvidia CUDACompute Unified Device Architecture. Programming Guide.* Versión 2.0.

Ochi Luiz S., Vianna Dalessandro S., Drummond Lucia M. A., Victor André O. (1998). A Parallel Evolutionary Algorithm for the Vehicle Routing Problem with Heterogeneous Fleet. Parallel and Distributed Processing. Lecture Notes in Computer Science. Vol. 1388 / 1998. pp. 216 – 224. Computer Science.

Ombuki B., Ross BJ, Hanshar F. (2006). *Multi-objective Genetic Algorithms for Vehicle Routing Problem with Time Windows.* Appl Intell; 24(1). pp. 17-30.

Owens John D., Houston Mike, Luebke David, Green Simon, Stone John E., Phillips James C. *GPU Computing. Graphics Processing Units – Powerful Programmable and Highly Parallel – are Increasingly Targeting General- Purpose Computing Applications.* IEEEXplore. Vol. 96. No. 5.

Pacheco Peter S. (1997). *Parallel Programming with MPI.* Morgan Kaufmann Publishers. ISBN. 1-55860-339-5. pp. 293.

Papadimitriou C., Steiglitz K. (1982). *Combinatorial Optimization: Algorithms and Complexity.* Prentice Hall. Englewood Cliffs, New Jersey, USA. ISBN 0-13-152462-3. pp. 496.

Papadimitriou C., Steiglitz, K. (1998). *Combinatorial Optimization: Algorithms and Complexity.* Dover Publications Inc. New York, USA.

Pérez J., Pazos R., Reyes G., Landero V., Frausto J., Cruz L. (2005). *A Statistical Learning-based Methodology for Algorithm Selection.* The ACM Journal of Experimental Algorithms. Vol. 10.

Pettey C.B., Lauze M.R., Grefenstette J.J. (1987). *A Parallel Genetic Algorithm.* Proceedings of the 2nd. International Conference on Genetic Algorithms and their Apllication (ICGA). John J. Greffenstette Ed. Lawrence Erlbaum Associates Publishers.

Pullen H., Webb M. (1967). *A Computer Application to a Transport scheduling Problem*. Computer Journal, 10:10-13.

Qi Chenming, Cui Shoumei, Sun Yunchuan. (2008). *Using Ant Colony System and Local Search Methods to Solve VRPTW.* IEEE Pacific-Asia Workshop on Computational Intelligence and Industrial Application. Vol. 2. ISBN:978-0-7695-3490-9. pp. 478-482.

Rabenseifner Rolf. (2003). *Hybrid Parallel Programming on HPC Platforms.* Proceedings of the Fifth European Workshop on OpenMP. EWOMP'03. Alemania.

Rabenseifner Rolf, Hager Georg, Jost Gabriele. (2009). *Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes.* IEEE Xplore. ISSN: 1066-6192. ISBN: 978-0-7695-3544-9. pp. 427-436.

Rochat Y., Taillard E.D. (1995). *Probabilistic Diversification and Intensification in Local Search for Vehicle Routing.* Journal of Heuristics. Vol.1. pp. 147-167.

Rodríguez-León Abelardo, Cruz-Chávez Marco Antonio, Rivera-López Rafael, Ávila-Melgar Erika Yesenia, Juárez-Pérez Fredy, Cruz-Rosales Martín Heriberto. (2010). *A Communication Scheme for an Experimental Grid in the Resolution of VRPTW using an Evolutionary Algorithm.* Electronics, Robotic and Automotive Mechanics Conference. IEEE Computer Society. DOI 10.1109/CERMA32010.8.

Roosta Seyed H. (2000). *Paralell Processing and Parallel Algorithms. Theory and Computation.* Springer-Verlag. ISBN. 0-387-98716-9. pp. 117-120. New York.

Ross Sheldon M. (2012). *Simulación.* Quinta edición. Prentice-Hall.

Rousseau L.M., Gendreau M., Pesant G. (2002). *Using Constraint-based Operators to Solve the Vehicle Routing Problem with Time Windows.* Journal of Heuristics. Klumer Academic Publishers. pp. 43-58.

Salveson, M. E. (1952). *On a Quantitative Method in Production Planning and Scheduling Econometrica*. 20(4): 554-590-1952.

Savelsbergh M.W.P. (1985). *Local Search for Routing Problems with Time Windows.* Transportation Science 29(2). pp. 156-166.

Sanders Jason, Kandrot Edward. *CUDA by Example. An Introduction to General Purpose GPU Programming.* Addison-Wesley Ed. ISBN-13. 978-0-13-138768-3. ISBN-10. 0-13-138768.5.

Siegfried Benkner, Brandes Thomas. (2001). *High-Level Data Mapping for Clusters of SMPs. High-Level Parallel Programming Models and Supportive Environments.* 6th International Workshop. Frank Mueller (Ed.). Springer-Verlag Berlin Heidelberg. ISBN: 3-540-41944-6.

Silberstein Mark. (2014). *GPUs: High Performanca Accelerators for Parallel Applications.* Ubiquity Symposium: The Multicore Transformation. DOI:10.1145/2618401. pp. 1-13.

Skolicki Azbigniew, De Jong Kenneth. (2005). *The Influence of Migration Sizes and Intervals on Island Models.* Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation. pp. 1295 – 1302. ISBN:1-59593-010-8. USA.

Sipser Michael. (2013). *Introduction to the Theory of Computation.* Second Edition. Thomson Course Technology. ISBN. 0-534-95097-3.

Solomon M. M. (1987). *Algorithms for Vehicle Routing and Scheduling Problems with Time Window Constraints*. INFORMS. Operations Research. Vol. 35, No. 2. 0030-364X/87/3502-0254. pp. 254 -265.

Stan Salvador, Chan Philip. (2004). *Determining the Number of Clusters / Segments in Hierarchical Clustering / Segmentation Algorithms*. 16th. IEEE International Conference on Tools with Artificial Intelligence. ISBN. 0-7695-2236-X.

Szymon Jagiello, Dominik Zelazny. (2013). *Solving Multi-criteria Vehicle Routing Problem by Parallel Tabu Search on GPU.* International Conference on Computational Science (ICCS). Elsevier.

Tan K. C., Chew Y. H., Lee L. H.. (2006). *A Hybrid Multiobjective Evolutionary Algorithm for Solving Vehicle Routing Problem with Time Windows.* Computational Optimization and Applications. Springer Science + Bussiness Media, Inc. Manufactured in The Netherlands. Vol. 34, pp. 115-151. DOI:10.1007/s10589-005-3070-3.

Tanese Reiko. (1989). *Distributed Genetic Algorithms.* In J. D. Schaffer Ed. Proceedings of the Third International Conference on Genetic Algorithms. pp. 434-439. Morgan Kaufmann.

Tarditi David, Puri Sidd, Oglesby Jose. (2006). *Accelerator: Using Data Parallelism to Program GPUs for General Purpose Uses.* Proccedings of the ASPLOS Conference. Vol. 4. Issue 11. pp. 325-335.

Uchida Akihiro, Ito Yasuaki, Nakano Koji. (2013). *An Efficient GPU Implementation of Ant Colony Optimization for the Traveling Salesman Problem.* International Conference on Computing, Networking and Communications (ICNC). DOI. 10.1109/ICNC.2012.22. ISBN. 978-1-4673-4624-5. pp. 94-102.

Wang Yuping. (2012). *A Hybrid Approach based on Ant Colony System for the VRPTW.* International Conference on WTCS. Springer-Verlag Berlin Heidelberg. pp. 327 – 333.

Whitley Darrell. (1993). *A Genetic Algorithm Tutorial.* Technical Report CS-93-103.

Whitley Darrell. (1993). *An Executable Model of a Simple Genetic Algorithm.* Foundations of Genetic Algorithms 2. Morgan Kaufmann Publishers. ISBN. 1-55860-263-1. pp. 45-62.

Whitley Darrell, Rana Soraya, Heckendorn Robert B. (1998). *The Island Model Genetic Algorithm: On Separability, Population Size and Convergence.* Journal of Computing and Information Technology.

Whitley L. Darrell, Starkweather Timothy. (1990). *GENITORII: A Distributed Genetic Algorithm.* Journal of Experimental and Theoretical Artificial Intelligence, 2:189 214.

Wieczorek Bozena. (2011). *Parallel Independent Simulated Annealing Searches to Solve the VRPTW.* T. Czachórski et al. (Eds.): Man-Machine Interactions 2, AISC 103. Springer-Verlag Berlin Heidelberg. pp. 377-384.

Wilkinson Barry. (2009). *Grid Computing. Techniques and Applications.* CRC Press. Taylor and Francis Group. ISBN-13: 978-1-4200-6954-9.

Wilt Nicholas. (2013). *The CUDA Handbook. A Comprehensive Guide to GPU Programming.* Pearson Education Inc. ISBN-13. 978-0-321-80946-9, ISBN-10. 0-321-80946-7. pp. 179-180.

Wodecki Mieczyslaw, Bozejko Wojciech, Karpiński, Pacut Maciej. (2014). *Multi-GPU Parallel Memetic Algorithm for Capacited Vehicle Routing Problem.* PPAM 2013 Parte II. LNCS 8385. Springer-Verlag Berlin Heidelberg. pp. 207-214.

Wu Chih-Sheng. (2003). *A Study of Heuristic Algorithms for Optimization and Clustering Problems.* Tesis para obtener el grado de Maestría en Administración de la Información. China.

Yang Xin-She. (2010). *Nature-Inspired Metaheuristic Algorithms.* ISBN-13. 978-1-905986-28-6. ISBN-10. 1-905986-28-9. UK

Yu B., Yang Z. Z., Yao B. Z. (2010). *A Hybrid Algorithm for Vehicle Routing Problem with Time Windows.* Expert Systems with Applications. Elsevier.

Envío de Tipos de Dato Estructura "struct" con Comunicación Colectiva en MPI para el VRPTW

El envío de tipos de dato *struct* en MPI no es una tarea trivial, ya que a diferencia de un tipo de dato primitivo, esta puede contener múltiples datos de diferente tipo, además de arreglos tanto unidimensionales como multidimensionales, lo cual complica considerablemente su envío, debido a que sus características no corresponden a ninguno de los tipos de datos manejados en MPI.

Para resolver la problemática descrita anteriormente, MPI permite la creación de tipos de datos derivados mediante la función MPI_Type_struct y MPI_Type_commit, cuya sintaxis se muestra en el código A.1.

```
MPI_Type_struct ( int bloques, int longitud_bloques[ ], MPI_Aint
desplazamientos[ ],  MPI_Datatype tipos_datos[ ], MPI_Datatype          (A.1)
*nuevo_tipo);
```

Para la generación del tipo de dato estructura utilizada en el algoritmo distribuido AGCD_VRPTW se utilizó la estrcutura mostrada en el código A.2.

```
typedef struct PobInicial  {
                        int Solucion[N-1];
                        int Inicio[VEH];
                        int Fin[VEH];
                        int TotDem[VEH];                (A.2)
                        float CostoVeh[VEH];
                        int NumVeh;
                        float Fitness;
                   } Poblacion [TamPob];
```

Por lo que la configuración para la generación del nuevo tipo de dato derivado de acuerdo a la sintaxis mostrada en el código A.1, se muestra en el código A.3.

```
MPI_Datatype  Individuo;

int bloques = 7;

int tipo[bloques] = {MPI_INT, MPI_INT, MPI_INT, MPI_INT,
MPI_FLOAT, MPI_INT, MPI_FLOAT};

int blocklen[bloques] = {N-1, VEH, VEH, VEH, VEH, 1, 1};

MPI_Aint  mov[bloques] = {0, (N-1)*sizeof(int),  ((N-1)+VEH) *                (A.3)
sizeof(int),  ((N-1)+(2*VEH))*sizeof(int),  ((N-1)+(3*VEH)) * sizeof(int),
(((N-1)+(3*VEH))*sizeof(int)) + (VEH * sizeof(float)),
(((N)+(3*VEH))*sizeof(int)) + (VEH * sizeof(float))};

MPI_Type_struct(bloques, bloque, mov, tipo, &Individuo);

MPI_Type_commit(&Individuo);
```

Una vez que el nuevo tipo de dato *Individuo* ha sido generado, es posible realizar el envío utilizando comunicación colectiva, para lo cual se utilizaron las funciones *MPI_Scatter*, la cual se encarga de repartir los datos a los procesos, y *MPI_Gather* que corresponde al proceso inverso de *MPI_Scatter*, es decir, recolecta los datos de los procesos para almacenarlos en un solo nodo denominado nodo maestro.

Cabe mencionar que para el caso del algoritmo genético se trabaja con poblaciones, por lo que fue necesario envíar un conjunto de individuos a cada proceso, lo cual se denomina *subpoblación*. Para esto fue necesario mapear la estructura a una matriz de estructuras, donde cada fila representa una subpoblación y cada columna, un individuo que la compone, como se muestra en la figura A.4

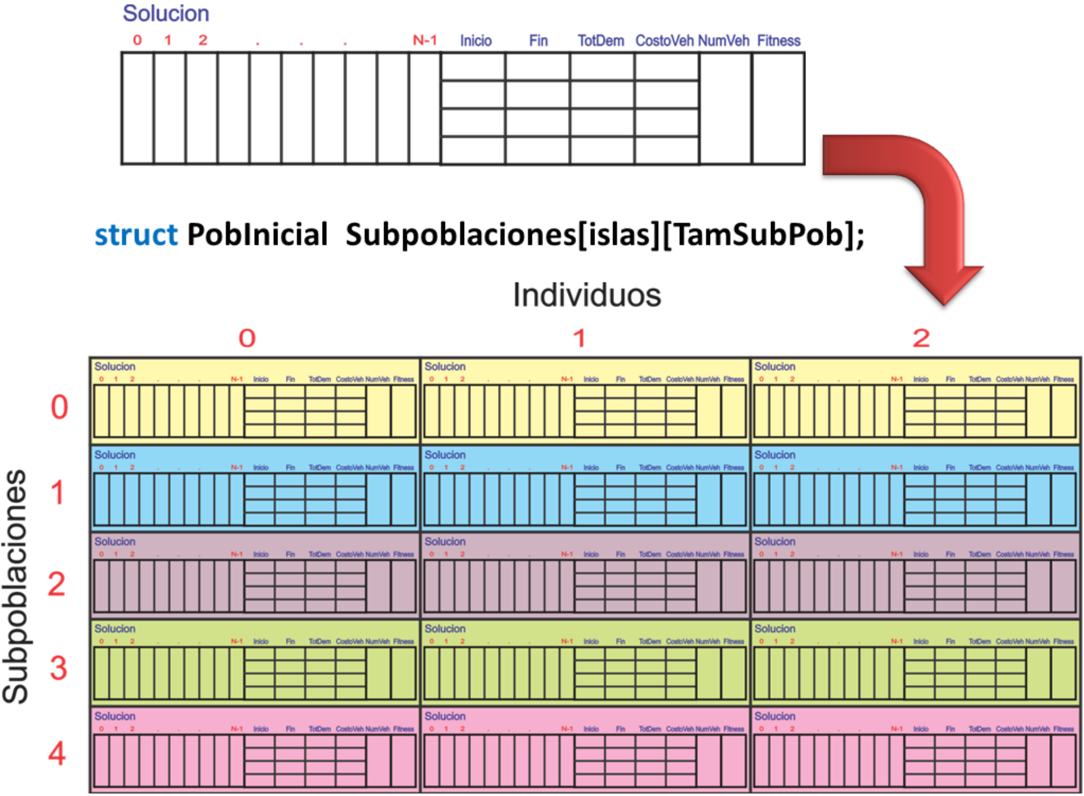


Figura A.4. *Mapeo de la estructura a una matriz de estructuras donde cada fila corresponde a una subpoblación*

La matriz de estructura, también llamada matríz de subpoblaciones debido a la información que contiene, se reparte por filas a cada proceso, para lo cual de aplicó el código A.5.

MPI_Scatter(&Subpoblaciones[rango], Subpoblacion, NuevoDato, Poblacion, 0, MPI_COMM_WORLD); (A.5)

Cada proceso recibe al mismo tiempo la información de la fila correspondiente a su *rango*. Una vez terminadas las funciones de cada proceso, se recogen las soluciones para ser concentradas en el proceso maestro, para lo cual se utiliza la función MPI_Gather, como se muestra en el código A.6.

MPI_Gather(&Poblacion, Subpoblacion, NuevoDato, &Subpoblaciones[i], Subpoblacion, NuevoDato, 0, MPI_COMM_WORLD); (A.6)

Código Fuente AGC-VRPTW

```
/*****************************************
* M.I. Alina Martínez Oropeza            *
*****************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>

#define N 101
#define VEHICLES 25
#define CAPACITY 200
#define POPULATION 1000
#define CROSSOVER 0.7
#define MUTATION 0.32
#define MUTA_LOCAL 0.3
#define CHROMUTA 3
#define ALPHA 1.9
#define BETHA 0.2
#define RO 0.86
#define DELTA 0.5
#define q0 0.4
#define T0 10
#define GENERATIONS 4500
#define COMPARE 99999

void READ_INSTANCE(int *);
int CLUSTERING();
void CALC_DISTANCES();
void SOLUCION();
void INIT_SOL(int ind, struct InitialPop *PopulationIni);
void CONST_SOLS(int ind, struct InitialPop *PopulationIni);
int TIME_CONSTRAINTS(int *, int *, int , int );
void ROUTE_COST(int , int , struct InitialPop *PopulationIni);
void STRUCTURE_INIT();
int CLOSENESS_INSERTION(int *, int , int , int *);
int DIRECT_SWAPINSERTION(int *, int , int , int *);
int SWAP_INSERTION(int *, int , int , int *);
int SWAPS_INSERTION(int *, int , int , int *);
void GENETIC(struct InitialPop *PopulationIni);
void CHROM_CROSSOVER(int IndMother, int IndFather, struct InitialPop
*PopulationIni);
void COOPERATIVE_MUTATION(struct InitialPop *PopulationIni);
```

```
int EXPLOITATION(int *vect_tabu, int *vect_aux, struct InitialPop
*Mutation, float (*Tij)[N], float *maximo, int index, int index1,
int Ant);
int EXPLORATION(int *vect_tabu, int *vect_aux, struct InitialPop
*Mutation, float (*Tij)[N], float *Pij, int index, int index1, int
Ant);
void LOCAL_UPDATING(float (*Tij)[N], struct InitialPop *Mutation,
int Ant);
void GLOBAL_UPDATING(float (*Tij)[N], struct InitialPop *Mutation);

struct benchmark {
                        int CUSTOMER;
                        int XCOORD;
                        int YCOORD;
                        int DEMAND;
                        int READY_TIME;
                        int DUE_DATE;
                        int SERVICE_TIME;
                  }instance[N];
struct clusterizing {
                        int VECTOR[N-1];
                        int LB[VEHICLES];
                        int UB[VEHICLES];
                        int TOTAL_DEMAND[VEHICLES];
                    }clusters, init_sol;
struct TimeWindows {
                        int CLIENT;
                        int START;
                        int SIZE;
                   } windows[N-1];
struct InitialPop {
                        int SOLUTION[N-1];
                        int LB[VEHICLES];
                        int UB[VEHICLES];
                        int TOTAL_DEMAND[VEHICLES];
                        float VEHCOST[VEHICLES];
                        int NUMVEH;
                        float FITNESS;
                  } PopulationIni[POPULATION],
                    SortPopulation[POPULATION], BestFitness;
struct FeaturesPop {
                        float Wik;
                        float Dik;
                   }TimesInd[N-1];

float MAT_DISTANCES[N][N];
int Vh=0;
FILE *rf;

int main()
{
   int m, ind=0, i=0, j=0, iter =0, x=0, iter=0;
   srand(time(NULL));
   READ_INSTANCE(&m);
```

```
   CALC_DISTANCES();
   STRUCTURE_INIT();
   ind=0;
   do{
      CLUSTERING();
      INIT_SOL(ind, PopulationIni);
      PopulationIni[ind].NUMVEH = Vh;
      if(PopulationIni[ind].NUMVEH <= VEHICLES)ind++;
         Vh = 0;
   }while(ind < POPULATION);
   do
   {
      GENETIC(PopulationIni); iter++;
   }while(iter<GENERATIONS);
   return 0;
}

void READ_INSTANCE(int *m)
{
   int i, j, MAT_AUX[N][7];
   int dato;   FILE *ff;

   if((ff=fopen("rc101.txt","r+"))==NULL)
   {
      printf("\n NO SE PUDO ABRIR EL ARCHIVO DE LECTURA ");
      system("pause");     exit(0);
   }
   fscanf(ff,"%d",m);
   for(i=0;i<N;i++)
   {
      for(j=0; j<7; j++)
      {
         fscanf(ff,"%d",&dato);
         MAT_AUX[i][j]=dato;
      }
   }   fclose(ff);
   for(i=0; i<N; i++)
   {
         instance[i].CUSTOMER = MAT_AUX[i][0];
         instance[i].XCOORD = MAT_AUX[i][1];
         instance[i].YCOORD = MAT_AUX[i][2];
         instance[i].DEMAND = MAT_AUX[i][3];
         instance[i].READY_TIME = MAT_AUX[i][4];
         instance[i].DUE_DATE = MAT_AUX[i][5];
         instance[i].SERVICE_TIME = MAT_AUX[i][6];
   }
}

int CLUSTERING()
{
   int index=0, i=0, j=0, position=0, centroid=0;
   int total_request=0,customer=0, vect_tabu[N];
   time_t timer1, timer2;
   float aux=0.0;
```

```
for(i=0; i<N; i++) vect_tabu[i] = 0;
do
{
   centroid = (rand()%(N-1))+1;
   while(vect_tabu[centroid] == 1) centroid = (rand()%(N-1))+1;
   clusters.VECTOR[index] = centroid;
   vect_tabu[centroid] = 1;
   clusters.LB[Vh] = index;
   total_request = instance[centroid].DEMAND;
   index++;
   if(index == (N-1))
   {
      clusters.UB[Vh] = index-1;
      clusters.TOTAL_DEMAND[Vh] = total_request;
   }
   while((total_request != 0)&&(index != (N-1)))
   {
      for(customer=1; customer<N; customer++)
      {
         if(vect_tabu[customer]==0)
         {
             aux = MAT_DISTANCES[centroid][customer];
             position = customer;
             customer = N;
             }
         }
         for(i=1; i<N; i++)
         {
            if((vect_tabu[i]!=1)&&(MAT_DISTANCES[centroid][i]<aux))
            {
               aux = MAT_DISTANCES[centroid][i];
               position = i;
            }
         }
         if(total_request + instance[position].DEMAND < CAPACITY)
         {
            total_request=total_request+instance[position].DEMAND;
            clusters.VECTOR[index] = position;
            vect_tabu[position] = 1;
            index++;
            if(index == (N-1))
            {
               clusters.UB[Vh] = index-1;
               clusters.TOTAL_DEMAND[Vh] = total_request;
            }
         }
         else if(total_request+instance[position].DEMAND==CAPACITY)
         {
            total_request=total_request+instance[position].DEMAND;
            clusters.VECTOR[index] = position;
            vect_tabu[position] = 1;
            clusters.UB[Vh] = index;
            clusters.TOTAL_DEMAND[Vh] = total_request;
            index++;
            total_request = 0;
```

```
                if(index == (N-1))
                {
                   clusters.UB[Vh] = index-1;
                   clusters.TOTAL_DEMAND[Vh] = total_request;
                }
             }
             else
             {
                clusters.UB[Vh] = index-1;
                clusters.TOTAL_DEMAND[Vh] = total_request;
                total_request = 0;
             }
          }
       }
       if((Vh + 1) < VEHICLES) Vh++;
       else
       {
          printf("\n No hay mas vehiculos disponibles !!!\n");
          system("\n pause");
       }
    }while(index < (N-1));
    return 0;
}


void CALC_DISTANCES()
{
    int i=0, j=0;
    float distanceA = 0.0, distanceB = 0.0;

    do
    {
       for(j=0; j<N; j++)
       {
          if(i==j) MAT_DISTANCES[i][j] = 0;
          else
          {
             distanceA =(instance[i].XCOORD - instance[j].XCOORD);
             distanceA = distanceA * distanceA;
             distanceB =(instance[i].YCOORD - instance[j].YCOORD);
             distanceB = distanceB * distanceB;
             MAT_DISTANCES[i][j] = sqrt(distanceA + distanceB);
             MAT_DISTANCES[j][i] = MAT_DISTANCES[i][j];
          }
       }
       i++;
    }while(i < N);
}

void INIT_SOL(int ind, struct InitialPop *PopulationIni)
{
    int aux=0, vehic=0, i=0, j=0, aux1=0;
    for(i=0; i<N-1; i++)
    {
```

```
         windows[i].CLIENT = clusters.VECTOR[i];
         windows[i].START = instance[clusters.VECTOR[i]].READY_TIME;
         windows[i].SIZE = instance[clusters.VECTOR[i]].DUE_DATE –
                           instance[clusters.VECTOR[i]].READY_TIME;
      }
      SORT_TW(windows);
      CONST_SOLS(ind, PopulationIni);
}

void CONST_SOLS(int ind, struct InitialPop *PopulationIni)
{
   int vh=0, i=0, j=0, k=0, TotalCost=0, depot=0, index=0,
       VectTabu[N-1], vehics=Vh, VhSelect = -1, flag=0, flag2=0,
       client=0, insertion=0, VehTabu[VEHICLES];
   float aux = 0;

   for(i=0; i<N-1; i++) VectTabu[i] = 0;
   for(i=0; i<VEHICLES; i++)
   {
      if(i >= Vh) VehTabu[i] = -1;
      else VehTabu[i] = 0;
   }
   for(vh=0; vh<Vh; vh++)
   {
      for(i= clusters.LB[vh]; i<=clusters.UB[vh]; i++)
      {
         if(i == clusters.LB[vh])
         {
            PopulationIni[ind].SOLUTION[index] = windows[i].CLIENT;
            PopulationIni[ind].LB[vh] = index;
            VectTabu[windows[i].CLIENT -1] = 1;
            if(MAT_DISTANCES[deposit][windows[i].CLIENT]<=windows[i].START)
               TimesInd[index].Wik = windows[i].START;
            else TimesInd[index].Wik =
                 MAT_DISTANCES[depot][PopulationIni[ind].SOLUTION[index]];
            TimesInd[index].Dik = TimesInd[index].Wik + instance
                                  [windows[i].CLIENT].SERVICE_TIME;
            PopulationIni[ind].TOTAL_DEMAND[vh] = instance[PopulationIni
                                        [ind].SOLUTION[index]].DEMAND;
            if(i == N-2)
            {
               PopulationIni[ind].UB[vh] = index;
               PopulationIni[ind].VEHCOST[vh] = TimesInd[index].Dik+
                 MAT_DISTANCES[PopulationIni[ind].SOLUTION[index]][depot];
               vh++;
            }
            else index++;
         }
         else if((i > clusters.LB[vh]) && (i <= clusters.UB[vh]))
         {
            if((MAT_DISTANCES[PopulationIni[ind].SOLUTION[index-1]][windows
                  [i].CLIENT] + TimesInd[index-1].Dik) <= windows[i].START)
            {
               PopulationIni[ind].SOLUTION[index] = windows[i].CLIENT;
               VectTabu[windows[i].CLIENT - 1] = 1;
```

```
                TimesInd[index].Wik = windows[i].START;
                TimesInd[index].Dik = TimesInd[index].Wik +
                        instance[windows[i].CLIENT].SERVICE_TIME;
                PopulationIni[ind].TOTAL_DEMAND[vh] +=
                instance[windows[i].CLIENT].DEMAND;
                index++;
            }
            else if((MAT_DISTANCES[PopulationIni[ind].SOLUTION[index-
                1]][windows[i].CLIENT] + TimesInd[index-1].Dik + instance
                [windows[i].CLIENT].SERVICE_TIME) <= instance[windows[i].
                CLIENT].DUE_DATE)
            {
                PopulationIni[ind].SOLUTION[index] = windows[i].CLIENT;
                VectTabu[windows[i].CLIENT - 1] = 1;
                TimesInd[index].Wik = MAT_DISTANCES[PopulationIni[ind].
                        SOLUTION[index-1]][windows[i].CLIENT] +
                                TimesInd[index-1].Dik;
                TimesInd[index].Dik = TimesInd[index].Wik + instance
                                [windows[i].CLIENT].SERVICE_TIME;
                PopulationIni[ind].TOTAL_DEMAND[vh]+= instance[windows[i].
                                        CLIENT].DEMAND;
                index++;
            }
            if(i == clusters.UB[vh])
            {
                PopulationIni[ind].UB[vh] = index - 1;
                PopulationIni[ind].VEHCOST[vh] = TimesInd[index-1].
                Dik + MAT_DISTANCES[PopulationIni[ind].SOLUTION
                [index-1]][depot];
                vh++;
            }
        }
    }
}
for(i=0; i<N-1; i++)
{
    j = aux = flag = 0;
    flag2 = VhSelect = -1;

    if(VectTabu[i] == 0)
    {
        for(j=0; j<VEHICLES; j++) VehTabu[j] = 0;
        client = i + 1;
        insertion = (rand()%2)+1;
        switch(insertion)
        {
            case 1: flag2=CLOSENESS_INSERTION(VehTabu,client,ind,VectTabu);
            break;
            case 2:flag2=DIRECT_SWAPINSERTION(VehTabu,client,ind,VectTabu);
            break;
            case 3: flag2 = SWAP_INSERTION(VehTabu, client, ind, VectTabu);
            break;
            case 2: flag2=SWAPS_INSERTION(VehTabu, client, ind, VectTabu);
            break;
        }
```

```
         if(flag2 == 0)
         {
            if(Vh+1 <= VEHICLES)
            {
               NEW_VEHIC(ind,client,VectTabu,VehTabu,PopulationIni);
               vehics = Vh;
            }
         }
      }
   }
   for(vh=0; vh<vehics; vh++)
      PopulationIni[ind].FITNESS += PopulationIni[ind].VEHCOST[vh];
}

int CLOSENESS_INSERTION(int *VehTabu, int client, int ind, int *VectTabu)
{
   int i=0, j=0, flag2=0, aux=0, flag=0, flag3=0, VhSelect=-1, vh=0,
       vehics = Vh, index=0, sizes=0;
   int vect_aux[size], vect_start[size], vect_size[size];

   do
   {
      j = aux = flag2 = flag3 = index = sizes = 0;
      VhSelect = -1;
      for(i=0; i<size; i++) vect_aux[i] = vect_start[i] = vect_size[i] = 0;
      flag = SEARCH_ROUTE(VehTabu, client, ind);
      if(flag == 1)
      {
         index = PopulationIni[ind].LB[VhSelect];
         sizes = (PopulationIni[ind].UB[VhSelect] - PopulationIni
                 [ind].LB[VhSelect]) + 1;
         for(i=0; i<=sizes; i++)
         {
            if(i<sizes)
            {
               vect_aux[i] = PopulationIni[ind].SOLUTION[index];
               vect_start[i] = instance[vect_aux[i]].READY_TIME;
               vect_size[i] = instance[vect_aux[i]].DUE_DATE -
                              instance[vect_aux[i]].READY_TIME;
               index++;
            }
            else
            {
               vect_aux[i] = client;
               vect_start[i] = instance[client].READY_TIME;
               vect_size[i] = instance[client].DUE_DATE - instance
                              [client].READY_TIME;
            }
         }
         SORT_WINDOWS(vect_aux, vect_start, vect_size, sizes);
         VhSelect=CANDIDATE(vect_aux, vect_start, vect_size, sizes);
         flag = TIME_CONSTRAINTS(vect_aux, vect_start, sizes, VhSelect);


         if(flag == 0)
```

```
         {
            VehTabu[VhSelect] = 1;
            for(vh=0; vh<Vh; vh++)
            {
               if(VehTabu[vh] == 0)
               {
                  flag2 = 1;
                  vh = vehics;
               }
            }
         }
         else
         {
            INSERT(PopulationIni, ind, vh);
            for(i=PopulationIni[ind].LB[VhSelect], j=0; i<=PopulationIni
                [ind].UB[VhSelect], j<=sizes; i++, j++)
               PopulationIni[ind].SOLUTION[i] = vect_aux[j];
            ROUTE_COST(VhSelect, ind, PopulationIni);
            flag3=1;
         }
      }

   }while((flag2 == 1) && (flag3 == 0));
   return flag3;
}


int DIRECT_SWAPINSERTION(int *VehTabu, int client, int ind, int *VectTabu)
{
   int i=0,j=0,sizes=0,sizes2=0,index=0,index2=0,flag=0,flag2=-1,
      flag3=-1, flag4=-1, VhOption=-1, VhSelect=-1, vh=-1, aux1=-1;
   int vect_aux[size],vect_start[size],vect_size[size],vect_aux2[size],
      vect_start2[size], vect_size2[size], vect_auxResp[size],
      vect_startResp[size], vect_sizeResp[size];
   int vect_auxResp2[size], vect_startResp2[size], vect_sizeResp2
       [size], VehDisp[VEHICLES];
   double aux=0;

   for(i=0; i<VEHICLES; i++)
   {
      if(i < Vh) VehDisp[i] = 0;
      else VehDisp[i] = -1;
   }
   flag=AVAILABILITY(VehDisp, PopulationIni, ind);
   if(flag == 1)
   {
      VhSelect = VehDisp[rand()%(j-1)];
      index = PopulationIni[ind].LB[VhSelect];
      sizes=(PopulationIni[ind].UB[VhSelect]- PopulationIni[ind].LB
            [VhSelect]) + 1;
      SORT_WINDOWS(vect_aux, vect_start, vect_size, sizes);
      flag2 = TIME_CONSTRAINTS(vect_aux, vect_start, sizes, VhSelect);
      if(flag2 == 1)
      {
         INSERT(PopulationIni, ind, vh);
```

```
   for(i=PopulationIni[ind].LB[VhSelect], j=0; i<=PopulationIni[ind].
      UB[VhSelect], j<=sizes; i++, j++)
      PopulationIni[ind].SOLUTION[i] = vect_aux[j];
   ROUTE_COST(VhSelect, ind, PopulationIni);
   VectTabu[client-1] = 1;
}
else
{
   VehTabu[VhSelect] = 1;
   for(i=0; i<=sizes; i++)
   {
      vect_auxResp[i] = vect_aux[i];
      vect_startResp[i] = vect_start[i];
      vect_sizeResp[i] = vect_size[i];
   }
   do
   {
      for(i=0; i<size; i++)
      {
         vect_aux2[i] = vect_size2[i] = vect_start2[i] = 0;
         vect_auxResp2[i] = vect_sizeResp2[i]=vect_startResp2[i]=0;
      }
      VhOption = rand()%Vh;
      while(VehTabu[VhOption] == 1) VhOption = rand()%Vh;
         index2 = PopulationIni[ind].LB[VhOption];
         sizes2=(PopulationIni[ind].UB[VhOption]-PopulationIni
                 [ind].LB[VhOption]) + 1;
         DIRECT_INSERTION(vect_aux2, vect_start2, vect_size2);
         SORT_WINDOWS(vect_aux2,vect_start2,vect_size2,sizes2);
         flag3 = TIME_CONSTRAINTS(vect_aux2, vect_start2, sizes2,
                 VhOption);
         if(flag3 == 1)
         {
            INSERT(PopulationIni, ind, vh);
            for(i=PopulationIni[ind].LB[VhOption], j=0; i<=
               PopulationIni[ind].UB[VhOption], j<=sizes2; i++, j++)
               PopulationIni[ind].SOLUTION[i] = vect_aux2[j];
            ROUTE_COST(VhOption, ind, PopulationIni);
            VectTabu[client-1] = 1;
            flag2 = flag3 = 1;
         }
         else
         {
            PERMUTE1to1(vect_aux2, vect_start2, vect_size2,
                        sizes2, PopulationIni);
            if(VectTabu[client-1] == 0)
            {
               for(i=0; i<Vh; i++)
               {
                  if(VehTabu[i] == 0)
                  {
                     flag4 = 1;
                     i = Vh;
                  }
                  else flag4 = 0;
```

```
                    }
                 }
              }
           }while((VectTabu[client-1] == 0) && (flag4 == 1));
        }
     }
     else flag2 = 0;
     return flag2;
}

int TIME_CONSTRAINTS(int *vect_aux, int *vect_start,int sizes, int
VhSelect)
{
   int i, deposit=0, flag=1;
   float Start[size], Finish[size];
   for(i=0; i < size; i++) Start[i] = Finish[i] = 0.0;
   for(i=0; i <= sizes; i++)
   {
      if(i == 0)
      {
         if(MAT_DISTANCES[deposit][vect_aux[i]] < instance[vect_aux
            [i]].READY_TIME)
            Start[i] = vect_start[i];
         else Start[i] = MAT_DISTANCES[deposit][vect_aux[i]];
         Finish[i] = Start[i] + instance[vect_aux[i]].SERVICE_TIME;
      }
      else if((MAT_DISTANCES[vect_aux[i-1]][vect_aux[i]] + Finish[i-1]) <=
              vect_start[i])
      {
          Start[i] = vect_start[i];
          Finish[i] = Start[i] + instance[vect_aux[i]].SERVICE_TIME;
      }
      else if((MAT_DISTANCES[vect_aux[i-1]][vect_aux[i]] + Finish[i-1] +
     instance[vect_aux[i]].SERVICE_TIME) <= instance[vect_aux[i]].DUE_DATE)
      {
         Start[i]=MAT_DISTANCES[vect_aux[i-1]][vect_aux[i]] + Finish[i-1];
         Finish[i] = Start[i] + instance[vect_aux[i]].SERVICE_TIME;
      }
      else
      {
         flag = 0;
         i = sizes;
      }
   }
   return flag;
}

void GENETIC(struct InitialPop *PopulationIni)
{
   int aux1=0, i, j, ind=0, IndFather=0, IndMother=0, P_Cross=0, best=-1;
   float aux=0, aux2=0, muta=0;
   struct InitialPop IndSon1, IndSon2, Aux;

   for(P_Cross=0; P_Cross<(POPULATION * CROSSOVER); P_Cross++)
   {
```

```
      best = SEARCH_BEST(PopulationIni);
      BestFitness = PopulationIni[best];
      IndMother = rand()%(POPULATION/2);
      IndFather = (POPULATION/2)+rand()%(POPULATION -(POPULATION/2));
      CHROM_CROSSOVER(IndMother,IndFather,PopulationIni,IndSon1,IndSon2);
      if((muta=rand()/(float)RAND_MAX)<=MUTATION)
      {
         Aux=IndSon1;
         COOPERATIVE_MUTATION(Aux);
      }
      else if((muta=rand()/(float)RAND_MAX)<=MUTATION)
      {
         Aux=IndSon2;
         COOPERATIVE_MUTATION(IndSon1);
      }
      UPDATE_POPULATION(IndSon1, IndSon2);
   }
}

void CHROM_CROSSOVER(int IndMother, int IndFather, struct InitialPop
*PopulationIni, struct InitialPop IndSon1, struct InitialPop IndSon2)
{
   int i=0, j=0, m=0, chrom1[VEHICLES/2], chrom2[VEHICLES/2], vh=0,
       flag=1, depot=0, CrossInd=0, cont=0, index=0;
   int CROSS=0;
   struct InitialPop Aux, Aux2;

   for(i=0; i<VEHICLES/2; i++) chrom1[i] = chrom2[i] = -1;
   if(PopulationIni[IndMother].NUMVEH < PopulationIni[IndFather].NUMVEH)
      CROSS = 1+(rand()%(PopulationIni[IndMother].NUMVEH / 2));
   else CROSS = 1+(rand()%(PopulationIni[IndFather].NUMVEH / 2));
   CHROM_MUTA(chrom1, chrom2, CROSS, IndMother, IndFather);
   for(cont=0; cont<CROSS; cont++)
   {
      for(j=PopulationIni[IndFather].LB[chrom2[cont]]; j<=
           PopulationIni[IndFather].UB[chrom2[cont]]; j++)
      {
         IndSon1.SOLUTION[index] = PopulationIni[IndFather].SOLUTION[j];
         if(j == PopulationIni[IndFather].LB[chrom2[cont]])
            IndSon1.LB[IndSon1.NUMVEH]=index;
         if(j == PopulationIni[IndFather].UB[chrom2[cont]])
         {
            IndSon1.UB[IndSon1.NUMVEH] = index;
            IndSon1.TOTAL_DEMAND[IndSon1.NUMVEH] = PopulationIni
                          [IndFather].TOTAL_DEMAND[chrom2[cont]];
            IndSon1.VEHCOST[IndSon1.NUMVEH] = PopulationIni
                                [IndFather].VEHCOST[chrom2[cont]];
            IndSon1.NUMVEH++;
         }
         index++;
      }
   }
   Aux2=PopulationIni[IndMother];
   SCANNING(IndSon1, Aux2);
   Aux = IndSon1;
```

```
   COST_INDIVIDUAL(IndSon1);
   for(cont=0; cont < CROSS; cont++)
   {
      for(j=PopulationIni[IndMother].LB[chrom1[cont]]; j<=PopulationIni
          [IndMother].UB[chrom1[cont]]; j++)
      {
         IndSon2.SOLUTION[index] = PopulationIni[IndMother].SOLUTION[j];
         if(j == PopulationIni[IndMother].LB[chrom1[cont]])
            IndSon2.LB[IndSon2.NUMVEH] = index;;
         if(j == PopulationIni[IndMother].UB[chrom1[cont]])
         {
            IndSon2.UB[IndSon2.NUMVEH] = index;
            IndSon2.TOTAL_DEMAND[IndSon2.NUMVEH] = PopulationIni
                                   [IndMother].TOTAL_DEMAND[chrom1[cont]];
            IndSon2.VEHCOST[IndSon2.NUMVEH] = PopulationIni
                                   [IndMother].VEHCOST[chrom1[cont]];
            IndSon2.NUMVEH++;
         }
         index++;
      }
   }
   Aux2=PopulationIni[IndFather];
   SCANNING(IndSon1, Aux2);
   Aux = IndSon2;
   COST_INDIVIDUAL(IndSon2);
}

void COOPERATIVE_MUTATION(struct InitialPop *Aux)
{
   int position=0, Ant=0, aux=0, i=0, j=0, index=0, index1=0,
       ChromMuta[CHROMUTA], vect_aux[N-1], vect_tabu[N-1], IndM=0,
       flag=0, LocalMuta=0, depot=0;
   float Tij[N][N],Pij[N-1],AUX[N-1], maximo[N-1], GAMMA=0, flag=-1;
   struct InitialPop Mutation[N-1],BestAntFitness;

   for(i=0; i<CHROMUTA; i++) ChromMuta[i] = -1;
   for(i=0; i<N; i++)
   {
      for(j=0; j<N; j++)
      {
        if(i == j) Tij[i][j] = 0;
        else Tij[i][j] = T0;
      }
   }

   for(LocalMuta=0; LocalMuta<(Aux.NUMVEH*MUTA_LOCAL); LocalMuta++)
   {
      SELECT_CHROMS(ChromMuta);
      for(i=0; i<N-1; i++) vect_aux[i] = -1;
      index=0;
      for(i=0; i<CHROMUTA; i++)
      {
         for(j=Aux.LB[ChromMuta[i]]; j<=Aux.UB[ChromMuta[i]]; j++)
         {
            vect_aux[index] = Aux.SOLUTION[j];
```

```
    index++;
    }
}
COLONY=index+1;
index=0;
for(IndM=0; i<COLONY; i++)
{
    Mutation[IndM].SOLUTION[0] = vect_aux[index];
    Mutation[IndM].TOTAL_DEMAND[Mutation[IndM].NUMVEH] =
                          instance[vect_aux[index]].DEMAND;
    Mutation[IndM].VEHCOST[Mutation[IndM].NUMVEH] = MAT_DISTANCES
  [depot][vect_aux[index]]+instance[vect_aux[index]].SERVICE_TIME;
}
for(Ant=0; Ant<COLONY; Ant++)
{
    index1 = 1;
    for(i=0; i<N-1; i++)
    {
        if(i == Mutation[Ant].SOLUTION[index1-1])
            vect_tabu[i] = 1;
        else vect_tabu[i] = 0;
    }
    do
    {
        flag=-1;
        if(index1 == Mutation[Ant].LB[Mutation[Ant].NUMVEH])
        {
            GAMMA = (float)(rand()%100)+100;
            if(GAMMA <= q0)
                aux =SEARCH_MAX(AUX);
            else
                aux =SEARCH_Pij(AUX);
            Mutation[Ant].SOLUTION[index1] = vect_aux[aux];
            Mutation[Ant].TOTAL_DEMAND[Mutation[Ant].NUMVEH] =
                                instance[vect_aux[aux]].DEMAND;
            if(MAT_DISTANCES[depot][vect_aux[aux]] <= instance
                [vect_aux[aux]].READY_TIME)
                Mutation[Ant].VEHCOST[Mutation[Ant].NUMVEH] =
                instance[vect_aux[aux]].READY_TIME + instance
                [vect_aux[aux]].SERVICE_TIME;
            else
                Mutation[Ant].VEHCOST[Mutation[Ant].NUMVEH] =
                MAT_DISTANCES[depot][vect_aux[aux]] + instance
                [vect_aux[aux]].SERVICE_TIME;
            vect_tabu[vect_aux[aux]-1] = 1;
            index1++;
        }
        else
        {
            GAMMA = (float) (rand()%100)/100;

            if(GAMMA <= q0)
                position = EXPLOITATION(vect_tabu, vect_aux,
                    Mutation, Tij, maximo, index, index1, Ant);
            else
```

```
                  position = EXPLORATION(vect_tabu, vect_aux,
                        Mutation, Tij, Pij, index, index1, Ant);
               flag = CONSTRAINTS(Mutation, Ant);
               if(flag == 0)
               {
                  COST_CALCULATION(Mutation, Ant);
                  if (index1 < (index-1))
                  {
                     Mutation[Ant].NUMVEH++;
                     Mutation[Ant].LB[Mutation[Ant].NUMVEH] = index1;
                  }
               }
            }
         }while(index1 < index);
         LOCAL_UPDATING(Tij, Mutation, Ant);
         for(i=0; i<=Mutation[Ant].NUMVEH; i++)
            Mutation[Ant].FITNESS += Mutation[Ant].VEHCOST[i];
      }
      aux=SearchBestAnt(Mutation);
      GLOBAL_UPDATING(Tij, Mutation, aux);
      if(Mutation[aux]<BestAntFitness.FITNESS)
         BestAntFitness = Mutation[aux];
   }
}

int EXPLOITATION(int *vect_tabu, int *vect_aux, struct InitialPop
*Mutation, float (*Tij)[N], float *maximo, int index, int index1,
int Ant)
{
   int i=0, aux=0;
   float temp=0;

   for(i=0; i<index; i++)
   {
      if(vect_tabu[vect_aux[i]] == 0)
         maximo[i] = pow(Tij[Mutation[Ant].SOLUTION[index1-1]][vect_aux
                     [i]], ALPHA) * pow((1.0/MAT_DISTANCES[Mutation[Ant].
                     SOLUTION[index1-1]][vect_aux[i]]), BETHA);
      else maximo[i] = 0;
   }
   for(i=0; i<index; i++)
   {
      if(maximo[i] > 0)
      {
         temp = maximo[i];
         aux = i;
         i = index;
      }
   }
   for(i=0; i<index; i++)
   {
      if(maximo[i] > temp)
      {
         temp = maximo[i];
         aux = i;
```

```
      }
   }
   return aux;
}

void LOCAL_UPDATING(float (*Tij)[N], struct InitialPop *Mutation, int Ant)
{
   int i=0, j=0;

   for(i=0; i<=Mutation[Ant].NUMVEH; i++)
   {
      for(j=Mutation[Ant].LB[i]; j<=Mutation[Ant].UB[i]; j++)
         Tij[Mutation[Ant].SOLUTION[j]][Mutation[Ant].SOLUTION[j+1]]=((1
          -DELTA)*Tij[Mutation[Ant].SOLUTION[j]][Mutation[Ant].SOLUTION
          [j+1]]) + (DELTA * T0);
   }
}

void GLOBAL_UPDATING(float (*Tij)[N], struct InitialPop *Mutation, int aux)
{
   int i=0, j=0, Ant=0, k=0, aux1=0;
   float aux=0, ATij=0, aux2=0;
   ATij = 1/Mutation[aux].FITNESS;
   for(i=0; i<=Mutation[aux].NUMVEH; i++)
   {
      for(j=Mutation[aux].LB[i]; j<=Mutation[aux].UB[i]; j++)
         Tij[Mutation[aux].SOLUTION[j]][Mutation[aux].SOLUTION[j+1]] = ((1-
         RO) * Tij[Mutation[aux].SOLUTION[j]][Mutation[aux].SOLUTION[j+1]])
         + (RO * ATij);
   }
}

int EXPLORATION(int *vect_tabu, int *vect_aux, struct InitialPop *Mutation,
float (*Tij)[N], float *Pij, int index, int index1, int Ant)
{
   int i=0, aux=0;
   float temp=0, sumatoria=0, AUX[N-1];

   for(i=0; i<index; i++)
   {
      if(vect_tabu[vect_aux[i]] == 0)
      {
         Pij[i] = pow(Tij[Mutation[Ant].SOLUTION[index1-1]][vect_aux[i]],
          ALPHA) * pow((1.0/MAT_DISTANCES[Mutation[Ant].SOLUTION[index1-
          1]][vect_aux[i]]), BETHA);
         sumatoria += Pij[i];
      } else Pij[i] = 0;
   }
   for(i=0; i<index; i++) AUX[i]=Pij[i];
   aux =SEARCH_MAX(AUX);
   return aux;
}

void STRUCTURE_INIT()
{
      int i, j;
```

```
	for(i=0; i<N-1; i++)
	{
		clusters.VECTOR[i]=0;
		init_sol.VECTOR[i]=0;
	}
	for(i=0; i<VEHICLES; i++)
	{
		clusters.LB[i]=0;
		clusters.UB[i]=0;
		clusters.TOTAL_DEMAND[i]=0;
		init_sol.LB[i]=0;
		init_sol.UB[i]=0;
		init_sol.TOTAL_DEMAND[i]=0;
	}
	for(i=0; i<N-1; i++)
	{
		windows[i].CLIENT=0;
		windows[i].START=0;
		windows[i].SIZE=0;
	}
	for(i=0; i <POPULATION; i++)
	{
		for(j=0; j<N-1; j++)
			PopulationIni[i].SOLUTION[j]=0;
		for(j=0; j<VEHICLES; j++)
		{
			PopulationIni[i].LB[j]=0;
			PopulationIni[i].UB[j]=0;
			PopulationIni[i].TOTAL_DEMAND[j]=0;
			PopulationIni[i].VEHCOST[j]=0;
		}
		PopulationIni[i].NUMVEH=0;
		PopulationIni[i].FITNESS=0;
	}
	for(i=0; i<N-1; i++)
	{
		TimesInd[i].Wik=0;
		TimesInd[i].Dik=0;
	}

	for(i=0; i<N-1; i++)
		BestFitness.SOLUTION[i] = 0;
	for(i=0; i<VEHICLES; i++)
	{
		BestFitness.LB[i] = BestFitness.UB[i] =
		BestFitness.TOTAL_DEMAND[i] = 0;
		BestFitness.VEHCOST[i] = 0;
	}
	BestFitness.NUMVEH = 0;
	BestFitness.FITNESS = COMPARE;

	Vh=0;
}
```

Código Fuente
Cálculo de la Distancia Hamming

```
/****************************************
* M.I. Alina Martínez Oropeza          *
****************************************/

#include "stdafx.h"
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

#define N 100
#define VEHICLES 25
#define POPULATION 30

int READ_SOLUTIONS(int *m);
void PRINT_SOLS();
int HAMMING_CALC(int (*DistHamming)[POPULATION]);
float AVERAGE_HAMMING(int (*DistHamming)[POPULATION]);

struct EVALUATION {
                            int SOLUTION[N];
                            int LB[VEHICLES];
                            int UB[VEHICLES];
                            int VEH;
                   } INDIVIDUALS[POPULATION];

int main()
{
      int i=0, j=0, m, ind=0,DistHamming[POPULATION][POPULATION];
      float Hamming=0;

      for(ind=0; ind<POPULATION; ind++)
      {
            for(i=0; i<N; i++)INDIVIDUALS[ind].SOLUTION[i] = 0;
            for(i=0; i<VEHICLES; i++)
               INDIVIDUALS[ind].LB[i] = INDIVIDUALS[ind].UB[i] = 0;
            INDIVIDUALS[ind].VEH = 0;
      }

      for(i=0; i<POPULATION; i++)
            for(j=0; j<POPULATION;j++)
```

```
                DistHamming[i][j] = 0;
      READ_SOLUTIONS(&m);
      HAMMING_CALC(DistHamming);
      Hamming = AVERAGE_HAMMING(DistHamming);
      return 0;
}

int READ_SOLUTIONS(int *m)
{
      int i, j, k, MAT_AUX[POPULATION][151];
      int dato, ind=0;
      FILE *ff;
      if((ff=fopen("SOLUTIONS5.txt","r+"))==NULL)
   {
       printf("\n NO SE PUDO ABRIR EL ARCHIVO DE LECTURA ");
         system("pause");
       exit(0);
   }
   fscanf(ff,"%d",m);

   for(i=0;i<POPULATION;i++)
   {
         for(j=0; j<151; j++)
         {
                fscanf(ff,"%d",&dato);
                MAT_AUX[i][j]=dato;
         }
   }
   fclose(ff);

   for(ind=0; ind<POPULATION; ind++)
   {
           for(i=0; i<N; i++)
                INDIVIDUALS[ind].SOLUTION[i] = MAT_AUX[ind][i];
           k=N;
           for(i=0; i<VEHICLES; i++)
           {
                INDIVIDUALS[ind].LB[i] = MAT_AUX[ind][k];
                INDIVIDUALS[ind].UB[i] = MAT_AUX[ind][k+1];
                k+=2;
           }
           INDIVIDUALS[ind].VEH = MAT_AUX[ind][150];
   }
   return 0;
}

int HAMMING_CALC(int (*DistHamming)[POPULATION])
{
      int ind,indAux,vh,vh2,i,j,hamming=0,Bandera=0,cont=0,dif=0;
```

```
      for(ind=0; ind<POPULATION-1; ind++)
      {
         for(indAux=ind+1; indAux<POPULATION; indAux++)
         {
            hamming=0; cont=0;
            for(vh=0; vh<INDIVIDUALS[ind].VEH; vh++)
            {
               for(i=INDIVIDUALS[ind].LB[vh]; i<=INDIVIDUALS[ind].
                    UB[vh]; i++)
               {
                  Bandera=0;
                  for(vh2=0; vh2<INDIVIDUALS[indAux].VEH; vh2++)
                  {
                     if(i==INDIVIDUALS[ind].LB[vh]) cont=0;
                        j=INDIVIDUALS[indAux].LB[vh2]+cont;
                     if((j<=INDIVIDUALS[indAux].UB[vh2])&&(j<=
                         INDIVIDUALS[ind].UB[vh]))
                     {
                        if(INDIVIDUALS[ind].SOLUTION[i] ==
                           INDIVIDUALS[indAux].SOLUTION[j])
                        {
                           Bandera = 1;
                           vh2 = INDIVIDUALS[indAux].VEH;
                        }
                     }
                  }
                  if(Bandera==0)
                     hamming++;
                     cont++;
               }
            }
            DistHamming[ind][indAux] = hamming;
         }
      }
      return 0;
}

float AVERAGE_HAMMING(int (*DistHamming)[POPULATION])
{
      int i, j;
      float average[POPULATION], Hamming=0;;

      for(i=0; i<POPULATION; i++)
            average[i] = 0;

      for(i=0; i<POPULATION; i++)
      {
            for(j=0; j<POPULATION; j++)
            {
                  average[i] += DistHamming[i][j];
```

```
        }
    }

    for(i=0; i<POPULATION-1; i++)
    {

        average[i] = average[i]/(POPULATION-(i+1));
        printf("%.2f\n", average[i]);
        Hamming += average[i]/POPULATION-1;
    }
    return Hamming;
}
```

Apéndice

Caracteristicas
NVidia Tesla C2070

```
################# Clúster #################

{gpu01}: /opt/NVIDIA_CUDA-5.5_Samples/1_Utilities/deviceQuery
[pmoreno@:deviceQuery]$(7)->./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 2 CUDA Capable device(s)

Device 0: "Tesla C2070"
  CUDA Driver Version / Runtime Version          5.5 / 5.5
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 5375 MBytes (5636554752 bytes)
  (14) Multiprocessors, ( 32) CUDA Cores/MP:     448 CUDA Cores
  GPU Clock rate:                                1147 MHz (1.15 GHz)
  Memory Clock rate:                             1494 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 786432 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65535),
3D=(2048, 2048, 2048)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (65535, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
```

```
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           2 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

Device 1: "Tesla C2070"
  CUDA Driver Version / Runtime Version          5.5 / 5.5
  CUDA Capability Major/Minor version number:    2.0
  Total amount of global memory:                 5375 MBytes (5636554752 bytes)
  (14) Multiprocessors, ( 32) CUDA Cores/MP:     448 CUDA Cores
  GPU Clock rate:                                1147 MHz (1.15 GHz)
  Memory Clock rate:                             1494 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 786432 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65535),
3D=(2048, 2048, 2048)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (65535, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           131 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
> Peer access from Tesla C2070 (GPU0) -> Tesla C2070 (GPU1) : No
> Peer access from Tesla C2070 (GPU1) -> Tesla C2070 (GPU0) : No

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 5.5, CUDA Runtime
Version = 5.5, NumDevs = 2, Device0 = Tesla C2070, Device1 = Tesla C2070
Result = PASS
```