UNIVERSIDAD AUTÓNOMA DEL
ESTADO DE MORELOS

FACULTAD DE CIENCIAS QUÍMICAS E INGENIERÍA

CENTRO DE INVESTIGACIÓN EN INGENIERÍA Y CIENCIAS APLICADAS

ALGORITMO GENÉTICO HÍBRIDO COOPERATIVO EN AMBIENTE GRID PARA TALLERES CON FLUJO FLEXIBLE

TESIS DOCTORAL
PARA OBTENER EL GRADO DE:

DOCTOR EN INGENÍERIA Y CIENCIAS APLICADAS
CON OPCIÓN TERMINAL EN TÉCNOLOGIA ELÉCTRICA

PRESENTA
M.C. FREDY JUÁREZ PÉREZ

ASESOR: DR. MARCO ANTONIO CRUZ CHÁVEZ

Cuernavaca Morelos, Febrero de 2013

Resumen

Los entornos de producción en serie son cada día más competitivos, debido a la demanda de los clientes que exige mantener un catálogo de productos amplio, por esta razón, se requiere de una programación eficaz y eficiente para la producción, que pueda satisfacer en tiempo y forma la demanda de sus clientes. Hay una gran brecha entre la literatura y los problemas reales de la industria, en donde la mayoría de la investigación está centrada en la solución de problemas pequeños, para los cuales casi siempre es posible encontrar la solución óptima, sin embargo, la simplificación en el planteamiento de estos problemas, deja fuera muchas limitaciones del mundo real, en contraparte, el tratamiento de problemas grandes presenta un area de oportunidad poco explorada. La industria necesita de sistemas de calendarización de la producción para satisfacer entornos reales y puedan obtener buenas soluciones en tiempos cortos.

El problema tratado en esta tesis doctoral se conoce como taller de flujo flexible (FFS, por sus siglas en inglés *Flexible Flow Shop*), consiste en un conjunto de trabajos a procesarse en un conjunto de etapas en serie, en donde cada etapa se compone de un conjunto de máquinas paralelas idénticas, a su vez cualquier máquina puede procesar cualquier trabajo uno a la vez y sin interrupciones, se considera una serie de restricciones incluyendo tiempos de inicio dependientes de la secuencia (SDST). Para tratar este problema, se propone un algoritmo genético híbrido cooperativo en ambiente Grid llamado AGHCGrid, que consta de tres metaheurísticas: algoritmos genéticos (GA), sistema de colonias de hormigas (SCH) y el recocido simulado (RS), junto con el uso del cómputo de alto rendimiento para la paralelización de algoritmos y su ejecución en ambiente Grid. Además, se proponen dos métodos: el primero para la sintonización distribuida automática aplicada en paralelo (SDAAP) que permite reducir el tiempo del análisis de sensibilidad del algoritmo, el segundo que es una extensión del primero (SDAAP-MI) que permite reducir el tiempo de la experimentación, ambos utilizan todos los núcleos de la Grid, pero solo el segundo puede ejecutarse en múltiples instancias.

El método híbrido cooperativo en ambiente Grid, que conjunta las tres metaheurísticas (AG,SCH,RS), para el problema de taller de flujo flexible con las restricciones que se manejan en esta tesis, no existe actualmente en la literatura, por lo que en esta tesis se estudia un problema de calendarización muy realista ayudando a cerrar la brecha entre la literatura y la práctica. También se propone un conjunto amplio de instancias de prueba para las cuales damos las cotas superiores obtenidas por el algoritmo propuesto. Así mismo, se realiza el análisis de la eficacia y el análisis de la eficiencia calculando la aceleración (Speedup), finalmente se comenta el desarrollo de interfaces gráficas para el usuario final en el ámbito de la transferencia tecnológica hacia la industria.

Abstract

The serial production environments are becoming more competitive, due to the demands of customers who require maintaining a broad product portfolio, therefore, requires a scheduling effective and efficient of production, which can satisfy the time customer demand. There is a big gap between literature and the real problems of the industry, where most research focuses on solving small problems, so it is almost always possible to find the optimal solution. However, simplified of problem statement leaves out many real-world constraints, in contrast, the treating a large problems represent major area of opportunity. The industry requires systems of production scheduling to meet real environments and can obtain good solutions in a short time.

The problem addressed in this PhD thesis is known as flexible flow shop (FFS), consists of a set of jobs to be processed in a series of stages in series, where each stage consists of a set of identical parallel machines, where any machine can process any work one at a time and uninterrupted, is considered a number of restrictions including sequence dependent setup times (SDST). To address this problem, we propose a hybrid algorithm cooperative in Grid environment called AGHCGrid, which consists of three metaheuristics: genetic algorithms (GA), ant colony system (SCH) and simulated annealing (RS), along with the use of high performance computing for parallelization algorithms and execution on Grid environment. In addition, we propose two methods: one for automatic distributed tuning applied in parallel (SDAAP) that reduces the time sensitivity analysis of the algorithm, the second is an extension of the first (SDAAP-MI) that can reduce the time experimentation, both use all cores of the Grid, but only the latter can run in multiple instances.

The cooperative hybrid method in Grid environment, that combines the three metaheuristics (AG, SCH, RS) for to the problem of flexible flow shop with restrictions that are handled in this PhD thesis, there is currently in the literature, so that This thesis studies a realistic scheduling problem helping to close the gap between literature and practice. It also proposes a comprehensive set of test instances for which we give the upper bounds obtained by our algorithm. Also, it makes the analysis of the effectiveness and efficiency analysis by calculating the Speedup, finally discusses the development of graphical interfaces for end users in the field of technology transfer to industry.

Agradecimientos

A mi asesor de tesis el Dr. Marco Antonio Cruz Chávez, por el apoyo que siempre me mostró y sigue mostrando, por su amistad, animo y paciencia.

A los integrantes de mi comité tutoral: Dra. Margarita Tecpoyotl Torres, Dr. Martín G. Martínez Rangel, Dr. Martín Heriberto Cruz Rosales, Dr. David Juárez Romero y Dr. Abelardo Rodríguez León, Dr. Álvaro Zamudio Lara, por sus acertadas observaciones.

Al Centro de Investigaciones en Ingeniería y Ciencias Aplicadas (CIICAp) que fue mi segunda casa.

A CONACYT por el apoyo económico.

Dedicatorias

A Dios por estar siempre conmigo.

A mis Padres Silvano y Gaudencia por el apoyo durante todo este tiempo que le dedique a mis estudios.

A mis hermanos Rodolfo, Griselda, Aurelia^{+}, Evelia y Silvano por su esfuerzo de tratar de ser mejores cada día.

A mis sobrinos que son luz de esperanza y alegría para toda la familia.

A mis compañeros de posgrado por compartir su conocimiento.

No estudio por saber más,
sino por ignorar menos.
Sor Juana Inés de la Cruz, 1695.

No basta saber, se debe también aplicar.
No es suficiente querer, se debe tambien hacer.
Johann Wolfgang von Goethe,
Los viajes de Wilhelm Meister, 1821.

Nomenclatura

AG	Siglas de Algoritmo Genético, traducido del inglés *genetic algorithm*.
B&B	Siglas en inglés el algoritmo de ramificación y poda (*Branch and Bound*).
BT	Siglas para la búsqueda Tabú.
Cluster	Agrupación de computadoras unidas a través una red local de alta velocidad que trabajan como una sola.
FFS	Siglas en inglés para el problema de taller de flujo flexible (*Flexible Flow Shop*).
FS	Siglas en inglés para el problema de taller de flujo (*Flow Shop*).
AGHCGrid	Siglas para el algoritmo genético híbrido cooperativo en ambiente Grid.
gLite	Es el middleware de próxima generación para la computación en Grid, diseñado a partir de la colaboración de más de 80 personas en 12 diferentes centros de investigación académica e industrial en europa.
Grid	Conjunto de Clusters geográficamente dispersos y unidos a través de una red global de alta velocidad que trabajan como una sola.
GUI	Siglas en inglés para describir una interfaz gráfica de usuario (*Graphical User Interface*).
Makespan o C_{max}	Tiempo total de término del último trabajo que sale del sistema (C_{max}).
MPI	Siglas en inglés para la interfaz de paso de mensajes (*Message Passing Interface*)
NP	Siglas en inglés para tiempo no polinomial (*No Polinomial Time*).
OCH	Siglas de Optimización por Colonia de Hormigas, traducido del inglés *ant colony optimization*.
RS	Siglas de Recocido Simulado, traducido del inglés *simulated annealing*.
RTT	Siglas en inglés para describir el tiempo que tarda un paquete enviado desde un emisor en volver a este mismo emisor habiendo pasado por el receptor de destino (*Round-Trip delay Time*).
SB	Siglas en inglés para el desplazamiento por cuello de botella (*Shifting Bottleneck*).
SCH	Siglas de Sistema de Colonia de Hormigas, traducido del inglés *ant colony system*.
SDAAP	Siglas de Sintonización Distribuida Automática Aplicada en Paralelo.
SDAAP-MI	Siglas de Sintonización Distribuida Automática Aplicada en Paralelo en Múltiples Instancias.
SDST	Siglas en inglés para los tiempos de inicio dependientes de la secuencia (Sequence Dependent Setup Times).

UB	Siglas en inglés para cota superior (*Upper Bound*).
VLAN	Siglas en inglés para describir una red virtual de área local (*Virtual Local Area Network*).
VPN	Siglas en inglés para describir una red privada virtual (*Virtual Private Network*).
FFS	Siglas en inglés de Flexible Flow Shop.
SDST	Siglas en inglés de Sequence Dependent Setup Times.
Speedup	Ganancia en aceleración que tiene un algoritmo secuencial a ser paralelizado sobre un conjunto de núcleos de procesamiento.
MPICH	Versión ampliamente usada de la librería de paso de mensajes MPI.
OpenMPI	Versión libre de la implementación de MPI.
LB	Siglas en inglés de Lower Bound, se refiere a una cota superior.
I2	Internet 2.

Símbolos usados para el FFS con tiempos de inicio dependientes de la secuencia (FFS-SDST).

m	Número de etapas para un FFS.
m_k	Número de máquinas en paralelo en la etapa k.
k	k-ésima etapa de m.
i	i-ésima máquina de la etapa k.
n	Número total de trabajos a ser secuenciados.
j	Trabajo $j \in n$.
p_{ijk}	Tiempo de procesamiento del trabajo j en la máquina i de la etapa k.
S_{iljk}	Tiempo de inicio dependiente de la secuencia entre los trabajos l y j en la máquina i de la etapa k.
O_{ijk}	Operación que involucra los tiempos de secuencia S_{iljk} y de procesamiento p_{ijk}.

Símbolos usados para sistema de colonia de hormigas (SCH).

α	Coeficiente de importancia alfa para la feromona.
β	Coeficiente de importancia beta aplicada para el costo.
γ	Coeficiente de evaporación gamma aplicada a la evaporación de feromona local.
δ	Coeficiente de evaporación delta aplicada a la evaporación de feromona global.
h	Número de hormigas en el sistema.
p	Criterio de paro de la búsqueda local.
q	Coeficiente entre 0 y 1 que determina la exploración/explotación.
$S_{hormiga}$	Solución inicial para SCH.

Símbolos usados para recocido simulado (RS).

t_o	Temperatura inicial.
m	Longitud de la cadena de Markov.
μ	Factor para el descenso de temperatura.
t_f	Temperatura final.
S_{metal}	Solución inicial para RS.

Símbolos usados para el AG.

T	Tamaño de la población
C	Tasa de cruce.
G	Número de generaciones
$S_{individuo}$	Solución inicial para el algoritmo genetico (AG).

Índice general

Índice de figuras

Índice de tablas

Capítulo 0

Algoritmos

Capítulo 1

INTRODUCCIÓN

En los entornos de producción una calendarización (scheduling) se refiere al tiempo de llegada de los trabajos que requieren ser procesados, y la secuenciación se refiere al orden en que los trabajos serán procesados [Churchman et al., 1969], estos términos se pueden utilizar indistintamente [Dudek et al., 1974], la calendarización de la producción esta limitado principalmente por la disponibilidad de recursos, el cual depende del entorno de producción.

La calendarización se ha vuelto muy importante en las últimas décadas debido principalmente a la diversidad de productos, tamaños de los pedidos y los tiempos de entrega que cada vez son más cortos, así mismo la competencia exige la personalización de los productos y el aumento de la calidad de la producción para poder competir en mercados internacionales, provocando que la calendarización de la producción sea cada vez más compleja [Thijs Urlings, 2010].

La optimización de la calendarización de la producción cobra importancia cuando permite aumentar la capacidad de producción o la satisfacción del cliente, sin que ello lleve a invertir en comprar nueva maquinaria, invirtiendo solo en el mantenimiento de la maquinaria actual, es decir, que se optimiza la calendarización de la producción con las máquinas existentes.

Este trabajo de investigación doctoral se enfoca en el problema referenciado como taller de flujo flexible, conocido en inglés como flexible flow shop o hybrid flow shop con tiempos de inicio dependientes de la secuencia (FFS-SDST) el cual se refiere como un problema de secuenciar un conjunto de operaciones asociados a un conjunto de trabajos que deben ser procesados sobre un conjunto de máquinas, en un sistema de estados de producción en serie como el mostrado en [Wang and Li, 2002].

El entorno del FFS-SDST se presenta por lo general en la industria de la manufactura y se asocia a la problemática de mejorar la eficiencia de los recursos disponibles en

un sistema para la producción en serie, este sistema esta conformado por una conjunto de etapas, en donde cada etapa esta conformado por un número idéntico de máquinas en paralelo, los trabajos a ser secuenciados [Santos et al., 1996] comienzan a ser procesados primero en la etapa uno, luego en la etapa dos y así sucesivamente hasta terminar en la última etapa, los trabajos al ser procesados pasan por las distintas etapas configuradas en serie, en cada etapa un trabajo puede ser procesado por cualquiera de las máquinas disponibles uno a la vez, requiriendo de solo una operación por cada etapa.

La secuencia de los trabajos inicialmente es determinada para la etapa uno, pero puede cambiar al pasar a las etapas restantes, cada que un trabajo va a ser procesado en una máquina, este máquina requiere de un tiempo de preparación antes de recibir el siguiente trabajo, este tiempo de preparación es dependiente de la secuencia (SDST), una vez que se determina la calendarización, es decir, la secuencia que tendrán los trabajos en cada una de las diferentes etapas, se procede a calcular el tiempo total de término conocido como makespan (C_{max}), es decir lo que busca el FFS-SDST es minimizar el tiempo total de término asociado a los costos de producción.

Un FFS-SDST tiene restricciones de capacidad, es decir que solo puede procesar un trabajo a la vez y cada trabajo tiene asociado un tiempo de procesamiento, además de una restricción para los tiempos de inicio dependientes de la secuencia entre dos trabajos, de modo que para resolver el problema hay que determinar la calendarización que tenga el menor costo con base en la función objetivo que es el makespan.

Para adentrarnos al estudio del FFS-SDST a continuación se explica la motivación que nos llevó al estudio de este problema, seguido de la explicación del problema similar pero de menor complejidad conocido como Flow Shop (FS) en la sección 1.2, después abordamos el problema de Flexible Flow Shop clásico que contempla pocas restricciones y un entorno ideal en la sección 1.3 y la notación usada para representar el problema en la sección 1.4, posteriormente se muestran los objetivos en la sección 1.5, los alcances en la sección 1.6, las contribuciones en la sección 1.7 y la organización de este trabajo de investigación doctoral en la sección 1.8.

1.1. Motivación.

A lo largo del tiempo se ha acumulado una cantidad considerable de literatura incluyendo una amplia gama de problemas con diversas características [Thijs Urlings, 2010], en donde los primeros estudios sobre calendarización fueron realizados por [Salveson, 1952], sin embargo, diversos autores [Ledbetter and Cox, 1977; Ford et al., 1987; McKay et al., 1988; Olhager and Ray., 1995] han observado una enorme brecha que hay entre la teoría y la práctica de la calendarización, muchos investigadores como [Dudek et al., 1992; MacCarthy and Liu, 1993] y [McKay et al., 2002], han criticado la investigación de la calendarización y en particular la literatura existente para la calendarización del Flow Shop con respecto a esta brecha.

Diferentes autores han hecho una revisión de la literatura [Graves, 1981; Allahverdi et al., 1999; Linn and Zhang, 1999; Vignier et al., 1999; Ruiz and Vázquez-Rodríguez, 2010; Ribas et al., 2010] y han encontrado conclusiones similares al utilizar diferentes entornos de programación. Se ha realizado una revisión estadística [Reisman et al., 1997] del problema Flow Shop entre los años 1952,1994 y concluye que de un total de 170 artículos revisados, sólo 5 (3 %) de los artículos hacen referencia a aplicaciones reales, y treinta y cuatro artículos (20 %) se refieren a aplicaciones que no estaban basadas en entornos del mundo real.

El FFS es un problema de optimización combinatorio clasificado como *NP-Duro* [Garey and Johnson, 1979; Mártinez, 2010], aunque hay una tendencia reciente para tratar de plantear formulaciones más realistas de los problemas de calendarización, todavía no hay muchos esfuerzos de investigación para considerar las limitaciones que encontramos en entornos de manufactura en el mundo real. La desventaja constante que prevalece, en las soluciones de tales problemas, es que las formulaciones son extremadamente complejas y las soluciones óptimas son muy difíciles de obtener, la unica manera de tratarlos es utilizar enfoques heurísticos [Shaukat and Luan, 1999] y metaheurísticas [Choong et al., 2001; Jin et al., 2006] para obtener buenas soluciones en tiempos razonables, por esta razón en este investigación doctoral se utilizan metaheurísticas.

En la actualidad no existe un algoritmo exacto (determinista) que pueda encontrar la solución óptima para un problema de FFS-SDST, excepto para instancias muy pequeñas, debido a que el tiempo requerido para obtener la solución óptima puede llegar a ser del orden de años, miles de años, incluso millones de años, esto se debe a que el espacio de soluciones presenta un crecimiento exponencial a medida que aumenta el tamaño de la instancia, por tal razón, solo se limita su uso a encontrar soluciones a problemas muy pequeños y a encontrar cotas superiores a instancias grandes con el fin de hacer comparaciones con otros algoritmos, por el contrario, las metaheurísticas son algoritmos de aproximación (no deterministas), las cuales se diseñan e implementan para que estén acotados en tiempo polinomial, es decir que tengan un tiempo de respuesta razonable (eficiencia) y sean capaces de entregar soluciones de gran calidad (eficacia).

A pesar de que las metaheurísticas entregan soluciones sub-óptimas y en algunos casos óptimas, su velocidad de respuesta esta limitado por las características del equipo de cómputo usado, la capacidad de memoria y velocidad de procesador son los dos componentes que limitan su desempeño, es por ello que se busca siempre utilizar las computadoras más potentes para obtener resultados en tiempos más cortos, el problema es que una computadora esta limitada a un determinado escalamiento en cuanto a número de procesadores y capacidad de memoria que pueda tener, por lo que el tamaño de la instancia de un problema de FFS-SDST que puede ser tratado esta limitado a estos dos recursos.

Para saltar esta limitante y poder tratar instancias de gran escala, es necesario usar equipo de cómputo de alto rendimiento conocido como cluster o Grid, generalmente llamado súper-cómputo, este nuevo mundo que es totalmente fascinante por el poder de cómputo que puede albergar, en la actualidad llega a utilizar hasta 1,572,864 núcleos de

procesamiento [1], representa el poderío de los países desarrollados para que sus científicos e ingenieros puedan encontrar soluciones a problemas de gran escala, en donde la clave de la utilización de estos recursos es lo que se conoce como paralelización o paralelismo.

En este sentido el presente trabajo de investigación doctoral esta motivado principalmente por la dureza del problema y el realismo de los tiempos de inicio dependientes de la secuencia para el FFS-SDST que se presenta en el mundo real, así mismo el poder tratar instancias más grandes mediante el diseño de un algoritmo paralelizado que haga uso de una plataforma Grid de alto rendimiento, presentando tiempos de respuesta razonables y soluciones de buena calidad, se presentan nuevos retos en cuanto al análisis, diseño, programación y experimentación, así mismo problemas de distribución, sincronización y cooperación de procesos que no se encuentran en algoritmos secuenciales.

1.2. Flow Shop.

Flow Shop (FS) es uno de los problemas más estudiados en el área de calendarización que consiste en asignar recursos a un conjunto de trabajos sobre un periodo de tiempo [Pinedo, 2008].

Este problema consiste en calendarizar un conjunto N de trabajos $N = \{1, 2, ..., n\}$ a procesarse en un conjunto M de máquinas $M = \{1, 2, ..., m\}$ en serie en un cierto orden que se obtiene a partir de una permutación. Cada trabajo j con $j \in N$ debe procesarse en cada máquina k con $k \in M$ de modo que los trabajos sigan un mismo camino $(1, 2, ..., m)$. Cada trabajo j requiere de un total de m operaciones con $m \in M$ para ser finalizada, es decir, una operación por cada máquina y cada máquina puede procesar solo un trabajo a la vez como se observa en la figura 1.1.

Si los trabajos conservan la misma secuencia entre las máquinas se le conoce como FS permutado, pero si se permite cambios en la secuencia al pasar de una máquina a otra entonces se le conoce como FS no permutado. El espacio de soluciones es calculado como $(n!)^m$ posibles soluciones de secuenciado de los trabajos, pero en el caso del FS permutado se reduce a $n!$ debido a que el orden de los trabajos entre las máquinas no cambia [Reza and Saghafian, 2008].

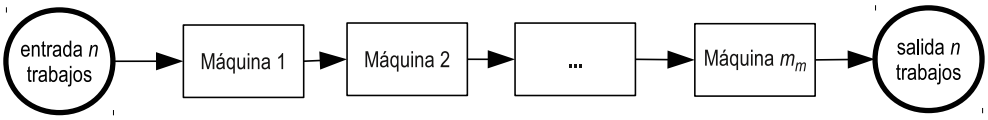

Figura 1.1. *Entorno Flow Shop*

Sin embargo a pesar de ser un problema NP-Completo si involucra más de dos máquinas [Garey and Johnson, 1979] y que es difícil de tratar por ser un problema de optimización combinatoria complejo, en entornos de producción real, es poco común

[1] ver sitio http://www.top500.org/

encontrarse sistemas que utilicen una sola máquina por cada operación [Naderi et al., 2010], por lo que se adoptan entornos con varias etapas y en cada etapa existe un número determinado de máquinas en paralelo, con el objetivo de buscar incrementar las capacidades de producción en términos de velocidad y calidad. Este entorno se refiere a un sistema flexible de producción que en términos formales se conoce como Flexible Flow Shop (FFS).

1.3. Flexible Flow Shop clásico.

El problema de FFS es un entorno de manufactura muy común, el cual consiste en un conjunto de n trabajos que deben ser procesados en un cierto orden sobre un conjunto de m etapas en serie, con base en una función objetivo. Existe un número determinado de variantes pero todos ellos comparten las siguientes características en común:

1. El número de etapas de procesamiento es de al menos 2.

2. En cada etapa k con $k \in M$, se tiene $M_k \geqslant 1$ máquinas en paralelo con al menos una etapa con $M_k > 1$.

3. Todos los trabajos son procesados siguiendo una misma ruta de producción que se conoce como flujo de trabajo (Flow Shop), es decir: etapa 1, etapa 2,...,etapa m.

4. Cada trabajo requiere un tiempo de procesamiento p_{jk} el cual refiere al tiempo de procesamiento del trabajo j con $j \in N$ en la etapa k con $k \in M$, también denominado operación O_{jk}, si el tiempo de procesamiento depende de la máquina donde se procesa el trabajo entonces se agrega el índice i (p_{ijk}) que indica que el trabajo j en la etapa k se procesa en la máquina i refiriéndose a la operación O_{ijk} al cual también se le agrega el índice i.

La flexibilidad del sistema se refiere a que un trabajo j puede omitir pasar por cualquier número de etapas k siempre y cuando se procese en al menos una etapa [Ruiz and Vázquez-Rodríguez, 2010], también se refiere al hecho de que un trabajo en determinada etapa k puede procesarse en cualquiera de las $M_k \geqslant 1$ máquinas en paralelo disponibles en cada etapa k [Lee and Vairaktarakis, 1979].

Para el problema de FFS clásico al que llamaremos "estándar", no se consideran tiempos de espera entre los trabajos, es decir están disponibles en el tiempo cero, todas las máquinas en cada etapa son idénticas, cualquier máquina solo puede realizar una operación a la vez y cualquier trabajo puede ser procesado por una solo máquina a la vez, los tiempos de inicio dependientes de la secuencia y dependientes de la máquina [Jungwattanakit et al., 2007] son despreciables, las interrupciones no están permitidas, p. ej. si se quisiera interrumpir un trabajo por otro de mayor prioridad no esta permitido, la capacidad de la cola (buffer) entre dos etapas es ilimitado y finalmente los datos

del problema se conocen por anticipado y son deterministas, es decir que no son datos dinámicos o adquiridos en tiempo de ejecución como se observa en la figura 1.2.

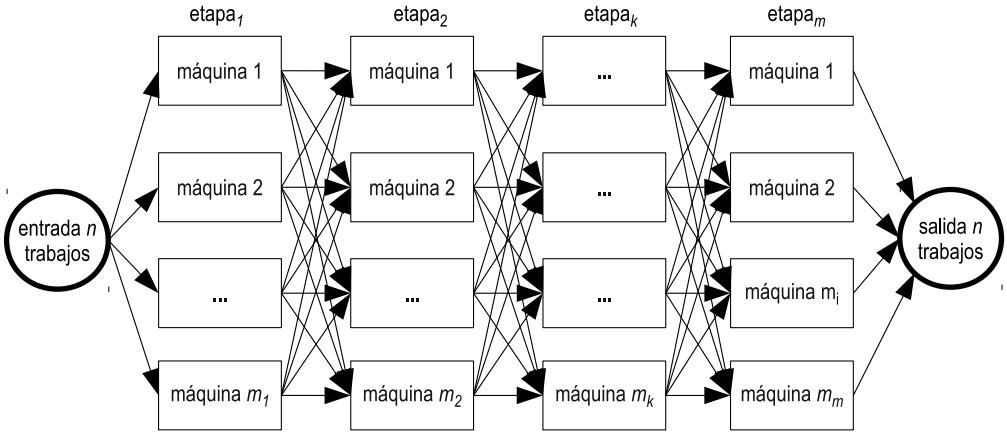

Figura 1.2. *Entono del Flexible Flow Shop. La figura representa un Flexible Flow Shop estándar, en donde las máquinas por cada etapa son idénticas, cada trabajo solo requiere de una operación por etapa y cualquier máquina puede hacerlo, así mismo los tiempos de inicio dependientes de la secuencia son despreciables y no se permiten interrupciones.*

La mayoría de las variantes del FFS no cumplen con las características de un problema clásico, cambian en dos o tres aspectos del problema, sin embargo puede ser tomado como punto de inicio mediante el cual es posible agregar o quitar características del problema para describir las diferentes variantes del FFS [Yaurima, et al; Kahraman et al., 2010].

Como se menciono anteriormente, el FFS es, en la mayoría de los casos *NP-Duro*, incluso en los casos en donde una etapa tiene solo dos máquinas y la otra solo una [Gupta, 1998; Lowa et al., 2008; Huang and Li, 1998], de la misma manera si se permiten interrupciones para detener un trabajo por otro de mayor prioridad y reanudarlo en un tiempo posterior, es extremadamente *NP-Duro*, incluso con dos etapas m [Hoogeveen et al., 1996]. Relajando el problema en relación al número de máquinas por etapa, restringiéndolo a una por etapa conocido como Flow Shop (FS) y en el caso de tener solo una etapa pero con múltiples máquinas en paralelo conocido como el problema de máquinas paralelas, ambas son *NP-Duro* [Garey and Johnson, 1979; Mártinez, 2010].

1.4. Flexible Flow Shop clasificación y notación.

La nomenclatura usada esta basada en [Pinedo, 2008], [Vignier et al., 1999] y [Ruiz and Vázquez-Rodríguez, 2010], en la cual los problemas son clasificados de acuerdo a la configuración de las máquinas (α), características del procesamiento de los trabajos y restricciones (β) y la función objetivo (γ). En esta tesis se considera un número finito de máquinas, trabajos y etapas indicado con i, j y k respectivamente, de los elementos asociados al trabajo j solo se toman en cuenta los tiempos de procesamiento p_{ijk} y se

omiten las fechas de inicio (r_j), fechas de entrega (d_j) y peso (ω_j) que a continuación se describen:

- p_{ijk} que refieren al trabajo j en la máquina i de la etapa k, el uso de máquinas paralelas idénticas omite el índice i debido a que las velocidades de las máquinas son idénticas y no dependen del trabajo j por tanto los tiempos de procesamiento se refieren como p_{jk}.

- Release date(r_j). Se refiere fecha tiempo de inicio, indica que el trabajo no puede comenzar antes de su tiempo de fecha programada de inicio.

- Due date(d_j). Se refiere a la fecha de entrega, indica la fecha límite para la entrega del producto o fecha límite de fabricación.

- Weight (ω_j). El peso se usa como factor para modelar su importancia como costo, volumen o prioridad.

Cada problema entonces puede ser descrito con una triada $\alpha|\beta|\gamma$. El parámetro α define la estructura de las máquinas como lo es el número de etapas, y el número y características de máquinas por etapa. α esta compuesto por 4 parámetros $\alpha_1, \alpha_2, \alpha_3, \alpha_4$. α_1 indica la configuración general de las máquinas, en este caso un Flexible Flow Shop. α_2 el número de etapas en el sistema. α_3 y α_4 igualmente describen las propiedades de las máquinas por etapa, p. ej. la notación $(\alpha_3\alpha_4)^k$ indican que son α_3 máquinas en paralelo del tipo α_4 en la etapa k. $\alpha_3 \in \{\emptyset$,PM,QM,RM$\}$, donde PM indica máquinas paralelas idénticas, QM máquinas paralelas uniformes y RM máquinas paralelas no relacionadas [Pinedo, 2008]. En el caso de que $\alpha_3 = \emptyset$ por definición indica que es una sola máquina.

El segundo elemento β indica las restricciones y consideraciones distintas de un problema estándar, entre las más comunes están:

- Release date(r_j). Indica que el trabajo no puede comenzar antes de su tiempo de inicio.

- Precedence constraints (prec). Indica que hay restricciones de procedencia entre los diferentes trabajos, esto quiere decir que un trabajo o varios trabajos no pueden comenzar a ser procesados antes de que sus antecesores terminen.

- Sequence dependent setup times (S_{ljk}). Representa el tiempo de inicio dependiente de la secuencia entre los trabajos l y j en la etapa k, s_{0jk} denota el tiempo de inicio para el trabajo j en la etapa k si el trabajo j es el primero asignado en esa máquina y S_{l0k} representa un tiempo de limpieza de la máquina después del trabajo l en la etapa k, si el trabajo l es el último asignado en esa máquina. Si el tiempo de inicio depende de la máquina en donde se procesa el trabajo entonces se agrega el índice i (S_{iljk}) indicando que el tiempo de inicio entre el trabajo l y j se da para la máquina i en la etapa k, lo que sugiere el uso de máquinas paralelas no

relacionadas y uniformes, en contraparte con el uso de máquinas paralelas idénticas donde $p_{ijk} = p_{jk}$. Si no aparece esta restricción, entonces se indica que los tiempos de inicio dependientes de la secuencia son despreciables o igual a cero, o que están incluidos en los tiempo de procesamiento (p_{jk}).

- Preemptions (prmp). Indica que un trabajo puede ser interrumpido por otro en cualquier momento, esto permite al calendarizador sustituir un trabajo por otro en la misma máquina, el tiempo de procesamiento que resta del trabajo interrumpido es recuperado al regresar a su máquina o puede ser procesado en otra máquina disponible si se trata de máquinas paralelas.

- Machine elegibility restrintions (M_j). Indica que el procesamiento del trabajo j esta restringido al conjunto de máquinas M_j en la etapa k.

- Blocking (block). Indica que el buffer entre dos etapas es limitado y una vez lleno, los trabajos deben esperar en la etapa anterior hasta que se libere espacio.

- Permutations (prmu). Indica que el orden de los trabajos se basa en la regla *First In Fisrt Out (FIFO)*, esto indica que el orden de ejecución de los trabajos es mantenido a lo largo de todo el sistema.

- Recirculations (recrc). Indica que los trabajo tienen permitido procesarse más de una vez en la misma etapa.

- Breakdowns (brkdwn). Indica que las máquinas no están disponibles todo el tiempo, esto puede ser debido a programas de mantenimiento.

- No-wait (nwt). Indica que los trabajos no pueden esperar entre dos máquinas sucesivas, esto quiere decir que un trabajo no puede comenzar a procesarse en la primera máquina hasta asegurar que el trabajo puede pasar a través de todo el sistema sin retardos, p. ej. en rollos de acero laminado que no pueden esperar debido a que se enfriarían.

- $p_j = p$. Indica que todos los tiempos de procesamiento son iguales a p.

- $size_{jk}$. Indica que una operación O_{jk} debe ser procesada por un número de máquinas igual a $size$ en la etapa k para ser completada.

El tercer elemento γ se refiere a la función objetivo, las cuales se muestran en la tabla 1.1, el máximo tiempo total de término $C_{max} = max_j C_j$, conocido como makespan, se refiere al tiempo de término de último trabajo que sale del sistema. El tiempo de flujo del trabajo j (F_{max}) es $F_j = C_j - r_j$ se refiere al tiempo que el trabajo pasa en el sistema sin tomar en cuenta el tiempo de llegada del trabajo.

El retraso del trabajo j (L_{max}) es $L_j = C_j - d_j$, donde d_j es el tiempo límite prometido de fabricación e implica que si es positivo ($C_j > d_j$) la operación terminó

tarde y si es negativo ($C_j < d_j$) la operación término temprano. La demora del trabajo j (T_{max}) es $T_j = max(C_j - d_j, 0) = max(L_j, 0)$ lo cual indica que la demora nunca es negativa, el tiempo de anticipación (E_{max}) es $E_j = max(d_j - C_j, 0)$ indica el tiempo de holgura con respecto a la fecha de entrega d_j de manera inversa al retraso, la unidad de penalización para la demora se toma $U_j \in \{0, 1\}$ donde toma el valor de 1 si $C_j - d_j > 0$ y 0 en otro caso.

También es común asociar un peso (ω_j) con el trabajo j con el fin de modelar su importancia, tal peso puede representar por ejemplo el costo del trabajo, su volumen o la prioridad relativa, entre otras posibilidades. Los elementos de C_j, F_j, T_j, etc. y sus homólogos ponderados $\omega_j C_j, \omega_j F_j, \omega_j T_j$, etc. se utilizan con frecuencia para describir las funciones objetivo descritas en la tabla 1.1.

NOTACIÓN	DESCRIPCIÓN	SIGNIFICADO
C_{max}	$max_j C_j$	makespan - Máximo tiempo total de término
F_{max}	$max_j(C_j - r_j)$	Máximo tiempo de flujo del trabajo
L_{max}	$max_j(C_j - d_j)$	Máximo tiempo de retraso
T_{max}	$max_j\{max(L_j, 0)\}$	Máximo tiempo de demora
E_{max}	$max_j\{max(d_j - C_j, 0)\}$	Máximo tiempo de anticipación
$\overline{C}$	$\sum C_j$	Tiempo total de término
$\overline{C^w}$	$\sum \omega_j C_j$	Tiempo total de término ponderado
$\overline{F}$	$\sum F_J$	Tiempo total de flujo
$\overline{F^w}$	$\sum \omega_j F_j$	Tiempo total de flujo ponderado
$\overline{T}$	$\sum T_J$	Tiempo total de retraso
$\overline{T^w}$	$\sum \omega_j T_j$	Tiempo total de retraso ponderado
$\overline{U}$	$\sum U_J$	Número de trabajos retrasados
$\overline{U^w}$	$\sum \omega_j U_j$	Número de trabajos retrasados ponderados
$\overline{E}$	$\sum E_J$	Tiempo total de anticipación
$\overline{E^w}$	$\sum \omega_j E_j$	Tiempo total de anticipación ponderado

Tabla 1.1. *Funciones objetivo más comunes. La tabla muestra las funciones objetivos estudiadas en la literatura, entre las cuales se encuentra el makespan que es el máximo tiempo total de término estudiado en esta tesis.*

Un ejemplo de esta notación para el FFS clásico descrito anteriormente es $FFS_m = ((PM^{(i)})_{i=1}^{m})||C_{max}$ donde FFS_m indica que es un Flexible Flow Shop con un número de etapas m, $(PM^{(k)})_{k=1}^{m}$ indica que esta conformada por un número determinado de máquinas en paralelo idénticas en la etapa k con un número de etapas desde k=*1* hasta m y C_{max} indica obtener el makespan.

Por último supongamos que tenemos un FFS limitado a cuatro etapas donde la etapa 1 tiene solo una máquina, la etapa 2,3 y 4 tienen un número cualquiera de máquinas paralelas no relacionadas, además tiene restricciones de tiempo de inicio de los trabajos (r_j), tiempos de inicio dependientes de la secuencia (S_{jk}) en la etapa 3 y una función objetivo que calcule el tiempo total de término ponderado $FFS_m =$

$(RM^{(1)}, ((RM^{(k)})_{i=2}^{m})|r_j, S_{jk}^{(3)}| \sum \omega_j C_j$

1.5. Objetivo de la investigación.

Los objetivos de esta investigación de tesis son los siguientes:

- Diseñar y programar un algoritmo paralelo e híbrido utilizando algoritmos genéticos, sistema de colonia de hormigas y recocido simulado, el cual deberá estar diseñado para ser ejecutado sobre un ambiente Grid, además de ser cooperativo para coordinar los esfuerzos realizados por los diferentes procesos distribuidos de manera uniforme sobre la plataforma de producción Grid Morelos[2].

- El diseño del algoritmo debe ser fácilmente escalable para utilizar un número elevado de procesos sin que con ello pierda la calidad de las soluciones obtenidas, o el tiempo de respuesta acotado en tiempo polinomial, se busca mantener la eficacia con respecto al secuencial, mejorando la eficiencia de manera proporcional al número de procesos y núcleos usados, en donde el balance ideal es asignar un proceso por cada núcleo sin sobrecarga.

- Llevar a cabo el análisis de sensibilidad para sintonizar los parámetros, posteriormente realizar la experimentación determinando si el algoritmo propuesto, mejora a medida que se utiliza un número mayor de procesos distribuidos sobre una plataforma Grid, para el problema de FFS-SDST con tiempos de inicio dependientes de la secuencia y finalmente con base en este análisis determinar si es recomendable o no, utilizar un algoritmo paralelo sobre ambiente Grid para tratar este tipo de problemas.

- Llevar a cabo el análisis de la aceleración (Speedup) del algoritmo propuesto para determinar el grado de aprovechamiento del algoritmo paralelo respecto a su contraparte secuencial.

Todos los objetivos de esta tesis tienen como punto central la paralelización y el uso de la Grid, pues se busca con ello determinar los efectos de la cooperación de procesos en este ambiente, en donde la cooperación se divide en un proceso maestro que coordina los esfuerzos de un conjunto de procesos esclavos. El proceso maestro utiliza algoritmos genéticos construyendo la población, aplicando los operadores de selección y cruce, delegando la parte de la mutación a los procesos esclavos que aplican primero una mutación cooperativa con sistema de colonia de hormigas y después una mutación iterativa con recocido simulado. La cooperación de procesos utiliza técnicas de distribución, comunicación y sincronización de procesos que se ejecutan sobre un ambiente Grid. Para llevar a buen término este tema de investigación se sigue la siguiente metodología:

[2] http://gridmorelos.uaem.mx

- Modelo. Se plantea el problema del FFS con tiempos de inicio dependiente de la secuencia como un modelo de programación lineal entera binaria y como un modelo de grafo disyuntivo.

- Método. Consiste en un método propuesto para el diseño de un algoritmo que permite encontrar soluciones óptimas acotado en tiempo polinomial, este método es un algoritmo de optimización combinatoria paralelo diseñado para correr bajo un ambiente Grid.

- Plataforma. Consiste en la utilización de la plataforma de producción denominada Grid Morelos, la cual es una plataforma Grid multi-cluster de alto rendimiento.

- Aplicación. Consiste en realizar el análisis, diseño y programación de la aplicación en C con interfaz de paso de mensajes MPI que implementa el método propuesto.

- Experimentación. Consiste en llevar a cabo la experimentación sobre esta plataforma, para determinar la eficiencia y eficacia del algoritmo propuesto.

La utilización de MPI permite la cooperación de procesos al utilizar paso de mensajes para coordinar las acciones realizadas por los procesos, se decidió por una paralelización con mínima comunicación debido al limitado ancho de banda que une los clusters, ya que una alta comunicación impacta directamente el desempeño del algoritmo elevando los tiempos computados de procesamiento, así mismo se decidió por la utilización de núcleos homogéneos con el fin de evitar tiempos de terminación asimétricos dados por velocidades diferentes de los núcleos de procesamiento.

Finalmente con base en la experimentación llevada cabo, utilizando diferentes versiones de MPI, se determina que el entorno ideal es utilizar el compilador C de Intel (icc) en combinación con las librerías de MPI de Intel (mpiicc), ya que es el que proporciona una mejor optimización en la generación de código, esta determinación se hizo después de utilizar el compilador de C de Linux (gcc) y las librerías de MPI de OpenMPI y MPICH.

1.6. Alcances de la investigación.

La variante para el problema de FFS que se trata en este trabajo de investigación, junto con las condiciones de ejecución para el algoritmo propuesto se definen a continuación:

- Se considera un número finito de máquinas, trabajos y etapas, así mismo solo se toman en cuenta los tiempos de procesamiento para cada trabajo y se omiten tiempos de inicio programados, tiempos de entrega y peso ponderado, así mismo no se permiten interrupciones conocidos como *preemtions* (su nombre en inglés).

- Se utilizan tiempos de inicio dependientes de la secuencia (SDST), esto quiere decir que dependiendo del orden de secuenciado entre los diferentes trabajos, existen diferentes tiempos de preparación entre cada par de trabajos.

- La función objetivo es minimizar el makespan (C_{max}).

- Proponer métodos para reducir los tiempos del análisis de sensibilidad y experimentación mediante el uso de los recursos en paralelo de la Grid.

- El algoritmo en paralelo deberá de utilizar un máximo número de núcleos de procesamiento que tenga la plataforma de producción Grid Morelos llegado el momento de análisis de sensibilidad y experimentación, que actualmente alcanza los 120 núcleos de procesamiento.

- Medir la eficiencia y eficacia al utilizar diferente número de procesos asignados a núcleos procesamiento, para determinar cual es el número de procesos óptimo del algoritmo propuesto con el cual se obtienen los mejores resultados.

1.7. Contribuciones de la tesis.

Las aportaciones de la presente investigación doctoral son:

- Diseño y programación del algoritmo híbrido AGHCGrid en ambiente Grid, compuesto por tres metaheurísticas (AG, SCH y RS), en donde la forma de integrar estas tres metaheurísticas en paralelo, además de la variante del problema FFS que trata y finalmente, su entorno de ejecución en ambiente Grid, no aparece en la literatura existente.

- Dos métodos propuestos para llevar a cabo el proceso de sintonización: Sintonización distribuida automática aplicada en paralelo (SDAAP) y SDAAP en múltiples instancias (SDAAP-MI), la primera se utiliza para el análisis de sensibilidad (sintonización) y la segunda para la experimentación del algoritmo propuesto ya sintonizado, ambos métodos solo funcionan al utilizar cómputo de alto rendimiento como cluster y ambientes Grid.

- Un conjunto de instancias de prueba con las cuales se llevan a cabo los trabajos de análisis de sensibilidad y experimentación, para las cuales se proponen las primeras cotas superiores encontradas con el algoritmo propuesto ejecutado sobre la Grid Morelos.

1.8. Organización de la tesis.

La presente investigación doctoral se organiza de la siguiente manera:

- Capítulo 1. Se presenta la introducción al problema FFS-SDST en donde se define el entorno del desarrollo del problema, los objetivos, alcance y las contribuciones de la tesis.

- Capítulo 2. Muestra la descripción del problema FFS, su clasificación y la notación usada para representar el problema, así mismo se muestran el uso de los tiempos de inicio dependientes de la secuencia (SDST), el modelo matemático, el grafo disyuntivo y finalmente se presenta un problema pequeño de ejemplo.

- Capítulo 3. Muestra una revisión de la literatura enfocándonos en los métodos aplicadas a la solución del FFS con diversas configuraciones, de entre los cuales se seleccionaron tres metaheurísticas que se integran para diseñar y programar el método propuesto en esta tesis.

- Capítulo 4. Se presenta la metodología de solución propuesta que consiste en 5 pasos: modelo, método, plataforma, aplicación y experimentación, se explican a detalle cada uno de estos pasos entre los cuales se encuentra el diseño y explicación del algoritmo paralelo AGHCGrid propuesto, sus representaciones simbólicas, generación de nuevas soluciones y su ejecución en un ambiente Grid.

- Capítulo 5. Se presenta el método propuesto para la sintonización distribuida automática, mostrando primero su funcionamiento secuencial y después en paralelo.

- Capítulo 6. Se presentan los resultados experimentales sobre la plataforma de producción Grid Morelos, primero se muestran los resultados del análisis de sensibilidad utilizando el método de sintonización distribuida automática aplicada en paralelo (SDAAP), después se presentan los resultados de la experimentación del algoritmo sintonizado utilizando un segundo método propuesto que consiste en una extensión del método de sintonización distribuida automática aplicada en paralelo ahora en múltiples instancias (SDAAP-MI), sobre diferente número de núcleos reduciendo el tiempo de experimentación, con el objetivo de medir la eficiencia y eficacia del algoritmo propuesto, entre el cual se incluye el cálculo de la acelaración (Speedup).

- Capítulo 7. Se presentan las conclusiones y trabajos futuros que se desprenden de esta tesis.

Capítulo 2

DESCRIPCIÓN DEL PROBLEMA

Es este Capítulo se explica en que consiste el problema de Flexible Flow Shop con tiempos de inicio dependientes de la secuencia (FFS-SDST) en la sección 2.1, después se muestra el modelo matemático en la sección 2.4, el modelo de grafo disyuntivo propuesto en la sección 2.3 y finalmente se muestra un ejemplo para un problema pequeño de 3 trabajos por dos etapas, con dos máquinas en paralelo por cada etapa y tiempo de inicio dependientes de la secuencia en la sección 2.5.

2.1. Flexible Flow Shop con tiempos de inicio dependientes de la secuencia.

Formalmente el problema de Flexible Flow Shop con tiempos de inicio dependientes de la secuencia y con una función objetivo para calcular el C_{max} (FFS-SDST) se describe como $FFS_m = (PM^{(i)})_{i=1}^{m}|S_{jk}|C_{max}$ siguiendo la notación de [Vignier et al., 1999].

Consiste en un conjunto N de trabajos $N = \{1, 2, ..., n\}$ a procesarse en un conjunto M de etapas en serie $M = \{1, 2, ..., m\}$, en donde cada etapa k con $k \in M$ tiene un conjunto $M_k = \{1, 2, ..., m_k\}$ de máquinas paralelas idénticas i con $i \in M_k$ para procesar los j trabajos con $j \in N$, entonces cada trabajo j en cada etapa k puede ser procesado por cualquiera de las máquinas i, esto es que cada trabajo j requiere de m operaciones O_{ijk} para terminar, una por cada etapa k, seleccionando en cada etapa k una máquina i del conjunto de máquinas paralelas M_k, en donde cada máquina i solo puede procesar un trabajo a la vez y una vez que comienzan a procesarse no pueden interrumpirse, los trabajos se procesan primero en la etapa 1, luego en la etapa 2 y así sucesivamente. Los tiempos de procesamiento p_{jk} se refieren al tiempo de procesamiento del trabajo j en la

etapa k.

Los SDST (S_{ljk}) representa el tiempo de inicio entre los trabajos l con $l \in N$ y j procesado después del trabajo l en la etapa k en cualquier máquina i, es decir, después de que la operación O_{ilk} salga de la máquina i y antes de que la operación O_{ijk} entre a la máquina i requiere de un tiempo de inicio S_{ljk}, S_{0jk} denota el tiempo de inicio de la máquina i antes del trabajo j en la etapa k, si el trabajo j es el primero asignado a la máquina i y S_{l0k} representa el tiempo de limpieza de la máquina i después del trabajo l en la etapa k, si el trabajo l es el último asignado en la máquina i, si los SDST dependen de la máquina donde se lleva a cabo la operación, entonces se agrega el índice i (S_{iljk}) indicando que los SDST se da entre los trabajos l y j en la máquina i de la etapa k.

Dos tipos de tiempos de inicio se toman en consideración en esta tesis: dependiente de la máquina y dependiente de la secuencia [Jungwattanakit et al., 2007]. Se consideran tiempos de inicio dependientes de la máquina (S_{0jk}), ocurriendo solo una vez cuando en trabajo j es el primer trabajo asignado a la máquina i en la etapa k, posteriormente solo se consideran los tiempos de inicio dependientes de la secuencia (S_{ljk}) entre los trabajos l y j sucesivos al procesarse en la máquina i de la etapa k. Finalmente los tiempos de limpieza (S_{l0k}) de la máquina i al terminar de procesar el último trabajo j en la etapa k, no se toman en cuenta para el cálculo del makespan debido a que el tiempo es independiente a la salida del trabajo j en la máquina i de la etapa k.

La función objetivo consiste en calcular el máximo tiempo total de término o makespan. El problema permite posibles cambios en la secuencia de los trabajos entre cada etapa aumentando considerablemente su espacio de soluciones que es calculado como $n!\left(\prod_{i=1}^{m} m_i\right)^m$ [Gourgand et al., 1999] en comparación con el clásico permutado $n!\,(m_k)^m$ y finalmente el Flow Shop $(n!)^m$ no permutado, se observa un crecimiento en el espacio de soluciones a medida que se va escalando desde el Flow Shop hacia el Flexible Flow Shop clásico y no permutado.

Para esta tesis se consideran los siguientes supuestos:

- Todos los trabajos están disponibles para ser procesados en el tiempo 0 (cero), considerando antes los tiempos de inicio dependiente de la secuencia (S_{ljk}) entre el trabajo l y j en la etapa k, más el tiempo de procesamiento del trabajo j en la etapa k (p_{jk}), ambos integran el costo de cada operación (O_{ijk}), observe que en este caso se hace uso del índice i porque las operaciones hacen referencia a una máquina especifica, sin embargo los tiempos de procesamiento no son dependientes de las máquinas debido a que son máquinas paralelas idénticas en velocidad y número.

- Todos los trabajos pasan a través del sistema en el mismo orden $(etapa_1, etapa_2, ..., etapa_m)$.

- Cada etapa k tiene el mismo número de máquinas paralelas idénticas m_k.

- No se consideran interrupciones de los trabajos (*prmp*).
- No se consideran bloqueos de las máquinas (*block*).
- No se consideran fechas de inicio programadas (r_j), fechas límite de entrega (d_j), ni peso (ω_j).
- Cada trabajo j en cada etapa k solo requiere de una operación y debe seleccionar solo una máquina de las m_k existentes.
- El orden de los trabajos puede cambiar de una etapa a otra, es decir se considera como no permutado.

La mayoría de los investigadores se centran en el estudio del problema clásico que supone un ambiente idóneo y no contempla situaciones reales, por tanto son pocos los investigadores que se centran en el estudio de problemas complejos y más apegados a la realidad. Una de las situaciones más reales son los tiempos de inicio dependientes de la secuencia (SDST), tratar con SDST supone un aumento en la dificultad de la calendarización [Allahverdi et al., 2008] y por ser una situación más real, la investigación con el uso de SDST crece día tras día.

2.2. Función objetivo.

La función objetivo que se estudia en esta tesis doctoral, es minimizar el tiempo total de término del último trabajo que sale del sistema, es decir, en una calendarización C_{jk}, denota el tiempo de término del trabajo j en la etapa k, por tanto C_j denota el tiempo de término del trabajo j en la última etapa m del sistema.

En este sentido $C_{max} = max(C_1, C_2, ..., C_j)$, indica que la función objetivo es minimizar el máximo tiempo total de término, que es encontrar una calendarización que tenga el tiempo total de término más corto de entre todos los trabajos desde C_1 hasta Cj que tenga el tiempo de término más largo, esto es $max(C_1, C_2, ..., C_j)$ el cual se le conoce como makespan o C_{max}.

En la sección 2.5 se da un ejemplo de como calcular el makespan para un problema pequeño.

2.3. Modelo de grafo disyuntivo propuesto.

Para el problema $FFS_m = (PM^{(k)})_{k=1}^{m} | S_{ljk} | C_{max}$ se propone un modelo de grafo disyuntivo a partir de [Kuo-Ching et al., 2007] en donde se muestra un grafo disyuntivo para el problema de Flow Shop, nuestra propuesta incluye la adición máquinas paralelas

idénticas M_k por cada etapa k como se observa en la figura 2.1. Cualquier solución de cualquier instancia puede ser asociada a este grafo $G = (O, C, D)$ donde O representa el conjunto nodos (vértices), C el conjunto de arcos (aristas) conjuntivos directos y D el conjunto de arcos (aristas) disyuntivos indirectos.

Los nodos O_{ijk} denotan las operaciones del trabajo j con $j \in N$ en cada etapa k con $k \in M$ en solo una máquina i con $i \in m_k$ de las disponibles como se muestra en la sección 2.1. C denota las relaciones de precedencia entre las operaciones de los diferentes trabajos al transitar por las etapas en serie. D representa la disyuntiva para seleccionar en cada etapa k el orden de procesamiento de los trabajos en la máquina i, en donde una máquina i es seleccionada mediante $i = \{1|2|...|m_k\}$ el cual usa el operador *or* para seleccionar solo una de las m_k disponibles, esto debido a que cada trabajo j en cada etapa k solo requiere de una operación y cualquier máquina i del conjunto m_k puede hacerlo.

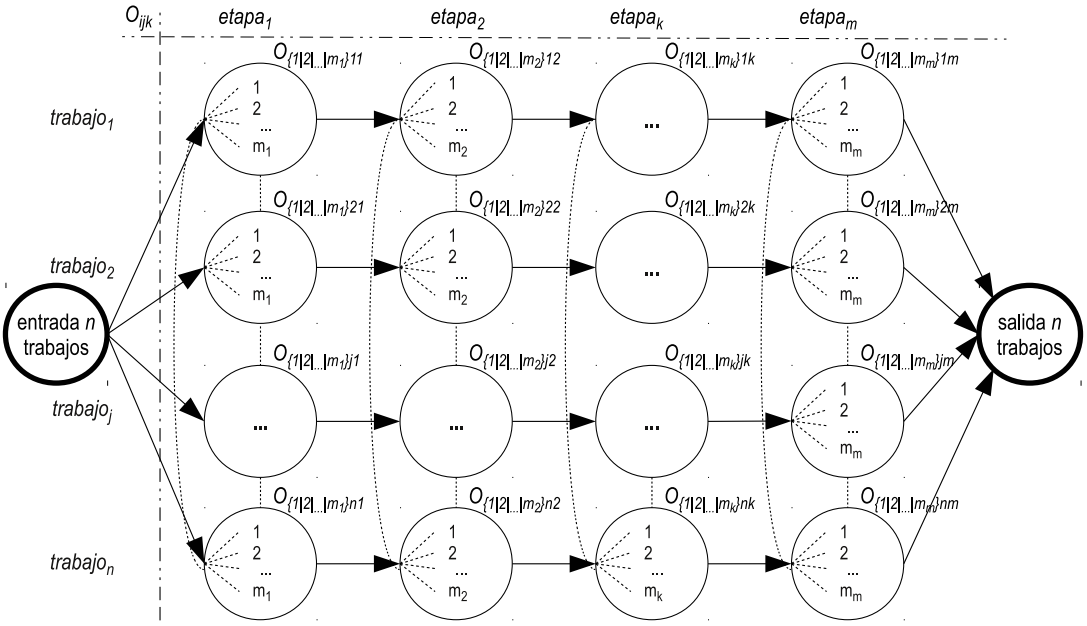

Figura 2.1. *Grafo disyuntivo propuesto para el FFS-SDST. La figura muestra el grafo disyuntivo propuesto para el problema* $FFS_m = (PM^{(k)})_{k=1}^{m}|S_{ljk}|C_{max}$ *donde* FFS_m *indica que es un Flexible Flow Shop con un número de etapas m,* $(PM^{(k)})_{k=1}^{m}$ *indica que esta conformada por un número determinado de máquinas en paralelo idénticas en la etapa k con un número de etapas desde k=1 hasta m,* S_{ljk}*, indica los tiempos de inicio dependiente de la secuencia entre los trabajos l y j en la etapa k y* C_{max} *indica que la función objetivo es minimizar el máximo tiempo total de término.*

El total de operaciones se calcula como $O_{total} = n \times m + 2$, que incluye dos operaciones ficticias con tiempos de procesamiento cero ($p_j = 0$) en la entrada y salida del grafo.

La construcción de una calendarización valida comienza en el nodo ficticio de entrada, seleccionando el orden de los trabajos a procesar en la $etapa_{k=1}$, que puede

consistir en la generación de una permutación de los trabajos $N = \{1, 2, ..., n\}$, paso seguido por cada trabajo j con $j \in N$ se selecciona una máquina $i = \{1|2|...|m_k\}$ del conjunto de máquinas paralelas idénticas m_k, la razón de seleccionar solo una máquina responde al hecho de que cada trabajo solo requiere de una operación y cualquier máquina del conjunto m_k puede hacerlo, esta operación (O_{ijk}) asocia los tres elementos máquina (i), trabajo(j) y etapa(k) respectivamente.

Este orden de trabajos j en máquinas i en cada etapa k resuelve la disyuntiva de los arcos D del grafo en cada etapa y los arcos conjuntivos C no permiten pasar a la siguiente etapa hasta completar todas las asignaciones restantes. Una vez completadas todas las asignaciones en la $etapa_{k=1}$, continúa con las demás etapas restantes k con $k \in M$ hasta llegar a la última etapa k en donde el proceso termina al posicionarse en el nodo ficticio de salida.

Conceptualmente los nodos ficticios (entrada y salida) nos ayudan a un mejor entendimiento visual del problema y técnicamente en programación nos ayudan a tener un mejor control del programa, esas son las dos razones principales para su utilización las cuales podrían en un momento determinado omitirse.

2.4. Modelo matemático.

La primera aproximación para poder resolver problemas de optimización combinatoria, es definir un modelo matemático para obtener soluciones óptimas, el inconveniente de formular este tipo de modelos es que solo nos permiten resolver instancias muy pequeñas y fáciles, ya que problemas grandes y complejos no pueden resolverse de esta manera debido al tiempo y a la memoria que consumen.

Un problema clásico de FFS con colas (buffer) ilimitadas puede ser convertido en uno sin colas pero con bloqueo de máquinas, esta razonamiento parte del hecho de que un trabajo l, una vez procesado en la etapa k, debe pasar a la etapa $k+1$, en donde si la etapa $k+1$ cuenta con una cola ilimitada, entonces el trabajo se encola y debe esperar hasta que la máquina se desocupe, pero si no cuenta con una cola, entonces la máquina en la etapa k se bloquea al finalizar de procesar el trabajo l hasta que la máquina en la etapa $k+1$ se desocupe y en este sentido ambos son equivalentes [Guinet and Solomon, 1996].

A continuación presentamos un modelo de programación lineal entera mixta (MIP) para minimizar el makespan (C_{max}) para un problema de Flexible Flow Shop con tiempos de inicio dependientes de la secuencia (FFS-SDST) propuesto por [Tavakkoli-Moghaddam and Safei, 2007].

La función objetivo es:

$$min \quad C_{max} \tag{2.1}$$

Las restricciones son:

$$\sum_{j=1}^{n_i} \sum_{l=0, l\neq k}^{K} x_{ijlk} = 1 \quad \forall i, k \tag{2.2}$$

$$\sum_{l=0, l\neq k}^{K} x_{ijlk} = \sum_{q=1, q\neq k}^{K+1} x_{ijkq} \quad \forall i, j, k \tag{2.3}$$

$$c_{ik} \geq p_{ik} + \sum_{j=1}^{n_i} x_{1j0k} s_{i0k} \quad \forall k \tag{2.4}$$

$$c_{ik} - c_{(i-1)k} \geq p_{1k} + \sum_{j=1}^{n_i} \sum_{l=0}^{K} x_{iljk} s_{ilk} \quad \forall i \geq 1, k \tag{2.5}$$

$$c_{ik} \geq p_{ik} + \sum_{j=1}^{n_i} \sum_{l=0, l\neq k}^{K} x_{ijlk} \left(s_{ilk} + d_{il}\right) \quad \forall i,\ k = 1, ..., K+1 \tag{2.6}$$

$$c_{ik} = d_{(i-1)k} + p_{ik} + \sum_{j=1}^{n_i} \sum_{l=0, l\neq k}^{K} x_{iljk} s_{ilk} \quad \forall i \geq 1, k \tag{2.7}$$

$$c_{mk} = d_{mk} \quad \forall k \tag{2.8}$$

$$C_{max} \geq c_{mk} \quad \forall k \tag{2.9}$$

$$x_{ijlk} \in \{0, 1\} \quad \forall i, j, l, k \quad ; \quad c_{ik}, d_{ik} \geq 0 \quad \forall i, k \tag{2.10}$$

La función objetivo de la expresión 2.1, es reducir el máximo tiempo total de término conocido como makespan (C_{max}), la restricción 2.2, asegura que cada trabajo k en cada etapa es asignado a una sola máquina después del trabajo l, la restricción 2.3, la cual es complementaria a la restricción 2.2, es una restricción para la calendarización

de los trabajos, la cual asegura que todos los trabajos siguen una secuencia y ruta bien definida a través de cada una de las máquinas en cada etapa, esta restricción determina que máquina en cada etapa debe ser calendarizada.

La restricción 2.4 calcula el tiempo total para el primer trabajo disponible en cada máquina en la etapa 1, así mismo, la restricción 2.5 calcula el tiempo total para el primer trabajo disponible en cada máquina en las otras etapas, además de garantizar que cada trabajo se procesa en todas las etapas anteriores con respecto al tiempo de inicio relativos al trabajo a procesar como del trabajo anterior, la restricción 2.6 controla la formación del bloqueo de las máquinas que deben esperar a que la máquina en la siguente etapa se desocupe.

La restricción 2.7 calcula el procesamiento de un trabajo en función del procesamiento de su predecesor en la misma máquina de una etapa, esta restricción controla la creación de tiempo de inactividad (bloqueo) de la máquina, ambas restricciones 2.6 y 2.7 aseguran que un trabajo no puede comenzar a ser procesado hasta que este disponible, es decir hasta que haya terminado de ser procesado en la etapa anterior y que el trabajo previo que actualmente se esta procesando en la etapa actual se haya completado.

La restricción 2.7 indica que el procesamiento de cada trabajo en cada etapa se inicia inmediatamente después de su salida de la etapa anterior, más el tiempo de inicio dependiente de la secuencia del trabajo anterior, es decir, que calcula el tiempo de salida en relación a cada trabajo en cada etapa excepto para la última etapa, la restricción 2.8 asegura que cada trabajo sale de la línea de producción al procesarse en la última etapa, la restricción 2.9 define el tiempo total de término, finalmente la restricción 2.10 asegura que la variable de decisión x_{iljk} solo toma valores booleanos.

2.5. Ejemplo de una instancia del problema.

Supongamos que tenemos un FFS-SDST integrado por 3 trabajos $N = \{1, 2, 3\}$ que pasan a través de un conjunto de 3 etapas en serie $M = \{1, 2, 3\}$ en donde cada etapa k con $k \in M$ esta compuesto de 2 máquinas paralelas idénticas $m_k = \{1, 2\}$ para procesar cualquiera de los j trabajos con $j \in N$ en cualquiera de las máquinas i con $i \in m_k$, el cual se denota como $FFS_3 = (PM^{(2)})_{k=1}^{2}|S_{ljk}|C_{max}$. Los tiempos de procesamiento para los trabajos se muestran en la tabla 2.1, en la cual solo se da un tiempo de procesamiento p_{jk} por cada trabajo j en cada etapa k, esto se debe a que todas las máquinas i en cada etapa k son idénticas $p_{ijk} = p_{jk}$ y solo requiere asignar un tiempo para todas.

Los otros dos tipos de máquinas que puede usarse para este problema son máquinas uniformes y máquinas no relacionadas, haciendo una interpretación de [Pinedo, 2008], el uso de máquinas paralelas uniformes asigna un tiempo de procesamiento para todos los trabajos j en cada máquina i de cada etapa k y finalmente las máquinas paralelas no

relacionadas asigna un tiempo de procesamiento por cada trabajo j en cada máquina i de cada etapa k.

p_{jk}	$etapa_{k=1}$	$etapa_{k=2}$	$etapa_{k=3}$
p_{1k}	7	9	5
p_{2k}	5	7	7
p_{3k}	9	5	9

Tabla 2.1. *Tiempos de procesamiento p_{ij} para un problema de $3 \times 3 \times 2$. La tabla muestra los tiempos de procesamiento p_{jk} para un problema de j=3 trabajos que deben procesarse sobre un conjunto de m=3 etapas en donde cada etapa k esta conformado por $m_k = 2$ máquinas en paralelo idénticas denotado como $FFS_3 = (PM^{(2)})_{k=1}^{2}|S_{ljk}|C_{max}$.*

Los tiempos de inicio dependientes de la secuencia SDST(S_{ljk}) se dan entre los trabajos l y j de la etapa k, haciendo omisión del índice i debido a que todas las máquinas son iguales como se muestra en la tabla 2.2. Del lado izquierdo tenemos los trabajos l y del lado derecho los trabajos j con $j, l \in N$ en forma de una matriz cuadrada para representar todas las combinaciones de los SDST divididos por etapas k, la interpretación es como sigue:

1. En el primer renglón de cada etapa k esta indicado con un 0 (cero), denota el tiempo de inicio dependiente de la máquina S_{0jk}, donde $l = 0$ indica que no hay un trabajo l previo a j y que j es el primer trabajo en ser asignado a la máquina i en la etapa k.

2. Los tiempos de inicio dependientes de la secuencia entre los trabajos l y j en la etapa k se establecen en cero solo cuando $l = j$, es decir el trabajo antecesor l y sucesor j no puede ser el mismo.

3. En la última columna esta indicado con un 0 (cero), denota el tiempo de limpieza S_{l0k}, donde $j = 0$ indica que no hay un trabajo j después de l y que l es el último trabajo en ser asignado a la máquina i en la etapa k, no se toman en cuenta para el cálculo del makespan debido a que el tiempo es independiente a la salida del trabajo del sistema, pero pueden ser considerados para otras funciones objetivo.

2.5.1. Optimización de la matriz de tiempos de inicio.

Con el objetivo de optimizar la matriz de los tiempos de inicio dependientes de la secuancia (SDST) que puede llegar a ser muy grande y facilitar con ello su manejo, además de proporcionar una manera elegante en su diseño, se genera una nueva matriz de la siguiente manera:

- Los tiempos de inicio indicados como $S_{111}, S_{221}, S_{331}$,$S_{112}, S_{222}, S_{332}$ y $S_{113}, S_{223}, S_{333}$, no son validos porque $l = j$ como lo marca el punto 2.

S_{ljk}	j_1	j_2	j_3	$0(limpieza)$
		$etapa_{k=1}(S_{ljk})$		
0(inicio)	S_{011}=2	S_{021}=1	S_{031}=3	S_{001}=0
l_1	S_{111}=0	S_{121}=2	S_{131}=1	S_{101}=0
l_2	S_{211}=1	S_{221}=0	S_{231}=1	S_{201}=0
l_3	S_{311}=2	S_{321}=3	S_{331}=0	S_{301}=0
		$etapa_{k=2}(S_{ljk})$		
0(inicio)	S_{012}=1	S_{022}=2	S_{032}=3	S_{002}=0
l_1	S_{112}=0	S_{122}=3	S_{132}=2	S_{102}=0
l_2	S_{212}=1	S_{222}=0	S_{232}=1	S_{202}=0
l_3	S_{312}=3	S_{322}=2	S_{332}=0	S_{302}=0
		$etapa_{k=3}(S_{ljk})$		
0(inicio)	S_{013}=3	S_{023}=2	S_{033}=1	S_{003}=0
l_1	S_{113}=0	S_{123}=3	S_{133}=1	S_{103}=0
l_2	S_{213}=2	S_{223}=0	S_{233}=1	S_{203}=0
l_3	S_{313}=3	S_{323}=2	S_{333}=0	S_{303}=0

Tabla 2.2. *Tiempos de inicio dependientes de la secuencia SDST para el problema de* $3 \times 3 \times 2$. *La tabla muestra los tiempos dependientes de la secuencia* S_{ljk} *para las etapas m={1,2 y 3}, se observa que los tiempos de limpieza al procesar el último trabajo j no son tomados en cuenta.*

- Los tiempos de limpieza S_{101}, S_{201}, S_{301}, S_{102}, S_{202}, S_{302} y S_{103}, S_{203}, S_{303}, no se toman en cuenta para el cálculo del makespan como lo establece el punto 3.

El resultado es una nueva matriz elegante de tamaño $N \times N \times N$ mostrada en la tabla 2.3, que es utilizada para la generación de las instancias de prueba a usar en esta tesis como se muestra en el apéndice B, esta nueva matriz requiere una lectura especial mostrada en el apéndice D para pasar de matriz compacta a una matriz de operación más directa en programación con fines de cálculo del makespan.

2.5.2. Grafo disyuntivo y de solución.

El grafo disyuntivo para el problema de la sección 2.5 se muestra en la figura 2.2, en el se puede apreciar el total de las operaciones $O = m \times n = 9$, más dos operaciones ficticias (entrada y salida) para un total de 11, los tiempos de procesamiento de cada operación aparecen en medio del conjunto de máquinas paralelas idénticas, los tiempos de inicio dependientes de la secuencia se especifican en la tabla 2.3. La disyuntiva aquí primero es determinar el orden de los trabajos en cada etapa k y después seleccionar la máquina i de la operación O_{jk} para determinar la operación O_{ijk} que involucra el índice de la máquina seleccionada de entre las dos posibles $i = \{1|2\}$ sin violar las restricciones de precedencia.

El total de operaciones posibles se muestran en la tabla 2.4, para la

$etapa_{k=1}(S_{ljk})$		
S_{011}=2	S_{021}=1	S_{031}=3
S_{211}=1	S_{121}=2	S_{131}=1
S_{311}=2	S_{321}=3	S_{231}=1
$etapa_{k=2}(S_{ljk})$		
S_{012}=1	S_{022}=2	S_{032}=3
S_{212}=1	S_{122}=3	S_{132}=2
S_{312}=3	S_{322}=2	S_{232}=1
$etapa_{k=2}(S_{ljk})$		
S_{013}=3	S_{023}=2	S_{033}=1
S_{213}=2	S_{123}=3	S_{133}=1
S_{313}=3	S_{323}=2	S_{233}=1

Tabla 2.3. *Optimización de los tiempos de inicio dependientes de la secuencia. La tabla muestra una matriz $N \times N \times N$ para representar los tiempos de inicio dependientes de la secuencia S_{ljk} optimizados a partir de la tabla 2.2.*

etapa$_{k=1}$ *y* trabajo$_{j=1}$ las dos operaciones O_{ijk} que relacionan la máquina i son $trabajo_{j=1}(O_{111}, O_{211})$, $trabajo_{j=2}(O_{121}, O_{221})$ y $trabajo_{j=3}(O_{131}, O_{231})$. Para la $etapa_{k=2}$, $trabajo_{j=1}(O_{112}, O_{212})$, $trabajo_{j=2}(O_{122}, O_{222})$ y $trabajo_{j=3}(O_{132}, O_{232})$. Finalmente la $etapa_{k=3}$, $trabajo_{j=1}(O_{113}, O_{213})$, $trabajo_{j=2}(O_{123}, O_{223})$ y $trabajo_{j=3}(O_{133}, O_{233})$.

El total de posibles operaciones es un múltiplo del número de máquinas por etapa m_k es decir $O_{posibles} = m \times n \times m_k = 18$, más dos operaciones ficticias (entrada y salida), pero al seleccionar solo una de las m_k máquinas por operación, quedan 11 operaciones que son las representadas en el grafo dirigido $G = (V, A)$ de la figura 2.3 para una posible solución a este problema. El orden en el cual son procesados los trabajos entre una etapa y otra es determinado mediante el uso de algoritmos de aproximación, esta tesis propone un algoritmo de aproximación AGHCGrid que será explicado en el capitulo 4 .

O_{ijk}	$etapa_{k=1}$		$etapa_{k=2}$		$etapa_{k=3}$	
máquina	1	2	1	2	1	2
trabajo 1	O_{111}	O_{211}	O_{112}	O_{212}	O_{113}	O_{213}
trabajo 2	O_{121}	O_{212}	O_{122}	O_{222}	O_{123}	O_{223}
trabajo 3	O_{131}	O_{231}	O_{132}	O_{232}	O_{133}	O_{233}

Tabla 2.4. *Total operaciones posibles para el problema de $3 \times 3 \times 2$. La tabla muestra un total de $m \times n \times m_k = 18$ operaciones posibles que darán lugar a solo 9 operaciones reales, más dos ficticias para el total de las 11 requeridas para una posible solución.*

El grafo dirigido G=(V,A) de la figura 2.3, donde V representa el conjunto de vértices (nodos) de las operaciones y A es el conjunto de aristas (arcos) que conectan estos vértices, donde cada arista(r, s) esta formado por un par de operaciones $(O_{origen}, O_{destino})$

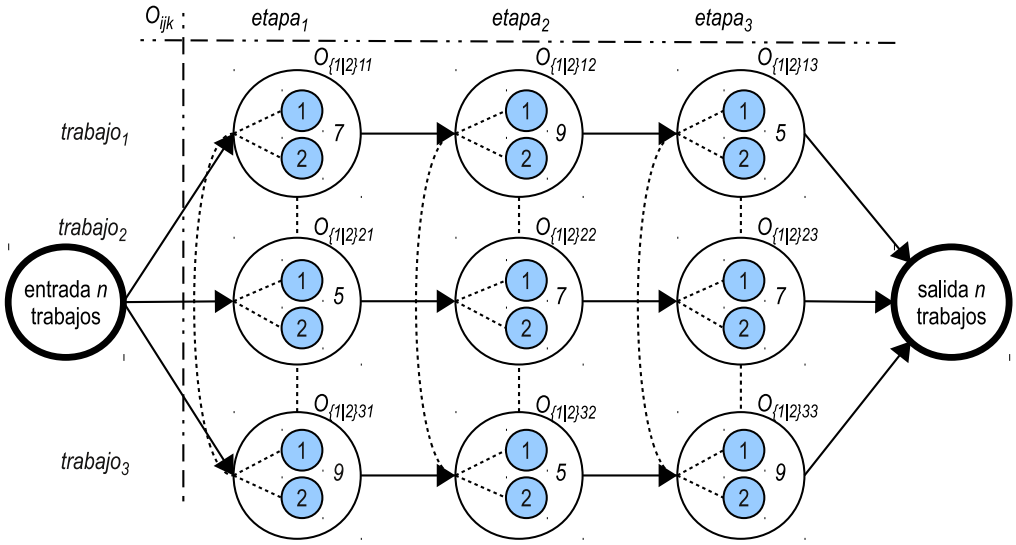

Figura 2.2. *Grafo disyuntivo para el problema de* $3 \times 3 \times 2$. *La figura muestra un total de 9 operaciones, más dos ficticias para un total de 11, en cada una de las 9 operaciones se debe de decidir utilizando algún método por una de las dos máquinas disponibles.*

tal que $r, s \in V$ y $r \neq s$ que representa una calendarización factible y donde cada operación (O_{ijk}) asocia los tres elementos máquina (i), trabajo(j) y etapa(k) respectivamente. Las operaciones (vértices) y las aristas (arcos) representados en el grafo son tomados de la tabla 2.4 que representa las 9 operaciones seleccionadas del conjunto posible de 18 operaciones mediante un método de selección, por último las operaciones descartadas son representadas de color opaco en la figura 2.3. El procesamiento de cada operación O_{ijk} del grafo solución esta conformado por dos elementos:

1. El tiempo se inicio dependiente de la secuencia S_{ljk}.

2. El tiempo de procesamiento p_{jk}.

La suma de ambos $costo_{uv} = S_{jk} + p_{jk}$ representan el costo de transitar por la arista(u,v) donde u y v corresponden a las operaciones ($O_{origen}, O_{destino}$), p. ej. el costo de transitar del nodo inicial (entrada) hacia la operación O_{111} es $S_{011} = 2$ con l=*0*, j=*1*,k=*1* que representa el tiempo de inicio dependiente de la máquina entre el trabajo l y j, este tiempo es dependiente de la máquina porque es el primero trabajo es ser asignado a la máquina 1 (i=*1*), debido a que no hay un trabajo previo a j ya que l=*0*, representa el primer trabajo(j=*1*) asignado a la máquina 1 de la etapa 1, más $p_{11} = 7$ con j=*1* y k=*1*, que representa el tiempo de procesamiento del trabajo 1 en la etapa 1, ambos tiempos se obtienen de las tablas 2.2 y 2.1 respectivamente, la suma de ambos es 9 que es el costo de transitar por la arista que une las operaciones ($entrada, O_{111}$).

Posteriormente el costo de transitar por la arista que une las operaciones (O_{111}, O_{131}) es $S_{131} + p_{31} = 1 + 9 = 10$, que es el tiempo de inicio entre los trabajos l=1 y j=3 en la etapa 1 más el tiempo de procesamiento del trabajo 3 en la etapa

1. Los demás costos de las subsecuentes tiempos de inicio y operaciones están representadas en las aristas correspondientes hasta llegar al nodo ficticio de salida el cual tiene un costo de 0.

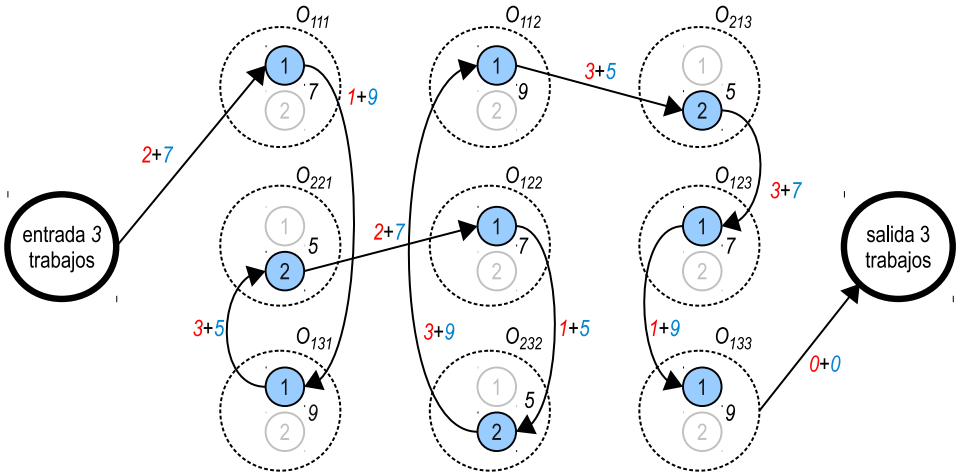

Figura 2.3. *Grafo de solución para el problema de* $3 \times 3 \times 2$. *La figura muestra el grafo de una posible solución al problema, en donde se representa la secuencia que ha de seguir los trabajos para ser procesados en las máquinas de cada etapa.*

2.5.3. Diagrama de Gantt y representación matricial.

La forma más comprensible de visualizar una calendarización es mediante el uso de una gráfica de Gantt donde el eje X representa el tiempo y el eje Y los recursos, la gráfica esta dividida en etapas y en cada etapa se sitúan las máquinas que corresponden a esa etapa, los tiempos de inicio y los tiempos de procesamiento de los trabajos que representan los tiempos de inicio y las operaciones se posicionan sobre el tiempo en la máquina y etapa correspondiente respetando el orden en la secuencia de la calendarización, los tiempos de los trabajos se van incrementando a medida que pasan a la siguiente etapa respetando los tiempos de terminación en la máquina y etapa anterior.

Por ejemplo el grafo dirigido de solución 2.3 es representado por la gráfica de Gantt de la figura 2.4, en la cual los tiempos de inicio dependientes de la máquina y de la secuencia están representados de color gris, las operaciones para los tiempos de procesamiento de los trabajos están representados de diferentes tonalidades de grises el trabajo 1, trabajo 2 y trabajo 3 respectivamente, y finalmente los tiempos de inicio que corresponden a los tiempos de limpieza que no son tomados en cuenta para el cálculo del makespan cuyo valor 0 (cero) están difuminados. El makespan correspondiente a esta calendarización corresponde a un valor de 37 para el trabajo 1 en la etapa 3, que es el último en salir del sistema y el cual tiene el máximo tiempo total de término.

Otra manera de representar el diagrama de Gantt anterior que esta estrechamente relacionado con el grafo de solución 2.3 es mediante el uso de una matriz de tamaño

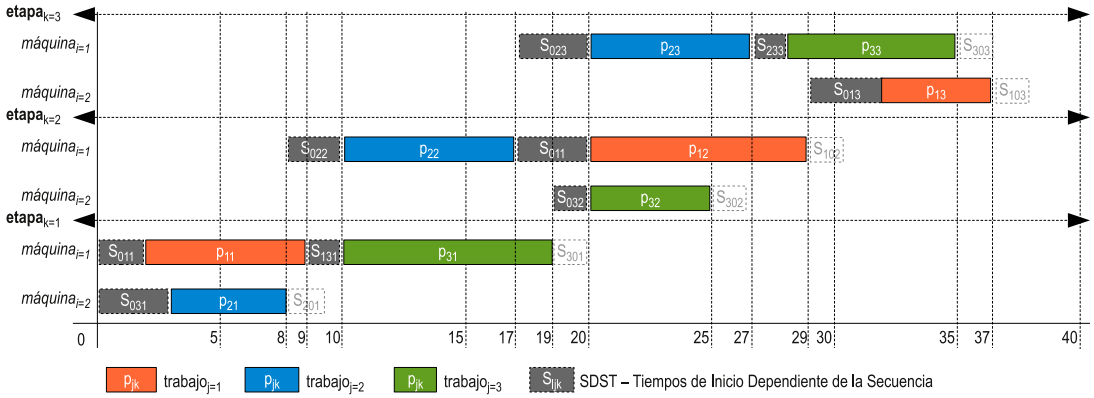

Figura 2.4. *Diagrama de Gantt para el problema de* $3 \times 3 \times 2$. *La figura muestra la calendarización de los trabajos indicando los tiempos de inicio* S_{ljk} *dependientes de la secuencia y los tiempos de procesamiento* p_{ij} *para los tres trabajos.*

$M \times N$, en donde las filas representan las etapas y los recursos, y las columnas la secuencia de los trabajos en dichos recursos, por ejemplo la representación del diagrama de Gantt 2.4 queda como sigue:

$$\begin{pmatrix} O_{111} & O_{221} & O_{131} \\ O_{212} & O_{122} & O_{232} \\ O_{213} & O_{223} & O_{133} \end{pmatrix}$$

La matriz contiene la secuencia de operaciones por cada etapa, en donde cada operación relaciona los índices i,j,k de la máquina, trabajo y etapa respectivamente, los tiempos de inicio pueden determinarse a partir de la secuencia de estas operaciones y los tiempos de procesamiento de los índices de la operación. Para incluir las dos operaciones ficticias (entrada,salida) estas se puede codificar sobre un vector de una sola dimensión de tamaño $M \times N + 2$, considerando que cada serie de operaciones múltiplo de N, después de la primera operación ficticia representa una etapa k, quedando como sigue:

$$\begin{pmatrix} O_{entrada} & O_{111} & O_{221} & O_{131} & O_{212} & O_{122} & O_{232} & O_{213} & O_{223} & O_{133} & O_{salida} \end{pmatrix}$$

La desventaja de estas representaciones es que no se pueden apreciar los tiempos de inicio y de procesamiento incluidos en cada operación, la ventaja es que para efectos de programación es más fácil el cálculo de las operaciones con el uso de matrices y vectores.

Capítulo 3

MÉTODOS APLICADOS AL FLEXIBLE FLOW SHOP

El FFS se encuentra en todo tipo de escenarios del mundo real, siendo muy frecuente en la industria, razón por la cual es de mucho interés que pueda ser aplicado a la industria de la manufactura, FFS ha atraído mucho la atención de los investigadores dada su complejidad de tipo *NP-Duro* y su relevancia práctica, razón por la cual ha sido ampliamente estudiado utilizando diversos técnicas, que incluyen métodos exactos, heurísticas y metaheurísticas.

Dentro de los métodos heurísticos Branch and Bound (B&B) es la técnica preferida para resolver de forma óptima un FFS, seguida de la programación matemática (MIP) como se muestra en la sección 3.1.

En cuanto a las heurísticas, las más sencillas consisten en reglas de atención, también conocidas como reglas de prioridad o despacho, las cuales son mostradas en la sección 3.2.

Finalmente las metaheurísticas, estas agregan un elemento aleatorio con la idea de que el uso repetido de esta técnica puede generar mejores resultados, entre las que destacan: algoritmos genéticos (AG), optimización por colonia de hormigas (OCH) y recocido simulado (RS), las cuales son mostradas en la sección 3.3.

3.1. Métodos exactos.

En la literatura se pueden encontrar diversos métodos para la resolución de problemas de optimización de forma exacta, entre los que destacan los métodos de programación lineal entera mixta (MILP) y ramificación y acotamiento conocido como B&B

(Branch and Bound) y . En esta sección nos centraremos en explicar estos dos métodos.

3.1.1. Programación Lineal Entera Mixta (MILP).

Una gran variedad de problemas de optimización provenientes de áreas de aplicación muy diversas y cuya resolución es muy compleja desde el punto de vista computacional referidos como problemas NP-Completos, pueden ser modelados a partir de la programación lineal entera mixta.

En particular, la mayor parte de los problemas de optimización combinatoria admiten modelos en esta familia. La modelización como programas lineales enteros mixtos ha resultado ser, en los últimos años, la herramienta más eficiente para la resolución exacta de instancias de estos problemas.

Un problema de programación lineal entera mixta (MILP) es un problema de programación lineal en el que algunas de las variables son enteras, si todas las variables enteras son binarias (0/1), el problema se denomina problema de programación lineal entera mixta. Si, por otra parte, todas las variables son enteras, el problema se denomina problema de programación lineal entera estricta, en ingeniería los problemas más frecuentes son los problemas de programación lineal entera mixta, estos problemas proporcionan un marco de modelado flexible y eficiente para formular y resolver muchos problemas de ingeniería.

Un problema de programación lineal entera mixta, se representa en forma estándar al minimizar:

$$Z = \sum_{j=1}^{n} c_j x_j \tag{3.1}$$

Sujeto a:

$$\sum_{j=1}^{n} c_{ij} x_j = b_i; \quad i = 1, 2, ..., m \tag{3.2}$$

$$x_{ij} > 0; \quad i = 1, 2, ..., m \tag{3.3}$$

$$x_{ij} \in \mathbb{N} \quad i = para\, todos\, o\, algunos\, j = 1, 2, ..., m \tag{3.4}$$

donde $\mathbb{N}$ es el conjunto {0,1,...,n}

La posibilidad de utilizar variables enteras o binarias amplía notablemente las posibilidades de modelización matemática, a diferencia del problema con variables reales, el número de soluciones de un modelo de programación lineal entera es finito, por lo que podría plantearse la posibilidad de encontrar la solución mediante la exploración de todas las soluciones posibles.

Sin embargo, el número de soluciones a explorar para un problema mediano puede ser muy elevado: en principio, para un problema con n variables enteras debemos explorar 2^n soluciones (excluyendo quizás algunas descartadas por las restricciones). Para $n = 30$, tenemos $2^{30} = 1,073,741,924$ soluciones posibles. Se han desarrollado metodologías que permiten explorar de manera más eficiente que la simple enumeración el conjunto de soluciones posibles. Gran número de estas metodologías emplean la lógica del branch and bound, y están incorporadas a la mayoría de programas informáticos que resuelven modelos lineales.

Para resolver un problema FFS se utilizan lenguajes algebraicos de modelado que tienen capacidad de indexación de las variables y ecuaciones, permiten cambiar sin dificultad las dimensiones del modelo, de forma natural separan datos de resultados. Desde el punto de vista del modelador permiten la detección de errores de consistencia en la definición y verificación del modelo. Desde el punto de vista del usuario simplifican drásticamente su mantenimiento, entre los lenguajes de modelado más conocidos se pueden mencionar: GAMS[1] y AMPL[2] por mencionar algunos.

3.1.2. Branch and Bound (B&B).

Los algoritmos basados en B&B son posiblemente los más empleados en la resolución exacta de problemas de optimización combinatoria, en un algoritmo de B&B cada estado se interpreta como un subconjunto de soluciones del problema original, en donde en el estado inicial, se tienen todas las soluciones y mediante el proceso de ramificación, un conjunto de soluciones se descompone en la unión disjunta de varios conjuntos, hasta llegar a conjuntos unitarios que representan una única solución del problema, este planteamiento hace que el espacio de búsqueda tenga estructura de árbol [González, 2011].

Los métodos de ramificación y poda son capaces de explotar en cierta medida conocimiento del dominio del problema, mediante heurísticos y reglas de atención, con el objetivo de guiar la búsqueda hacia regiones buenas del espacio de soluciones. Los componentes fundamentales de los algoritmos de ramificación y poda son: un esquema de ramificación, un método de cálculo de cotas inferiores, un método de cálculo de cotas superiores, y una estrategia de control, mediante esta técnica se han podido resolver de forma satisfactoria un gran número de problemas de optimización complejos y problemas de satisfacción de restricciones.

[1] http://www.gams.com/
[2] http://www.ampl.com/

El método de B&B utiliza un árbol de búsqueda [Cruz, 2005], en donde la ramificación es una técnica enumerativa y puede encontrar todas las posibles soluciones de un problema dado, para poder encontrar la mejor solución se procede a acotar el espacio de soluciones, dando lugar a una búsqueda en un espacio de soluciones más pequeño.

Para lograr esto, se comienza por dividir el espacio de soluciones en varios subespacios y mediante una técnica que nos permita saber en cual de todos los subespacios se puede localizar la solución óptima, para este subespacio se continúa dividiendo en forma sucesiva hasta llegar a la solución óptima, para ello se requieren tener buenas cotas que permitan definir los subespacios de soluciones generados.

Cada rama del árbol que se genera representa una solución y cada nodo del árbol representa un subespacio de soluciones, considerando al nodo como la raíz de las ramas. Para elegir la siguiente rama que acerque más al óptimo, es necesario utilizar un método que encuentre una cota inferior conocido como LB (Lower Bound) que es la mayor obtenida en las otras ramas, pero que no debe rebasar una cota superior que se establece como un dato conocido, la cota superior es conocida como UB (Upper Bound) y es un valor por arriba y lo más cercano posible a la solución óptima, esto último evitara que la cota superior se aleje de la solución sin que se tenga conocimiento de esto. El acotamiento a los subespacio de soluciones evitara hacer búsquedas innecesarias en otro subespacio de soluciones en donde no se encuentre el óptimo.

3.1.3. Revisión de la literatura - Método exactos.

Branch and Bound (B&B) es sin lugar a dudas, la técnica más utilizada para encontrar la solución óptima al problema de FFS y representa casi toda la investigación hasta el momento, sin embargo, se ha concentrado en versiones simplificadas del problema, en donde el escenario más simple tan solo considera sólo dos etapas con una sola máquina en la primera etapa y dos máquinas idénticas en la segunda etapa, para este caso especifico donde ya es considerado NP-Duro [Gupta, 1998].

El artículo más antiguo que se encontró con B&B es un algoritmo propuesto por [Rao, 1970], más recientemente [Bolat et al., 2005] estudiaron el mismo problema obteniendo soluciones aproximadas utilizando B&B, heurísticas y algoritmos genéticos. Otro método exacto para este problema sin utilizar tiempos de espera entre las dos etapas fue tratado por [Guirchoun et al., 2005], de manera similar pero ahora permitiendo tiempos de espera entre las dos etapas se trato en [Arthanary and Ramaswamy, 1971] y también por [Mittal and Bagga, 1973].

Tratando el problema con más dureza, encontramos problemas con dos etapas y cualquier número de máquinas paralelas idénticas en la segunda etapa que se han estudiado recientemente por [Lee and Kim, 2004] quienes propusieron un método B&B para minimizar la tardanza total, para este tipo de problema se han tratado de manera exitosa hasta 15 trabajos, para los cuales se obtiene la solución óptima en tiempos

razonables, el caso en donde la primera etapa puede tener tener cualquier número de máquinas idénticas y la etapa dos sólo una, es estudiado en [Gupta et al., 1997], para el cual los autores proponen un B&B que es capaz de obtener buenas soluciones en un tiempo razonable.

Para dos y tres etapas con máquinas uniformes en paralelo en cada etapa [Dessouky et al., 1998] son los primeros en proponer un B&B. [Salvador, 1973] estudio la variante no-wait del problema FFS proponiendo un B&B que explora sólo las permutaciones que pueden asignar un trabajo a la primera máquina disponible en cada etapa. Recientemente [Choi and Lee, 2009] han estudiado el problema en dos etapas con varias máquinas idénticas en paralelo en cada etapa para minimizar la tardanza total, en donde los autores proponen un método B&B en combinación con una heurística.

El más antiguo B&B para tratar el FFS clásico en su forma general mostrado en la sección 1.3, en donde se puede tener cualquier número de etapas y cualquier número de máquinas en paralelo por etapa, es estudiado por [Brah and Hunsucker, 1991; Brah, 1998], en la cual, la estructura de árbol propuesta es una adaptación de la que presentada por primera vez en [Bratley and Florian, 1975] que trata el problema de máquinas paralelas que solo tienen una etapa, y que ha sido el método más utilizado cuando se quiere tratar con un número indefinido de etapas.

A pesar de que en su momento [Brah and Hunsucker, 1991] propusieron sofisticadas cotas inferiores, solo podían resolver de manera óptima instancias muy pequeñas de ocho trabajos por dos etapas, donde cada etapa tenia tres máquinas en paralelo que podía ser resuelto en varias horas. [Rajendran and Chaudhuri, 1992a] también estudió el mismo problema y aunque no se observa una comparación directa con los resultados de [Brah and Hunsucker, 1991], si se observa un mejor rendimiento del algoritmo, de manera similar [Rajendran and Chaudhuri, 1992b] y [Vignier et al., 1999] utilizan la misma metodología pero para minimizar el flow time.

Para tratar un FFS de m-etapas, la estrategia que generalmente se sigue al utilizar algoritmos basados en B&B, es construir primero la solución para la etapa 1, luego para la etapa 2 y así sucesivamente hasta la etapa m. En [Carlier and NTron, 2000] se muestra una nueva estrategia en cada punto de decisión de la etapa crítica que es el cuello de botella, en donde un trabajo es seleccionado si el makespan para esa calendarización es más pequeño que una cierta cota superior (UB), otra estrategia igualmente exitosa es la de incorporar métodos no exactos (heurísticas) al B&B para encontrar cotas superiores como los mostrados por [Portmann et al., 1998; Morita and Shio, 2005], en donde se han usado reglas de despacho en combinación con algoritmos genéticos para generar las cotas superiores al inicio de cada etapa.

Algunos autores como [Sherali et al., 1990], han utilizado implícitamente B&B a través de la programación matemática, es decir, representan su problema como un modelo de MIP y luego usan un resolvedor, tal es el caso de [Gooding et al., 1994] que utiliza la programación matemática para modelar un FFS con una función objetivo para minimización los costos de producción. Otro caso es [Sawik, 2000] que modeló una línea

de flujo flexible con bloqueos, que posteriormente mejoro en [Sawik, 2001]. [Pearn et al., 2005] propuso un modelo matemático y una heurística para un problema complejo de embalaje con tiempos de inicio dependientes de la secuencia, finamente a pesar del éxito relativo de los algoritmos exactos, todavía son incapaces de resolver instancias de tamaño medio y grandes, así mismo los problemas del mundo real son demasiado complejos para ser tratados por métodos exactos, por lo que se requiere el uso de heurísticas.

3.2. Heurísticas.

Una heurística es un método desarrollado con base en la experiencia que permite obtener soluciones aproximadas de un problema sin garantizar la solución óptima [Mártinez, 2010; Wetzel, 1983]. Este tipo de métodos se aplican cuando no es posible encontrar una solución óptima por un método exacto, debido a la naturaleza y complejidad del problema, las heurísticas usadas ampliamente para tratar instancias del FFS son las reglas de atención.

3.2.1. Reglas de atención (dispatching rules).

Conocidas también como reglas de prioridad o despacho, son ampliamente utilizadas para tratar de resolver problemas de calendarización, permiten seleccionar el siguiente trabajo a ser procesado por una máquina determinada, a la hora de estar construyendo una calendarización, las ventajas que presentan estas heurísticas es que son muy rápidas y su implementación muy sencilla, en contraparte su desventaja es que proporcionan en la mayoría de los casos una baja calidad en las soluciones, por lo que para mejorar la calidad, se utilizan en combinación con otros métodos.

Las reglas de atención manejan la prioridad, en donde un conjunto de trabajos que están esperando ser procesados en una determinada máquina, una vez desocupada, el trabajo de mayor prioridad es asignada a esa máquina, así mismo, su facilidad de uso le permite ser utilizados por otros algoritmos para modificar el criterio de selección a la hora de construir soluciones heurísticas [González, 2011].

Las reglas de atención se clasifican de varias formas, [Jackson, 1957] propuso dos clasificaciones.

1. Reglas estáticas. Las reglas estáticas no dependen del tiempo sino sólo de los datos de los trabajos y las máquinas

2. Reglas dinámicas. Las dinámicas sí dependen del tiempo, por lo que en un instante de tiempo un trabajo j puede tener mayor prioridad que un trabajo l, y en otro instante de tiempo puede ocurrir lo contrario.

Las reglas de atención también se clasifican con base en la información que manejan [Pinedo, 2005].

1. Reglas locales. Las reglas locales sólo manejan información perteneciente a la cola de la máquina en la que el trabajo está esperando

2. Reglas globales. Manejan las reglas locales, además de la información relacionada con otras máquinas.

Existe una gran cantidad de reglas de atención como los mostrado en [Lawrence, 1984; Holthaus and Rajendram, 1997], entre las reglas de atención básicas más utilizadas podemos destacar las siguientes:

- CP (camino crítico): se utiliza cuando los trabajos están sujetos a restricciones de precedencia. Se selecciona el trabajo cuyo tiempo de procesamiento total es mayor, considerando únicamente las actividades precedentes a la actual.

- CR (ratio crítico): selecciona el trabajo con el menor ratio crítico, definido como el cociente entre el tiempo que falta hasta el due date del trabajo dividido por el tiempo de procesamiento restante del trabajo.

- EDD (instante de finalización más temprano): esta regla selecciona el trabajo que tenga el due date más próximo.

- FCFS (primera en llegar, primera en ser atendida): esta regla selecciona el trabajo que antes llega a la cola de la máquina. Ese será el trabajo con un tiempo de inicio más temprano, y por ello también es llamada a veces ERD (instante de inicio más temprano).

- LFJ (trabajo menos flexible): esta regla se suele utilizar cuando existen máquinas en paralelo que no son idénticas. Si cada trabajo j puede ser procesado por un subconjunto M_j de esas máquinas, se seleccionará aquel trabajo que pueda ser procesado por el menor subconjunto M_j . Es decir, el trabajo con menos alternativas de procesamiento posibles.

- LNS (mayor número de sucesores): esta regla se puede utilizar cuando los trabajos están sujetos a restricciones de precedencia, y selecciona el trabajo que tenga un mayor número de trabajos sucesores.

- LPT (tiempo de procesamiento más largo): esta regla da más prioridad a los trabajos cuyo tiempo de procesamiento es mayor. Cuando se tienen máquinas en paralelo esta regla tiende a equilibrar sus cargas de trabajo. Esto es debido a que suele resultar ventajoso mantener trabajos con tiempos de procesamiento cortos para más adelante, porque estos trabajos serán útiles al final para equilibrar la carga de las máquinas.

- MS (mínima holgura): selecciona el trabajo que en el momento de estar libre una máquina tiene el menor tiempo de holgura. El tiempo de holgura de un trabajo j se define como $max(d_j - pr_j - t, 0)$, siendo d_j su due date, pr_j su tiempo de procesamiento restante, y t el instante de tiempo actual.

- MWKR (mayor trabajo restante): elige al trabajo con un mayor tiempo de procesamiento restante.

- SIRO (servicio en orden aleatorio): de acuerdo con esta regla, cuando una máquina esta libre el siguiente trabajo es seleccionado de forma aleatoria de entre los trabajos que están esperando para utilizar dicha máquina.

- SPT (tiempo de procesamiento más corto): se selecciona el trabajo cuyo tiempo de procesamiento es más pequeño.

- SQNO (cola más corta en la siguiente operación): selecciona el trabajo con la menor cola a la siguiente máquina en su ruta. La longitud de la cola a la siguiente máquina puede ser medida de distintas formas, por ejemplo el número de trabajos esperando en cola o la cantidad total de trabajo esperando en cola.

- S/RPT (holgura dividida entre el tiempo de procesamiento restante): la prioridad asignada a cada trabajo es la holgura, calculada de la misma forma que en la regla MS, pero esta vez dividida por el tiempo de procesamiento restante del trabajo.

- SST (tiempo de setup más corto): selecciona el trabajo que tendría un tiempo de setup más pequeño si se planificara a continuación.

- SSTPT (tiempo de setup y de procesamiento más corto): selecciona el trabajo que tenga más pequeña la suma de su tiempo de procesamiento restante y su tiempo de setup si se planificara a continuación.

- TSPT (tiempo de procesamiento más corto interrumpido): esta regla da prioridad a los trabajos de acuerdo con la regla SPT, con la excepción de los trabajos que han estado esperando más tiempo de un máximo fijado. Esos trabajos pasan al frente de la línea de espera en algún orden especificado (utilizando, por ejemplo, la regla FCFS).

- WSPT (tiempo de procesamiento ponderado más corto): los trabajos se seleccionan en orden decreciente de w_j/p_j , siendo w_j el peso asignado al trabajo j y p_j su tiempo de procesamiento en la máquina considerada. Cuando todos los trabajos tienen una ponderación igual, la regla WSPT es equivalente a la regla SPT.

En [Vepsalainen and Morton, 1985; Pinedo, 2005] se muestra el uso de reglas de atención básicas y compuestas, en donde las reglas de atención básicas presentan un buen desempeño al utilizar un único objetivo, pero que para problemas más complejos, se requiere el uso combinado de varias reglas básicas.

3.2.2. Desplazamiento por cuello de botella (SB).

El desplazamiento por cuello de botella conocido comúnmente como SBP (*Shifting Bottleneck*), es un algoritmo de búsqueda local propuesto por [Adams et al., 1998], el método esta basado en la idea empírica de que el rendimiento de un sistema depende de la correcta utilización de los recursos que escasean [González, 2011].

Este procedimiento ha sido ampliamente utilizado para tratar el JSSP [Cruz, 2005] el cual utiliza el grafo disyuntivo de [Roy and Sussmann, 1964] y se utiliza para obtener buenas soluciones al JSSP, tambien ha sido utilizado en [Cheng et al., 2001] para el problema de máquinas paralelas no relacionadas.

3.2.3. Revisión de la literatura - Heurísticas.

Un conjunto amplio de reglas de atención son usadas para determinar la secuencia de los trabajos, estas reglas constituyen las políticas generales a seguir para la clasificación y la asignación de trabajos a máquinas que serán asignados en las diferentes etapas del problema. Las reglas de atención se han utilizado como herramientas de solución para muchos problemas de importancia teórica y práctica, en [Guinet and Solomon, 1996] por ejemplo, las reglas de atención se utilizan para minimizar el makespan (C_{max}) y el máximo tiempo de demora (T_{max}) para un FFS de m-etapas. En [Kochhar and Morris, 1987; Kochhar et al., 1988] se estudia este mismo problema pero con la posibilidad de saltarse etapas (skipping), estos autores proponen reglas de atención alternativas combinándolo con un algoritmo de búsqueda local.

En [Takaku and Yura, 2005] se estudian este mismo problema con reglas de atención pero con una función objetivo para calcular el tiempo total de retraso ($\overline{T}$) sin memoria intermedia (buffers). En [Gupta and Tunc, 1991; Sriskandarajah and Sethi, 1989] se propusieron un conjunto de heurísticas de reglas de atención para el problema en dos etapas investigado por [Gupta, 1998]. En [Tsubone et al., 1996] se estudia también un FFS con dos etapas, compuesta por una única máquina de dosificación en la primera etapa y una restricción para la selección de un subconjunto de máquinas en la segunda etapa (M_j). En [Kyparisis and Koulamas, 2001] se usan una serie de reglas de atención para evaluar diferentes funciones objetivo de fecha de entrega (de dates), makespan (C_{max}) y tiempo de flujo ($\overline{F}$).

El problema de FFS con m-etapas con el objetivo de demora total ($\overline{T}$) fue tratado por [Verma and Dessouky, 1999], ellos trataron un problema de m-etapas con máquinas paralelas uniformes y trabajos idénticos, así también investigaron el rendimiento de las reglas de atención y obtuvieron cotas inferiores. En [Jayamohan and C. Rajendran, 2000] es cuando se comienzan a explorar la idea de utilizar diferentes reglas de atención en diferentes etapas del problema. Las reglas de atención son especialmente adecuados para hacer frente a entornos complejos, dinámicos e impredecibles, de ahí su popularidad en

la práctica, por ejemplo en [Paul, 1979] se plantea un FFS con dos etapas para darle una solución a la industria de envases de vidrio, en donde el autor propuso varias reglas de atención.

En [Narasimhan and Panwalkar, 1984; Narasimhan and Mangiameli, 1987] y [Kadipasaoglu et al., 1997] los autores aplicaron las reglas de atención a un simple problema de dos etapas para la fabricación de cable, con una sola máquina en la primera etapa, en donde los autores probaron problemas con un límite de 70 trabajos y hasta cuatro máquinas en la segunda etapa. Otro problema real, esta vez basada en la manufactura textil, fue abordado en [Guinet, 1991] con varias heurísticas. En [Agnetis et al., 1997] estudiaron un problema complejo de FFS que se da en la fabricación de automóviles, en donde diferentes funciones objetivo fueron estudiadas.

Una serie de heurísticas sofisticadas utilizan la estrategia de divide y vencerás, en donde el problema original se divide en subproblemas más pequeños que se resuelven uno a la vez y sus soluciones se integran en una solución completa al problema original, en [Vairaktarakis and Elhafsi, 2000] se estudia un problema de dos etapas, el cual es dividido en una serie de múltiples FS con el fin de reducir la flexibilidad del enrutamiento y reducir los costos de producción.

En [Suresh, 1997] se estudia un problema en dos etapas y lo divide en dos problemas con máquinas paralelas, una por cada etapa, donde los tiempos de inicio para la segunda etapa son los tiempos de terminación de los trabajos en la primera etapa. Las estrategias de divide y vencerás son particularmente eficaces para los problemas de m-etapas, que son variantes del desplazamiento por cuello de botella conocido como SB (Shifting Bottleneck), en donde el SB funciona bajo el principio de dar prioridad absoluta a la etapa que tiene el cuello de botella, maximizando de esta manera la productividad de toda el FS.

En [Choong et al., 2001] se propone un enfoque SB para un problema de m-etapas para minimizar el makespan. En [Yang, 1998] se propone una heurística SB para la minimizar de la demora ponderada total ($\overline{T^w}$), también para un problema de m-etapas, otros autores que explotan la idea SB son [Acero-Domínguez and Patermina-Arboleda, 2004] donde estudian la función objetivo del makespan. En [Lee et al., 2004] se estudia la demora total y en [Chen and Chen, 2008] para el número de trabajos retrasados, en [Chen and Chen, 2009] los mismos autores han propuesto heurísticas similares, pero para el objetivo de la demora total ($\overline{T}$).

Un número relativamente pequeño de heurísticas han sido propuestos para las variantes del problema de m-etapas, por ejemplo se propone en [Ding and Kittichartphayak, 1994; Sawik, 1993] para el problema de m-etapas con cualquier número de máquinas por etapa, más tarde [Santos et al., 2001] propuso una heurística de mejora para el FFS de m-etapas para el makespan. En [Sawik, 1995] se propone una heurística simple para la minimización de makespan para un FS similar pero con memoria intermedia limitada (buffers).

En [Sevastianov, 2002; Kyparisis and Koulamas, 2006] se presenta igualmente una heurística para los FFS con m-etapas utilizando máquinas paralelas uniformes para obtener el makespan, finalmente la mayoría de las heurísticas a la medida para aplicaciones del mundo real, son para problemas relativamente simples con dos o tres etapas solamente. En [Wittrock, 1985] se presenta un sistema real de producción electrónica, en donde autor considera tres etapas idénticas, máquinas en paralelo por etapa y la característica de que los trabajos pueden omitir (skipping) una o dos de las tres etapas de procesamiento.

3.3. Metaheurísticas.

La comunidad científica que trabaja en al área de optimización combinatoria, ha desarrollado en los últimos años una serie de estrategias que aumentan el desempeño de las heurísticas, las cuales son conocidas como metaheurísticas.

Las metaheurísticas utilizan un elemento aleatorio al uso de las heurísticas y la aplican de manera repetitiva, con la idea de que esto puede mejorar los resultados encontrados por las heurísticas simples, tal es el caso de recocido simulado (RS) mostrado en la sección 3.3.3, que esta compuesto por una búsqueda local iterada e incluye un elemento aleatorio para aceptar soluciones malas que permiten escapar de los óptimos locales, también estan los algoritmos genéticos (AG) mostrado en la sección 3.3.1, que están compuestos por una población de individuos, en donde cada individuo es una posible solución, el AG aplica una serie de operadores de selección, cruce y mutación para realizar una exploración del espacio de soluciones.

Finalmente también esta optimización por colonia de hormigas (ACO) mostrado en la sección 3.3.2, en donde la idea principal de este método consiste en converger en el camino más corto entre la fuente de comida y el hormiguero, ACO es capaz de aplicar búsquedas locales cooperativas en donde las hormigas cooperan al utilizar rastros de feromonas y marcar los caminos más visitados, los cuales potencialmente conllevan a encontrar al mejor solución del problema. Estos tres metaheurísticas, donde algunas de ellas pueden incluir una búsqueda tabú (TS) son las más ampliamente usadas para tratar problemas de FFS, otras metaheurísticas también han sido usadas esporádicamente como por ejemplo optimización por enjambre de partículas (OEP), sistema inmune artificial (SIA), redes neuronales (RN) y otras.

3.3.1. Algoritmos genéticos (AG).

Lo algoritmos genéticos (AG) fueron propuestos por [Holland, 1975] y se basan en conceptos de evolución biológica, en ella se postula que los individuos mejor adaptados tendrán más posibilidades se sobrevivir y pasar su carga genética a las nuevas generaciones, con base en esta analogía, los AG son métodos de optimización combinatoria de

búsqueda global, los cuales están conformados por una población de individuos, en donde cada individuo representa una solución para un problema especifico, el AG mide la capacidad de supervivencia de cada individuo evaluando su aptitud o costo dada por la función objetivo asociada el problema, así mientras mayor sea la aptitud de un individuo, mayor serán sus probabilidades de supervivencia y por tanto su carga genética aportará soluciones de buena calidad que se aproximen al óptimo generación tras generación.

El método básico propuesto por [Holland, 1975] y descrito como componentes básicos por [Goldberg, 1989], se muestra en el algoritmo 3.1, en donde una nueva población *P'*, se obtiene a partir de la población anterior P, generación tras generación, aplicando operadores de selección, cruce y mutación como se muestra a continuación:

Algoritmo 3.1 Algoritmo Genético Básico (AG).

```
1: P = Población_Inicial()        ▶ Se construye población inicial aleatoriamente
2: G = 0        ▶ Contador de número de generaciones
3: mientras no se cumpla criterio de paro o convergencia del algoritmo hacer
4:     P' = {}        ▶ Crear población inicial vacia
5:     mientras población P' no esta completa hacer
6:         Seleccionar padres(p1, p2) de P
7:         Aplicar operadores genéticos de cruce con probabilidad p_s
8:         si Se ha producido el cruce de padres(p1, p2) entonces
9:             Aplicar operadores genéticos de mutación con probabilidad p_m
10:            Evaluar descendientes
11:            Agregar descendiente a la población P'
12:        si no
13:            Agregar padres a la población P'
14:        fin si
15:    fin mientras
16:    G = G + 1        ▶ Se incremena el contador de número de generaciones
17:    P = P'        ▶ Se establece la nueva población como la actual
18: fin mientras
```

- Línea 1. Se construye una población inicial de manera aleatoria.
- Línea 2. Se utiliza un determinado número G de generaciones, que se inicializa a *G=0*.
- Línea 3. Mientras el criterio de paro no se cumpla, el cual puede estar determinado por el número de generaciones G establecido ó por la convergencia del algoritmo sobre un tiempo dado, se llevan a cabo las siguientes acciones.
- Línea 4. Se establece la nueva población *P'* como vacía.
- Línea 5 y 6. Mientras la nueva población *P'* no este completa, se lleva a cabo el proceso de selección, el cual puede estar determinado por algún método de selección como el de ruleta [Blickle and Thiele, 1995] o selección por torneo [Díaz, 2008].
- Línea 7. Una vez seleccionados los padres, se aplica un operador de cruzamiento

para llevar a cabo el proceso de exploración del espacio de soluciones, con base a un factor de probabilidad p_m que determina cuando si y cuando no se lleva a cabo el cruce, los operadores de cruce más comúnmente utilizados son cruce de 1 y 2 puntos y cruce uniforme [Araujo and Cervigón, 2009].

- Línea 8-11. Si se determina que los padres se han cruzado, entonces de aplica un operador de mutación para llevar a cabo el proceso de explotación del espacio de soluciones, el operador de mutación aplica una pequeña perturbación al individuo para generar una solución vecina de la solución actual, en donde la probabilidad de mutación suele ser muy baja, por lo general entre el 0.5 % y el 2 %.

- Línea 12 y 13. En caso contrario los padres seleccionados pasan de forma directa a la nueva población *P'*.

- Línea 16. Se incrementa el número de generaciones.

- Línea 17. Se reemplaza la población anterior P por la nueva población *P'*.

Finalmente, para este algoritmo propuesto por [Holland, 1975], posteriormente se han creado numerosas variantes, entre las que destaca prescindir de la población temporal y aplicar los operadores genéticos de cruce y mutación directamente sobre la población. Otra variante consiste en la manera que se remplaza la población, para el cual existen diversas opciones entre las que destacan.

- Reemplazo de padres. Los padres se reemplazan por los hijos.

- Reemplazo de individuos similares. Los individuos de la población actual, serán reemplazados por los individuos descendientes que tenga una aptitud similar.

- Reemplazo de los peores individuos. Los peores individuos de la población actual serán reemplazados aleatoriamente por los nuevos descendientes.

- Reemplazo aleatorio: Los individuos de la población actual serán reemplazados aleatoriamente.

3.3.2. Optimización por colonia de hormigas (OCH).

Las hormigas son insectos sociales que viven en colonias colaborando entre ellas, esta colaboración les permite mostrar un comportamiento complejo para buscar alimento, en este sentido poseen una habilidad para encontrar los caminos más cortos entre su hormiguero y las fuentes de comida. Mientras se mueven entre el hormiguero y las fuentes de comida depositan a lo largo del camino una sustancia llamada feromona.

Las hormigas prefieren de manera probabilista los caminos con mayor concentra-

ción de esta substancia cuando encuentran una intersección y deben elegir un camino, cuanto más fuerte es el rastro, más probabilidades hay de que las hormigas elijan ese camino. Debido a que las hormigas depositan feromona por el camino elegido, esto lleva a reforzar los caminos seleccionados con una concentración de feromona más elevada, así este comportamiento permite a las hormigas encontrar el camino más corto que será el camino que tenga la más alta concentración de feromona.

En la literatura se han propuesto diversos algoritmos que siguen la metaheurística OCH, entre los que se encuentran: Sistema hormiga (SH) [Dorigo and Maniezzo, 1991], sistema de hormigas elitistas (SHE) [Dorigo, 1992], hormiga Q (Ant-Q) [Gambardella and Dorigo, 1995], sistema de colonia de hormigas (SCH) [Dorigo and Gambardella, 1996], sistema de hormigas max-min (SHMM) [Stützle and Hoos, 1996], sistema de hormigas con ordenación (SHO) [Bullnheimer et al., 1997] y el sistema de la mejor peor hormiga (SMPH) [Cordón et al., 2000].

De todas las técnicas desarrolladas, la contribución más interesante es la que se refiere como sistema de colonia de hormigas (SCH), debido a que introduce la evaporación paso a paso con el objetivo de hacer una mayor exploración del espacio de soluciones, a partir de ahí, las subsecuentes técnicas solo introducen pequeñas variaciones técnicas, es por esta razón, que para esta tesis doctoral seleccionamos esta técnica base, que es la más representativa.

3.3.2.1. Sistema hormiga (SH).

Las hormigas al principio se mueven aleatoriamente depositando feromona en los primeros caminos, transcurrido un tiempo las hormigas empezarán a explorar los caminos más prometedores, mientras recorren van reforzando los caminos transitados anteriormente. Las hormigas que encuentran una fuente de comida regresan más rápidamente lo que conlleva a reforzar los caminos más cortos, el resto de las hormigas estará transitando por caminos más largos, por tanto, los caminos más cortos tendrán una concentración mayor de feromona, lo que permitirá a las siguientes hormigas elegir el camino más corto, finalmente este proceso convergerá en el camino más corto de todos los posibles caminos entre el hormiguero y la fuente de comida.

La convergencia se complementa con la evaporación de la feromona con el transcurso del tiempo, así los caminos menos recorridos pierden progresivamente feromona y por tanto serán menos atractivos por las hormigas. El tiempo de evaporación de la feromona es un factor importante ya que puede llevar a quedar estancado en un óptimo local rápidamente debido a la persistencia de la feromona, esto se debe que las hormigas preferirán el camino con más concentración de feromona aun en los casos en que una hormiga encuentre un camino más corto.

El sistema hormiga fue el primer algoritmo propuesto en la literatura [Dorigo and Maniezzo, 1991], esta compuesto por un ciclo principal que le permite construir

soluciones, en donde cada hormiga k de un total de m hormigas, actualiza la cantidad de feromona τ_{ij} existente en el camino del estado i al estado j con base en la siguiente formula 3.5.

$$\tau_{ij} \leftarrow (1-\rho) \times \tau_{ij} + \sum_{k=1}^{m} \triangle \tau_{ij}^{k} \tag{3.5}$$

en donde ρ es la tasa de evaporación de feromona, m es el número de hormigas y $\triangle \tau_{ij}^{k}$ es la cantidad de feromona que se sumará al camino i,j por las m hormigas, este incremento esta determinado por la siguiente ecuación 3.6.

$$\triangle \tau_{ij}^{k} = \begin{cases} \frac{Q}{L_k} & si\ la\ hormiga\ k\ recorre\ el\ camino\ (i,j) \\ 0 & en\ otro\ caso \end{cases} \tag{3.6}$$

en donde Q es una constante y L_k es la longitud de la ruta de la *k-ésima* hormiga.

Durante el proceso de construcción de soluciones, una hormiga k posicionada en el estado i debe calcular la probabilidad de transitar hacia el siguiente estado j al estar construyendo su camino parcial s^p con base en la siguiente ecuación 3.7.

$$p_{ij}^{k} = \begin{cases} \frac{\tau_{ij}^{\alpha} \times \eta_{ij}^{\beta}}{\sum_{c_{il} \in N(s^p)} \tau_{ij}^{\alpha} \times \eta_{ij}^{\beta}} & si\ c_{il} \in N(s^p) \\ 0 & en\ otro\ caso \end{cases} \tag{3.7}$$

donde $N(s^p)$ es el conjunto de estados no visitados por la hormiga k, los parámetros α y β que determinan la importancia entre la cantidad de feromona τ_{ij} y la información heurística η_{ij} calculada con base en la ecuación 3.8.

$$\eta_{ij} = \frac{1}{d_{ij}} \tag{3.8}$$

donde d_{ij} es la distancia entre los estados i,j.

3.3.2.2. Sistema de hormigas Max-Min (SHMM).

Este algoritmo propuesto por [Stützle and Hoos, 1996], tiene la variante de que maneja un límite superior (τ_{max}) y un límite inferior (τ_{min}) para la cantidad de feromona que puede ser depositada en el camino de i a j, y que es calculada con base en la ecuación 3.9.

$$\tau_{ij} \leftarrow \left[(1-\rho) \times \tau_{ij} + \sum_{k=1}^{m} \triangle \tau_{ij}^{best} \right]_{\tau_{min}}^{\tau_{max}} \tag{3.9}$$

esto significa que cuando la cantidad de feromona asociado a $\tau_{ij} > \tau_{max}$ sobrepasa su límite superior, entonces $\tau_{ij} = \tau_{min}$, por el contrario si $\tau_{ij} < \tau_{min}$ sobrepasa su límite inferior, entonces $\tau_{ij} = \tau_{max}$. El incremento de feromona $\triangle \tau_{ij}^{best}$ esta dado por la ecuación 3.10.

$$\triangle \tau_{ij}^{best} = \begin{cases} \frac{Q}{L_{best}} & si\ la\ hormiga\ k\ recorre\ el\ mejor\ camino \\ 0 & en\ otro\ caso \end{cases} \tag{3.10}$$

donde L_{best} es la longitud del camino de la mejor hormiga k.

3.3.2.3. Sistema de colonia de hormigas (SCH).

Sin duda la mejor contribución a la OCH, fue propuesta por [Dorigo and Gambardella, 1996], en donde se propone una mejora de la actualización de feromona con el fin de aumentar la exploración del espacio de soluciones, esta mejora establece dos tiempos para la actualización de feromona:

1. Actualización de feromona local. Esta actualización es aplicada por todas las hormigas paso a paso, es decir, se aplica después de que una hormiga k posicionada en el estado i transita hacia el estado j, mediante la ecuación 3.11.

$$\tau_{ij} = (1-\varphi) \times \tau_{ij} + \varphi \times \tau_0 \tag{3.11}$$

   donde $\varphi \in (0, 1]$ es el coeficiente de evaporación de feromona y τ_0 es la cantidad inicial de feromona. El objetivo de esta evaporación paso a paso es aumentar la exploración, en el sentido de que una evaporación provocará que las siguientes hormigas seleccionen otros estados al construir su camino.

2. Actualización global de feromona. Esta actualización se aplica hasta el final de cada ciclo, por la hormiga que tiene la mejor solución, con base en la formula 3.12.

$$\tau_{ij} = \begin{cases} (1-\rho) \times \tau_{ij} + \rho \times \triangle \tau_{ij} & si\ (i,j)\ pertenece\ al\ mejor\ camino \\ \tau_{ij} & en\ otro\ caso \end{cases} \tag{3.12}$$

3.3.3. Recocido simulado (RS).

Recocido simulado (RS) fue propuesto por [Kirkpatrick et al., 1983], es un algoritmo de búsqueda local iterada, en donde RS simula estocásticamente el enfriamiento lento de un sistema físico, permitiendo ayudar a resolver problemas de optimización, el proceso de simular el enfriamiento lento de un metal, al pasar del estado liquido al estado solido y poder conseguir un solido de mínima entropía, se apoya en el uso del algoritmo de metrópolis [Metrópolis at al, 1953], el algoritmo generalizado de RS se muestra en el algoritmo 3.2, el cual se explica a continuación.

Algoritmo 3.2 Algoritmo de recocido simulado (RS).

```
1:  funcion RS(t_o, m, μ, t_f)
2:      T = t_0
3:      S = S_0
4:      repetir
5:          para i, i = 1, 2, ..., m hacer
6:              S' = N(S)
7:              si (C(S') ≤ C(S)) entonces
8:                  S = S'
9:              si no
10:                 d = C(S') − C(S)
11:                 r = aleatorio[0, 1]
12:                 si (r ≤ e^(−d/T)) entonces
13:                     S = S'
14:                 fin si
15:             fin si
16:         fin para
17:         T = T . μ
18:     hasta (T ≤ t_f)
19: fin funcion
```

- Línea 1. Se reciben los parámetros (t_o, m, μ, t_f) donde t_o es la temperatura inicial, m es el número de ciclos que repite la longitud de la cadena de markov, μ es el coeficiente de enfriamiento, t_f es la temperatura final.

- Línea 2. La metaheurística de RS inicia con una temperatura T alta que influye en la aceptación de casi todas las nuevas soluciones S' generadas a partir del vecindario $N(S)$ en la línea 6, después T gradualmente disminuye al ser multiplicado por el coeficiente μ en la línea 17.

- Línea 3. Se asigna a S la solución inicial S_0 genera de manera aleatoria.

- Linea 4. Corresponde al ciclo para el descenso de temperatura dado por el factor μ.

- Línea 5. Corresponde al ciclo de metrópolis con número de iteraciones igual a la longitud de la cadena de Markov dado por m.

- Línea 6. Se genera una nueva solución *S'* a partir del vecindario de *N(S)* al aplicar una perturbación a la solución actual *S*.

- Línea 7 y 8. Se evalúa el costo de las dos soluciones y se comparan, si el costo de la nueva solución *S'* mejora o iguala el costo de la solución actual *S*, en el entendido que se trata de minimizar, entonces en la línea 8 se toma la nueva solución *S'* como la solución actual *S*.

- Línea 9-15. En caso contrario, se determina la diferencia con base en el costo entre la nueva solución *S'* y *S* en la línea 10 y se genera un número aleatorio r entre 0 y 1 en la línea 11, en la línea 12 y 13 se determina por la función de probabilidad de Boltzmann si se acepta o no la nueva solución S' a pesar de que es peor que la actual *S*, esto aceptación es lo que le permite escapar de óptimos locales.

- Línea 16. Termina el ciclo de metrópolis.

- Línea 17. Se decrementa la temperatura en un factor μ.

- Línea 18. Termina el algoritmo de RS si la temperatura ha bajado lo suficiente hasta alcanzar al temperatura final t_f.

3.3.4. Revisión de la literatura - Metaheurísticas.

La mayoría de las metaheurísticas aplicadas al FFS intentan restringir el espacio de búsqueda, a las cuales se acceden mediante las permutaciones de los n trabajos, la idea es construir con base en esta permutación una calendarización. En [Low, 2005] se considera el estudio de máquinas paralelas no relacionadas en cada estado mediante RS para minimizar el tiempo de flujo total ($\overline{F}$) y considerando también tiempos de inicio dependientes de la secuencia (SDST).

En [Allahverdi and Al-Anzi, 2006; Jin et al., 2006] se presentan varias heurísticas y un método para un problema de FFS de m-etapas tratado también mediante RS que modela las peticiones de un sistema cliente servidor, además se presenta un límite inferior para evaluar el desempeño del algoritmo, en donde los autores se equivocaron y fue corregida después por [Haouari and Hidri, 2008]. En [Naderi et al., 2009a] se utilizó RS para resolver un FFS con tiempos de inicio dependientes de la secuencia y tiempos de transporte entre las etapas de un vehículo guiado de forma automática, para este mismo problema pero sin limitaciones de tiempo se estudia en [Naderi et al., 2009b].

Otros autores han utilizado una representación en donde una permutación para cada máquina en cada etapa se mantiene, esto es referido como orden de operación de procesamiento o representación exacta, en [Janiak et al., 2007] se emplea este sistema y se proponen varios métodos de RS con el fin de hacer frente a un espacio de soluciones grande.

Los algoritmos genéticos (AG) se han utilizado también ampliamente [Xiao et al., 2000] se utilizó un AG para buscar en el espacio de soluciones una permutación para la solución del problema de FFS de m-etapas con una función objetivo para calcular el makespan, el mismo problema con tiempos de inicio dependientes de la secuencia fue tratado en [Kurz and Askin, 2004a] usando AG con una representación de claves aleatorias (RKGA), el cual superó varias otras heurísticas especializadas, entre ellos los de [Kurz and Askin, 2004b] tratados un año antes.

Un problema real de una empresa de procesamiento de cheques que considera la recirculación y el cálculo del retraso total ponderado fue tratado con un AG por [Bertel and Billaut, 2004]. En [Ruiz and Maroto, 2006] un AG fue empleado para minimizar el makespan en un FFS de m-etapas con máquinas paralelas no relacionadas, tiempos de inicio dependientes de la secuencia y una restricción para la selección de un subconjunto (M_j), cabe mencionar que esta propuesta fue superior a una amplia gama de heurísticas y metaheurísticas como OCH, BT, RS y procedimientos deterministas, esta propuesta también obtuvo mejores calendarizaciones que los generados manualmente por el personal de la empresa.

En [Ying and Lin, 2006] se propone una metaheurística de optimización de colonia de hormigas (OCH) para un problema de multiprocesamiento con relaciones de precedencia y mostró resultados superiores a [Oguz et al., 2004]. En [Alaykyran et al., 2007] también se propone una OCH para un FFS de m-etapas, ellos con este método produjeron mejores resultados que algunos B&B. Otras metaheurísticas hay sido utilizados, tal es el caso optimización por enjambre de partículas (PSO) propuesto por [Tseng and Liao, 2008].

Finalmente comentar que desafortunadamente, el problema de FFS aun si se restringe a su modelo clásico, todavía es imposible hacer una comparación con algoritmos ya que la mayoría de los autores realiza experimentaciones con sus propios algoritmos y sus propias instancias generadas al azar, en algunos casos instancias del mundo real muy particulares, es más no existe un conjunto de benchmarks para los cuales se puedan tener como referencia para un problema de FFS.

Capítulo 4

METODOLOGÍA DE SOLUCIÓN

La metodología seguida en esta tesis, se muestra en la figura 4.1, esta basado en alcanzar los objetivos propuestos en 5 pasos que son:

1. Modelo. Consiste en plantear el problema del FFS con tiempos de inicio dependiente de la secuencia como un modelo de programación lineal entera binaria mostrado en la sección 2.4 y como un modelo de grafo disyuntivo propuesto mostrado en la sección 2.3.

2. Método. Consiste en un método propuesto para el diseño de un algoritmo que permite encontrar soluciones óptimas acotado en tiempo polinomial, este método es mostrado en la sección 4.1, el resultado es un algoritmo de optimización combinatoria mostrado en el algoritmo 4.1 denominado AGHCGrid, es un algoritmo paralelo diseñado para correr bajo un ambiente Grid [Foster and Kesselman, 2004].

3. Plataforma. Consiste en la utilización de la plataforma de producción denominada Grid Morelos, la cual es una plataforma Grid multi-cluster de alto rendimiento mostrada en la sección 4.2.

4. Aplicación. Consiste en realizar el análisis, diseño y programación de la aplicación en C con interfaz de paso de mensajes MPI mostrado en la sección 4.3, que implementa el método propuesto en el punto 2.

5. Experimentación. Consiste en llevar a cabo la experimentación sobre esta plataforma como se muestra en el capitulo 6, consiste en la propuesta de un método de sintonización distribuida automática aplicada en paralelo denominada SDAAP mostrado en el capitulo 5 y aplicado en la sección 6.3, así mismo, la experimentación del algoritmo mediante el método SDAAP-MI mostrado en la sección 6.4.3, determinando su eficacia como se muestra en la sección 6.4 y finalmente la medición de la eficiencia de la aceleración (Speedup) del algoritmo calculando la aceleración

y su eficiencia mostrado en la sección 6.5.

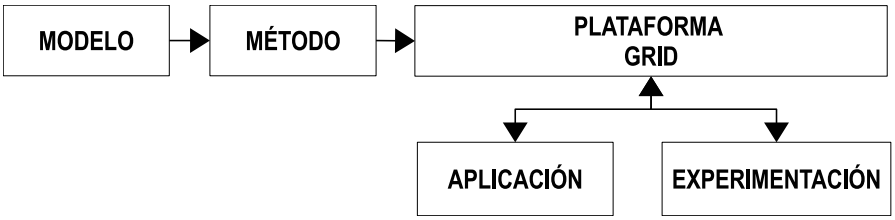

Figura 4.1. *Metodología propuesta. Consiste en partir de un modelo matemático, posteriormente desarrollar un método aplicado en paralelo que es implementado sobre la plataforma de producción Grid Morelos de alto rendimiento, mediante la programación en C con paso de mensajes MPI de la aplicación, después llevar a cabo el análisis de sensibilidad con la sintonización de los parámetros del algoritmo mediante un método de sintonización distribuida automática aplicada en paralelo (SDAAP) y finalmente realizar la experimentación para analizar su eficiencia y eficacia.*

4.1. Método.

Unos de los principales aportes de esta tesis es el método propuesto mostrado en la figura 4.2, que consiste en la hibridación tres metaheurísticas: 1) Algoritmo Genético [Araujo and Cervigón, 2009], 2) Sistema de colonia de hormigas [Dorigo et al., 2006] y 3) Recocido simulado [Martinez, 2008] en un algoritmo llamado AGHCGrid y ejecutarlo en un ambiente Grid. El algoritmo genético es un algoritmo cooperativo que combina los esfuerzos del sistema de colonia de hormigas y recocido simulado. El AG realiza la parte de la exploración utilizando selección y cruzamiento, mientras SCH y RS la explotación mediante búsquedas locales [Negenman, 2001] cooperativas e iterativas respectivamente.

Utilizando la nomenclatura de genéticos, SCH hace una mutación cooperativa utilizando variaciones de los rastros de feromona en las soluciones encontradas por las hormigas y RS una mutación iterativa al aplicar pequeñas perturbaciones aleatorias a la solución actual, esta doble mutación se espera ayude a mejorar la eficacia del algoritmo.

El algoritmo AGHCGrid propuesto no se ejecuta en solo núcleo, en lugar de ello el método considera el uso de un ambiente Grid, el cual se refiere como una Grid multicluster de cómputo intensivo denominada Grid Morelos, esta integrada por tres clusters de alto rendimiento, el primero llamado Cuexcomate localizado en el Centro de Investigación en Ingeniería y Ciencias Aplicadas de la Universidad Autónoma del Estado de Morelos, el segundo llamado Texcal localizado en la Universidad Politécnica del Estado de Morelos y el tercero llamado Nopal localizado en el Instituto Tecnológico de Veracruz.

Estos clusters están separados geográficamente pero unidos a través de I2 como se muestra en la sección 4.2, el diseño del método considera la ejecución en paralelo de

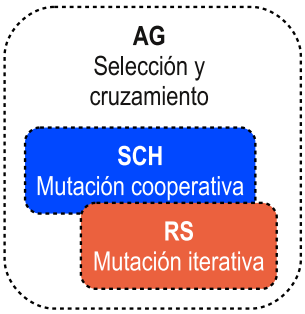

Figura 4.2. *Método de hibridación propuesto. Consiste en la hibridación tres tipos de metaheurísticas: AG, SCH y RS, además de implementar la cooperación entre ellas. El AG aplica selección y cruzamiento, SCH y RS implementa mutación cooperativa y mutación iterativa respectivamente, en donde la doble mutación se espera ayude a aumentar la eficacia del algoritmo, todo ello sobre un ambiente Grid.*

un conjunto de procesos distribuidos uniformemente sobre la Grid, con la finalidad de cooperar con los resultados encontrados haciendo uso de la interfaz de paso de mensajes MPI, como el mostrado en [Cruz et al., 2010b; 2009d], en donde se utiliza la interfaz de mensajes MPI para paralelizar el problema de distribución de agua y el problema de máquinas paralelas no relacionadas respectivamente, a partir de sus algoritmos secuenciales [Cruz et al., 2009b] para el problema de máquinas paralelas no relacionadas y [Cruz et al., 2009c;e] para el problema de distribución de agua.

El método propuesto se basa en el diseño del algoritmo 4.1, denominado AGHCGrid, el cual es mostrado en la sección 4.1.1 y explicado en la sección 4.1.2, así mismo se explica como se llevan a cabo la distribución de procesos en la sección 4.1.3, comunicación de procesos en la sección 4.1.4, sincronización de procesos en la sección 4.1.5 y cooperación de procesos en la sección 4.1.6, todo ello diseñado para ser aplicado sobre la Grid, así mismo se explica como se llevan a cabo los procesos de construcción de la población inicial, selección y cruzamiento del AG en las secciones 4.1.7, 4.1.8 y 4.1.9 respectivamente, mutación cooperativa en la sección 4.1.10 con SCH y mutación iterativa en la sección 4.1.11 con RS.

4.1.1. Algoritmo cooperativo AGHCGrid en ambiente Grid.

El algoritmo genético híbrido cooperativo en ambiente Grid 4.1, denominado AGHCGrid se centra en la hibridación que tiene como objetivo realizar búsquedas globales (AG), búsquedas locales iterativas (RS) y búsquedas locales cooperativas (SCH) sobre el espacio de soluciones utilizando la Grid Morelos como se muestra en la figura 4.3.

Esta determinación parte del hecho de que los algoritmos genéticos son muy buenos para realizar búsquedas globales pero casi siempre sufren de convergencia prematura, mientras que los algoritmos de búsqueda local como RS y SCH son muy buenos para

refinar las búsquedas pero también tienden a estancarse en óptimos locales, de tal manera que ambos métodos se complementan a la perfección y es por tal motivo que en la literatura muchos autores han tratado de mejorar sus algoritmos hibridando estas dos técnicas [González, 2011].

Nuestra propuesta a diferencia de [Figielska, 2009], que hacen una doble hibridación de las metaheurísticas de algoritmos genéticos con RS y otros que usan un algoritmo memético que incluye algún método de búsqueda local [Tavakkoli-Moghaddam et al., 2009], nosotros incluimos una segunda metaheurística de SCH adicional a RS para combinarlo con el algoritmo genético, con el objetivo principal de incrementar la refinación de una búsqueda local iterativa tradicional como RS, agregando una búsqueda local con cooperación con SCH.

La metaheurística de SCH es cooperativa y esta inspirada en el comportamiento real de las hormigas para encontrar el camino más corto entre el hormiguero y una fuente de comida, dicha cooperación esta basada en el uso de sustancias químicas referidas como feromonas que permiten a las hormigas tener una comunicación indirecta denominada stigmergy [Grassé, 1959] con el uso de sustancias químicas.

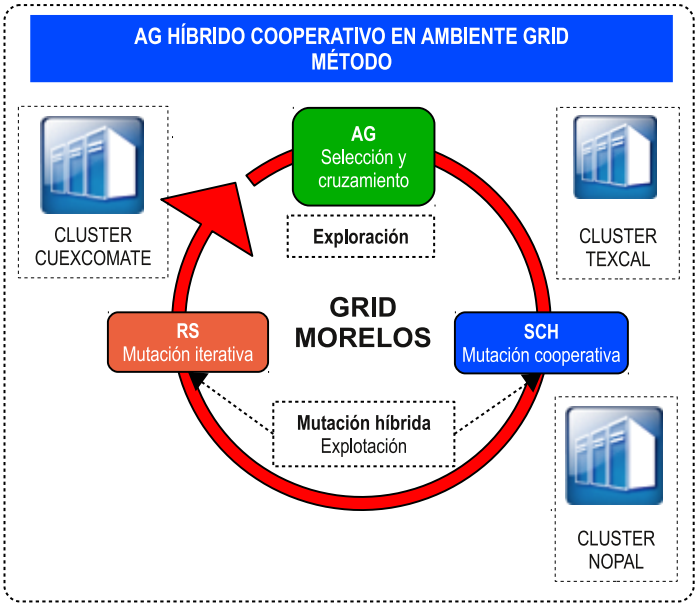

Figura 4.3. *Método AGHCGrid propuesto y su ejecución en ambiente Grid. Consiste en ejecutar el algoritmo desarrollado 4.1 integrado por tres metaheurísticas: AG, SCH y RS sobre la plataforma de producción Grid Morelos mostrado en la sección 4.2 y habilitar la cooperación entre los procesos.*

A diferencia de un método secuencial, el método propuesto ejecuta el algoritmo sobre la Grid en forma paralela, similar al utilizado en [Rodriguez-León et al., 2010; Cruz et al., 2009a], es decir que existen múltiples instancias del mismo programa corriendo sobre la Grid en un tiempo dado, similar al utilizado en [Cruz et al., 2012b; 2010a]. El esquema implementado para el control de los procesos del algoritmo es un esquema

maestro/esclavo en donde el proceso maestro asigna las tareas a los procesos esclavos como el mostrado en [Belkadi et al., 2006].

4.1.2. Interpretación del algoritmo AGHCGrid.

El algoritmo AGHCGrid es un programa que se ejecuta en múltiples instancias sobre la Grid, asigna de manera automática identificadores de manera inequívoca para cada proceso que se esta ejecutando, con este identificador se selecciona el segmento de código que debe ejecutar cada proceso dependiendo de su ámbito, es decir si el proceso es el cero entonces ejecuta el segmento de código para el maestro y si es diferente de cero entonces ejecuta el segmento de código para los procesos esclavos, el algoritmo diseñado y propuesto para esta tesis, es el algoritmo 4.1 y su explicación de detalla a continuación:

Algoritmo 4.1 AGHCGrid distribuido en ambiente Grid

1: /* NOTA: las funciones envia y recibe se bloquean hasta completarse */
2: **funcion** $AGHCGrid(proceso_{id},\ id_{max})$
3: **si** $(proceso_{id} == esclavo)$ **entonces**
4: $S_{hormiga} = \{\emptyset\}$
5: $C_{max}(S_{mejorlocal}) = 999999$ ▶ minimizar
6: **fin si**
7: **repetir**
8: **si** $(proceso_{id} == maestro)$ **entonces**
9: $recibe\,(proceso_i, poblacion[individuo_i], bloqueante)\ ,\ i = 1, 2, ..., id_{max}$
10: $poblacion = seleccion(op\_ruleta, poblacion)$
11: $poblacion = cruzamiento(tasacruce, op\_circular, poblacion)$
12: $envia(proceso_i, poblacion[individuo_i]),\ i = 1, ..., id_{max}$
13: **si no**
14: $S'_{hormiga} = SCH(h, \alpha, \beta, \gamma, \delta, p, q, UB, S_{hormiga})$ ▶ mutación cooperativa
15: $S_{metal} = S'_{hormiga} \xrightarrow{f} rs$
16: $S'_{metal} = RS(t_o, m, \mu, t_f, S_{metal})$ ▶ mutacion iterativa
17: $S_{individuo} = S'_{metal} \xrightarrow{f} individuo$
18: $S_{mejorlocal} = min\left(C_{max}(S_{mejorlocal}, S_{hormiga}, S'_{hormiga}, S'_{metal})\right) \xrightarrow{f} individuo$
19: $envia(proceso_{maestro}, S_{individuo})$
20: $recibe(proceso_{maestro}, S'_{individuo}, bloqueante)$
21: $S_{hormiga} = S'_{individuo} \xrightarrow{f} sch$
22: **fin si**
23: **hasta** (completar el total de generaciones)
24: **si** $(proceso_{id} == maestro)$ **entonces**
25: $recibe(proceso_i, poblacion[individuo_i]),\ i = 1, ..., id_{max}$
26: $S_{mejorglobal} = min(C_{max}(poblacion[individuo_i])),\ i = 1, 2, ..., id_{max}$
27: **retornar** $(S_{mejorglobal})$ ▶ mejor solución
28: **si no**
29: $envia(proceso_{maestro}, S_{mejorlocal})$
30: **fin si**
31: **fin funcion**

1. Línea 2. El algoritmo recibe como parámetros el identificador del número de proceso ($proceso_{id}$) así como el total de procesos que se están ejecutando al mismo tiempo en paralelo (id_{max}).

2. Línea 7-23. Declara un un ciclo principal de mejora continua que es igual al número de generaciones del algoritmo genético, dentro del cual están las secciones de código para el maestro y esclavos.

3. Línea 8. Si el $proceso_{id}$ corresponde al proceso maestro, entonces se llevan a cabo las siguientes acciones (línea 9-12):

   a) Línea 9. El maestro esta en espera hasta recibir las soluciones de los procesos esclavos para la construcción de la población, que esta integrado por un vector de individuos de tamaño id_{max}, donde $id_{max} = P_{max} - 1$, siendo P_{max} el número de núcleos de procesamiento seleccionados para la ejecución del algoritmo, no tomándose en cuenta el $proceso_0$ por ser el maestro, quedando id_{max} procesos esclavos.

   El vector se inicializa con la recepción de los mensajes provenientes desde el $proceso_1$ hasta el $proceso_{id_{max}}$, los cuales son almacenados en el vector población con índice desde el $individuo_1$ hasta el $individuo_{id_{max}}$ respectivamente, quedando claro la relación entre el tamaño de la población y el número de procesos esclavos, el individuo recibido es la mejor solución encontrada como resultado de aplicar búsquedas locales con SCH y RS, no se toma en cuenta el $proceso_0$ porque es el maestro.

   b) Línea 10. Construye una nueva población al aplicar selección con el operador ruleta a fin de conservar los individuos mejor adaptados de la población actual.

   c) Línea 11. Construye una nueva población al aplicar el operador de cruzamiento circular con una tasa de cruzamiento para realizar una exploración del espacio de soluciones a la población actual.

   d) Línea 12. Distribuye la población a los procesos esclavos, donde cada esclavo recibe un individuo de la población que corresponde al identificador del proceso, es decir envía el $individuo_1$ hasta el $individuo_{id_{max}}$ al $proceso_1$ hasta el $proceso_{id_{max}}$ respectivamente, en este punto los procesos esclavos están en espera de recibir la solución inicial para aplicar la mutación en la línea 20.

4. Línea 4. En caso contrario el $proceso_{id}$ corresponde a un proceso esclavo, entonces se llevan a cabo las siguientes acciones (línea *3-6 y 14*-21):

   a) Línea 3-6. Solo la primera vez que se inicia estas tareas en cada proceso, no se cuentan con una solución inicial enviada por el proceso maestro para el

sistema de colonia de hormigas ($S_{hormiga}$), debido a que en el primer ciclo es donde se producen al final las primeras soluciones, se establece $S_{hormiga} = \{\emptyset\}$ y el valor de la función objetivo $C_{max}(S_{mejorlocal}) = 999999$ a un número muy grande porque la función objetivo es de minimizar.

b) Línea 14. Calcula la solución $S'_{hormiga}$ dada por la ecuación 4.1, haciendo una llamada al algoritmo 4.2 del SCH, donde h es el número de hormigas, α es el coeficiente de importancia *alfa*, β es el coeficiente de importancia *beta*, γ es el coeficiente de evaporación *gamma* para la transición local, δ es el coeficiente de evaporación *delta* para una transición global, q es el coeficiente que divide la exploración/explotación, p es el criterio de paro del algoritmo, UB es la mejor cota superior conocida y $S_{hormiga}$ es la solución inicial con la que el sistema de colonia de hormigas inicia su búsqueda.

Si es la primera vez se inicializa en vació $S'_{hormiga} = \{\emptyset\}$ en la línea 4, de lo contrario es recibida del proceso maestro en la línea 20. La función SCH aplica una mutación cooperativa guiada por los rastros de feromona que contiene la solución inicial $S_{hormiga}$ y la mejor solución encontrada entonces es devuelta por esta función.

$$S'_{hormiga} = SCH(h, \alpha, \beta, \gamma, \delta, p, q, UB, S_{hormiga}) \tag{4.1}$$

c) Línea 15. Calcula la solución S_{metal} dada por la ecuación 4.2, aplica una función de transformación f que convierte la mejor solución encontrada por SCH ($s'_{hormiga}$) en una solución inicial para RS, por tanto S_{metal} es equivalente a $S'_{hormiga}$ ya que la función de transformación no altera la calendarización ni tampoco el valor de la función objetivo, solo realiza un cambio de contexto en la representación de la solución.

$$S_{metal} = S'_{hormiga} \xrightarrow{f} rs \tag{4.2}$$

Línea 16: Calcula la solución S'_{metal} dada por la ecuación 4.3, hace una llamada al algoritmo 4.4 de RS, donde t_o es la temperatura inicial, m es el número de ciclos que repite la longitud de la cadena de markov, μ es el coeficiente de enfriamiento, t_f es la temperatura final y S_{metal} es la solución inicial con la que RS inicia su búsqueda. La función RS aplica una mutación iterativa guiada por la solución inicial (S_{metal}) y la mejor solución encontrada entonces es devuelta por esta función.

$$S'_{metal} = RS(t_o, n, \mu, t_f, S_{metal}) \tag{4.3}$$

d) Línea 17. Calcula la solución $S_{individuo}$ dada por la ecuación 4.4, aplica una función de transformación f que convierte la mejor solución S'_{metal} encontrada por RS en una solución inicial basada en individuos, por tanto

$S_{individuo}$ es equivalente a S'_{metal}, esta transformación prepara al algoritmo para enviar la solución encontrada al proceso maestro en forma de individuo para que pueda construir la población.

$$S_{individuo} = S'_{metal} \xrightarrow{f} individuo \tag{4.4}$$

e) Línea 18. Calcula, transforma y guarda la mejor solución encontrada hasta el momento $S_{mejorlocal}$, seleccionando la que tenga un costo menor entre la actual $S_{mejorlocal}$, la recibida del proceso maestro como resultado de la selección, cruzamiento y convertida a solución inicial para SCH ($S_{hormiga}$), la encontrada por SCH ($S'_{hormiga}$) y la encontrada por RS (S'_{metal}) dada por la ecuación 4.5, finalmente aplica una función de transformación f que convierte la mejor solución local encontrada en una solución inicial basada en individuo dada por la ecuación 4.6, con el fin de para poder enviarla al proceso maestro al final del algoritmo.

$$S_{mejorlocal} = min(S_{mejorlocal}, S_{hormiga}, S'_{hormiga}, S'_{metal}) \tag{4.5}$$

$$S_{mejorlocal} = S_{mejorlocal} \xrightarrow{f} individuo \tag{4.6}$$

f) Línea 19. La mejor solución local encontrada por SCH y RS es enviada al proceso maestro, en este punto el proceso maestro esta en espera de recibir las soluciones de todos los procesos esclavos para construir la población, aplicar selección y cruzamiento en la líneas 9,10 y 11 respectivamente.

g) Línea 20. Cada uno de los procesos esclavos esta en espera hasta recibir del proceso maestro la solución inicial correspondiente $S'_{individuo}$, después de terminar de construir la población, aplicar los operadores de selección y cruzamiento, el proceso se bloquea hasta recibir el mensaje, después continúa.

h) Línea 21. Calcula la solución inicial ($S_{hormiga}$), dada por la ecuación 4.7, aplicando una función de transformación f que convierte la solución recibida del maestro ($S'_{individuo}$) en una solución inicial basada en SCH, siendo equivalente $S'_{individuo}$ y $S_{hormiga}$, esta transformación prepara al algoritmo para la siguiente generación que inicia con una mutación cooperativa con SCH.

$$S_{hormiga} = S'_{individuo} \xrightarrow{f} sch \tag{4.7}$$

5. Línea 24. Cuando termina el ciclo principal de las generaciones, si el $proceso_{id}$ corresponde al proceso maestro, entonces se llevan a cabo las siguientes acciones (líneas 25-27):

a) Línea 25. El proceso maestro recibe las mejores soluciones de cada uno de los procesos esclavos contenida en $S_{mejorlocal}$, recibiendo mensajes provenientes

desde el $proceso_1$ hasta el $proceso_{id_{max}}$, guardándolos en el vector población con índice desde el $individuo_1$ hasta el $individuo_{id_{max}}$ respectivamente.

b) Línea 26. Calcula $S_{mejorglobal}$ dada por la ecuación 4.8, con $i = 1, ..., id_{max}$ es decir la mejor solución con coste mínimo del conjunto de las mejores soluciones recibidas de los procesos esclavos.

$$S_{mejorglobal} = min(C_{max}(poblacion[individuo_i])), i = 1, ..., id_{max} \quad (4.8)$$

c) Línea 27. Finalmente retorna la mejor solución global encontrada por el algoritmo ($S_{mejorglobal}$).

6. Línea 28. En caso contrario el $proceso_{id}$ corresponde a un proceso esclavo, entonces se llevan a cabo la siguientes acción:

a) Línea 29. Cada proceso esclavo envía su mejor solución encontrada ($S_{mejorlocal}$) a lo largo de todo el ciclo de mejora, determinado por el número de generaciones.

Los detalles a cerca de la distribución, comunicación, sincronización y cooperación de procesos, además de la construcción de la población, selección, cruce y mutación son expuestos a continuación.

4.1.3. Distribución de procesos.

La distribución de procesos del algoritmo AGHCGrid se define de la siguiente manera: Dada una distribución uniforme de procesos sobre la Grid con base en la ecuación 4.26, que representa la suma de núcleos s de cada procesador r de cada nodo q de todos los clusters p que conforman la Grid, en donde a cada proceso es asignado un identificador único con base en la ecuación 4.27, que va desde *0* hasta id_{max}, debido a que la identificación de los procesos con MPI empieza en 0 (cero) y no en 1, y donde $id_{max} = (P_{max}) - 1$.

La distribución de procesos y sus identificadores se puede definir formalmente como una relación uno a uno entre cada proceso y cada núcleo ($proceso_{id} \rightarrow núcleo_{p,q,r,s}$) de la siguiente forma: $\{proceso_0 \rightarrow núcleo_{1,1,1,1}, proceso_1 \rightarrow núcleo_{1,1,1,2}, ..., proceso_{id_{max}} \rightarrow núcleo_{p,q,r,s}\}$, finalmente el $proceso_0$ es el proceso maestro y el resto de los procesos son procesos esclavos que tienen un $proceso_{id} \neq 0$.

El algoritmo AGHCGrid es un algoritmo paralelo, la distribución de procesos sobre la Grid requiere de calcular el número total de núcleos de la Grid y asignar los correspondientes identificadores únicos a cada proceso, en donde cada proceso es una instancia en ejecución del algoritmo. La idea de una distribución lo más uniforme posible en cuanto

a tiempo de término de cada proceso, radica en el uso del esquema maestro/esclavo, en donde el proceso maestro debe esperar a que todos los procesos esclavos terminen y envíen sus resultados, en este sentido el proceso maestro debe esperar por el más lento, mientras que los que ya terminaroń deben esperar también.

Es posible asignar más de un proceso por núcleo a lo que se le conoce como sobrecarga [Richard et al., 1997], en el entendido de que el tiempo de procesamiento se incrementara de manera considerable, pero si el tiempo total de término del algoritmo sigue siendo tratable entonces puede tomarse esta decisión.

Existen situaciones sobre las cuales puede requerir consideraciones en esta distribución y es cuando la diferencia entre los nodos más rápidos y los más lentos es muy marcada, tanto, que los tiempos de procesamiento para un proceso pueden ser del doble o más para una misma tarea con respecto a los más rápidos, p. ej. en la figura 4.4 a) se observa el comportamiento del tiempo de término de la ejecución de los procesos sobre recursos homogéneos, donde todos tienden a terminar igual, en la figura 4.4 b) se observa el comportamiento de recursos heterogéneos donde el $proceso_2$ asignado al $núcleo_{1,1,1,3}$ tarda en doble del $proceso_1$, en este caso es preferible asignar dos procesos al $núcleo_{1,1,1,2}$ para que ambos procesos tiendan a terminar al mismo tiempo.

Por último si los recursos más lentos repercuten seriamente en la eficiencia del algoritmo lo mejor es no utilizarlos. En esté algoritmo propuesto no fue necesario incrementar más allá de un proceso por núcleo debido a que las diferencias no fueron tan marcadas.

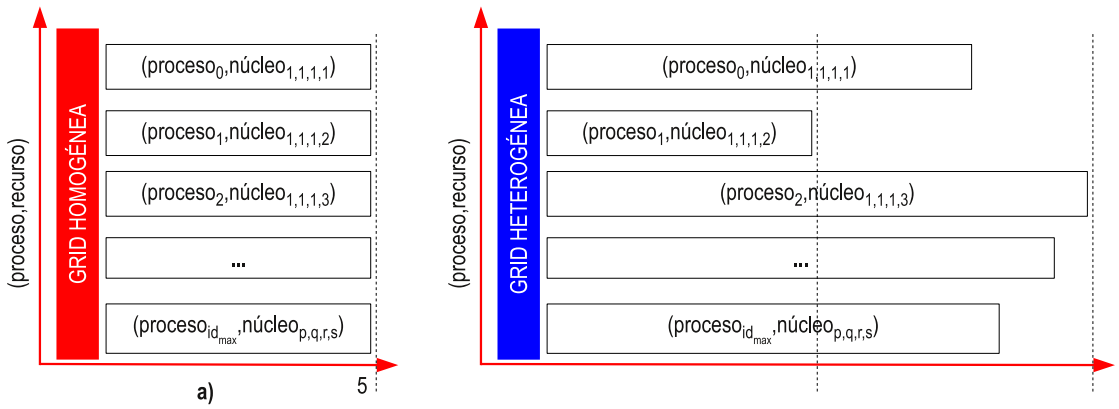

Figura 4.4. *Tiempo de término de los procesos. Dos tipos de tiempo se observan en relación al uso de plataformas homogéneas y heterogéneas, en donde las homogéneas tienden a ser uniformes en sus tiempos de término y las heterogéneas tienden a tener tiempos de espera que pueden ser cortos o largos dependiente de la discrepancia entre los recursos.*

La ubicación física del proceso maestro queda asignado al nodo maestro ubicado en el cluster CIICAp, mientras que el resto de los procesos esclavos se distribuyen en toda la Grid, esta manera de designar las ubicaciones de los procesos es la más lógica, pero no es una regla universal, ya que el nodo maestro puede estar en cualquier núcleo de cualquier nodo de la Grid, además de que puede ser cualquiera de los P_{max} procesos

con un identificador $proceso_{id} \neq 0$.

4.1.4. Comunicación de procesos.

Los procesos distribuidos por el algoritmo AGHCGrid en la sección 4.1.3 sobre la Grid, en donde un proceso de comunicación puede describirse formalmente como un conjunto de procesos $P = \{proceso_0, proceso_1, ..., proceso_{id_{max}}\}$, que se comunican entre el proceso maestro y los procesos esclavos cada determinado tiempo, en donde cada proceso es autónomo pero dependiente de la relación maestro esclavo.

Cada $proceso_{id}$ se ejecuta en un $núcleo_{p,q,r,s}$ de la Grid, donde p es el cluster, q el nodo, r el procesador y s el núcleo, de tal manera que un $proceso_{id}$ es asignado a un solo núcleo s, esto es la relación $(proceso_{id} \rightarrow núcleo_{p,q,r,s})$, para este conjunto de procesos, la comunicación se da en dos sentidos: de esclavos a maestro $(proceso_{id} \rightarrow proceso_0)$, con $id = 1, ..., id_{max}$ y de maestro a esclavos $(proceso_0 \rightarrow proceso_{id})$, con $id = 1, ..., id_{max}$.

La comunicación que utiliza el algoritmo propuesto es una comunicación punto a punto entre el proceso maestro y esclavo en ambas direcciones [Jiang et al., 2004], es decir primero el proceso maestro recibe las soluciones de los procesos esclavos en una relación de muchos a uno y después el proceso maestro devuelve nuevas soluciones a los esclavos en una relación de uno a muchos como se observa en la figura 4.5 a).

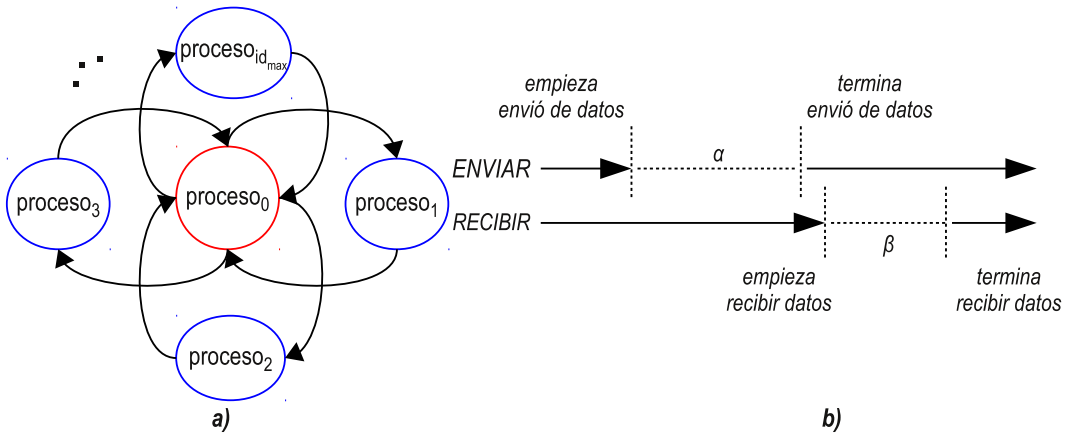

Figura 4.5. *Comunicación de procesos de algoritmo AGHCGrid. La comunicación se da en ambos sentidos, primero de los procesos esclavos hacia el maestro y después del proceso maestro a los esclavos, en cada comunicación el proceso que envía emplea un tiempo α y el que recibe se bloquea un tiempo β hasta recibir el mensaje esperado.*

El intercambio de información utiliza un mecanismo de paso de mensajes en donde uno envía y otro recibe como se observa en la figura 4.5 b) que funciona de la siguiente manera.

- El mecanismo tiene un comportamiento bloqueante, es decir, que mientras un

proceso espera recibir mensajes, se mantiene en espera (bloqueado) hasta recibirlos.

- El proceso esclavo que envía, se bloquea un tiempo α mientras deposita el mensaje en una cola (buffer) del proceso maestro continuando inmediatamente después.

- El proceso maestro recibe y se bloquea un tiempo β mientras da lectura al mensaje de su cola (buffer).

- Si el proceso que recibe da lectura al mensaje antes de que este listo, el proceso se bloquea hasta que el mensaje llegue.

- Cuando el proceso maestro recibe, debe esperar por el mensaje de cada uno de los procesos esclavos en el orden $proceso_1, proceso_2, ..., p_{id_{max}}$.

- Cuando los procesos esclavos reciben, deben esperar a que el proceso maestro reparta los resultados a los procesos anteriores en el orden $proceso_1, proceso_2, ..., p_{id_{max}}$ y esperar su turno.

Durante el proceso de enviar y recibir, solo el proceso que recibe puede bloquearse esperando indefinidamente por un mensaje, de tal forma que debemos asegurar que el número de mensajes enviados sea el mismo número que los mensajes recibidos, finalmente para evitar esta situación de bloqueo indefinido de los procesos se procede a la sincronización.

4.1.5. Sincronización de procesos.

Dada una distribución de procesos como el mostrado en la sección 4.1.3, sobre los cuales existe una comunicación como la mostrada en la sección 4.1.4, el proceso de sincronización puede describirse formalmente como un conjunto de procesos $P = \{proceso_0, proceso_1, ..., proceso_{id_{max}}\}$ que se distribuyen en la Grid, en donde un proceso p con con $p \in P$ es asignado a un solo núcleo s en una relación $(proceso_{id} \rightarrow núcleo_{p,q,r,s})$.

A su vez el tiempo de término de cada proceso se define como μ_p, en donde el tiempo total de término del proceso más lento se define como $C_{max\mu} = max(\mu_0, \mu_1, \mu_p, \mu_{id_{max}})$, por tanto el tiempo que un proceso p que ya ha terminado debe esperar con respecto al más lento es $\ell_p = C_{max\mu} - \mu_p$, finalmente el proceso de sincronización requiere de fijar un punto en todos los procesos p, incluido el maestro, en el cual todos convergen en el tiempo $t_s = \mu_p + \ell_p$, y será en ese momento cuando se inicie el proceso de envió y recepción como se muestra en la figura 4.6.

Cuando el algoritmo AGHCGrid mediante el proceso maestro, debe recolectar las primeras soluciones producidas por los procesos esclavos, se debe realizar un proceso de sincronización para asegurar que las soluciones de todos los procesos esclavos están

listos para ser enviados y el proceso maestro esta listo para recibirlos, esta sincronización responde al hecho de que no todos los procesos esclavos terminan al mismo tiempo, y en este sentido los que terminan antes deben esperar por los que aun no lo hacen.

Una consideración muy importante para plantear la sincronización, radica en el hecho de que el proceso maestro no puede esperar un tiempo muy largo debido a que, esto provocaría que el algoritmo tuviera una terminación temprana al no recibir respuesta de los procesos esclavos, este tiempo muy largo puede ser el tiempo del último proceso esclavo en terminar, si el proceso maestro decide empezar la espera de recibir a partir del momento en que los procesos empiezan a trabajar, por tanto la sincronización debe establecerse en el tiempo en el cual todos los procesos esclavos terminaron su ciclo correspondiente [Thakur et al., 2005].

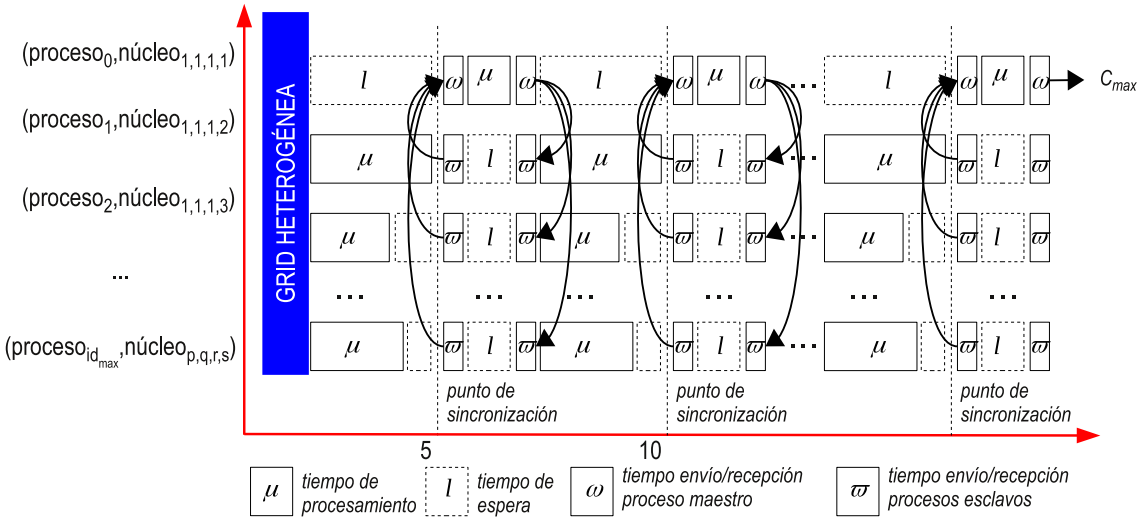

Figura 4.6. *Sincronización de procesos del algoritmo AGHCGrid. Se observa que el proceso*$_1$ *tiene el tiempo* μ_1 *más largo de término de procesamiento* $C_{max\mu} = max(\mu_0, \mu_1, \mu_p, \mu_{id_{max}})$ *con un tiempo de espera* $\ell_1 = 0$*, todos los demás procesos deben ajustar su tiempo de espera* $\ell_p = C_{max\mu} - \mu_p$ *en relación al tiempo de término más largo para lograr una sincronización uniforme al tiempo* $t_1 = \mu_1 + \ell_1 = 10$.

Una de las líneas de investigación que se escapa a esta tesis es buscar la manera de evitar los tiempos de espera largos que se da al usar nodos de procesamiento lentos y rápidos, haciendo uso de la migración de procesos o de otra técnica que permitiera un uso más uniforme de los tiempos de procesamiento [Dejan et al., 2000]. Para evitar en lo más posible este tipo de problemas, se hace uso solo de aquellos nodos que no proveen una penalización de espera muy grande y que nos da un comportamiento sub-uniforme de los tiempos de procesamiento y espera, a la hora de llevar a cabo el proceso de sincronización.

4.1.6. Cooperación de procesos.

Dada una distribución con base en la sección 4.1.3, una comunicación con base en la sección 4.1.4 y una sincronización con base en la sección 4.1.5, formalmente la cooperación de procesos se da sobre un conjunto de procesos $P = \{proceso_0, proceso_1,..., proceso_{id_{max}}\}$ que se distribuyen en la Grid, en donde un proceso p con con $p \in P$ es asignado a un solo núcleo s en una relación $(proceso_{id} \rightarrow núcleo_{p,q,r,s})$, para los cuales se definen 4 tiempos:

1. Tiempo de procesamiento real μ_p.

2. Tiempo de espera $\ell_p = max(\mu_0, \mu_1, \mu_p, \mu_{id_{max}}) - \mu_p$.

3. Tiempo de envió/recepción del nodo maestro ω_0.

4. Tiempo de envio/recepción de los procesos esclavos $\varpi_p, p \neq 0$.

La cooperación de procesos se define como el ciclo de mejora continúa en relación al número de veces que el proceso maestro interactúa con los procesos esclavos, donde en cada ciclo se espera encontrar una mejora en la calidad de la solución como se muestra en la figura 4.7.

Una vez establecido el proceso de sincronización, es posible establecer el mecanismo de cooperación utilizando paso de mensajes con MPI [Jia, 2007], consiste en que el maestro recolecte las soluciones obtenidas por todos los procesos esclavos, realize la selección y cruzamiento, y los regrese nuevamente a los nodos esclavos en un ciclo de mejora continua determinado por un criterio de paro (generaciones).

4.1.7. Construcción de la población para el algoritmo genético.

El proceso de construcción de la población para el algoritmo AGHCGrid, esta formado por individuos y puede describirse formalmente de manera similar a la cooperación de procesos descrita en la sección anterior 4.1.6, es decir, como un conjunto de procesos que se distribuyen en la Grid $P = \{proceso_0, proceso_1, ..., proceso_{id_{max}}\}$, en donde un proceso p con con $p \in P$ es asignado a un solo núcleo s en una relación $(proceso_{id} \rightarrow núcleo_{p,q,r,s})$, el $proceso_0$ almacena las soluciones encontradas desde el $proceso_1$ hasta el $proceso_{id_{max}}$ en un vector *población* de tamaño id_{max} formado por individuos, en donde las relaciones entre las soluciones de los procesos e individuos corresponden a $(proceso_1 \rightarrow individuo_1, proceso_2 \rightarrow individuo_2, ..., proceso_{id_{max}} \rightarrow individuo_{id_{max}})$, observe que el tamaño de la población es directamente igual al número de procesos distribuidos sobre la Grid.

El algoritmo utiliza una población generacional [Araujo and Cervigón, 2009], es

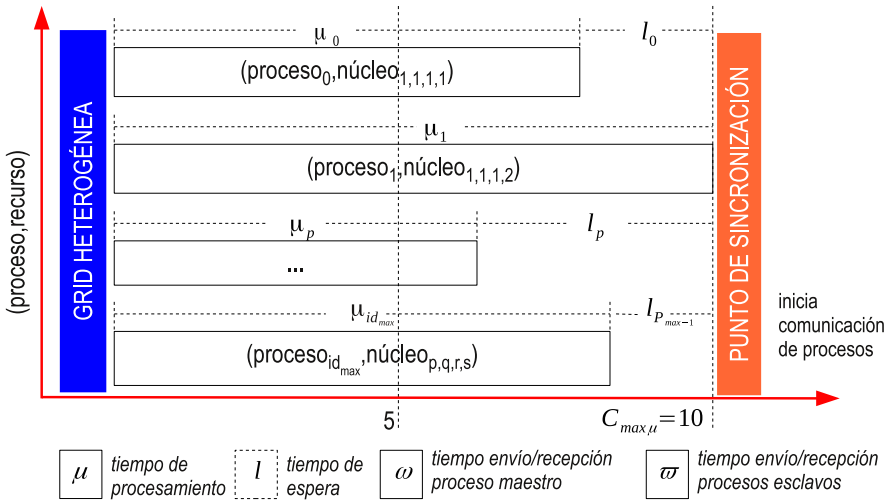

Figura 4.7. *Cooperación de procesos del algoritmo AGHCGrid. Los procesos esclavos consumen un tiempo total* $t_s = \mu_p + \ell_p$*, del cual* μ_p *es el tiempo real de procesamiento y* ℓ_p *el tiempo de espera con respecto al más lento definido como* $max(\mu_0, \mu_1, \mu_p, \mu_{id_{max}})$*, el proceso maestro solo espera el tiempo de sincronización* $t < t_s$*, la tarea de envió/recepción de soluciones consumen un mismo tiempo* $\omega_0 = \varpi_p, p \neq 0$*, cambiando solo la dirección de los mensajes de maestro a los esclavos o de los esclavos al maestro, el proceso maestro consume un tiempo de procesamiento* μ_0 *dedicado a la mejora de las soluciones recibidas mientras que los procesos esclavos esperan un tiempo igual, finalmente en el último ciclo el proceso maestro ya no distribuye las mejoras, en lugar de ello devuelve la mejor solución al método que lo ha invocado.*

decir que se sustituye por una nueva población cada generación, para poder construir una población como se muestra en la figura 4.8, el proceso maestro debe esperar a recibir las mejores soluciones encontradas por cada uno de los procesos esclavos y asignarlos al vector población de tamaño id_{max}, donde cada posición del vector corresponde a la mejor solución enviada desde cada proceso esclavo como un individuo. Observe que la implementación del algoritmo es genérico de modo que puede indicarse con índices de acceso base cero o base uno[1].

4.1.8. Selección.

El proceso de selección para una población formado por individuos basado en la sección 4.1.7, puede describirse formalmente como el conjunto de individuos perteneciente al $proceso_0$ contenidos en el vector $Poblacion = \{individuo_1, individuo_2, ..., individuo_{id_{max}}\}$, donde cada $individuo_i \in Poblacion$ con $i = 1, ..., id_{max}$, corresponde a la solución recibida del $proceso_i$, en una relación dada por $(proceso_1 \rightarrow individuo_1, proceso_2 \rightarrow individuo_2, ..., proceso_{id_{max}} \rightarrow individuo_{id_{max}})$, para el cual se lleva a cabo una selección por ruleta, en donde los individuos más aptos tiene más probabilidad de seleccionarse [Araujo and Cervigón, 2009].

[1] se refiere a la forma en que lenguajes de programación acceden al primer elemento de un vector, con índice 0 o con índice 1.

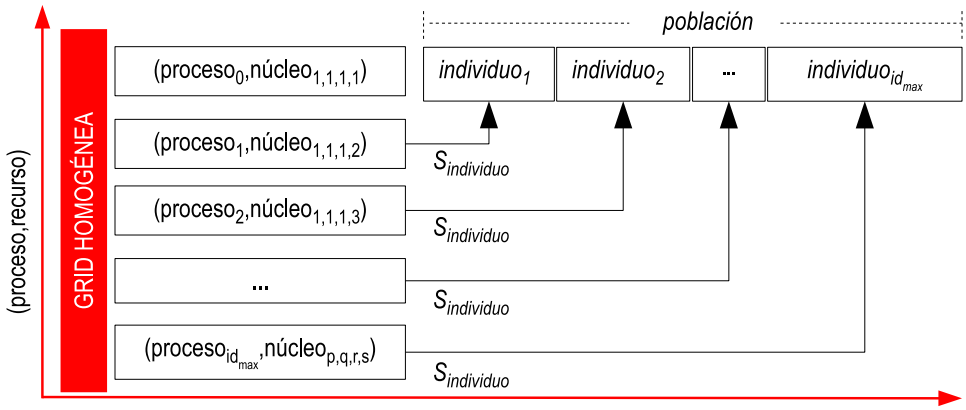

Figura 4.8. *Construcción de la población inicial. Los individuos de la población se reemplazan en cada generación, el vector de población formada por individuos es manejada por el proceso*$_0$ *maestro que se ejecuta en el nucleo*$_{1,1,1,1}$*, el proceso de llenado del vector consiste en asignar el mejor individuo encontrado por el primer proceso*$_1$ *a la primera posición del vector, el mejor individuo encontrado por el proceso*$_2$ *en la segunda posición del vector, hasta el mejor individuo encontrado en el proceso*$_{id_{max}}$ *en la última posición* id_{max} *del vector.*

En esté caso la función objetivo es de minimizar, por tanto los individuos más aptos son los que tienen menor costo, la manera de hacer que los individuos que tengan menor costo correspondan a los que tienes más probabilidades de seleccionarse es aplicar la inversa del costo, es decir que favorece a los individuos que tienen menor costo asegurando que tengan la adaptación más alta. Los pasos a realizarse son los siguientes.

1. La probabilidad de selección $prob_i$ de un $individuo_i$ es proporcional a su adaptación relativa, que se calcula con la ecuación 4.9, de la siguiente manera.

$$prob_i = \frac{C_{max}(individuo_i)^{-1}}{\sum_{j=1}^{id_{max}} C_{max}(individuo_j)^{-1}} \ , i = 1, 2, ..., id_{max} \tag{4.9}$$

   donde $C_{max}(individuo_i)^{-1}$ es la inversa del costo del makespan para el individuo i y $\sum_{j=1}^{id_{max}} C_{max}(individuo_j)^{-1}$ es la suma de la adaptación de la población, que se calcula como la suma de la inversa del costo de la función objetivo, asociada a cada individuo j de la población, es decir, la suma de la inversa del costo del makespan desde el individuo *1* hasta el individuo id_{max}

2. Después calculamos las probabilidades acumuladas para cada $individuo_i$ con la ecuación 4.10, de la siguiente forma.

$$\begin{aligned} acum_0 &= 0 \\ acum_i &= prob_1 + ... + prob_i \ , i = 1, 2, ..., id_{max} \end{aligned} \tag{4.10}$$

3. El criterio de selección, consiste en generar un número aleatorio uniformemente

distribuido sobre un intervalo [0,1] con la ecuación 4.11,

$$r = aleatorio[0, 1] \tag{4.11}$$

4. y calcular la ubicación del individuo dentro del vector de población con la ecuación 4.12, de la siguiente forma.

$$sel_i = acum_{i-1} < r < acum_i \tag{4.12}$$

Los pasos 1 y 2 solo se calculan una vez, y los pasos 3 y 4 se calculan cada vez que queramos seleccionar un individuo con base en su adaptación.

Dada la siguiente tabla 4.1 Supongamos que deseamos seleccionar 3 individuos por el método de la ruleta, para ello generamos tres números aleatorios con base en la ecuación 4.11.

$$r_1 = 0.8751$$
$$r_2 = 0.1391$$
$$r_3 = 0.2582$$

Con base en el método de la ruleta, los individuos seleccionados son: $individuo_7$, $individuo_1$ e $individuo_3$, ya que se cumple las condiciones de 4.12:

$$(acum_6 < r_1(0.8751) < prob_7) \rightarrow individuo_7$$
$$(acum_0 < r_2(0.1391) < prob_7) \rightarrow individuo_1$$
$$(acum_2 < r_3(0.2582) < prob_7) \rightarrow individuo_3$$

$Individuo_i$	1	2	3	4	5	6	7	8
Adaptación C_{max}	4	1	1	2	3	2	5	2
$prob_i$	0.2	0.05	0.05	0.1	0.15	0.1	0.25	0.1
$acum_i$	0.2	0.25	0.3	0.4	0.55	0.65	0.9	1.0

Tabla 4.1. *Selección por ruleta. La adaptación es el costo de la función objetivo (C_{max}), las probabilidades ($prob_i$) se calculan con base en la ecución 4.9 y los acumulado de las probabilidades ($acum_i$) se calculan con base en la ecuación 4.12.*

4.1.9. Cruzamiento.

Una vez terminado el proceso de selección mostrado en la sección anterior 4.1.8, a los individuos resultantes se les aplica un proceso de cruce anular mostrado en la figura

4.9, que consiste en intercambiar los cromosomas de dos padres para producir dos hijos [Pavez et al., 2009], de tal manera que se construya una nueva población para llevar a cabo la exploración del espacio de soluciones.

El proceso de cruzamiento mediante el operador de ruleta y una $tasacruce \in (0, 1]$ para una población formado por individuos $Poblacion = \{individuo_1, individuo_2, ..., individuo_{id_{max}}\}$, se da tomando dos individuos padres $(padre_i, padre_{i+1})$ con $i = 1, 3, ..., id_{max}$ que producen al aplicar un cruce circular de los cromosomas c, dos individuos hijos $(hijo_i, hijo_{i+1})$.

Tomando como base una representación circular de los individuos, se define un punto de cruce i_c y un tamaño de cruce i_l, a el punto determinado de forma aleatoria $i_c = aleatorio[individuo_1, individuo_{id_{max}}]$ y a la longitud del intercambio determinado de forma aleatoria $i_l = aleatorio[1, \frac{L-1}{2}]$ respectivamente, donde L es la longitud del individuo. El cruce se realiza intercambiando los cromosomas de los padres a partir del punto i_c e intercambiando los siguientes i_l cromosomas c para producir dos hijos $(padre_i[c_{i_c}, c_{i_c+i_l}], padre_{i+1}[c_{i_c}, c_{i_c+i_l}]) \rightarrow (hijo_i, hijo_{i+1})$ con $i = 1, 3, ..., id_{max}$, siempre y cuando al generar un número aleatorio r con base en la sección 4.11, este comprendido en el rango de la tasa de cruce $0 < r \leq tasacruce$.

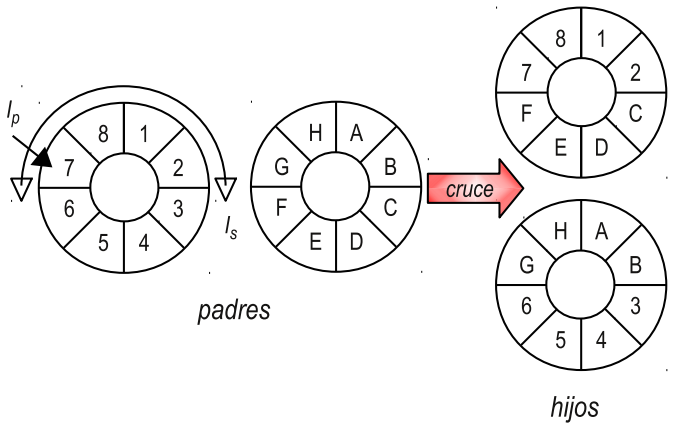

Figura 4.9. *Cruce circular. Dado dos padres que representan soluciones, en donde el* $padre_1 = \{1, 2, 3, 4, 5, 6, 7, 8\}$, *el* $padre_2 = \{A, B, C, D, E, F, G, H\}$, *con una longitud en cromosomas de los padres de 8, se calcula el punto de cruce* $i_c = 7$ *y una longitud de cruce* $i_l = 4$, *resultando dos hijos donde el* $hijo_1 = \{1, 2, C, D, E, F, 7, 8\}$ *y el* $hijo_2 = \{A, B, 3, 4, 5, 6, G, H\}$.

Una características que presenta este cruce, es que conserva el orden de la secuencia, desde la etapa *1* hasta la etapa n, de tal forma que un cruce circular en un solo punto, solo intercambia el orden de los trabajos y no el orden de las etapas, por lo que un intercambio aleatorio causaría soluciones infactibles.

4.1.10. Mutación cooperativa con sistema de colonia de hormigas.

Después de aplicar la selección mostrada en la sección 4.1.8 y cruzamiento mostrado en la sección 4.1.9, donde cada proceso esclavo recibe del proceso maestro una nueva solución en forma de individuo como se indica en la sección 4.1.6, se aplica una función de transformación f que convierte la solución recibida del maestro ($S'_{individuo}$) en una solución inicial para la metaheurística SCH como se muestra en la ecuación 4.7.

Esta transformación solo cambia la representación de la solución a una estructura de datos diferente, de tal manera que la calendarización y el costo de la función objetivo es el mismo, la solución representa solo un camino de todos los posibles que se encuentran en el grafo dirigido *G=(V,A)* construido a partir del grafo disyuntivo $G = (O, C, D)$ mostrado en la sección 2.3, que representa un problema de calendarización, donde V representa el conjunto de vértices (nodos) de las operaciones y A es el conjunto de aristas (arcos) que conectan estos vértices, donde cada arista$(r, s) \rightarrow a_{r,s}$ esta formado por un par de vértices que representan operaciones $(O_{origen}, O_{destino})$ tal que $r, s \in V$ y $r \neq s$, y donde cada operación (O_{ijk}) asocia los tres elementos máquina (i), trabajo(j) y etapa(k) respectivamente.

La figura 4.10, muestra un grafo que representa todas las aristas posibles por las que una hormiga k puede transitar $(a^k_{r,s})$ al momento de construir soluciones, el número de aristas (arcos) de este problema son {6,60,60,30} para las etapas inicial, 1, 2, 3 y final respectivamente, las cuales son numeradas desde 0 hasta n, donde n=156 que es el número total de aristas del grafo. Para que una hormiga k posicionada en el vértice r, seleccione el siguiente vértice s, en donde el siguiente vértice s de la hormiga k se define como el vecindario de r, se tiene que $s = N_k(r)$.

Una solución de SCH esta compuesta por un conjunto de vértices, donde cada vértice representa una operación (O_{ijk}), que representa la secuencia de transiciones entre las operaciones del grafo dirigido G(V,A), esto es $S_{hormiga} = \{v_{inicial} \rightarrow v_1, v_1 \rightarrow v_2, ..., v_n \rightarrow v_{final}\}$, en donde cada par de vértices *(r,s)* unidos producen una arista $(a_{r,s})$ como sigue: $(v_r, v_s) \rightarrow a_{r,s}$ sobre la que se asocia la información de los rastros de feromona (τ) y costo (η) de transitar por esa arista, por lo que la solución se representa por aristas $S_{hormiga} = \{a_{v_{inicial},v_1}, a_{v_1,v_2}, ..., a_{v_n,v_{final}}\}$, donde $v_{inicial}$ y v_{final} representan el punto de entrada y salida de los trabajos respectivamente.

Finalmente cada arista puede ser diferenciada de forma inequívoca con un identificador, lo que nos daría la posibilidad de tener una solución compacta $S_{hormiga} = \{id_1, id_2, ..., id_n\}$, donde cada id une el total de vértices u operaciones que se calculan con la ecuación 4.13, donde *inicial* y *final* son los vértices de entrada y salida respectivamente:

$$v_{max} = inicial + (m \centerdot n \centerdot k) + final \tag{4.13}$$

de los cuales, cada trabajo solo requiere seleccionar una de las m máquinas disponibles, por tanto una solución válida solo contendrá un número de vértices $inicial + (m =$

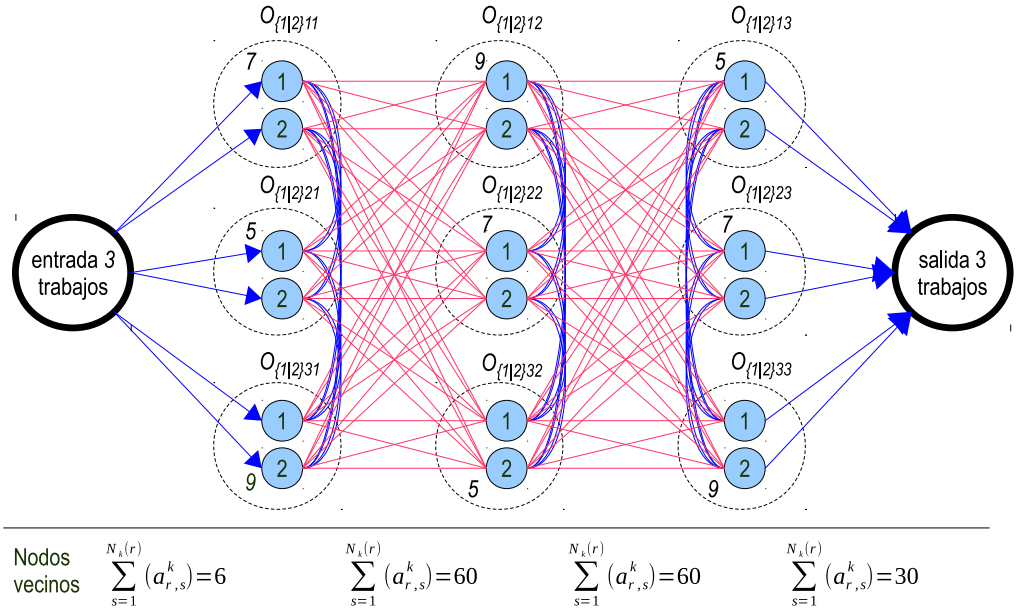

Figura 4.10. *Grafo G=(V,A) para SCH. Dado un problema cualquiera representado por un grafo disyuntivo G=(O,C,D), en este caso ejemplificamos un FFS-SDST integrado por 3 trabajos* $N = \{1, 2, 3\}$ *que pasan a través de un conjunto de 3 etapas en serie* $M = \{1, 2, 3\}$ *en donde cada etapa* k *con* $k \in M$ *esta compuesto de 2 máquinas paralelas idénticas* $m_k = \{1, 2\}$ *para procesar cualquiera de los* j *trabajos con* $j \in N$ *en cualquiera de las máquinas* i *con* $i \in m_k$.

$1 \,.\, n \,.\, k) + final$, seleccionado a partir del total de posibles operaciones dado por la ecuación anterior.

4.1.10.1. Construcción de soluciones factibles con SCH.

En SCH las hormigas artificiales construyen caminos recorriendo desde el inicio hasta el final el grafo dirigido G=(V,A) sin violar las restricciones, moviéndose de una operación r a otra operación s a través de las arista $a_{r,s}$ que tiene asociada una cierta cantidad de feromona $(\tau_{r,s})$ y el costo $(\eta_{r,s})$, en donde el costo esta compuesto por el tiempo de procesamiento (p_{ijk}) y el tiempo de inicio dependiente de la secuencia (S_{ljk}).

Cada vez que se mueve por una arista $a_{r,s}$, evapora feromona y guarda el camino como una solución parcial S hasta completarla, finalmente cuando todas las hormigas han terminado de construir sus caminos, se deposita feromona sobre los arcos de la mejor solución, de modo que las siguientes hormigas en transitar por el grafo se guíen por las nuevas cantidades de feromona que primero fueron evaporadas y luego depositadas, teniendo preferencia por aquellas aristas con mayor concentración de feromona.

La idea de una evaporación por donde las hormigas van pasando se contrapone al hecho natural de las hormigas de depositar, pero constituye la idea principal de que las

hormigas exploren nuevos caminos por donde existe menos cantidad de feromona, así mismo las hormigas también pueden optar por la construcción de caminos vecinos a la mejor solución reafirmando la explotación.

El algoritmo 4.2, que se muestra a continuación, construye soluciones factibles por las hormigas como se indica en [Dorigo et al., 2006; Maria and Dorigo, 1996], este algoritmo es una adaptación mejorada con reinicio dado por $S_{hormiga}$, en donde la llamada a esta función, esta integrada dentro del algoritmo propuesto AGHCGrid 4.1, la cual es llamada en la línea 14, este algoritmo aplica una búsqueda local cooperativa con reinicio basada en la solución inicial $S_{hormiga}$, que se explica a continuación:

Algoritmo 4.2 Búsqueda local cooperativa SCH.

1: **funcion** SCH$(h, \alpha, \beta, \gamma, \delta, p, q, UB, S_{hormiga})$
2: /* inicializa */
3: $\tau_0 = (n \cdot m \cdot k \cdot UB)^{-1}$ ▶ calcula feromona inicial
4: $\tau_{r,s} = \tau_0, \ \forall a_{r,s} \in G(V, A)$ ▶ inicializa feromona en todas las aristas
5: $\tau_{r,s} = [C_{max}(S_{hormiga})]^{-1}, \ \forall a_{r,s} \in S_{hormiga}$ ▶ feromona solución inicial
6: $C_{max}(S_{mejorlocal}) = 999999$ ▶ minimizar
7: /* ciclo principal */
8: **repetir**
9: **para** $\forall(hormiga_k), \ con \ k = 1, 2, ..., h$ **hacer** ▶ construye solución y evapora
10: $S[hormiga_k] = Construye\_Solucion(\alpha, \beta, \gamma, q, hormiga_k)$
11: **fin para**
12: /* calcula mejor solución local */
13: $S_{mejorlocal} = min(C_{max}(S_{mejorlocal}), C_{max}(S[hormiga_k])), \ k = 1, 2, ..., h$
14: /* deposita feromona sobre los arcos de la mejor solución */
15: **para** $\forall(hormiga_k), \ con \ k = 1, ..., h$ **hacer**
16: **para** $\forall(arista(r, s)) \in S[hormiga_k]$ **hacer**
17: $$\tau_{r,s} = \begin{cases} (1-\delta) \cdot \tau_{r,s} + \delta \cdot \nabla\tau_{r,s} & si \ a_{r,s} \in S_{mejorlocal} \\ \tau_{r,s} & en \ otro \ caso \end{cases}$$
18: **fin para**
19: **fin para**
20: **hasta** (completar el criterio de paro p)
21: **retornar** $(S_{mejorlocal})$ ▶ mejor solución
22: **fin funcion**

1. Línea 1. El algoritmo recibe como parámetros $(h, \alpha, \beta, \gamma, \delta, p, q, UB, S_{hormiga})$ donde h es el número de hormigas, α es el coeficiente de importancia *alfa*, β es el coeficiente de importancia *beta*, γ es el coeficiente de evaporación *gamma* para la transición local, δ es el coeficiente de evaporación *delta* para una transición global, p es el criterio de paro, q es el coeficiente que divide la exploración/explotación, UB es la mejor cota superior conocida y $S_{hormiga}$ es la solución inicial con la que el sistema de colonia de hormigas inicia su búsqueda, en donde si es la primera vez $S_{hormiga} = \{\emptyset\}$.

2. Línea 3. La fase de inicialización consiste en calcular el monto de feromona mínimo τ_0 que contendrá cada una de las aristas representadas en el grafo G(V,A) de la figura 4.10, que se determina con la siguiente ecuación 4.14, donde n es el número

de trabajos, m el número de máquinas por etapa, k el número de etapas y UB la mejor cota superior conocida del problema a resolver.

$$\tau_0 = (n \centerdot m \centerdot k \centerdot UB)^{-1} \tag{4.14}$$

3. Línea 4. Asignamos el monto de feromona calculado (τ_0) a cada una de las aristas del grafo G=(V,A) con la siguiente ecuación 4.15.

$$\tau_{r,s} = \tau_0, \ \ \forall a_{r,s} \in G(V, A) \tag{4.15}$$

4. Línea 5. Se deposita feromona sobre las aristas de la solución inicial recibida ($S_{hormiga}$), que se calcula como la inversa de la calidad de la solución obtenida a través de evaluar la función objetivo $C_{max}(S_{hormiga})$, si es la primera vez el valor de $S_{hormiga} = \{\emptyset\}$ y por tanto no se deposita feromona, de lo contrario aplica la ecuación 4.16.

$$\tau_{r,s} = [C_{max}(S_{hormiga})]^{-1}, \ \ \forall a_{r,s} \in S_{hormiga} \tag{4.16}$$

5. Línea 6. Finalmente se inicializa la variable destinada a contener la mejor solución encontrada $C_{max}(S_{mejorlocal}) = 999999$ a un número muy alto debido a que la función objetivo es minimizar, haciendo notar que $S_{hormiga}$ se refiere al camino de la hormiga y $C_{max}(S_{mejorlocal})$ a la evaluación de la función objetivo que consiste en calcular el makespan(C_{max}).

6. Línea 8-20. Contiene el ciclo de la búsqueda local que consiste en la generación de soluciones para las k hormigas, evaporación de feromona paso a paso, actualización de la mejor solución y adición de feromona sobre la mejor solución, en un ciclo determinado por el número de iteraciones p.

   a) Línea 9-11. Genera las soluciones S para cada una de las $hormigas_k$ y las guarda en un vector $S[hormiga_k]$ *con* k=1,2,...,h, a partir de la llamada a $Construye\_Solucion()$ con los parámetros $\alpha, \beta, \gamma, q, hormiga_k$. La construcción de las soluciones se realiza haciendo una llamada a la función que aplica el algoritmo 4.3, usando una regla de transición proporcional pseudoaleatoria.

   b) Línea 13. En cada iteración se calcula la mejor solución que tenga el menor costo evaluando la función objetivo y asignando el camino de la mejor hormiga, de entre la mejor solución encontrada hasta el momento ($S_{mejorlocal}$) y las soluciones S generadas por cada una de las hormigas en el vector $S[hormiga_k]$ *con* k=*1,2,...,*h con base en la ecuación 4.17

$$S_{mejorlocal} = min(C_{max}(S_{mejorlocal}), C_{max}(S[hormiga_k])), \ \ k = 1, 2, ..., h \tag{4.17}$$

   a) Línea 15-19. Recorre las soluciones del vector $S[hormiga_k]$ construidas por cada una de las $hormigas_k$ con k=1,2,...,h, evapora y deposita feromona solo

en aquellos aristas que pertenecen a la mejor solución encontrada hasta el momento ($S_{mejorlocal}$), donde $\delta \in (0, 1]$ es un coeficiente de evaporación, se aplica la siguiente ecuación 4.18.

$$\tau_{r,s} = \begin{cases} (1-\delta) \cdot \tau_{r,s} + \delta \cdot \nabla\tau_{r,s} & si\ a_{r,s} \in S_{mejorlocal} \\ \tau_{r,s} & en\ otro\ caso \end{cases} \tag{4.18}$$

donde el incremento de feromona es la inversa del costo de la función objetivo de la mejor solución dado por la ecuación 4.19

$$\nabla\tau_{r,s} = C_{max}(S_{mejorlocal})^{-1} \tag{4.19}$$

7. Línea 21. Una vez terminado el ciclo principal se retorna la mejor solución encontrada $S_{mejorlocal}$.

La llamada a la función *Construye_Solucion*() desde el algoritmo 4.2, aplica el algoritmo 4.3, para la construcción de soluciones por cada hormiga k se muestra a continuación:

Algoritmo 4.3 Búsqueda local cooperativa SCH - Construye_Solución().

1: **funcion** $Construye\_Solucion(\alpha, \beta, \gamma, q, k)$
2: /* inicializa */
3: $r = entrada$ ▶ posiciona hormiga k en el vértice inicial
4: $S_k = \{r\}$ ▶ agrega el vértice inicial r a la solución de la hormiga k
5: **repetir**
6: /* selecciona siguiente vertice de la hormiga y lo agrega a la solución */
7: $r = aleatorio[0, 1]$ ▶ número aleatorio que decide que caso aplicar
8:
$$sig_{r,s}^{k} = \begin{cases} maximo_{s \in S_k(r)} \left[(\tau_{r,s})^{\alpha} \cdot (\eta_{r,s})^{\beta}\right] & si\ s \in S_k(r) \wedge r \leq q \wedge s \notin tabu_k \\ ruleta\left(\dfrac{(\tau_{r,s})^{\alpha} \cdot (\eta_{r,s})^{\beta}}{\sum_{u \in S_k(r)} (\tau_{r,u})^{\alpha} \cdot (\eta_{r,u})^{\beta}}\right) & si\ s \in S_k(r) \wedge r > q \wedge s \notin tabu_k \\ 0 & en\ otro\ caso \end{cases}$$
9: $S_k = S_k + sig_s$ ▶ agrega vértice seleccionado a la solución
10: /* evapora feromona paso a paso y avanza al vértice seleccionado */
11: $\tau_{r,s} = (1-\gamma) \cdot \tau_{r,s} + \gamma \cdot \tau_0$
12: $r = s$
13: **hasta** $(r = salida)$
14: **retornar** (S_k) ▶ solución construida por la hormiga k
15: **fin funcion**

La construcción de soluciones se realiza paso a paso, en donde cada paso usa una regla de transición proporcional pseudoaleatoria para seleccionar el siguiente vértice, una hormiga empieza su recorrido en el vértice inicial, recorre todo el grafo G=(V,A) hasta llegar el vértice de salida, el algoritmo 4.3 se explica a continuación:

1. Línea 1. El algoritmo recibe como parámetros $\alpha, \beta, \gamma, q, k$ descritos anteriormente.

2. Línea 3. Posiciona a la hormiga k en el vértice inicial.

3. Línea 4. Inicializa la solución S de la hormiga k (S_k) agregando el primer vértice r visitado.

4. Línea 5-13. Ciclo principal que recorre todo el grafo G=(V,A) por la hormiga k avanzando paso a paso hasta llegar al vértice final.

   a) Línea 7. Genera un número $r \in [0, 1]$ aleatorio uniformemente distribuido entre 0 y 1.

   b) Línea 8. Se utiliza una lista tabú [Wardono and Fathi, 2004] para seleccionar un conjunto de aristas $a_{r,s}$ del vecindario $s \in N_k(r)$ de aristas s que son alcanzables por una hormiga k posicionada en el nodo r, siempre y cuando la arista s no haya sido visitada anteriormente ($s \notin tabu_k$) por la hormiga k, para este conjunto de aristas $a_{r,s}$ que cumplen estas dos condiciones, se lleva a cabo una tercera comparación entre r y q, donde $q \in [0, 1]$ es un factor que determina cuando se aplica una explotación o una exploración.

   Si $r \leq q$, las hormigas aplican una explotación del conocimiento obtenido en términos de la cantidad de feromona $(\tau_{r,s})$ y costos de transición $(\eta_{r,s})$, es decir, se selecciona la siguiente mejor arista entre la mayor cantidad de feromona $(\tau_{r,s})$ y un menor costo de transición $(\eta_{r,s})$, que esta dada por la función $maximo\ \left[(\tau_{r,s})^{\alpha} \cdot (\eta_{r,s})^{\beta}\right]$, en donde $\alpha, \beta \in [0, 1]$ representan el factor de importancia para la información de feromona y costo respectivamente.

   Si $r > q$ se calcula la probabilidad de seleccionar la siguiente arista $a_{r,s}$, que es es proporcional a su adaptación relativa determinada por $\dfrac{(\tau_{r,s})^{\alpha} \cdot (\eta_{r,s})^{\beta}}{\sum_{u \in S_k(r)} (\tau_{r,u})^{\alpha} \cdot (\eta_{r,u})^{\beta}}$, a la cual se aplica una selección por ruleta, por tanto la ecuación 4.20, es la usada para para seleccionar el siguiente vértice s por una hormiga k posicionada en el vértice r.

$$sig^{k}_{r,s} = \begin{cases} maximo_{s \in S_k(r)} \left[(\tau_{r,s})^{\alpha} \cdot (\eta_{r,s})^{\beta}\right] & si\ s \in S_k(r) \wedge r \leq q \wedge s \notin tabu_k \\ ruleta\left(\dfrac{(\tau_{r,s})^{\alpha} \cdot (\eta_{r,s})^{\beta}}{\sum_{u \in S_k(r)} (\tau_{r,u})^{\alpha} \cdot (\eta_{r,u})^{\beta}}\right) & si\ s \in S_k(r) \wedge r > q \wedge s \notin tabu_k \\ 0 & en\ otro\ caso \end{cases} \tag{4.20}$$

5. Línea 9. Agrega el vértice s seleccionado al vértice actual r de la hormiga k contenida en la solución S_k.

6. Línea 11. Al avanzar del vértice r en vértice s, sobre la arista $a_{r,s}$, se disipa feromona con el fin de hacerla menos atractiva para las hormigas siguientes, dándole oportunidad de seleccionar otras aristas aun no exploradas, se aplica la ecuación 4.21, donde γ es un coeficiente de evaporación de feromona paso a paso.

$$\tau_{r,s} = (1-\gamma) \cdot \tau_{r,s} + \gamma \cdot \tau_0 \tag{4.21}$$

7. Línea 12. La hormiga avanza al vértice s seleccionado, repitiéndose los pasos de las líneas 7,8 hasta completar el ciclo dado por p.

8. Línea 14. Al terminar el ciclo de construcción de la solución por parte de la hormiga k, retorna la solución de la hormiga k (S_k).

4.1.11. Mutación iterativa con recocido simulado.

Después de aplicar una mutación cooperativa con SCH en la sección 4.1.10, en donde la mejor solución encontrada ($S_{mejorlocal}$) es retornada al algoritmo AGHCGrid (algoritmo 4.1) línea 14, que la recibe en $S'_{hormiga}$, después aplica una función de transformación f que convierte la solución basada en aristas para SCH en una solución inicial para RS (S_{metal}) aplicando la ecuación 4.2, esta transformación solo cambia la representación de la solución a una estructura de datos diferente, de tal manera que la calendarización y el costo de la función objetivo es el mismo.

Una solución inicial para RS es $S_{metal} = \{O_1, O_2, ..., O_{n \cdot m \cdot k}\}$ esta compuesta por un conjunto de operaciones O en forma de triadas (i, j, k) de un grafo G=(V,A) como el mostrado en la figura 4.14, que representa el orden en el cual deben ser procesadas las operaciones, por tanto una solución S esta representada por una serie de operaciones en forma de triadas igual al número de operaciones que tiene el grafo disyuntivo que representa el problema, que en general es calculado como $n \cdot m \cdot k$, en donde una sucesión de k triadas representa una etapa y en el cual el orden de los trabajos i y máquinas j varían a lo largo de las etapas k indicando el orden el la cual son procesados los trabajos j sobre las máquinas i de cada etapa k.

4.1.11.1. Construcción de soluciones factibles con RS.

A diferencia de SCH, RS aplica una búsqueda local iterativa introducida por [Kirkpatrick et al., 1983], el cual consiste en un método probabilístico de búsqueda local iterada, en donde cada iteración hace uso de un ciclo de metrópolis que involucra la función de probabilidad de Boltzmann dada por la ecuación 4.24, que puede aceptar soluciones que no mejoren la calidad de la solución, pero que le permiten escapar de los óptimos locales, al involucrar la temperatura T y la diferencia de temperatura entre la solución actual y nueva a partir de la ecuación 4.25.

El nombre del método proviene de su similitud con el proceso metalúrgico de recocido lento de metales [Alem et al., 2009] con el que se consigue un sólido de mínima entropía, en donde RS sigue un esquema similar para ayudar a resolver problemas de optimización, el cual inicia con la mejor solución ($S_{actual} = S_{metal}$) encontrada por SCH y recibida como parámetro de entrada en RS, la metaheurística de RS inicia con una temperatura T alta que influye en la aceptación de casi todas las nuevas soluciones (S_{nuevo}) generadas a partir del vecindario $N(S)$, después T gradualmente disminuye al ser multiplicado por el coeficiente μ dado por la ecuación 4.22.

$$T = T \cdot \mu \tag{4.22}$$

Esto hace que la aceptación de movimientos sea cada vez más y más selectiva a medida que la temperatura desciende, finalmente solo acepta aquellos movimientos que mejoren la solución S_{actual}, un movimiento, mutación o perturbación aplicado a partir de un intercambio aleatorio de un par de operaciones (i, j, k) de S_{actual} genera un nuevo vecino llamado S_{nuevo} que esta dado por la ecuación 4.23.

$$S_{nuevo} = N(S) \,|\, \{S_{nuevo} \in S : S_{actual} \xrightarrow{\sigma} S_{nuevo}\} \tag{4.23}$$

Donde S_{nuevo} es parte del espacio de soluciones de S, es decir un vecino $N(S)$ que se obtiene al realizar una perturbación σ a S_{actual} para obtener una nueva solución S_{nuevo}, en donde la aceptación de las nuevas soluciones se da solo si $C_{max}(S_{actual}) \geq C_{max}(S_{nuevo})$ o si para un número aleatorio uniformemente distribuido $r \in [0, 1]$ que involucra la función de Boltzmann [Figielska, 2009] cumple la siguiente condición dada por la ecuación 4.24.

$$r \leq e^{\frac{-S_{dif}}{T}} \tag{4.24}$$

Donde para temperaturas T altas hay mayores probabilidades de aceptación y donde S_{dif} es la diferencia del incremento de temperatura dada por la ecuación 4.25.

$$S_{dif} = C_{max}(S_{nuevo}) - C_{max}(S_{actual}) \tag{4.25}$$

El criterio de aceptación se da dentro del ciclo de metrópolis el cual ejecuta un número de repeticiones igual a la longitud de la cadena de markov [Figielska, 2009], es decir un número de movimientos con los cuales se cubre el total de combinaciones para la estructura de vecindad usada, que para esté caso es igual al total de operaciones requeridas para procesar todos los trabajos en todas las etapas, donde cada trabajo solo requiere una sola máquina, el tamaño de la vecindad esta dado por $n \cdot m \cdot ((n \cdot m) - 1)$, el ciclo externo que hace uso de metrópolis es el que controla la temperatura. El algoritmo 4.4 de RS aplicado a el método se muestra a continuación.

La llamada a $RS(t_o, n, \mu, t_f, S_{metal})$ como función integrada del algoritmo AGHC-Grid 4.1 en la línea 16 se explica a continuación:

1. Línea 1. El algoritmo recibe como parámetros $(t_o, m, \mu, t_f, S_{metal})$ donde t_o es la temperatura inicial, m es el número de ciclos que repite la longitud de la cadena de

Algoritmo 4.4 Búsqueda local iterativa RS.

```
1: funcion RS(t_o, m, μ, t_f, S_metal)
2:     T = t_0                                   ▶ temperatura inicial
3:     C_max(S_mejorlocal) = 999999              ▶ minimizar
4:     S_actual = S_metal                        ▶ inicia con la mejor solución encontrada por SCH
5:     repetir
6:         para i, i = 1, 2, ..., m hacer        ▶ número de veces la cadena de MARKOV
7:             S_nuevo = {S_nuevo ∈ S : S_actual --σ--> S_nuevo}   ▶ genera vecino N(S)
8:             si (C_max(S_nuevo) ≤ C_max(S_actual)) entonces
9:                 S_actual = S_nuevo
10:                si (C_max(S_actual) ≤ C_max(S_mejorlocal)) entonces
11:                    S_mejorlocal = S_actual
12:                fin si
13:            si no
14:                S_dif = C_max(S_nuevo) − C_max(S_actual)   ▶ incremento de temperatura
15:                r = aleatorio[0, 1]
16:                si (r ≤ e^(−S_dif / T)) entonces
17:                    S_actual = S_nuevo
18:                fin si
19:            fin si
20:        fin para
21:        T = T . μ                              ▶ decrementa temperatura
22:    hasta (T ≤ t_f)                            ▶ hasta llegar a la tempratura minima
23:    retornar (S_mejorlocal)                    ▶ mejor solución
24: fin funcion
```

markov, μ es el coeficiente de enfriamiento, t_f es la temperatura final y S_{metal} es la solución inicial con la que RS inicia su búsqueda que fue encontrada por SCH.

2. Línea 2. Se inicia la temperatura T a una temperatura t_o que será disminuida conforme avanza la búsqueda.

3. Línea 3. Se inicia la solución $S_{actual} = S_{metal}$, donde S_{metal} fue la mejor solución encontrada por SCH.

4. Línea 4. Se establece el valor de la función objetivo $C_{max}(S_{mejorlocal}) = 999999$ para la solución $S_{mejorlocal}$ que contendrá la mejor solución de la búsqueda local iterada a un número muy grande debido a que el objetivo es de minimizar.

5. Línea 5-22. Constituye el ciclo iterativo externo que implementa un algoritmo de metrópolis interno de la búsqueda local, este ciclo externo es el que controla la temperatura, que disminuye al final de cada ciclo de metrópolis en la línea 21 y termina hasta que la temperatura T sea menor o igual a la temperatura final t_f en la línea 23.

6. Línea 6-20. Constituye el ciclo interno de metrópolis que se ejecuta un múltiplo de m veces la cadena de markov y en cada ciclo trata de mejorar la solución S_{actual}

al generar una nueva solución vecina (S_{nuevo}) de la siguiente manera:

a) Línea 7. Genera un nuevo vecino de *N(S)* con base en la ecuación 4.23.

b) Línea 8-12. Si el costo de la función objetivo de la solución S_{nueva} mejora o iguala el costo de la función objetivo de la solución S_{actual}, entonces se reemplaza la actual por la nueva ($S_{actual} = S_{nueva}$) en la línea 9, S_{actual} siempre toma la mejor solución encontrada, pero debido a que acepta soluciones de menor calidad en la línea 17 por la ecuación de Boltzmann 4.24, entonces se pierde la mejor solución, para que esto no suceda, en la línea 10, $S_{mejorlocal}$ siempre contendrá la mejor solución encontrada hasta el momento, por tanto si la solución S_{actual} mejora o iguala a $S_{mejorlocal}$ en cuanto al costo de la función objetivo, entonces se reemplaza por la nueva ($S_{mejorlocal} = S_{actual}$) en la línea 11.

c) Línea 13-19. Si el costo de la función objetivo de la solución S_{nueva} no mejora o iguala el costo de la función objetivo de la solución S_{actual}, entonces se busca su aceptación por la probabilidad que implementa la ecuación de Boltzmann 4.24, que involucra 3 parámetros y requiere tres acciones: 1) determinar la diferencia de temperatura entre la solución S_{nueva} y S_{actual} dada por la ecuación 4.25, en la línea 14, 2) determinar el valor de un número aleatorio r entre 0 y 1 distribuido uniformemente en la línea 15 y 3) tomar el valor de la temperatura T actual.

Si se cumple la condición de la línea 16, entonces la solución actual se cambia por una de menor calidad en la línea 17, esto le permite explorar otra región del espacio de soluciones escapando de los óptimos locales, si no se cumple la condición entonces procede a otro ciclo donde genera un nuevo vecino, buscando una mejor solución que la actual.

7. Línea 21. Al finalizar el ciclo de metrópolis en la línea 20, se decrementa la temperatura T con base en la ecuación 4.22.

8. Línea 22. Termina la búsqueda local iterada al cumplirse la condición de parada en donde la temperatura T es menor o igual a la temperatura final.

9. Línea 23. La mejor solución encontrada por RS es devuelta al algoritmo AGHCGrid 4.1 principal.

4.1.12. Consideraciones de diseño para el algoritmo AGHCGrid.

Una de las consideraciones que se tienen que hacer a la hora de diseñar algoritmos paralelos y ejecutarlos sobre una Grid heterogénea, es su calendarización, es decir la planificación de como se van a ejecutar los trabajos j sobre los nodos q de cada cluster

p de la Grid, tomando en cuenta el número de núcleos $D_{p,q,r}$ que hay por nodo. Si bien la distribución de tareas puede ser de manera uniforme en cuanto al número de recursos a usar, lo que más impacta es la velocidad de los recursos asignados por su naturaleza heterogénea, en este sentido la calendarización de trabajos en núcleos puede ser visto como un problema de máquinas paralelas no relacionadas [Mártinez, 2010], donde el tiempo de procesamiento p_{ij} de un algoritmo j es dependiente del núcleo i a utilizar, por tanto el tiempo de término será el makespan $C_{max} = max(C_1, ..., C_j)$.

Otro factor que hay que tomar en cuenta es la latencia y ancho de banda de las comunicaciones entre los diferentes clusters k que están conectados a través de I2, la manera en que se relaciona el ancho de banda y la latencia es que mientras el ancho de banda aumenta, la latencia baja y mientras la latencia aumenta, el ancho de banda disminuye. Como regla general las velocidades de conexión internas de cada cluster permanecen estables, esto quiere decir que la ejecución del algoritmo en un cluster no presenta mayores problemas de envió de datos entre los nodos debido a problemas de ancho de banda si al menos se cuenta con una conexión Gigabit, pero preferiblemente Infiniband. Los problemas se dan en las interconexiones de cada cluster y es generalmente cuando se quiere enviar datos de un nodo q del cluster A a un nodo p del cluster B cuando se presentan los cuellos de botella por las altas latencias de la conexión debido principalmente a la distancia y el uso de esa red por otras aplicaciones.

Esta combinación de recursos heterogéneos y cuellos de botella entre las comunicaciones de los clusters, es lo que tomamos en cuenta para el diseño del algoritmo AGHCGrid de optimización combinatoria, en un estudio experimental llevado a cabo en la Grid Morelos[2], se concluye que el diseño debe estar orientado a optimizar el uso de comunicaciones entre nodos alejados geográficamente, de tal manera que cada proceso j que se ejecuta en cada núcleo s de cada procesador r de cada q de cada cluster p sea lo más autónomo posible y no utilice comunicaciones excesivas que puedan degradar su eficiencia.

4.2. Plataforma.

Formalmente una plataforma Grid de cómputo de intensivo tipo multi-cluster se define como un conjunto A de clusters de alto rendimiento $A = \{cluster_1, cluster_2, ..., cluster_a\}$ alejados geográficamente y conectados por Internet 2, en donde cada cluster p con $p \in A$ esta compuesto por un conjunto $B_p = \{1, 2, ..., b_p\}$ de nodos, cada nodo q de procesamiento con $q \in B_p$ de cada cluster p esta compuesto por un conjunto $C_{p,q} = \{1, ..., c_{1,1}, ..., c_{p,q}\}$ de procesadores $\geqslant 1$, en donde cada nodo q se encuentra conectado en paralelo a través de un switch de alta velocidad con su respectivo cluster p, a su vez cada procesador r con $r \in C_{p,q}$ de cada nodo q de cada cluster p esta compuesto por un conjunto $D_{p,q,r} = \{1, ..., d_{1,1,1}, ..., d_{p,q,r}\}$ de núcleos de

[2]Estudio Experimental del Efecto del Paso de Mensajes en ambiente Grid para el Desarrollo de Sistemas que Tratan con Problemas NP-Completos

procesamiento con memoria compartida.

Cada núcleo s con $s \in D_{p,q,r}$ de cada procesador r de cada nodo q de cada cluster p se refiere como $t_{p,q,r,s}$ como se observa en la figura 4.11. Los diferentes nodos q que conforman la Grid pueden ser homogéneos o heterogéneos, es decir que los nodos q que conforman un cluster p pueden diferir en la arquitectura del procesador, velocidad o en el número de núcleos. Finalmente la Grid permite despachar en forma paralela un conjunto de J de trabajos (procesos) $J = \{1, 2, ..., n\}$ y distribuirlos en forma uniforme entre el conjunto de clusters.

Dada una plataforma Grid el número óptimo de procesos que se pueden distribuir de manera uniforme y ejecutar de manera paralela es directamente proporcional al número total de núcleos de procesamiento existentes en la Grid de manera global que se calcula con la siguiente ecuación:

$$P_{max} = \sum_{k=1}^{p} \sum_{i=1}^{q} \sum_{j=1}^{r} \sum_{l=1}^{s} t_{kijl} \tag{4.26}$$

Esto es la suma de núcleos s de cada procesador r de cada nodo q de todos los clusters p que conforman la Grid, la identificación de cada uno de los procesos se basa en la usada por MPI, en donde para un total de procesos P_{max} ejecutándose en paralelo, un identificador se define como:

$$id = \{0, 1, ..., id_{max}\} \, donde \, id_{max} = (P_{max}) - 1 \tag{4.27}$$

Es decir que MPI identifica los procesos desde *0* hasta id_{max} para un total de procesos desde 1 hasta P_{max}, en donde $id_{max} = (P_{max}) - 1$.

El objetivo principal que se buscó desde el principio en el diseño de la Grid Morelos y se comprobó con las pruebas experimentales, fue que la Grid pudiera disponer para su uso, del total de los núcleos existentes en la Grid con base en la ecuación 4.26. A diferencia de otros proyectos en lo que se ha trabajado como el caso de EELA [3] y GISELA [4] que son sistemas distribuidos más robustos basados en el software gLite, en los que se busco alternativas de trabajo pero que finalmente quedaron descartados por su limitante de solo poder despachar tareas hacia un solo site (cluster) sin considerar la Grid como la suma de los recursos disponibles globalmente.

[3] http://www.eu-eela.eu/
[4] http://www.gisela-grid.eu/

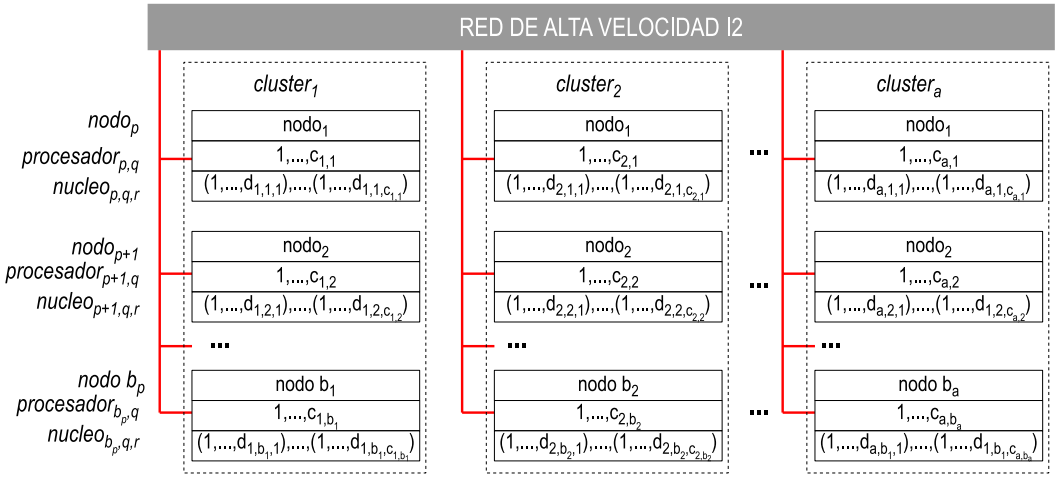

Figura 4.11. *Plataforma Grid - Definición formal.*

4.3. Aplicación.

Para poder llevar a buen término la implementación del método AGHCGrid descrito por el algoritmo 4.1 como una aplicación, se muestra en la sección 4.3.1, como se lleva a cabo la planeación de la ejecución sobre la plataforma de producción Grid Morelos descrita en la sección 4.2, en la sección 4.3.3, se explican las representaciones simbólicas usadas para cada metaheurística de AG, SCH y RS, así mismo se muestra como se llevan a cabo las transformaciones simbólicas que se dan para una solución, al pasar de una metaheurística a otra en el ciclo de mejora continua del algoritmo AGHCGrid, en la sección 4.3.4 se muestra como se generan nuevas soluciones al aplicar una mutación cooperativa descrita en la sección 4.1.10 con SCH y en la sección 4.1.11, se muestra como se generan nuevas soluciones al aplicar una mutación iterativa con RS, ambas mutaciones sobre los procesos esclavos a partir de una solución inicial acotada.

Para el AG se muestra como se envían las soluciones encontradas por los procesos esclavos al proceso maestro, como el proceso maestro construye la población inicial, aplica el operador de selección descrita en la sección 4.1.8, aplica el operador cruce descrito en la sección 4.1.9 y como son devueltas las nuevas soluciones a los procesos esclavos.

La aplicación (programa) fue desarrollada en el lenguaje de programación C y la librería de paso de mensajes MPI compatible con las versiones de MPI de Intel, OpenMPI y MPICH que implementan el estándar MPI-1 y MPI-2.

4.3.1. Planeación de la ejecución.

La aplicación ejecuta el algoritmo en forma paralela con el fin de ubicar múltiples instancias del mismo programa corriendo en un tiempo dado sobre la Grid como se muestra en la figura 4.12, el algoritmo AGHCGrid esta integrado por dos segmentos de código principales: uno para el proceso maestro y otro para los procesos esclavos. El segmento de código maestro alberga la codificación para la implementación de la selección y cruzamiento, mientras que el segmento de código para los esclavos alberga la codificación para la implementación de las mutaciones cooperativas con SCH y de las mutaciones iterativas con RS.

Finalmente ambos segmentos implementan mecanismos de paso de mensajes para comunicarse entre ellos con independencia de su ubicación geográfica, la cual involucra las técnicas para la distribución de procesos mostradas en la sección 4.1.3, de comunicación en la sección 4.1.4, de sincronización en la sección 4.1.5 y de cooperación en la sección 4.1.6.

La planeación de la ejecución tiene como objetivo hacer uso del máximo número núcleos disponibles en la Grid, que puede ser obtenido de la ecuación 4.26, con el fin de hacer una máxima exploración y explotación del espacio de soluciones. También se toma en cuenta que no todos los recursos son iguales, el trabajar con una Grid implica que los recursos pueden ser heterogéneos, por tanto lo que aplica es usar aquellos recursos que no degraden la eficiencia del algoritmo.

La técnica usada para seleccionar los recursos, consiste en lanzar un mismo proceso en toda la Grid por un tiempo suficientemente largo y medir cuanto tarda en cada nodo, después hacer un análisis nodo por nodo y determinar cuales son demasiado lentos para no incluirlos o asignarles menos procesos y cuales son muy rápidos para asignarles más procesos, siempre pensando en aproximar los tiempos de término de todos los procesos hacia un mismo tiempo para evitar los tiempos de espera que se determina con la sincronización mostrada en 4.6.

4.3.2. Construcción del grafo G=(V,A) para sistema de colonia de hormigas.

Para las metaheurísticas de RS y AG las estructuras de vecindad implementadas permiten encontrar nuevas soluciones *N(S)* aplicando pequeñas perturbaciones (mutaciones) o grandes perturbaciones como el cruzamiento, pero siempre sobre soluciones S completas.

En el caso de SCH no se manejan soluciones completas, sino que es un procedimiento constructivo paso a paso, en donde para cada nueva solución S una hormiga se enfrenta al problema de decidir qué camino seguir en el espacio de búsqueda con el fin de avanzar en la construcción de una solución, para elegir por dónde ir, se basa en los

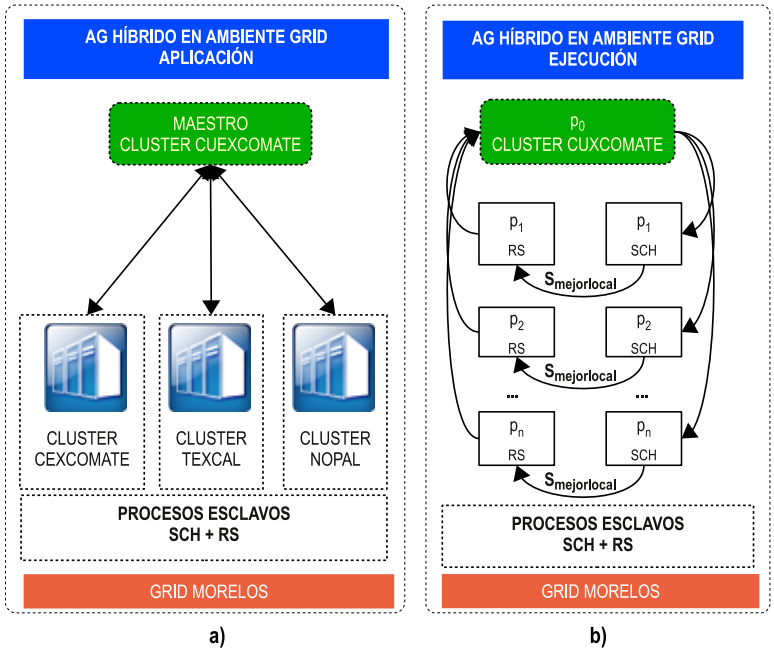

Figura 4.12. *Ejecución de la aplicación en ambiente Grid. En la figura a) se muestran los tres cluster a usar para la aplicación: Cuexcomate(CIICAp), Texcal(UPEMOR) y Nopal(ITV), donde el proceso maestro esta asignado a un nodo del cluster Cuexcomate y los procesos esclavos están repartidos en los nodos seleccionados de los tres clusters. En la figura b) la ejecución del algoritmo sobre la Grid asigna de manera automática identificadores a cada proceso como se menciona en la sección 4.1.3, el esquema implementado para el control es un maestro/esclavo en donde el proceso maestro coordina las acciones de los procesos esclavos, en este esquema se implementa la comunicación punto a punto entre cada proceso esclavo y el proceso de control maestro, esta comunicación permite al término de cada búsqueda enviar y recibir soluciones al proceso de control.*

tiempos de procesamiento, tiempos de inicio dependientes de la secuencia y en la cantidad de feromona que hay sobre los arcos, por esta razón la estructura constructiva no se ajusta a una estructura de vecindad, al final lo que se está definiendo con la información de las aristas es una heurística que ayuda a guiar la búsqueda.

SCH esta basado en una estructura de búsqueda constructiva, si bien podría comportarse como una estructura de vecindad, la diferencia esta en que no parte de una solución completa S y las cantidades de feromonas son una cuestión dinámica en tiempo de ejecución, las cuales más que intercambiarse, dependen de incrementarse y disminuirse con base en las decisiones tomadas por las hormigas al recorrer las aristas.

La estructura constructiva implementada tiene especial énfasis en la optimización del número de aristas para ser menor que una matriz cuadrada tradicional de tamaño $((m \cdot n \cdot m_k) + 2)^2$, donde m es el número de etapas, n el número de trabajos y m_k es el número de máquinas en paralelo que hay en cada etapa m, para calcular el número exacto de aristas de un grafo G=(V,A) se utiliza la siguiente ecuación 4.28, que se explica

en la figura 4.13.

$$(m_k \centerdot n)+(((m_k \centerdot n)\centerdot 2)-m_k)\centerdot(m_k \centerdot n)\centerdot(m-1)+((((m_k \centerdot n)-m_k)+1)\centerdot(m_k \centerdot n)) \quad (4.28)$$

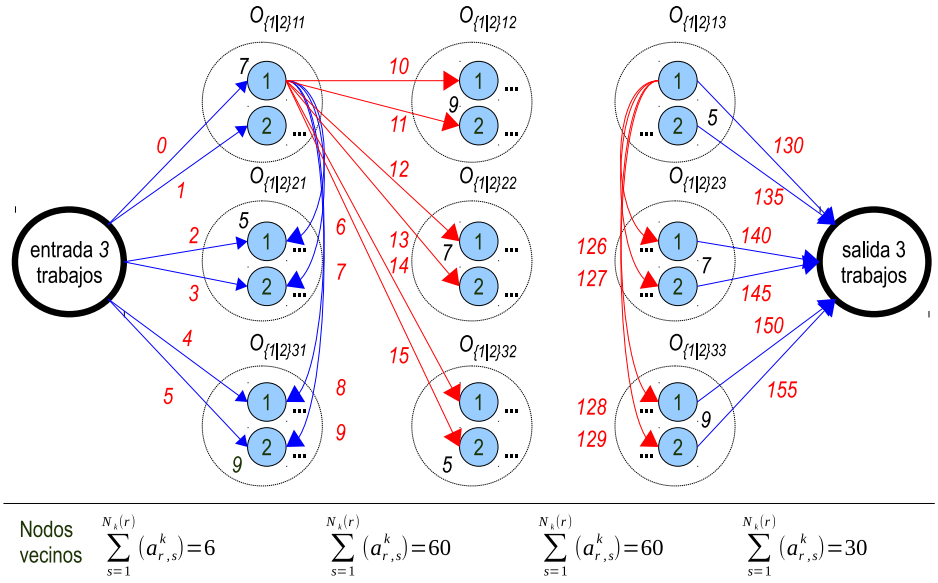

Figura 4.13. *Cálculo del número de aristas del grafo G=(V,A) para SCH. Dado un FFS-SDST integrado por 3 trabajos* $N=\{1,2,3\}$*, 3 etapas en serie* $M=\{1,2,3\}$ *y 2 máquinas paralelas idénticas* $m_k=\{1,2\}$ *por etapa, el número total de vértices se calcula con base en la ecuación 4.28, que sustituida quedaría:* $(2.3)+(((2.3).2)-2).(2.3).(3-1)+((((2.3)-2)+1).(2.3))=156$ *aristas, que se enumeran desde 0 hasta 155, en comparación con las* $((2.3.3)+2)^2=400$ *aristas de una matriz cuadrada.*

Para la construcción del grafo dirigido G=(V,A) con el número exacto de aristas dada por la ecuación 4.28, se utiliza el algoritmo 4.5, que construye un grafo con memoria estática usando vectores para acelerar el acceso, en lugar de memoria dinámica como tradicionalmente se construyen.

Cada arista esta basada en la estructura mostrada en las líneas 18-22 del algoritmo 4.5, que contiene información de las operaciones que esta uniendo, de tal forma que es posible saber de forma directa la máquina(i), trabajo(j) y etapa(k) referida como una operación O_{ijk}, en lugar de estar transformando tres índices (i,j,k) a partir de dos posiciones x,y de una matriz cuadrada que solo tiene dos índices, la ventaja se consigue al calcular los índices (i,j,k) una sola vez al momento de construir el grafo y posteriormente solo leyendo la estructura propuesta que consume menos tiempo que calcular las posiciones de una matriz cada vez que se requiera, la estructura también permite saber cual es el vecindario de una hormiga k posicionada en el vértice r, es decir $N_k(i,j,k)$, el grafo dirigido de la figura 4.13 se construye con base en el algoritmo 4.5.

En la figura 4.13, se muestra que desde la etapa inicial a la etapa 1 hay 6 aristas, si la hormiga selecciona el arco 0 para ir a la primera operación, las posibilidades aumentan a 4 aristas (6 a 9) más las 6 aristas necesarias para pasar a la siguiente etapa (10 a 15)

en caso de ser la última operación, en total 10 aristas, multiplicado por 6 que son todas las posibles rutas de la hormiga desde la etapa inicial suman 60 aristas para la etapa 1, más 60 de la etapa 2 que es similar a la etapa 1, excepto en la última etapa en donde solo quedan 4 aristas (126-129) y la arista 130 para ir a la etapa final, en total 5, multiplicado por la 6 operaciones en total de la última etapa serian 30, en total 6+60+60+30=156 aristas que corresponden a todas las posibilidades que una hormiga k posicionada en el vértice r, seleccione el siguiente vértice s, en donde el siguiente vértice s de la hormiga k se define como el vecindario de r, es decir $s = N_k(r)$.

Algoritmo 4.5 Construye grafo dirigido G=(V,A) para SCH.

```
1:  funcion Construye_Grafo(t_0)
2:      j_i = j_f = i_i = i_f = a_i = a_f = 0
3:      para (k = entrada, k < salida; k++) hacer
4:          j_i = j_f
5:          if (k == entrada || k == salida) hacer j_f += 1 else j_f += n
6:          para (j = j_i, j < j_f; j++) hacer
7:              i_i = i_f
8:              if (k == entrada || k == salida) hacer i_f += 1 else i_f += m
9:              para (i = i_i, i < i_f; i++) hacer
10:                 a_i = a_f
11:                 selecciona(k)
12:                     caso entrada    : |N(i,j,k)| = m . n
13:                     caso salida - 1 : |N(i,j,k)| = ((m . n) - m) + 1)
14:                     otro caso       : |N(i,j,k)| = ((m . n) . 2) - m)
15:                 fin selecciona
16:                 v_origen(i,l,k) = O_{i,j,k}
17:                 para (a = a_i, i < a_i + |N(i,j,k)|; a++) hacer
18:                     arista[a].id = (a - a_i) + 1
19:                     arista[a].origen_{i,j,k} = V_origen(i,l,k)
20:                     arista[a].feromona = t_0
21:                     arista[a].costo = S_{l,j,k} + p_{i,j,k}
22:                     arista[a].destino_{i,j,k} = Siguiente(N(i,j,k))
23:                 fin para
24:                 a_f = a_i + |N(i,j,k)|
25:             fin para
26:         fin para
27:     fin para
28: fin funcion
```

El algoritmo recibe como parámetro de entrada la cantidad mínima de feromona a partir de la ecuación 4.14, recorre todas las etapas desde el inicio hasta el fin (línea 3), después calcula en número de trabajos por cada etapa (línea 5) y recorre estos trabajos (línea 6), posteriormente calcula el número de máquinas que hay por trabajo (línea 8) y recorre estas máquinas (línea 9). Para cada combinación de etapa, trabajo, máquina que conforma una operación O_{ijk}, se toma como vértice origen $V_{origen}(i, l, k) = O_{ijk}$, se calcula su vecindario *N(i,j,k)* que depende de la etapa (líneas 12-15), el cual consiste en el número de aristas que conectan a V_{origen} con cada uno de las posibles operaciones que una hormiga puede transitar.

Finalmente se recorre el vecindario conformado por aristas, estableciendo en cada paso la operación origen y destino que componen cada arista, además de la cantidad de feromona inicial (τ_{rs}), costo de transitar la arista (η_{rs}) y un identificador único para cada arista.

4.3.3. Representaciones simbólicas de soluciones para SCH, AG y RS.

En esta tesis se utilizan tres representaciones simbólicas: SCH, RS y AG, las cuales se utilizan para codificar las soluciones generadas y aplicar transformaciones. Para explicar la codificación se muestra un grafo disyuntivo en la figura 2.2 de la sección 2.5.2, que representa un FFS-SDST de $3 \times 3 \times 2$ integrado por 3 trabajos $N = \{1, 2, 3\}$, 3 etapas en serie $M = \{1, 2, 3\}$ y 2 máquinas paralelas idénticas $m_k = \{1, 2\}$ por etapa, en esta figura todos los trabajos pueden elegir procesarse en cualquiera de las dos máquinas disponibles que hay en cada etapa en cualquier orden, partiendo de la entrada, pasando por la etapa 1, 2, 3, hasta llegar a la salida.

4.3.3.1. Representación simbólica de soluciones para SCH.

La representación simbólica utilizada para codificar las soluciones por las hormigas esta basada en los identificadores de las aristas referidas como arcos como se explica en la sección 4.1.10, en donde cada arista conecta dos vértices y en donde cada vértice es una operación. Una posible solución $S_{hormiga}$ para el problema de la figura 2.2 es mostrado en la figura 4.14, en donde los arcos disyuntivos pasan a ser arcos conjuntivos para poder representar el orden en el cual se procesan los trabajos sobre las máquinas en cada etapa, la figura muestra un grafo dirigido de solución G=(V,A) para una de las muchas posibles soluciones.

En la figura 4.14, se muestra que las aristas se enumeran a partir de la ecuación 4.28, quedando finalmente una solución valida basada en identificadores de aristas $S_{hormiga} = \{0, 9, 56, 42, 89, 116, 70, 132, 138, 150\}$, los cuales unen un total de 11 vértices debido a que cada trabajo solo requiere una de las dos máquinas disponibles, donde v_0, v_{19} corresponden a los vértices inicial y final respectivamente, las aristas enumeradas pueden mapearse a las aristas $a_{r,s}$, que a su vez pueden mapearse a vértices (v_r, v_s) correspondientes.

Para poder asignar y recuperar la secuencia de operaciones que componen una solución S, se hace uso de una estructura de datos con la cual se implementa un grafo dirigido G=(V,A) tal como se muestra en la sección 4.3.2, el cual es un arreglo de tamaño $m \centerdot n + 1$ que contiene la secuencia de los aristas que una hormiga k recorrió desde la entrada hasta la salida, en la figura 4.15 a) se encuentra representada la solución $S_{hormiga} = \{0, 9, 56, 42, 89, 116, 70, 132, 138, 150\}$, donde cada arista une dos vértices, como se observa en la figura 4.14, la solución $S_{hormiga}$ puede ser representada como

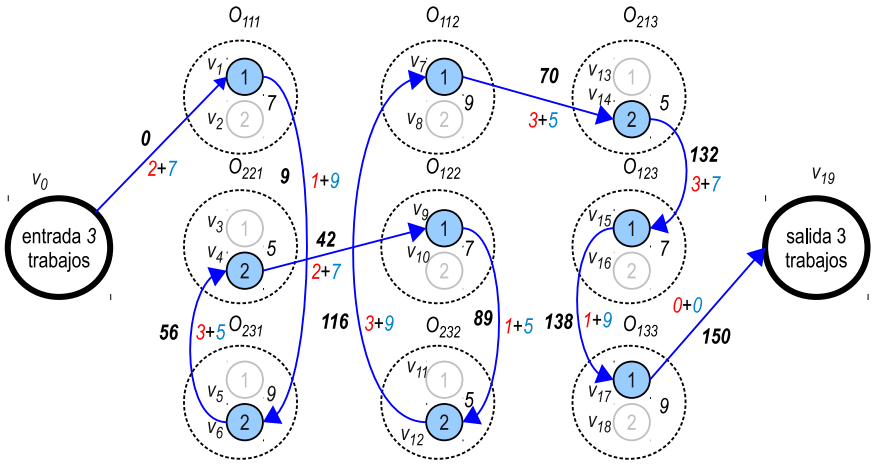

Figura 4.14. *Grafo de solución G=(V,A) para SCH. Dado un FFS-SDST integrado por 3 trabajos* $N = \{1, 2, 3\}$*, 3 etapas en serie* $M = \{1, 2, 3\}$ *y 2 máquinas paralelas idénticas* $m_k = \{1, 2\}$ *por etapa, los vértices que componen el grafo representan las operaciones y se enumeran con un identificador único* $v_r \rightarrow v_{id}$ *con* $id = 0, 1, ..., v_{max} = 19$ *a partir de la ecuación 4.13.*

una secuencia de vértices $v_o \rightarrow v_1$,$v_1 \rightarrow v_6$,$v_6 \rightarrow v_4$,$v_4 \rightarrow v_9$,$v_9 \rightarrow v_{12}$,$v_{12} \rightarrow v_7$,$v_7 \rightarrow v_{14}$,$v_{14} \rightarrow v_{15}$,$v_{15} \rightarrow v_{17}$,$v_{17} \rightarrow v_{19}$ y en donde cada vértice representa una operación, por tanto la misma solución $S_{hormiga}$ puede ser representada por una secuencia de operaciones $O_{entrada} \rightarrow O_{111}$,$O_{111} \rightarrow O_{231}$,$O_{231} \rightarrow O_{221}$,$O_{221} \rightarrow O_{122}$,$O_{122} \rightarrow O_{232}$,$O_{232} \rightarrow O_{112}$,$O_{112} \rightarrow O_{213}$,$O_{213} \rightarrow O_{123}$,$O_{123} \rightarrow O_{133}$,$O_{133} \rightarrow O_{salida}$.

Los tres casos representan la misma solución, pero la primera es la que proporciona un nivel de manejo técnico más fino, más compacto y más versátil, prueba de ello nos referimos a la figura 4.15 b) en donde el acceso a los datos clave para el cálculo del makespan están disponibles en forma inmediata a partir del número de arista *id* incluido en la solución $S_{hormiga}$, el acceso a los datos se logra posicionando un puntero sobre el índice *id* del vector, dichos datos son asignados mediante el uso del algoritmo 4.5, en la etapa de inicialización, el cual es parte del algoritmo AGHCGrid.

Finalmente en la figura 4.15 c), al extraer la información contenida en el vector referenciado por el *id* de las aristas, podemos procesar las operaciones en términos de i, j, k, costo de transitar esa arista ($\eta_{r,s}$) y cantidad de feromona ($\tau_{r,s}$), la interpretación de esta solución se da de la siguiente manera: el trabajo 1 se asigna a la máquina 1 en la etapa 1 con un tiempo de inicio 2 más un tiempo de procesamiento 7 para un total de 9 y una cantidad de feromona final de 0.0323, el trabajo 3 se asigna a la máquina 2 en la etapa 1 con un tiempo de inicio 1 más un tiempo de procesamiento 9 para un total de 10 y una cantidad de feromona final de 0.2083, y así sucesivamente hasta llegar al último trabajo 3 que se asigna a la máquina 1 en la etapa 3 con un tiempo de inicio 1 más un tiempo de procesamiento 9 para un total de 10 y una cantidad de feromona final de 0.0162.

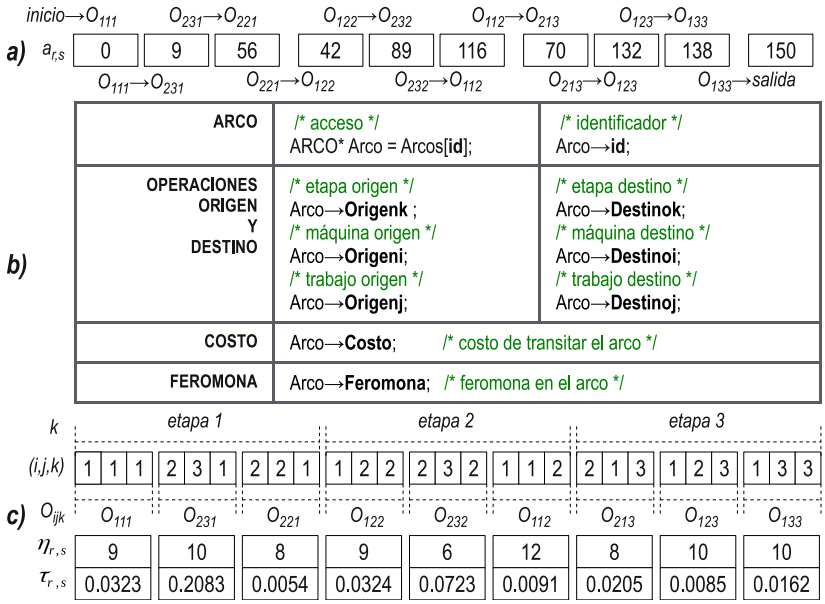

Figura 4.15. *Representación simbólica para SCH. El almacenamiento físico de una solución $S_{hormiga}$ esta representado en a) donde la secuencia de identificadores (id) corresponden a aristas del grafo 4.14, los cuales unen dos operaciones O_{ijk} que pueden ser obtenidos en b) accediendo a la estructura de datos mediante el índice id de la arista, además de el costo ($\eta_{r,s}$) de transitar la arista que consiste en el tiempo de inicio dependiente de la secuencia más el tiempo de procesamiento y la cantidad de feromona final ($\tau_{r,s}$), la representación de estos datos vitales para el cálculo del makespan pueden ser vistos en c) como una representación simbólica obtenida a partir de la solución $S_{hormiga}$.*

4.3.3.2. Representación simbólica de soluciones para RS.

La representación simbólica para codificar las soluciones en RS esta basado en pares (i,j), a los cuales se le aplica una división entera para calcular la etapa k y poder obtener una triada (i,j,k) que representan la secuencia de operaciones O_{ijk} a calendarizar como se explica en la sección 4.1.11.

Para el mismo problema anterior la solución para RS seria: $S_{metal} = \{(1,1),(2,3),(2,2),(1,2),(2,3),(1,1),(2,1),(1,2),(1,3)\}$, al cual le hace falta el componente k de la etapa, que se obtiene aplicando una división $(índice/j = 3) + 1$ quedando la secuencia de operaciones $O_{111} \rightarrow O_{231} \rightarrow O_{221} \rightarrow O_{122} \rightarrow O_{232} \rightarrow O_{112} \rightarrow O_{213} \rightarrow O_{123} \rightarrow O_{133}$ como se observa en la figura 4.16 a).

El acceso a los datos para el cálculo del makespan se obtienen al acceder mediante el índice al vector que contiene los datos de los vecinos en S_{metal} para obtener los componentes (i,j,k), el costo del tiempo de procesamiento p_{jk} viene dado por la máquina i donde va a procesarse el trabajo j en la etapa k, para el cual existe un vector de tamaño $m \cdot n$ que es igual al número de máquinas m por el número de trabajos n del sistema, pensando a futuro utilizar máquinas paralelas no relacionadas donde el tiempo

de procesamiento depende del trabajo y la máquina a usar, pero que en esta tesis se ajusta a máquinas paralelas idénticas conservando el número un máquinas para facilitar la transición.

Finalmente el tiempo de inicio dependiente de la secuencia S_{ljk} se obtiene accediendo al vector que contiene los tiempos de inicio con los índices (l,j,k) donde l es el trabajo inmediatamente anterior al trabajo j a ser procesado en la etapa k como se observa en 4.16 b).

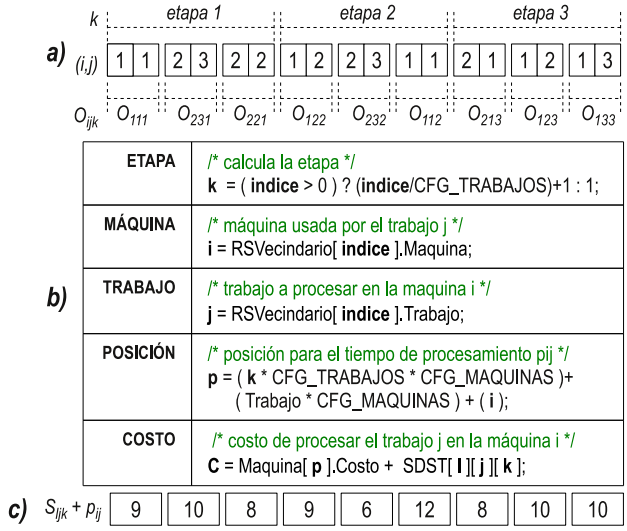

Figura 4.16. *Representación simbólica para RS. El almacenamiento físico de una solución S_{metal} esta representado en a) donde la secuencia de pares (i,j) corresponden a la secuencia de máquinas i que procesan trabajos j en la etapa k, donde k se calcula a partir del índice mediante una división entera (índice/j)+1, en b) la solución S_{metal} esta contenida en el vector RSVecindario, a partir de donde se obtienen los componentes (i,j,k), con los cuales se obtienen los tiempos de procesamiento p_{jk} y los tiempos de inicio dependientes de la secuencia S_{ljk}, la representación de estos datos vitales para el cálculo del makespan pueden ser vistos en c) como una representación simbólica obtenida a partir de la solución S_{metal}.*

Finalmente en la figura 4.16 c) después de acceder los datos de los tiempos de procesamiento y de inicio podemos obtener el costo $C = p_{jk} + S_{ljk}$ por operación para el cálculo del makespan, como resultado la secuencia de operaciones, tiempos de procesamiento e interpretación se ajusta exactamente igual al descrito para la representación simbólica de SCH mencionado anteriormente, con excepción de los rastros de feromona que no se utilizan en RS.

4.3.3.3. Representación simbólica de soluciones para AG.

La representación simbólica para codificar soluciones con AG esta basado en individuos, donde cada individuo codifica una solución de la misma forma que RS, es decir pares en (i,j) para los cuales también se tiene que determinar la etapa k a fin de poder

obtener una triada (i,j,k) que representan la secuencia de operaciones O_{ijk} a calendarizar, además de determinar para cada individuo su aptitud o adaptación que equivale a calcular el makespan de la solución que representa, seguido de su probabilidad en relación al total de individuos de la población, los cuales son necesarios para aplicar el método de selección por ruleta y cruzamiento anular tal como se explica en las secciones 4.1.8 y 4.1.9 respectivamente.

En este caso la solución codificada por un individuo seria: $S_{individuo} = \{O_{111} \rightarrow O_{231} \rightarrow O_{221} \rightarrow O_{122} \rightarrow O_{232} \rightarrow O_{112} \rightarrow O_{213} \rightarrow O_{123} \rightarrow O_{133}\}$, para la cual el cálculo de su makespan $C_{max}(S_{individuo}) = 37$ que equivale a su adaptación o aptitud, así mismo tiene una probabilidad de 0.5 en relación al resto de los individuos como se observa en la figura 4.17 a).

De igual manera que RS, los datos para el cálculo del makespan se obtienen accediendo al vector que contiene los genes en $S_{individuo}$ para obtener los componentes (i,j,k), el costo del tiempo de procesamiento p_{jk}, finalmente el tiempo de inicio dependiente de la secuencia S_{ljk} se obtiene accediendo al vector que contiene los tiempos de inicio con los índices (l,j,k) donde l es el trabajo inmediatamente anterior al trabajo j a ser procesado en la etapa k como se observa en 4.17 b).

Finalmente al igual que RS, en la figura 4.17 c) después de acceder los datos de los tiempos de procesamiento y de inicio podemos obtener el costo $C = p_{jk} + S_{ljk}$, por operación para el cálculo del makespan, como resultado la secuencia de operaciones, tiempos de procesamiento e interpretación se ajusta exactamente igual al descrito para la representación simbólica de RS mencionado anteriormente, adicionalmente contienen información de su adaptación y probabilidad.

4.3.3.4. Ciclo de transformaciones simbólicas.

El proceso de generar nuevas soluciones es un ciclo de mejora continua como se muestra en la figura 4.18, donde las primeras soluciones se obtienen en los procesos esclavos de manera separada a partir de un estado vació para $S_{hormiga} = \{\emptyset\}$ y se finaliza obteniendo una solución completa con SCH como se menciona en la sección 4.1.10.

Esta mejor solución se utiliza como solución inicial para RS, pero debido a que las soluciones entre SCH y RS difieren tanto en su representación simbólica como en su almacenamiento físico, es necesario hacer una transformación sobre la base de la ecuación 4.2, para pasar de una solución final con SCH, que esta integrada por una secuencia de aristas como se muestra en la sección 4.3.3.1, a una secuencia de pares (i,j) como se muestra en la sección 4.3.3.2, esta transformación requiere de obtener la secuencia de componentes (i,j,k) de cada arista representada en $S_{hormiga}$, a la vez que se almacena físicamente en S_{metal}, RS entonces puede iniciar su búsqueda local iterada a partir de una solución inicial.

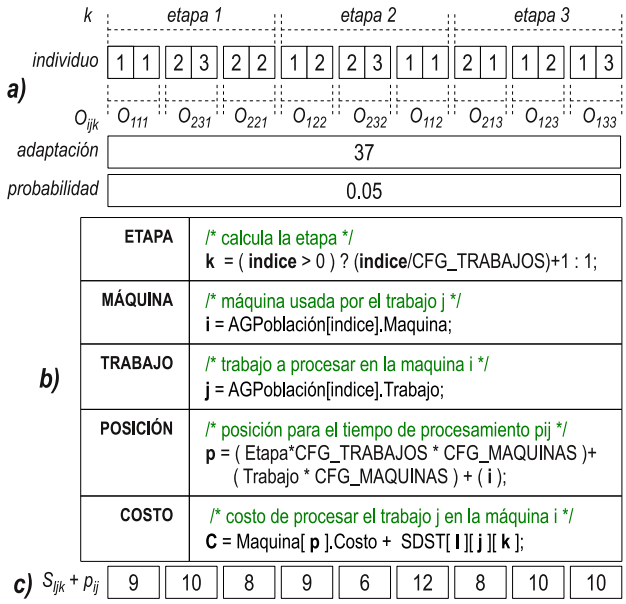

Figura 4.17. *Representación simbólica para AG. El almacenamiento físico de una solución $S_{individuo}$ utiliza una estructura AGPoblación similar a RS de la figura 4.16, en a) se observan los pares de genes (i,j) repartidos por etapas k, donde cada etapa k se calcula mediante una división entera (índice/j)+1 a partir del índice con el cual se recorre y se accede a los datos de la solución, también se observa el valor de la adaptación y probabilidad que se calcula para cada individuo de la población, en b) se observa como se obtienen los componentes (i,j,k) del vector AGPoblación, a partir de los cuales se obtienen los tiempos de procesamiento p_{jk} y los tiempos de inicio dependientes de la secuencia S_{ljk}, la representación de estos datos vitales para el cálculo del makespan pueden ser vistos en c) como una representación simbólica obtenida a partir de la solución $S_{individuo}$.*

Al finalizar RS en todos los procesos, cada proceso hace una transformación de su mejor solución S_{metal} a una solución de tipo individuo $S_{individuo}$, sobre la base de la ecuación 4.4, para que el proceso maestro pueda integrar la población del AG, en donde cada individuo esta integrado por pares (i,j) de genes de manera similar a RS como se observa en la sección 4.3.3.3, de tal forma que para esta transformación el acceso a los componente (i,j) es más directa por su similitud y el tiempo requerido es mínimo.

Finalmente cuando el proceso maestro distribuye las nuevas soluciones que son el resultado de aplicar selección y cruzamiento, los procesos esclavos deben llevar a cabo una transformación de individuo a SCH sobre la base de la ecuación 4.7, para completar el primer ciclo de mejora continua, en donde a partir de este segundo ciclo SCH a diferencia del primero inicia a partir de una solución inicial, para llevar a cabo esta transformación es necesario pasar la secuencia de componentes (i,j,k) de una solución $S_{individuo}$ a su correspondiente secuencia de números de arista para una solución $S_{hormiga}$.

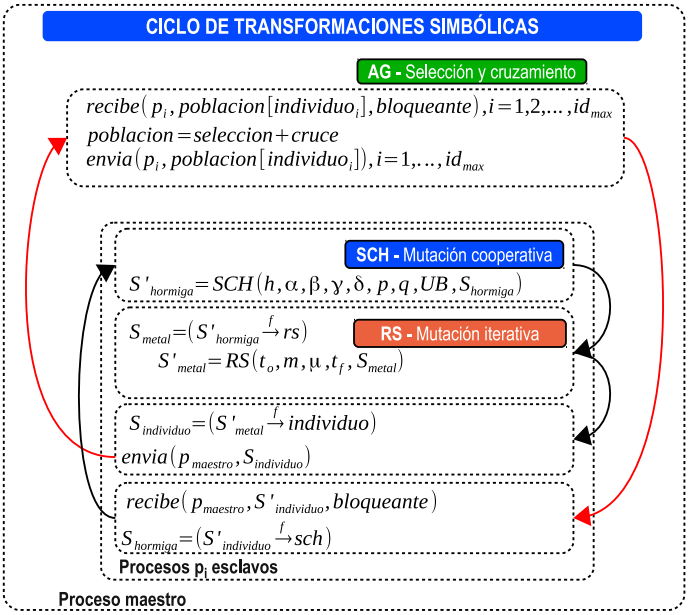

Figura 4.18. *Ciclo de transformaciones simbólicas para el algoritmo AGHCGrid. Comienza con un primer ciclo en donde se inicia aplicando SCH con una solución inicial* $S_{hormiga} = \{\emptyset\}$ *en la llamada a la función* $S'_{hormiga} = SCH(h, \alpha, \beta, \gamma, \delta, p, q, UB, S_{hormiga})$*, posteriormente la mejor solución se transforma a RS con* $S_{metal} = S'_{hormiga} \xrightarrow{f} rs$ *y se manda a llamar a la función* $S'_{metal} = RS(t_o, m, \mu, t_f, S_{metal})$*, la mejor solución entonces se transforma a un individuo* $S_{individuo} = S'_{metal} \xrightarrow{f} individuo$*, esta transformación la llevan a cabo todos los procesos esclavos y lo envían al maestro para que construya la población del AG* $envia(proceso_{maestro}, S_{individuo})$*, el proceso maestro aplica selección y cruzamiento y devuelve las soluciones a los procesos esclavos, en donde cada proceso esclavo recibe la solución inicial para SCH* $recibe(proceso_{maestro}, S'_{individuo}, bloqueante)$*, pero antes debe hacer una transformación de individuo a SCH* $S_{hormiga} = S'_{individuo} \xrightarrow{f} sch$*, completando el primer ciclo a partir del cual SCH comienza con una solución inicial continuando con el ciclo de mejora continua del algoritmo.*

4.3.4. Generación de soluciones.

El algoritmo AGHCGrid utiliza el ciclo de transformaciones simbólicas que involucra al proceso maestro y a todos los procesos esclavos, y este determinado por el máximo número de generaciones del AG, el proceso de transformaciones simbólicas es descrito en la sección 4.3.3 y sigue la secuencia $SCH \rightarrow RS \rightarrow AG \rightarrow SCH$ como se muestra en la figura 4.18, donde SCH+RS se realizan en los procesos esclavos y AG en el proceso maestro, a partir de las cuales se comienza la generación de nuevas soluciones, a continuación se muestra el mecanismo mediante el cual se consiguen generar nuevas soluciones, para las tres metaheurísticas usadas en esta tesis mediante las cuales se lleva a cabo el el proceso de exploración y explotación.

4.3.4.1. Generación de soluciones para SCH con reinicio.

Para el caso de SCH se hace uso del algoritmo 4.2 y 4.3 para generar una nueva solución cada vez, en donde cada paso hace uso de la función de transición pseudoaleatoria sobre la base de la ecuación 4.20,como se indica en el algoritmo 4.3, para el caso del primer ciclo de mejora continua no existe solución inicial por tanto todos las aristas del grafo G=(V,A) contienen la misma cantidad mínima de feromona ($\tau_{r,s} = t_0$), donde t_0 se calcula con base en la ecuación 4.14, y se aplica con base en la ecuación 4.15, con lo que se tiene la misma probabilidad de seleccionar un arco del vecindario $sig^k_{r,s} = N_k(r)$, como se muestra en la figura 4.19.

Para el segundo ciclo de mejora continua donde ya existe una solución previa como resultado de la transformación ($S_{hormiga} = S'_{individuo} \xrightarrow{f} sch$) dada por la ecuación 4.7, se deposita feromona adicional sobre las aristas de $S_{hormiga}$ con base en la ecuación 4.16, por lo que seleccionar un arco del vecindario $sig^k_{r,s} = N_k(r)$, esta influenciado por los vértices reforzados de $S_{hormiga}$.

Seleccionar el siguiente nodo s a partir de una hormiga k posicionada en el nodo r, esta dado por la ecuación 4.20, pero dado que la selección también incluye el costo de transitar la arista $a_{r,s}$, entonces la transición esta condicionada a la distancia $\eta_{r,s} = (S_{ljk} + p_{jk})^{-1}$ que es la inversa de tiempo de inicio dependiente de la secuencia (S_{ljk}) entre el trabajo l seguido del trabajo j en la etapa k, más el tiempo de procesamiento p_{jk} del trabajo j en la etapa k.

Para equilibrar la toma de decisión entre las cantidades de feromona y la distancia que hay sobre las aristas del grafo, se manipulan los factores de importancia α, β, de tal manera que se pueda favorecer con una mayor ponderación los rastros de feromona o distancia, con la finalidad de seleccionar el mejor valor que de los mejores resultados y evite caer en óptimos locales.

Otros dos factores que influyen en la calidad de las soluciones generadas son r,q, donde $r \in [0, 1]$, es un número generado de forma aleatoria, que al ser comparado con q determina cuando una hormiga k elige aplicar una explotación de las cantidades de feromonas acumuladas sobre las aristas de vecindario $N_k(r)$ y cuando se aplica una selección tipo ruleta con base en las probabilidades que tiene cada vecino como se observa en la ecuación 4.20.

Finalmente con respecto al control de la feromona que es la parte dinámica del algoritmo, las hormigas llevan a cabo dos tareas: 1) reducir la cantidad de feromona en cada transición en el sentido de incrementar el descubrimiento de nuevos caminos favoreciendo la exploración con base en la ecuación 4.21, esto debido que al tener menos cantidad de feromona se vuelve menos atractivo para las hormigas, 2) posteriormente al terminar de construir el camino completo y determinar la mejor solución global, todas las hormigas depositan feromona sobre los arcos que coincidan con la mejor solución global con base en la ecuación 4.18, como se muestra en la figura 4.20.

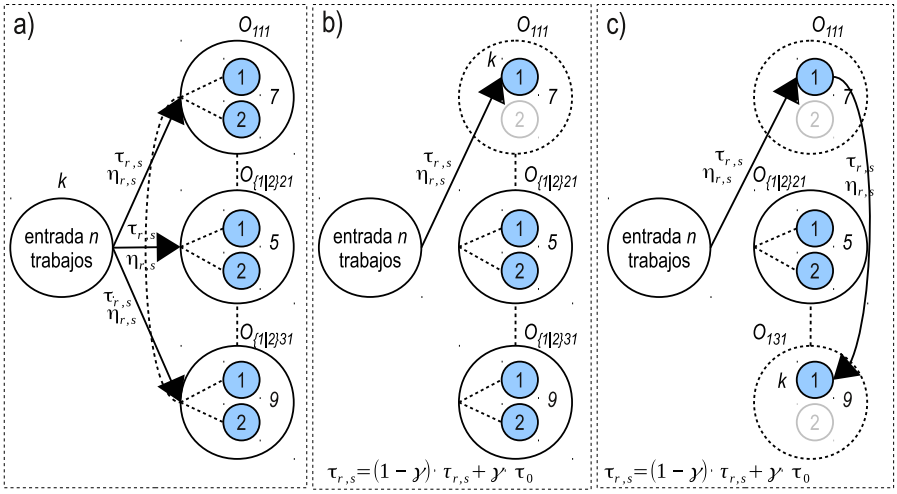

Figura 4.19. *Inicio de construcción de soluciones paso a paso para SCH. En la figura a) muestra una hormiga k posicionada en el vértice inicial r, la cual selecciona el siguiente vértice s a partir del vecindario alcanzable* $sig^k_{r,s} = N_k(r)$ *dada por la ecuación 4.20. En la figura b) se muestra a la hormiga k después de seleccionar procesar el trabajo 1 en la máquina 1 de la etapa 1 (*O_{111}*), a la vez que aplica una actualización de feromona paso a paso dado por la ecuación 4.21, la operación* O_{211} *que corresponde seleccionar la máquina 2 ya no se toma en cuenta debido a que solo se requiere una de las dos disponibles. En la figura c) se muestra a la hormiga transitar hacia la operación* O_{131} *con una solución parcial* $S_k = \{inicio, O_{111}, O_{131}\}$.

4.3.4.2. Generación de soluciones para RS con reinicio.

La generación de nuevas soluciones para RS se muestra en la figura 4.21, haciendo uso del algoritmo 4.4, en donde se inicia con una solución completa S_{metal} resultado de la transformación simbólica ($S_{metal} = S'_{hormiga} \xrightarrow{f} rs$) dada por la ecuación 4.2, la cual obtiene la mejor solución encontrada por SCH, que a su vez es un ciclo de mejora continua con un número de ciclos determinado por el número de generaciones.

Un nuevo vecino se obtiene al aplicar la ecuación 4.23, la cual obtiene un nuevo vecino *N(S)* a partir de la solución actual S_{metal}, el mecanismo utilizado consiste en aplicar una perturbación que modifique la secuencia de procesamiento de los trabajos asignados a determinada etapa y cambiar la máquina asignada al procesamiento de dicho trabajo, la naturaleza del problema no permite intercambiar operaciones entre diferentes etapas debido a la condición de ejecución en serie, los pasos para obtener una nueva solución son los siguientes.

1. Seleccionar de forma aleatoria una etapa k de las m etapas del sistema.

2. Sobre la etapa k seleccionada, seleccionar un par aleatorio de trabajos (j, j') con $j \neq j'$ y se intercambian entre si para obtener un par (j', j).

3. Cambiar de forma aleatoria la máquina i e i' asignadas a procesar el trabajo j y

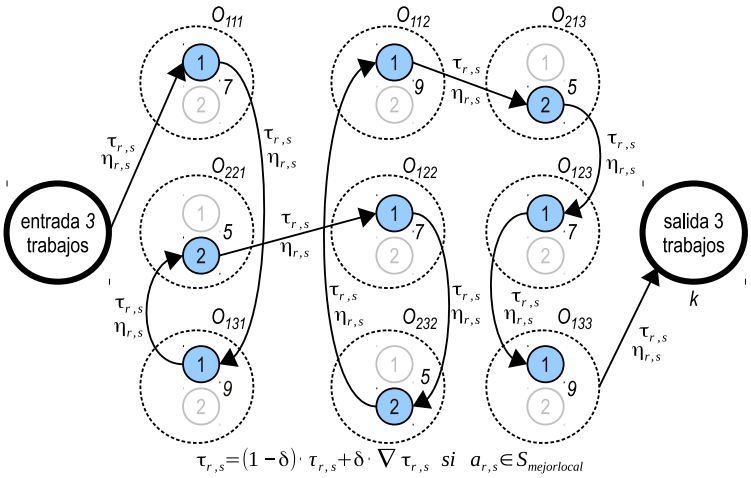

Figura 4.20. *Construcción final de soluciones paso a paso para SCH. Al terminar la hormiga k de construir la solución paso a paso, queda una solución final* $S_k = \{entrada, O_{111}, O_{231}, O_{221}, O_{122}, O_{232}, O_{112}, O_{213}, O_{123}, O_{133}, salida\}$*, una vez que todas las hormigas k construyen sus soluciones correspondientes, estas se comparan con* $S_{mejorglobal}$ *para determinar nuevamente la mejor solución global, finalmente esta solución es usada para compararse con las k soluciones y aplicar un incremento de feromona con base en la ecuación 4.18.*

j' respectivamente, las nuevas máquinas se seleccionan de ente las M_k existentes para la etapa *k*.

Durante el proceso de diseño del mecanismo se observó que esta pequeña perturbación muestra mejores resultados que perturbar más de una etapa *k*, el perturbar todas las etapas no mostró mejores resultados.

4.3.4.3. Generación de soluciones en AG.

La generación de nuevas soluciones en el proceso maestro se divide en tres fases:

1. Construcción de la población generacional en el proceso maestro, en donde cada individuo de la población, corresponde con la mejor solución encontrada en cada proceso esclavo, como resultado de aplicar una explotación mediante SCH+RS, como se muestra en la sección 4.1.7.

2. Construcción de una nueva población a partir de la aplicación de los operadores de selección, mostrado en la sección 4.1.8 y cruzamiento mostrado en la sección 4.1.9.

3. Distribución de la nueva población a cada proceso esclavo para que apliquen SCH+RS con reinicio, haciendo posible la cooperación de procesos aplicando el método mostrado en la sección 4.1.6.

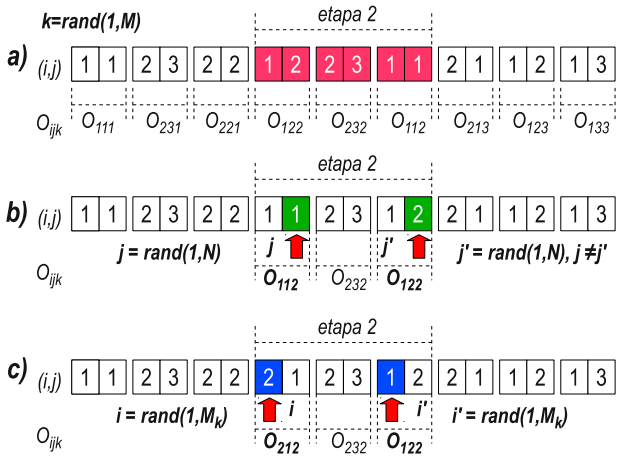

Figura 4.21. *Construcción de soluciones para RS. En la figura a) se muestra como seleccionar de manera aleatoria una etapa k mediante k=rand(1,M) = 2 para aplicar la perturbación. En la figura b) se muestra como para la etapa k=2 se intercambian de posición dos trabajos (j,j') seleccionados de manera aleatorio con* $j = rand(1, M_k) = 1$ *y* $j' = rand(1, M_k) = 2$ *con* $j \neq j'$*. En la figura c) se muestra como para los trabajos (j=1,j'=2) se cambian las máquinas asignadas (1,1) por las máquinas (2,1) generados de forma aleatoria. Al finalizar la perturbación, la secuencia de la etapa k=2 cambio de* $O_{122} \rightarrow O_{232} \rightarrow O_{112}$ *a* $O_{212} \rightarrow O_{232} \rightarrow O_{122}$*, para la etapa k=2.*

Los tres pasos se explican a continuación: el proceso de construcción de la población generacional comienza en cada proceso esclavo aplicando SCH sin reinicio o con reinicio dependiendo si es el primer ciclo de mejora continua o no, seguido de RS con reinicio, entonces la mejor solución local sigue las transformaciones simbólicas $individuo \xrightarrow{f} sch \xrightarrow{f} rs \xrightarrow{f} individuo$ como se observa en la figura 4.18.

Antes de ser enviado al proceso maestro, los procesos de comunicación hace uso de la biblioteca de paso de mensajes de MPI entre el proceso maestro y los esclavos, siguiendo el método mostrado en la sección 4.1.4 y 4.1.6, donde primero $MPI\_Recv() \leftarrow MPI\_Send()$ envían los procesos esclavos al maestro y posteriormente $MPI\_Send() \rightarrow MPI\_Recv()$ distribuye la población a los procesos esclavos a partir del proceso maestro.

Para poder llevar a cabo este mecanismo con el mínimo tiempo se aplica el método de sincronización de procesos descrito en la sección 4.1.5. La construcción de la población aplica el método mostrado en 4.1.7, para integrar una población individuo por individuo recibidos de cada uno de los procesos esclavos, donde cada individuo es la mejor solución local a cada proceso en cada generación del algoritmo como se observa en la figura 4.22.

Una vez construida la población se procede a realizar la selección por el método de la ruleta de los individuos mejor adaptados, aplicando el operador de selección mostrado en la sección 4.1.8 y posteriormente se procede a realizar la exploración, de las solucio-

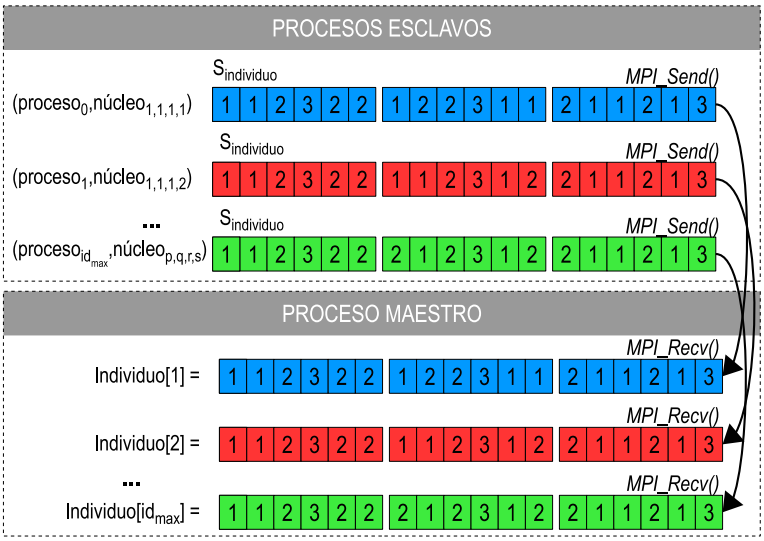

Figura 4.22. *Construcción de la población generacional. En la figura se observa como cada proceso p_{id} con $id = 0, ..., id_{max}$ que se ejecuta en el núcleo$_{p,q,r,s}$, envía su mejor solución local encontrada con MPI_ Send() y el proceso maestro lo recibe con MPI_ Recv() en cada iteración del algoritmo, donde cada iteración corresponde a una generación. Cada proceso antes de enviar la solución debe aplicar una búsqueda local cooperativa con reinicio con SCH seguida de una búsqueda local iterativa con reinicio con RS y las transformaciones correspondientes, en la figura se observa como el tamaño de la población es directamente proporcional al número de procesos esclavos y su correspondencia es uno a uno.*

nes recibidas de los procesos esclavos, para obtener una nueva población, aplicando el operador cruce mostrado en la sección 4.1.9.

Una vez que se obtiene la nueva población y se aplica la exploración, se procede a repartir las nuevas soluciones a los procesos esclavos como se observa en la figura 4.23, desde el proceso maestro hacia los procesos esclavos en una relación uno a uno, en donde los procesos esclavos al recibir la nueva solución, aplicarán las transformaciones simbólicas correspondientes e iniciaran una nueva búsqueda aplicando SCH+RS.

4.3.4.4. Generación de números pseudo-aleatorios en la Grid.

Dado una semilla aleatoria inicial k, los subsecuentes números $k + 1$ generados por el algoritmo AGHCGrid de forma aleatoria, son los que proveen su comportamiento estocástico. Para poder llevar a cabo la experimentación en donde no se repitan los mismos números, es necesario iniciar cada vez con una semilla k diferente, una manera de hacerlo es generarlos a partir del tiempo actual dado en segundos, esto asegura que una semilla inicial k, obtenida a partir del tiempo dado en segundos sobre un nodo cualquiera siempre es diferente.

Para poder recrear una experimentación que ha dado buenos resultados, primero

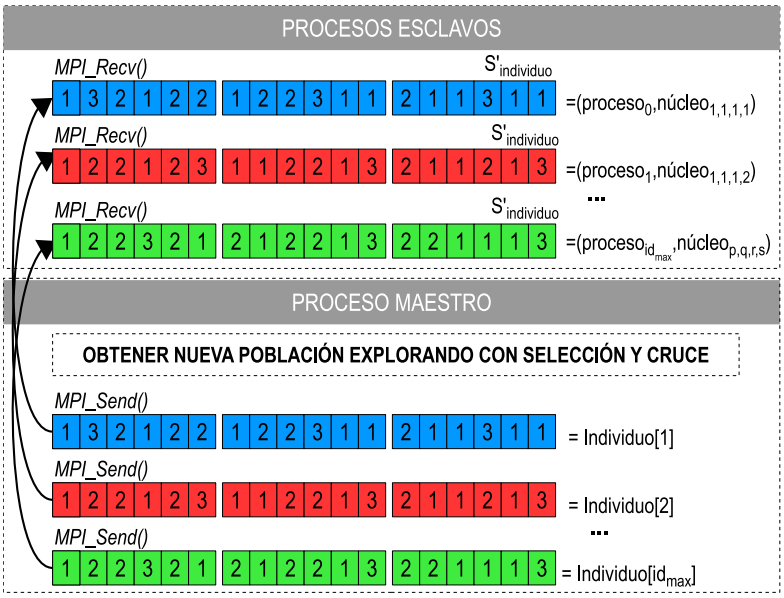

Figura 4.23. *Selección y exploración del espacio de soluciones. En la figura se observa como cada individuo de la población, es sometido a una exploración compuesta de un proceso de selección y cruce para obtener una nueva población, que posteriormente es distribuido de manera uniforme en una relación uno a uno, con cada proceso esclavo para iniciar una nueva búsqueda en el proceso de mejora continua para encontrar el mejor valor.*

es necesario guardar la semilla k con la que inicio el experimento, posteriormente establecerla como semilla inicial para volver a obtener los mismos números aleatorios y obtener una recreación exacta de la experimentación cuantas veces se requiera, de esta forma cada proceso que se crea con MPI genera una semilla inicial k diferente, por tanto si se tienen n procesos es necesario primero guardar las n semillas k con las que se inicio el experimento y posteriormente establecerlas como semillas iniciales k en cada proceso para poder recrear la experimentación de forma exacta.

Para evitar guardar n semillas iniciales para un grupo de procesos, al algoritmo AGHCGrid solo genera una semilla inicial k sobre el proceso maestro y esta es replicada a todos los n procesos esclavos, que a su vez generan una variación de la semilla inicial k del maestro agregando el factor del número de proceso, con esto se consigue generar semillas aleatorias diferentes para cada proceso a partir de solo una semilla inicial k, de esta forma es posible recrear una experimentación sobre un grupo de procesos estableciendo la semilla inicial k que deseamos recrear únicamente sobre el proceso maestro.

4.4. Análisis de la complejidad del algoritmo AGHCGrid.

Dos clases de complejidad se analizan: complejidad temporal *T(n)* y complejidad espacial *E(n)*. El algoritmo GHCGrid tiene una complejidad espacial *T(n)*, si para cualquier instancia de entrada de tamaño n, ocupa como máximo *T(n)* cantidad de memoria,

así mismo, el algoritmo GHCGrid tiene una complejidad temporal *T(n)*, si para cualquier problema de tamaño n, requiere de ejecutar un total de *T(n)* instrucciones para resolverlo, ambas complejidades *T(n)* y *E(n)* representan la cantidad de memoria y el tiempo requerido para resolver un problema de tamaño n en forma secuencial [Mártinez, 2010].

Para resolver el problema utilizando un cluster o una Grid se hace uso de la teoria de la complejidad paralela [Ian Parberry, 1987]. El algoritmo GHCGrid tiene una complejidad espacial paralela *E(n,m)*, si para cualquier instancia de entrada de tamaño n, ocupa como máximo *E(n,m)* cantidad de memoria distribuida en m nodos, así mismo, el algoritmo GHCGrid tiene una complejidad temporal paralela *T(n,p)*, si para cualquier problema de tamaño n, requiere de ejecutar un total de *T(n,p)* instrucciones en paralelo utilizando p núcleos de procesamiento de un cluster o Grid. para resolverlo, ambas complejidades *T(n,p)* y *E(n,p)* representan la cantidad de memoria distribuida en p nodos y el tiempo requerido en paralelo al utilizar p núcleos para resolver un problema de tamaño n.

La complejidad temporal para el algoritmo GHCGrid ejecutado sobre un solo núcleo p, tiene un comportamiento secuencial, y esta dada por la ecuación 4.29.

$$T_s(n,1) = \sigma(AG) + \frac{\varphi(SCH + RS)}{1} \tag{4.29}$$

en donde $\sigma(AG)$, indica que el algoritmo genético se ejecuta en forma secuencial en el proceso maestro, mientras $\varphi(SCH + RS)$, indica que los algoritmos de colonia de hormigas y recocido simulado, se ejecutan en paralelo en los procesos esclavos, utilizando el mismo núcleo que el algoritmo genético, es decir, el algritmo AGHCGrid se ejecuta en un solo núcleo, lo que le da un comportamiento secuencial.

La complejidad temporal para el algoritmo GHCGrid paralelizado al utilizar p núcleos esta dada por la ecuación 4.30.

$$T_p(n,p) = \sigma(AG) + \frac{\varphi(SCH + RS)}{p} \tag{4.30}$$

en donde $\sigma(AG)$, es similar al indicado en la ecuación 4.29, indica la parte que no se esta paralelizando y que se ejecuta en forma secuencial como el proceso maestro, mientras $\varphi(SCH + RS)$, indica la parte que se paraleliza como procesos esclavos, al utilizar p núcleos de la Grid, en donde el proceso maestro se ejecuta en un núcleo diferente al utilizado por los procesos esclavos, en una relación uno a uno como el indicado en la sección 4.1.3.

La figura 4.24, muestra el conjunto de funciones que integran el algoritmo AGHC-Grid propuesto, del lado izquierdo se encuentran las funciones utilizadas por el proceso

maestro y del lado derecho las funciones utilizadas por los procesos esclavos, la figura muestra un arbol de llamadas a funciones, para las cuales se requiere calcular su complejidad temporal.

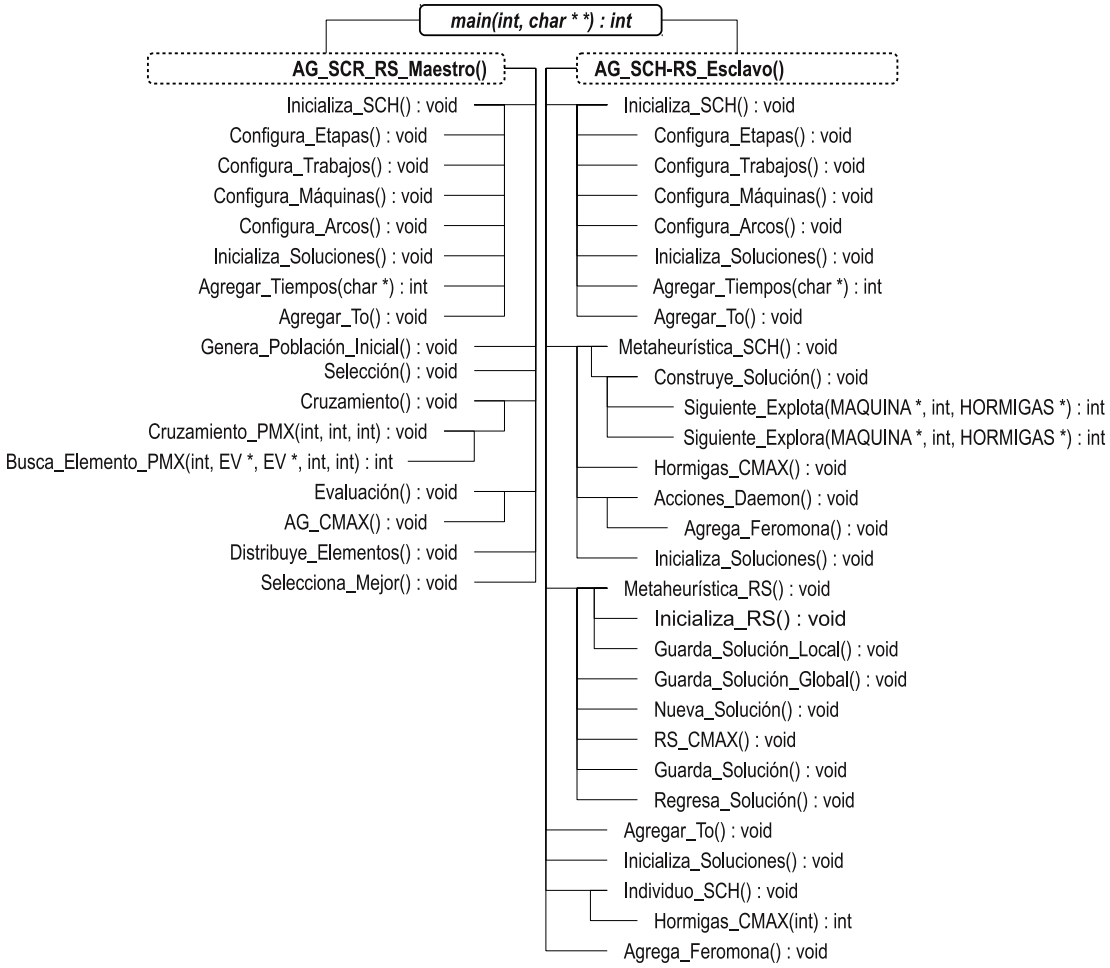

Figura 4.24. *Arbol de llamadas a funciones del algoritmo AGHCGrid.*

La tabla 4.2, muestra la complejidad temporal $T(n)$ de cada una de las funciones que integran el algoritmo AGHCGrid, en donde $T(n)$ representa el número de instrucciones simples que el algoritmo ejecuta: como el realizar una operación aritmética, hacer asignaciones, comparaciones, etc. Para poder calcular la complejidad temporal del algoritmo, es necesario hacer un análisis del código fuente y sumar paso a paso el número de instrucciones que contiene, un ejemplo de ello es mostrado en el apéndice A, la complejidad mostrada en la tabla 4.2, para cada una de las funciones y en general para todo el algoritmo, se considera con base en el peor de los casos.

Sobre la base de la ecuación 4.30, y de la funciones temporales calculadas en la tabla 4.2, se tiene que la complejidad temporal del algoritmo AGHCGrid es:

$$
\begin{aligned}
T_p(n,p) = \quad & 30G\,m\,m_k n\,T + 5G\,m\,n^2T + 4G\,m\,n\,T^2 + 29G\,m\,n\,T + \\
& 19G\,m\,T + 2G\,n\,T + 3G\,T^2 + 64G\,T + 13G + 5h\,m\,n + 9h + \\
& 34km^2m_kn^2 + 2m\,{m_k}^3n^2 - m\,{m_k}^3n + 30m\,{m_k}^2n^2 - \\
& 32m\,{m_k}^2n + 27m\,m_kn + 8mn^2 + 37m\,n + 29m - 3n + 6T + 39 + \\
& (9hm^2n^2p + 299h\,m^2np - 299h\,m\,m_kp + 23h\,m\,n\,p + \\
& 224h\,m\,p + 3h\,m_kp + 3h\,n\,p + 27h\,p + 4m^2n^2p + 18m\,n\,p + \\
& 15p + 3) + (30 + 3m + 2mk + 5n + 26mn + 7m^2mkn^2(-1 + \\
& mn)\left(n!\left(\prod_{i=1}^{m} m_i\right)^m\right) + 2m\,n(m((7m_k + 34)n + 3) + 2m_k + \\
& 5n + 34) + 17m\,n + 22)/p
\end{aligned}
$$

Por tanto la complejidad asintótica del algoritmo propuesto AGHCGrid, en base en el peor de lo casos es: $O\left(30 + 3m + 2mk_k + 5n + 26mn + 7m^2m_kn^2(-1 + mn)\left(n!\left(\prod_{i=1}^{m} m_i\right)^m\right)\right)$, la complejidad de RS se obtuvo sobre la base de [Aarts and Van Laarhoven, 1985], los pasos seguidos se muestran en el apéndice A.

Función	T(n)
AG_SCH_RS_Maestro	$30Gmm_knT + 5Gmn^2T + 4GmnT^2 + 29GmnT + 19GmT + 2GnT + 3GT^2 + 64GT + 13G + 5hmn + 9h + 34km^2m_kn^2 + 2mm_k{}^3n^2 - mm_k{}^3n + 30mm_k{}^2n^2 - 32mm_k{}^2n + 27mm_kn + 8mn^2 + 37mn + 29m - 3n + 6T + 39$
Inicializa_SCH	$30 - 3n + 34k\,m^2m_k\,n^2 + h(9 + 5m\,n) + m(29 - (-37 - 27m_k + 32m_k{}^2 + m_k{}^3)n + 2(4 + 15m_k{}^2 + m_k{}^3)n^2)$
Configura_Etapas	$3 + 10m$
Configura_Trabajos	$3 + 4m + 12m\,n$
Configura_Maquinas	$3 + 4m(1 + n + 5m_k\,n)$
Configura_Arcos	$2 + m(4 - (-4 - 5m_K + 15m_k{}^2 + m_k{}^3)n + 2m_k{}^2(15 + m_K)n^2)$
Inicializa_Soluciones	$2 + h(9 + 5m\,n)$
Agregar_Tiempos	$13 - 3n + m(7 + 17n + 8n^2)$
Agregar_To	$2 + m(2 - 17m_K)m_k\,n + 34k\,m^2m_k\,n^2$
Genera_Poblacion_Inicial	$4 + 23T$
Seleccion	$3 + 2(7 + 2m\,n)T + 3T^2$
Cruzamiento	$2 + (7 + m(17 + 25n + 5n^2))T + 4m\,n\,T^2$
Cruzamiento_PMX	$14 + 5n^2 + n(25 + 4T)$
Busca_Elemento_PMX	$1 + 2T$
Evaluacion	$2 + (15 + 2n + m(2 + 30m_K\,n))T$
AG_CMAX	$2(6 + m + n + 15m\,m_k\,n)$
Distribuye_Individuos	$5T + 2$
Selecciona_Mejor	$6T + 2$
AG_SCH_RS_Procesos	$11G\left(30 + 3m + 2m_k + 5n + 26mn + 7m^2m_kn^2(-1 + mn)\left(n!\left(\prod_{i=1}^{m} m_i\right)^m\right)\right) + 11G(hp(m(n(9mn + 299m + 23) - 299m_k + 224) + 3(m_k + n + 9)) + 3h(5mn + 3) + m(m(2n(n(17k\,m_k + 7m_k + 2p + 34) + 4) + 17) + n(-17m_k{}^2 + 16m_kn + mk + 10n + 18p + 112) + 5) + 3n + 17p + 53) + mn(5h + m_k(34kmn + 2m_k(m_k + 15)n - m_k(m_k + 32) + 27) + 8n + 37) + 9h + 29m - 3n + 35$
Metaheuristica_OCH	$9hm^2n^2p + 299hm^2np - 299hmm_kp + 23hmnp + 224hmp + 3hm_kp + 3hnp + 27hp + 4m^2n^2p + 18mnp + 15p + 3$
Construye_Solucion	$6 + m(221 - 299m_k) + 299m^2n$
Siguiente_Explota	$8 - 8mk + 8m\,n$
Siguiente_Explora	$9 - 15m_k + 15m\,n$
Hormigas_CMAX	$3(4 + m_k + n) + m(3 + 23n)$
Acciones_Daemon	$10 + 6h + 10m\,n + (4 + 9h)m^2n^2$
Agrega_Feromona	$2 + n(4m + 4m^2n + 9h\,m\,2n)$
Metaheuristica_RS	$30 + 3m + 2m_k + 5n + 26mn + 7m^2m_kn^2(-1 + mn)\left(n!\left(\prod_{i=1}^{m} m_i\right)^m\right) + 2mn(m((7m_k + 34)n + 3) + 2mk + 5n + 34) + 17mn + 22$
Inicializa_RS	$9m\,n + 6$
Guarda_Solucion_Local	$4m\,n + 2$
Guarda_Solucion_Global	$4m\,n + 2$
Nueva_Solucion	$22 + 2m_k + 2n$
RS_CMAX	$8 + 3m + 3n + 26m\,n + 7m\,m_k\,n$
Guarda_Solucion	4
Regresa_Solucion	5
Individuo_SCH	$21 + 3m + 17m^2 + 3n + 23m\,n - 5m\,m_k\,n + 16m\,m_k\,n^2$

Tabla 4.2. *Complejidad temporal T(n) de las funciones del algoritmo AGHCGrid.*

Capítulo 5

SINTONIZACIÓN DISTRIBUIDA AUTOMÁTICA APLICADA EN PARALELO

La implementación y propuesta de una sintonización distribuida automática aplicada en paralelo (SDAAP) en esta tesis, se propone con el fin de acortar los tiempos de sintonizado de los parámetros de las tres metaheurísticas que se utilizan (AG,SCH,RS), ya que el hacerlo en forma secuencial en una sola máquina llevaría demasiado tiempo dada la cantidad de parámetros que tienen. Parte del tiempo invertido en este proceso se incrementa debido a que requiere de la asistencia del usuario para realizar tareas de edición, compilación, ejecución y recolección de resultados en forma manual y solo se aprovecha el tiempo que el usuario pueda mantenerse atento a lo que suceda.

Estos tiempos se pueden reducir si contamos con un conjunto de máquinas disponibles para realizar estas tareas, pero aun así las tareas manuales se verían multiplicadas para el usuario, el cual tendría muchas posibilidades de equivocarse sin un control manual exacto. La propuesta es poder llevar a cabo este proceso de principio a fin de forma automática, sobre una sola máquina o sobre un conjunto de máquinas agrupadas en la forma de un cluster o una Grid, en la sección 5.1, se muestra una sintonización secuencial automática que no requiere asistencia del usuario en todo momento y en la sección 5.2, se muestra como se lleva a cabo una sintonización distribuida automática sobre un cluster o Grid.

5.1. Sintonización secuencial automática.

Dado un conjunto de parámetros del algoritmo propuesto AGHCGrid $F = \{s_1, s_2, ..., s_f\}$ a sintonizar, en donde cada parámetro k con $k \in F$ esta compuesto

por una serie de valores $F_k = \{v_1, v_2, ..., v_{f_k}\}$, el proceso de sintonización o análisis de sensibilidad es determinar acorde a la función objetivo la siguiente permutación p_s que obtenga los mejores resultados en la fase de experimentación.

$$p_s = \{v_1^{mejor} \in F_1, v_2^{mejor} \in F_2, ..., v_k^{mejor} \in F_k\} \tag{5.1}$$

El procedimiento para la sintonización consiste en generar primero las permutaciones sobre los valores $v \in F_1$, mientras se mantiene fijo el primer elemento de los parámetros restantes ($p_s = \{v \in F_1, F_2^1, ..., F_k^1\}$), esto permite ejecutar el algoritmo AGHCGrid con la permutación correspondiente 30 veces hasta agotar todas las permutaciones, el mejor resultado sobre la media de las 30 pruebas para cada permutación permite determinar el mejor valor $v_1^{mejor} \in F_1$ del primer parámetro.

Una vez fijado el mejor valor del primer parámetro $v_1^{mejor} \in F_1$, se procede a generar las permutaciones sobre los valores $v \in F_2$, mientras se mantiene fijo el primer elemento de los parámetros restantes ($p_s = \{v_1^{mejor} \in F_1, v \in F_2, , ..., F_k^1\}$), para poder determinar el mejor valor $v_2^{mejor} \in F_2$ del segundo parámetro, una vez determinado el mejor valor del segundo parámetro, se continúa sucesivamente fijando los parámetros restantes hasta fijar el último parámetro $v_k^{mejor} \in F_k$ del parámetro k, para finalmente tener la permutación que estadísticamente da los mejores resultados en el proceso de experimentación con base en la ecuación 5.1, mostrada anteriormente, en general el número de permutaciones que necesita evaluar el algoritmo para determinar la mejor permutación viene dado por la siguiente ecuación 5.2.

$$p_{total} = |F_1| + |F_2| + \ ... \ + |F_k| \tag{5.2}$$

En la forma secuencial, el número total de permutaciones mostradas en la ecuación 5.2, para fijar un conjunto de parámetros con base en la ecuación 5.1, deben ser evaluadas por un solo núcleo de una máquina cualquiera, mientras que en una forma distribuida, deben ser evaluadas por un conjunto de núcleos distribuido en un conjunto de máquinas, existiendo varias formas de llevar a cabo la sintonización de los parámetros.

La forma más básica consiste en escribir las permutaciones directamente en el código fuente del algoritmo, compilarlo para cada permutación generada y ejecutarlo un total de 30 veces en forma manual, para cada permutación sobre un solo núcleo o sobre un conjunto de núcleos, podemos observar que es una forma muy ineficaz y muy lenta, requiere nuestra asistencia en todo momento a afectos de llevar a cabo las modificaciones del código fuente, compilarlo y ejecutar el algoritmo.

Rápidamente podemos intuir que la primera mejora consiste en modificar el código fuente y agregarle un ciclo principal que ejecute el algoritmo un total de 30 veces, de esta forma solo tendríamos que llevar a cabo una ejecución del algoritmo para cada

permutación, tomando en cuenta que al hacer las permutaciones de los parámetros en forma manual hay muchas probabilidades de equivocarnos.

Una mejora que se propone para evitar equivocaciones al escribir directamente en el código fuente las permutaciones y que no requiera nuestra asistencia en todo momento, es generarlas automáticamente por un proceso independiente y pasar los parámetros de la permutación al algoritmo, el cual los recibe como argumentos de entrada como se muestra en la figura 5.1, para evitar copiar y pegar los resultados de la salida estándar, estos son redirigidos a un archivo de salida incremental, finalmente para evitar modificar y agregar un ciclo principal que altere el comportamiento original del algoritmo, el generador de permutaciones llama a ejecución 30 veces al algoritmo con los parámetros de la permutación correspondiente.

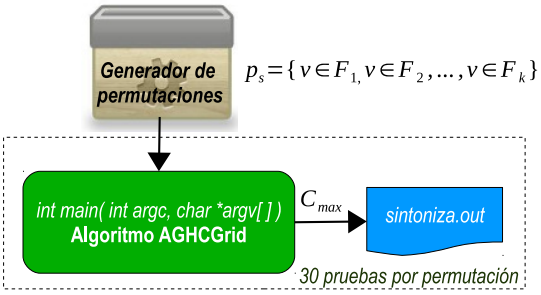

Figura 5.1. *Generación de permutaciones y ejecución. Para sintonizar un conjunto de 4 parámetros, como es el caso de RS se tiene por ejemplo que* $F = \{t_0, m, \mu, t_f\}$*, donde* $t_0 = \{50, 25\}$*,* $m = \{100, 50, 20\}$*,* $\mu = \{0.98, 0.985, 0.90\}$ *y* $t_f = \{0.1, 0.001, 0.001\}$ *el total de permutaciones requeridas para sintonizar los parámetros es* $p_{total} = |t_0|+|m|+|\mu|+|t_f| = |2|+|3|+|3|+|3| = 11$*, el generador de permutaciones pasa los parámetros permutados del primer parámetro como argumentos de entrada al algoritmo que escribe los resultados de la ejecución en un archivo incremental además de llamar a ejecución un total de 30 veces por cada permutación, después se procede con el segundo parámetro, y así sucesivamente.*

Cuando el generador completa todas las permutaciones del primer parámetro $v \in F_1$, el archivo de salida incremental contiene un listado de todas las permutaciones con sus respectivos valores de los parámetros y los resultados de las 30 pruebas como se muestra en el apéndice C, las características del archivo de salida contiene los siguientes datos:

1. Un encabezado con comentarios.

2. Fecha, hora de inicio de la sintonización y nombre de la máquina donde se ejecuta.

3. Número de permutación, directorio de trabajo, nombre del algoritmo y la permutación de los parámetros a evaluar.

4. 30 pruebas enumeradas del 1 al 30 en donde cada línea contiene:

   a) Número de prueba.

b) Tiempo de ejecución en segundos.

c) Semilla utilizada para generar los números aleatorios para efectos de recrear la experimentación posteriormente.

d) Valor de la función objetivo C_{max}.

5. Número total de pruebas efectuadas, fecha y hora de término.

Para poder determinar la mejor permutación con base en los resultados escritos en este archivo de salida, se propone crear un proceso independiente que lleva a cabo el análisis de los resultados, el cual consiste en leer el archivo de salida, calcular la media de las 30 pruebas para cada permutación, y seleccionar el parámetro correspondiente a la mejor media que corresponde al mejor valor $v_1^{mejor} \in F_1$ de las permutaciones efectuadas, mientras se mantienen fijo el resto de los parámetros al primer elemento, este método se utiliza para obtener la solución parcial $p_s = \{v_1^{mejor} \in F_1, F_2^1, ..., F_k^1\}$.

Una vez fijado el primer parámetro, se procede a fijar el segundo parámetro $v \in F_2$, utilizando el generador de permutaciones que pasa como argumentos de entrada los valores permutados del segundo parámetro mientras mantiene fijo el mejor valor encontrado para el primer parámetro y los demás restantes de la forma $p_s = \{v_1^{mejor} \in F_1, v \in F_2, , ..., F_k^1\}$, el generador de permutaciones llama a ejecución un total de 30 veces por cada permutación al algoritmo, el cual escribe cada vez los resultados obtenidos de la ejecución en el archivo de salida en forma incremental.

Una vez agotadas todas las permutaciones del parámetro, se procede de igual forma a determinar con base en la media de las 30 pruebas, la mejor media que corresponde a un valor $v \in F_2$ y entonces fijar el segundo mejor parámetro $v_2^{mejor} \in F_2$ y obtener la solución parcial $p_s = \{v_1^{mejor} \in F_1, v_2^{mejor} \in F_2, ..., F_k^1\}$, el generador de permutaciones continúa fijando los demás parámetros hasta fijar el último parámetro k para finalmente obtener la mejor permutación que estadísticamente da los mejores resultados $p_s = \{v_1^{mejor} \in F_1, v_2^{mejor} \in F_2, ..., v_k^{mejor} \in F_k\}$ con base en la ecuación 5.1.

Para separar los resultados con base en el parámetro sintonizado, se propone escribir tantos archivos de salida como parámetros a sintonizar existan, anteponiendo al nombre del archivo de salida, el nombre del parámetro, esto con el fin de no mezclar los resultados de todas las permutaciones en un solo archivo y facilitar el proceso de análisis de los resultados como se muestra en la figura 5.2.

De esta forma, para una secuencia de sintonización de parámetros: temperatura(t_0), markov(m), factor(μ), frozen(t_f), se generan los archivos de salida: *temperatura.out*, *markov.out*, *factor.out* y *frozen.out*, en donde cada archivo contiene los resultados de la evaluación de la función objetivo, al terminar de evaluar el conjunto de permutaciones, un proceso analiza los resultados y obtiene la mejor permutación para fijar el parámetro correspondiente y pasar al siguiente.

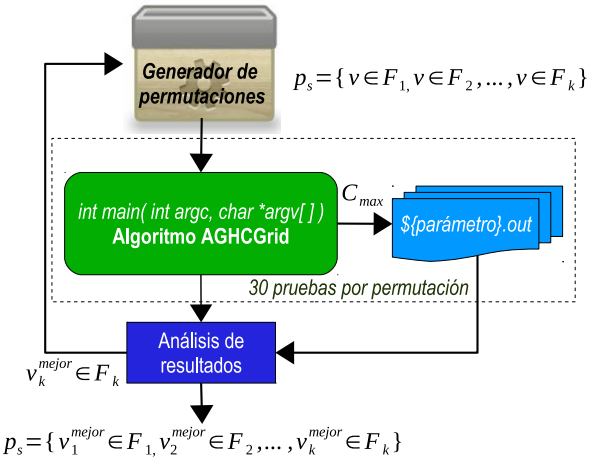

Figura 5.2. *Sintonización secuencial de parámetros. Para sintonizar un conjunto de 4 parámetros* $F = \{t_0, m, \mu, t_f\}$*, donde* $t_0 = \{50, 25\}$*,* $m = \{100, 50, 20\}$*,* $\mu = \{0.98, 0.985, 0.90\}$ *y* $t_f = \{0.1, 0.001, 0.001\}$*, se generan las permutaciones* $\{50, 100, 0.98, 0.1\}$ *y* $\{25, 100, 0.98, 0.1\}$*, el generador de permutaciones llama a ejecución 30 veces al algoritmo AGHCGrid por cada permutación, el cual recibe como argumentos de entrada los valores de la permutación, que escribe los resultados en forma incremental en un archivo que comienza con el nombre del parámetro a sintonizar, al agotar todas las permutaciones del primer parámetro se calcula la mejor media a partir del archivo de resultados y se fija, para el caso de que* $v_1^{mejor} = 50$*, se repite el proceso y se generan las permutaciones del segundo parámetro* $\{50, 100, 0.98, 0.1\}$*,* $\{50, 50, 0.98, 0.1\}$ *y* $\{50, 120, 0.98, 0.1\}$*, este proceso se repite hasta fijar todos los parámetros y obtener la mejor permutación* $p_s = \{v_1^{mejor} \in F_1, v_2^{mejor} \in F_2, ..., v_k^{mejor} \in F_k\}$ *que estadísticamente da los mejores resultados.*

5.2. Sintonización Distribuida Automática Aplicada en Paralelo.

La sintonización distribuida automática aplicada en paralelo (SDAAP) propuesta consiste en distribuir de manera uniforme, las permutaciones de un conjunto de parámetros $F = \{s_1, s_2, ..., s_f\}$ a sintonizar, en donde cada parámetro k con $k \in F$ esta compuesto por una serie de valores $F_k = \{v_1, v_2, ..., v_{f_k}\}$, sobre un cluster o una Grid, para la cual existe un conjunto de nodos r con su respectivo número de núcleos t de procesamiento $G = \{r_1 : t_1, r_2 : t_2, ..., r_n : t_n\}$. De la misma forma que en la sección anterior, el problema es determinar acorde a la función objetivo la permutación p_s que obtenga los mejores resultados en la fase de experimentación con base en la ecuación 5.1.

Existen dos problemas al intentar sintonizar en forma distribuida los parámetros sobre un cluster o una Grid: El primero es realizar el proceso de distribución uniforme de las permutaciones de cada parámetro, el segundo es como sincronizar los procesos una vez que se agotan todas las permutaciones de un parámetro para tomar el mejor valor y continuar la sintonización del siguiente.

Para ejemplificar el primer caso: Supongamos que queremos sintonizar en for-

ma distribuida un conjunto de 4 parámetros $F = \{t_0, m, \mu, t_f\}$ para RS, donde $t_0 = \{25, 50, 75, 100\}$, $m = \{10, 20, 30\}$, $\mu = \{0.98, 0.985, 0.988, 0.900\}$, $t_f = \{0.1, 0.001, 0.001\}$ y que los recursos de un cluster o Grid son $G = \{1:1, 2:1, 3:2\}$, es decir el nodo 1 tiene 1 núcleo, el nodo 2 un núcleo y el nodo 3 dos núcleos, en total 4 núcleos para llevar a cabo una distribución uniforme, es necesario repartir las 4 permutaciones $v \in F_1$ del primer parámetro $\{25, 10, 0.98, 0, 1\}$, $\{50, 10, 0.98, 0, 1\}$, $\{75, 10, 0.98, 0, 1\}$ y $\{100, 10, 0.98, 0, 1\}$ sobre los 4 núcleos disponibles para para fijar el primer parámetro $v_1^{mejor} \in F_1$, para llevar a cabo este proceso se derivan tres tipos de distribución:

1. Una distribución perfecta, es cuando existe el mismo número de procesadores como permutaciones de cierto parámetro a sintonizar sobre el cluster o Grid.

2. Una distribución sobrecargada, es cuando existen más permutaciones que procesadores sobre el cluster o Grid, en este caso se reparten las primeras permutaciones sobre los procesadores disponibles y conforme se van desocupando se van asignando las permutaciones restantes hasta agotarse, en cuyo proceso algunos procesos terminaran antes que otros y es aquí donde se ve reflejado con más énfasis el proceso de sincronización para determinar cuando se ha terminado de repartir todos las permutaciones y cuando todos los procesos ya han terminado.

3. Una distribución sobrada, es cuando hay más procesadores disponibles que permutaciones, en este caso los procesadores que no reciben permutaciones deberán esperar ociosos a recibir una permutación del siguiente parámetro a sintonizar.

La distribución de permutaciones sobre el conjunto G de núcleos para el primer parámetro $v \in F_1$ es perfecta, asigna procesos en una relación uno a uno de la forma ($nodo$: $nucleo$: $permutación$) como se observa en la figura 5.3, quedando repartidas de la siguiente manera: $(1:1:\{25, 10, 0.98, 0, 1\})$, $(2:1:\{50, 10, 0.98, 0, 1\})$, $(3:1:\{75, 10, 0.98, 0, 1\})$, $(3:2:\{100, 10, 0.98, 0, 1\})$, procediendo inmediatamente la ejecución del algoritmo a sintonizar en cada nodo y núcleo asignado con su valor correspondiente un total de 30 veces, si la distribución fue sobrecargada, los valores restantes de F_1 quedan en espera de que terminen los actuales asignados y se procede nuevamente a distribuir los valores hasta terminar.

Cuando se termina de distribuir y evaluar todas las permutaciones de primer parámetro $v \in F_1$, se procede a seleccionar el mejor valor sobre la media de las pruebas realizadas, supongamos que los mejores resultados se consiguen con el parámetro $v_1^{mejor} = 100$, donde $v_1^{mejor} \in F_1$, por tanto este valor queda fijo y se procede de igual forma a distribuir los parámetros de $v \in F_2$ que resultan en un distribución sobrada donde el núcleo 2 del nodo 3 permanece en espera, la distribución es como sigue: $(1:1:\{100, 10, 0.98, 0, 1\})$, $(2:1:\{100, 20, 0.98, 0, 1\})$, $(3:1:\{100, 30, 0.98, 0, 1\})$, $(3:2:\{\emptyset\})$, se procede de igual forma que el paso anterior, a determinar el mejor parámetro $v_2^{mejor} \in F_2$ sobre la media de las 30 pruebas, y así se continúa sucesivamente hasta fijar el último parámetro $v_k \in F_k$.

La manera de aprovechar de forma adecuada todos los núcleos es asegurarse que

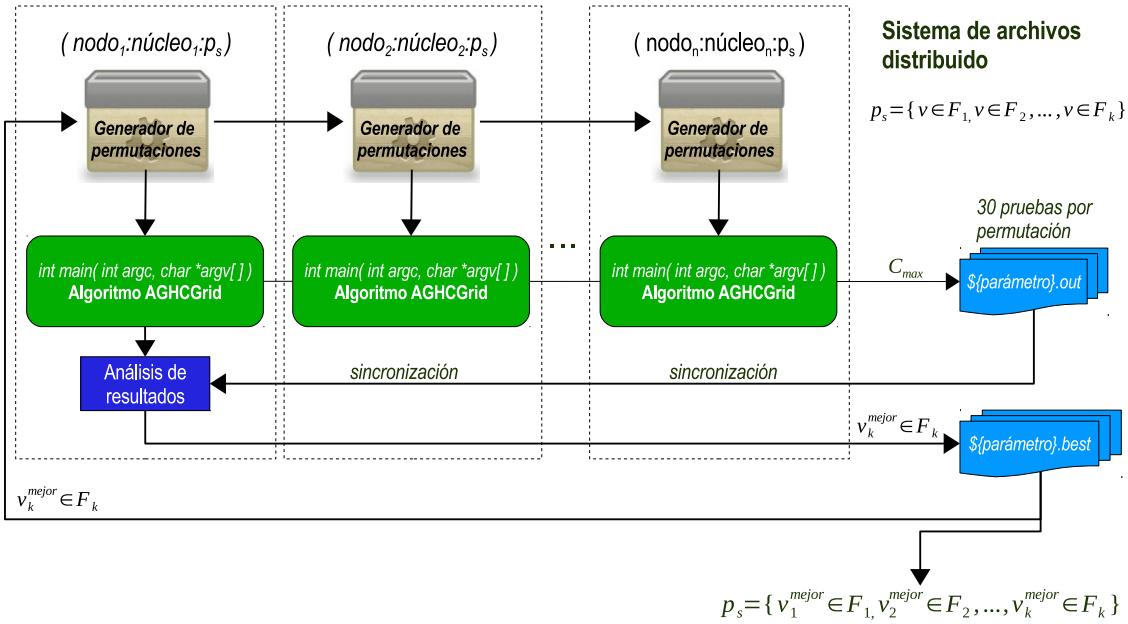

Figura 5.3. *Sintonización distribuida automática aplicada en paralelo SDAAP. Sobre un cluster o Grid se distribuyen los valores de los parametros a sintonizar de la forma (nodo : nucleo : permutación), en cada núcleo, de cada nodo, se evaluan las permutaciones y los resultados se escriben sobre el sistema de archivos distribuido, al agotarse las permutaciones de un párametro, el proceso maestro selecciona el mejor valor y repite el proceso para hasta agotar todos los parámetros.*

puede llevarse a cabo una distribución perfecta, de lo contrario al final se reparten menos valores que núcleos disponibles, lo que conlleva a tener núcleos en espera.

Otra manera de ver la distribución de permutaciones, es que cada núcleo de cada nodo recibe del generador de permutaciones un conjunto de parámetros que debe de evaluar secuencialmente, en donde cada proceso que se ejecuta es de la forma (*nodo* : *nucleo* : *permutación*). Para poder hacer una diferenciación entre los resultados que generan los procesos de diferentes nodos y núcleos, se asocia al nombre del archivo de resultados el nodo, núcleo y parámetro que se evaluó, así para el ejemplo anterior se generan los siguiente archivos de resultados: {1_1_,2_1_,3_1_,3_2_}*temperatura.out*, {1_1_,2_1_,3_1_,3_2_}*markov.out*, {1_1_,2_1_,3_1_,3_2_}*factor.out* y {1_1_,2_1_,3_1_,3_2_}*frozen.out,* de esta forma es posible determinar para cada parámetro evaluado en que nodo y núcleo se obtuvo el resultado.

Para el segundo caso que consiste en determinar cuando se ha terminado de evaluar las permutaciones distribuidas a los nodos y núcleos para determinado parámetro, con el fin de de calcular y fijar el mejor, se hace uso del sistema de archivos distribuido, en donde se provee de un único espacio de almacenamiento común para todos los nodos del cluster o Grid, es decir, que todos los archivos de resultados asociados a los procesos, en donde cada proceso escribe los resultados, de cada una de las 30 pruebas línea a línea, como se muestra en el apéndice C, puede ser visto por los demás nodos con independencia

de su ubicación, gracias al sistema de archivos distribuido, de esta manera cada proceso puede leer el archivo de resultados de los demás procesos y determinar el mejor valor una vez que todos han escrito sus respectivos resultados en archivos.

Para que el generador de permutaciones pueda determinar cual es el mejor valor, para un determinado parámetro y con base en él, puede generar las permutaciones del siguiente parámetro, éste debe esperar a que todos los procesos en ejecución terminen y escriban su archivo de resultados correspondiente, después designa un solo proceso (tradicionalmente el cero) para que encuentre el mejor valor de entre el conjunto de archivos asociado a la evaluación de ese parámetro y con este valor escriba un segundo archivo que será usado por el generador de permutaciones para fijar el parámetro actual y pueda continuar generando las permutaciones del siguiente parámetro.

Cada proceso asociado a un nodo y núcleo, escribe un archivo de control cuando esté termina, así el generador de permutaciones determina el momento exacto, en el que los demás procesos han terminado mediante el recuento del número de archivos escritos por los procesos asociados al actual parámetro, así un número de archivos igual al número de núcleos significa que ya se puede proceder a fijar el mejor valor. Una vez fijado un parámetro, este proceso se repite hasta fijar todos los parámetros restantes.

Capítulo 6

RESULTADOS EXPERIMENTALES

Este capítulo de la tesis se centra en la experimentación del algoritmo AGHCGrid, con el propósito de mostrar su desempeño en cuanto a su eficiencia y eficacia, utilizando la plataforma de producción Grid Morelos. Primeramente describimos en la sección 6.1 la plataforma de producción Grid Morelos usada y la creación de las instancias de prueba en la sección 6.2, después se muestra el análisis de sensibilidad sobre la Grid en la sección 6.3 que comprende la secuencia de sintonización y finalmente se muestra como se lleva a cabo la aplicación de la sintonización distribuida automática aplicada en paralelo (SDAAP) en la sección 6.3.1.

Después se muestra la eficacia del algoritmo sobre la Grid en la sección 6.4, que comprende la planeación para el anclaje de núcleos en la sección 6.4.2, la planeación de la distribución de los procesos en la sección 6.4.2, la eficacia automática aplicada en paralelo en múltiples instancias en la sección 6.4.3, el análisis de la eficacia de la cooperación de los procesos en la sección 6.4.4, el cual comprende el análisis de la instancias pequeñas, medianas y grandes en las secciones 6.4.4.1, 6.4.4.2 y 6.4.4.3 respectivamente.

Finalmente se muestra la eficiencia del algoritmo AGHCGrid sobre la Grid en la sección 6.5, el cual comprende la preparación del anclaje de núcleos en la sección 6.4.1, el análisis de la eficiencia mediante el cálculo de la aceleración (Speedup) y su eficiencia en la sección 6.5.1 y el cálculo de la sobrecarga en la sección en la sección 6.5.2.

6.1. Plataforma de producción Grid Morelos.

La plataforma de producción Grid Morelos, la cual es a la fecha la primera Grid que se encuentra en funcionamiento en el país [Cruz et al., 2012a], se define como una

Grid multi-cluster heterogénea de cómputo intensivo, está integrada por tres clusters de alto rendimiento que se encuentran geográficamente dispersos, pero unidos a través de I2, como se muestra en la tabla 6.1, donde $GRID = \{cluster_{cuexcomate}, cluster_{texcal}, cluster_{nopal}\}$, el primero se encuentra localizado en el CIICAp[1], el segundo en la UPEMOR[2] y el tercero en el ITV[3], los primeros dos clusters son idénticos y el último diferente en número de nodos, procesadores y núcleos. Los recursos totales comprenden 36 procesadores, 156 núcleos de procesamiento, 325.5 GB de memoria RAM, 31 TB de almacenamiento distribuido y 1792 núcleos en 4 tarjetas NVIDIA Tesla Fermi, la distribución por cluster se describe a continuación[4].

El cluster del CIICAp esta conformado por un nodo maestro y cinco nodos esclavos $cuexcomate = \{nodo01, nodo02, nodo03, nodo04, nodo05, nodo06\}$, los recursos que a continuación se describen se encuentran distribuidos desde el nodo01 hasta el nodo06 respectivamente: 11 procesadores $proc = \{2, 2, 2, 2, 2, 1\}$, 66 núcleos $n = \{12, 12, 12, 12, 12, 6\}$, 158 GB de RAM $mem = \{24, 24, 24, 24, 24, 36\}$, 15 TB de almacenamiento $hd = \{12_{TB}, 0.5_{TB}, 0.5_{TB}, 0.5_{TB}, 0.5_{TB}, 1_{TB}\}$, en el nodo06 se encuentran 896 núcleos en dos GPU's Tesla Fermi con 448 núcleos cada uno, dos tipos de comunicaciones Gigabit Ethernet e Infiniband a 40Gbs. El cluster Texcal de la Upemor tiene exactamente las mismas características que el cluster Cuexcomate del CIICAP.

El cluster del ITV esta conformado por un nodo maestro y trece nodos esclavos $nopal = \{nodo01, nodo02, ..., nodo14\}$, los recursos que a continuación se describen se encuentran distribuidos desde el nodo01 hasta el nodo14 respectivamente: 14 procesadores $proc = \{1, 1, ..., 1\}$, 28 núcleos $n = \{2, 2, ..., 2\}$, 13.5 GB de RAM $mem = \{0.5, 1, 1, ..., 1\}$, 1 TB de almacenamiento $hd = \{60_{GB}, 80_{GB}, 80_{GB}, ..., 80_{GB}\}$, comunicaciones Gigabit Ethernet.

GRID MORELOS											
CLUSTER CUEXCOMATE (66 Núcleos)						CLUSTER TEXCAL (66 Núcleos)					
Nodo01 12 núcleos	Nodo02 12 núcleos	Nodo03 12 núcleos	Nodo04 12 núcleos	Nodo05 12 núcleos	Nodo06 6 núcleos	Nodo01 12 núcleos	Nodo02 12 núcleos	Nodo03 12 núcleos	Nodo04 12 núcleos	Nodo05 12 núcleos	Nodo06 6 núcleos
CLUSTER NOPAL (30 Núcleos)											
Nodo01 2 núcleos	Nodo02 2 núcleos	Nodo03 2 núcleos	Nodo04 2 núcleos	Nodo05 2 núcleos	Nodo06 2 núcleos	Nodo07 2 núcleos	Nodo08 2 núcleos	...	Nodo14 2 núcleos		

Tabla 6.1. *Recursos de la Grid Morelos. La plataforma Grid Morelos, se define como una Grid heterogénea del alto rendimient,o debido a que esta conformada por nodos con diferente arquitectura de procesador, velocidad y comunicaciones.*

El resultado de esta hibridación en el número de nodos por cluster, número de procesadores por nodos, número de núcleos por procesador, velocidades y arquitectura

[1] Centro de Investigaciones en Ingeniería y Ciencias Aplicadas
[2] Universidad Politécnica del Estado de Morelos
[3] Instituto Tecnológico de Veracruz
[4] Información obtenida del sitio http://www.gridmorelos.uaem.mx:8080

de 32 y 64 bits, es una plataforma Grid heterogénea formada por clusters de multicomputadoras simétricas [Quinn, 2004], la cual es una situación más real de entornos de producción debido a la dificultad de conseguir recursos idénticos. Por tal razón en la Grid Morelos conviven recursos de diferente capacidad, es decir recursos que a lo largo del tiempo se han anexado a la Grid para incrementar su capacidad.

6.2. Generación aleatoria de instancias de prueba.

Con base en [Ruiz and Stutzle, 2008; Ruiz and Vázquez-Rodríguez, 2010] se codificó en C un programa como se muestra en el apéndice D, para generar instancias aleatorias de prueba mostradas en la tabla 6.1, de la siguiente manera:

- 5 grupos de trabajos $N = \{20, 40, 60, 80, 140\}$, 3 grupo de etapas $K = \{2, 4, 8\}$.
- 4 grupos de máquinas $M = \{2, 3, 4, 5\}$.
- Tiempos de procesamiento p_{ij} uniformemente distribuidos entre [1,99].
- Tiempo se iniciado dependientes de la secuencia S_{ij}, uniformemente distribuidos entre [1,25], [1,50], [1,100] y [1,125] correspondientes al 25 %, 50 %, 100 % y 125 % de radio con respecto a los tiempos de procesamiento.

En total 5x3x4x4 = 240 combinaciones y de cada combinación se generaron 5 instancias, en total 1200 instancias (archivos).

Las 1200 instancias se generaron automáticamente en modo texto e incluye encabezados de autoría, descripción del sistema y etapas como se muestra en el apéndice B.

Instancia	N	K	M	p_{ij}	S_{ij}	#	Total
1	{20}	{2,4,8}	{2,3,4,5}	{[1,99]}	{[1,25],[1,50],[1,100],[1,125]}	5	240
2	{40}	{2,4,8}	{2,3,4,5}	{[1,99]}	{[1,25],[1,50],[1,100],[1,125]}	5	240
3	{60}	{2,4,8}	{2,3,4,5}	{[1,99]}	{[1,25],[1,50],[1,100],[1,125]}	5	240
4	{80}	{2,4,8}	{2,3,4,5}	{[1,99]}	{[1,25],[1,50],[1,100],[1,125]}	5	240
5	{140}	{2,4,8}	{2,3,4,5}	{[1,99]}	{[1,25],[1,50],[1,100],[1,125]}	5	240
					Gran total		1,200

Figura 6.1. *Instancias de prueba. Para cada una de las instancias de prueba generadas, que van de 1 a 5, el número total de instancias se obtiene al combinarlas con el número de etapas que son 3, por el número de máquinas por cada etapa que son 4 y por el número de los tiempos de inicio dependientes de la secuencia que son 4, finalmente por cada combinación se crean 5 instancias.*

Lo anterior fue resultado la revisión hecha por [Ruiz and Vázquez-Rodríguez, 2010] que nos indica que la mayoría de los autores usa instancias generadas aleatoriamente

debido a que no existen benchmarks, otros más se enfocan en problemas reales de la industria por lo que sus datos son específicos para determinada problemática real y aunado a las muchas variantes del FFS, se concluye que no encontramos benchmarks que se ajusten a nuestro problema. Por lo anterior se procedió a generar nuestras propias instancias con base en los tamaños que los autores han estado resolviendo, los cuales son: problemas entre 50 y 80 operaciones solamente, menor a 15 o 20 trabajos en máximo 5 etapas [Ribas et al., 2010].

La nomenclatura usada para nombrar las instancias de prueba fue FFS_SDST_nxmxkxr_i, en donde:

- FFS_SDST se refiere al tipo del problema (*del inglés Flexible Flow Shop with Sequence Dependent Setup Times*),
- n es el número de trabajos,
- m el número de etapas,
- k el número de máquinas paralelas por etapa,
- r el radio de los tiempos de inicio dependientes de la secuencia con respecto a los tiempos de procesamiento e i el número problema que va del 1 al 5.

Del total de instancias generadas, se tomaron 5 representativas. Los resultados en la sintonización de cada una de las instancias, se muestran en la sección 6.3.2 y los resultados experimentales de cada una de las instancias se muestran en la sección 6.4, finalmente la acelaración (Speedup) de una instancia representativa se muestra en la sección 6.5.

6.2.1. Clasificación de las instancias de prueba.

Las instancias creadas con base en [Ruiz and Stutzle, 2008; Ruiz and Vázquez-Rodríguez, 2010] en la sección anterior 6.2, se clasifican en instancias pequeñas {1,2}, medianas {3,4} y grandes {5} mostradas en la tabla 6.2.

Instancia	Nombre	Tamaño
1	FFS_20x4x4x25	chica
2	FFS_40x4x4x25	
3	FFS_60x4x4x25	mediana
4	FFS_80x4x4x25	
5	FFS_140x4x4x25	grande

Figura 6.2. *Clasificación de las instancias. Las instancias propuestas y generadas de forma aleatoria, se clasifican en pequeñas, medianas y grandes, con base al tratamiento que se le da actualmente en la literatura.*

Las instancias pequeñas son tratadas incluso con métodos exactos y sirven de referencia para iniciarse en el mundo de la optimización, las instancias medianas, son las que actualmente se tratan en la comunidad científica, y las instancias grandes, son para las cuales existe un nicho de oportunidad, debido a que solo pueden tratarse con ayuda del computo de alto rendimiento, como el usado en la presente tesis doctoral.

6.3. Análisis de sensibilidad en la Grid.

Cuando se término de diseñar y programar el algoritmo paralelo AGHCGrid, los coeficientes de control (parámetros) que le dan vida y comportamiento al algoritmo, fueron sintonizados, es decir, se llevó a cabo un estudio del comportamiento del algoritmo, al realizar cambios en los parámetros, con el objetivo de encontrar aquellos valores que estadísticamente produjeron los mejores resultados en la salida del algoritmo, esto es, que mejoran la eficacia del algoritmo. Los rangos de valores asignados a cada variable de control, fueron aquellos en los cuales la calidad de los resultados siguió siendo buena, además de que nos permitió saber, que tan sensible es el algoritmo a cambios en los valores de los parámetros.

La metodología utilizada para llevar a cabo esta actividad, fue una sintonización distribuida automática explicada en el capítulo 5, en esta sección, se muestra como se aplica esta metodología, para realizar los trabajos del análisis de sensibilidad sobre la plataforma de producción Grid Morelos, las actividades comprenden la secuencia de sintonización de los parámetros del algoritmo AGHCGrid, estos parámetros comprenden tres metaheurísticas que son: Sistema de colonia de hormigas, recocido simulado y algoritmo genético. Así mismo se muestra como la ejecución en paralelo de los procesos de sintonización sobre la Grid, permite reducir los tiempos y abarcar un rango mayor de valores asociados a cada parámetro, finalmente se muestra los resultados del análisis de sensibilidad y la convergencia.

A continuación se muestra como se aplica la metodología explicada en el capítulo 5 para realizaron los trabajos del análisis de sensibilidad sobre la plataforma de producción Grid Morelos.

6.3.1. Implementación de la Metodología de Sintonización Distribuida Automática Aplicada en Paralelo (SDAAP).

El algoritmo AGHCGrid 4.1, mostrado en la figura 4.2, está conformado por un algoritmo genético cooperativo que combina, los esfuerzos del sistema de colonia de hormigas y recocido simulado, estas tres metaheurísticas están compuestas por un grupo de parámetros que son:

- $SCH = (h, \alpha, \beta, \gamma, \delta, p, q)$, donde h es el número de hormigas, α es el coeficiente

de importancia *alfa*, β es el coeficiente de importancia *beta*, γ es el coeficiente de evaporación *gamma* para la transición local, δ es el coeficiente de evaporación *delta* para una transición global, q es el coeficiente que divide la exploración/explotación y p es el criterio de paro.

- $RS = (t_o, m, \mu, t_f)$, donde t_o es la temperatura inicial, m es el número de veces que repite la longitud de la cadena de markov, μ es el coeficiente de enfriamiento y t_f es la temperatura final.

- $AG = (T, C, G)$, donde T es el tamaño de la población, que es directamente proporcional al número de procesos usados, C la tasa de cruce y G el número de generaciones.

Finalmente la secuencia de sintonización del los parámetros sigue el siguiente orden: $h \to \alpha \to \beta \to \gamma \to \delta \to p \to q$, $t_o \to m \to \mu \to t_f$ y $G \to C \to T$. En lo que respecta a la secuencia de sintonizado de los métodos, SCH y RS se sintonizan primero, estos dos métodos por ser independientes pueden sintonizarse en paralelo, finalmente se sintoniza AG.

Del conjunto de instancias de prueba generados aleatoriamente en la sección 6.2, se tomaron 5 instancias con diferente número de trabajos y tiempos de inicio dependientes de la secuencia S_{ljk}, conservando el mismo número de etapas m, número de máquinas en paralelo k, como se observa en la tabla 6.2. La metodología usada para sintonizar se compone de 5 pasos y se describe a continuación.

INSTANCIA	DESCRIPCIÓN
FFS_SDST_20x4x4x4x25	n=20,m=4,k=4,S_{ij}=25%
FFS_SDST_40x4x4x4x25	n=40,m=4,k=4,S_{ij}=25%
FFS_SDST_60x4x4x4x25	n=60,m=4,k=4,S_{ij}=25%
FFS_SDST_80x4x4x4x25	n=80,m=4,k=4,Sij=25%
FFS_SDST_140x4x4x4x25	n=140,m=4,k=4,S_{ij}=25%

Tabla 6.2. *Instancias de prueba utilizados. Para el análisis de sensibilidad se tomaron en cuenta 5 instancias de prueba con n={20,40,60,80,140} trabajos y conservando el mismo número de etapas m=4, número de máquinas en paralelo por cada etapa k=4 e igual rango de los tiempos dependientes de la secuencia $S_{ij} = 25\,\%$, finalmente se tomo la primera instancia i=1 de las 5 creadas para cada combinación.*

1. Diseño y programación de los algoritmos secuenciales SCH y RS. Dado que el algoritmo AGHCGrid esta conformado por dos metaheurísticas SCH y RS, se requiere llevar a cabo la sintonización de estos dos algoritmos, la literatura muestra que existen dos formas de llevarlo a cabo que son:

   - Una sintonización independiente de la instancia. Esto es que para un conjunto

de instancias de diferente tamaño, solo se utiliza la más representativa del conjunto y se lleva a cabo el proceso de sintonización, entonces se utilizan estos mejores parámetros para todo el conjunto de instancias.

- Una sintonización dependiente de la instancia. Esto es que para un conjunto de instancias de diferente tamaño, para cada instancia del conjunto se lleva a cabo el proceso de sintonización, obteniendo tantos valores de parámetros como instancias hay en el conjunto.

Desde luego, es fácil observar que por comodidad en los tiempos de sintonización, muchos autores utilizan una sintonización independiente de la instancia, ya que de la segunda forma, para esté caso tendríamos que llevar a cabo 5 sintonizaciones, una por cada instancia de prueba mostrada en la tabla 6.2, más aún, considerando el hecho de que son dos métodos a sintonizar debemos considerar entonces 10 sintonizaciones.

También es fácil deducir, que la primera opción debe ser considerada si no se cuenta con una plataforma Grid, o un equipo de cómputo científico suficientemente poderoso como para llevar a cabo una sintonización dependiente de la instancia, de modo que para esté caso, utilizaremos la segunda forma. Una de las ventajas de contar con una plataforma Grid, es que nos permite multiplicar por el número de núcleos, los esfuerzos y reducir con ellos los tiempos en la misma proporción, por último, no olvidar que la complejidad de llevar a cabo tales tareas, sobre una plataforma Grid aumenta y es un precio en tiempo y dedicación, que no todos están dispuestos a pagar, pensando en ello, es que se decidió elaborar el mecanismo para llevar a cabo esta tarea en forma automática y paralela, como resultado de esa necesidad, se propone en el capitulo 5, la sintonización distribuida automática aplicada en paralelo (SDAAP).

2. Definir los rangos de evaluación para SCH y RS. Los rangos de evaluación encontrados en la literatura para RS en este tipo de problemas, muestran que utilizan un rango de valores que denominan bajos, medios y altos para los 4 parámetros (t_o, m, μ, t_f). Para el caso de SCH, los valores para el factor de importancia α y β fluctúan entre 0 y 5, para los factores de evaporación de feromona γ, δ y p, que es el factor de importancia entre llevar a cabo una exploración o una explotación, se muestran que los rangos están entre 0 y 1.

   Finalmente h y q, que son los dos factores más determinantes por ser el número de hormigas y el número de ciclos, muestran que el número de hormigas, pueden llegar a ser hasta de 200 y algunos autores lo igualan al número de trabajos. Para el caso del número de ciclos, se pueden manejar de más de más de 2000, pero que mucho depende de las capacidades del equipo. Algunos autores mantienen fijo $\alpha = 1$ [Kuo-Ching et al., 2007] y solo varían el factor de importancia β. Para el caso de RS después de realizar una primera sintonización, se pasa a una refinación de los rangos, en donde los rangos a evaluar, para las dos metaheurísticas, se muestran

en la tabla 6.3.

PARÁMETRO	SISTEMA DE COLONIA DE HORMIGAS - SERIE DE VALORES
Hormigas	h={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20}
Alfa	α={1}
Beta	β={0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0}
Feromona-L	γ={0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9}
Feromona-G	δ={0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9}
Expl-Expl	q={0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9}
Ciclos	p={500,750,1000,1250,1500,1750,2000}
PARÁMETRO	RECOCIDO SIMULADO - SERIE DE VALORES
Temperatura	t_o={1,10,20,30,40,60,50,70,80,90,100,110,120,140,160,180,200,220,240, 260,280,300,320,360,380,400,420,440,460,500}
Markov	m={1,2,3,4,5,6,7,8,9,10,12,14,16,18,20,22,24,26,28,30,34,38,42,46,50,55, 60,65,70,75}
Factor	ɥ={0.986,0.80,0.814,0.821,0.828,0.835,0.842,0.849,0.856,0.863,0.870, 0.877,0.884,0.891,0.898,0.905,0.912,0.919,0.926,0.933,0.940,0.947,0.954, 0.961,0.968,0.975,0.982,0.984,0.988,0.990}
Temperatura	t_f={0.8,0.6,0.4,0.2,0.01,0.08,0.06,0.04,0.03,0.02,0.001,0.009,0.008,0.007, 0.006,0.005,0.004,0.003,0.002,0.0001,1}

Tabla 6.3. *Los rangos utilizados para evaluar para SCH y RS, estuvieron basados en la literatura, el número máximo de hormigas es h=20 y el número de ciclos varia desde p=500 hasta p=2000 con incrementos de 500, puede apreciarse que se fue conservador en estos dos parámetros, que son los más determinantes para SCH. Para RS la serie de valores para cada parámetro, fue mucho más grande que SCH, en las pruebas experimentales previas, se observó el consumo de tiempo de ambos algoritmos y fue sobre éste tiempo, que se tomaron los rangos de valores, para que los tiempos de espera no fueran excesivos.*

Con respecto al tamaño de la serie de valores establecidos para cada parámetro, puede observarse que es la misma para todas las instancias de prueba, si bien la sintonización es dependiente de la instancia y esto hace que tengamos que hacer 10 sintonizaciones, también se deduce que para las instancias pequeñas los tiempos de espera son menores que las grandes, entonces se podrían alargar la serie de valores para las instancia pequeñas, a fin de hacer una mejor selección probabilística de los parámetros.

La manera de proceder fue la selección de la misma serie de parámetros para SCH y la misma serie de parámetros RS, de tal manera que la evaluación fuera bajo las mismas condiciones y que los tiempos de espera fueran los adecuados, para no incurrir en tiempos de espera muy largos, aún con todo el poder de procesamiento que tiene la Grid.

3. Aplicar el método SDAAP para SCH y RS. La aplicación del método propuesto en la sección 5.2, requiere de la configuración de una serie de valores especificados

en un archivo de configuración (.conf), los valores especificados aquí, corresponden a parámetros como se muestra en el apéndice E, el contenido de estos valores especifica:

a) El nombre de los nodos y el número de núcleos que se usarán en cada nodo de la Grid para llevar a cabo el proceso de sintonización.

b) Los nombres de los parámetros a sintonizar y la serie de valores de cada parámetro.

c) La secuencia de sintonización de los parámetros especificados, esto es, indicar que parámetro se sintoniza en primer lugar, cual en segundo, y así sucesivamente.

d) La mejor cota conocida UB, este parámetro es interesante, puesto que no hay cotas conocidas y este cota afecta directamente el comportamiento del algoritmo SCH, se utilizó la mejor cota conseguida con la sintonización de RS.

e) El archivo ejecutable que corresponde a SCH o RS ajustado para tratar una instancia de prueba como se muestra en el apéndice B, dependiente del tamaño de la instancia. El código fuente debe ser ajustado para manejar la instancia en particular, debe ser compilado y depositado en el directorio de trabajo, este proceso se repite tantas veces como instancias de prueba existan, en nuestro caso 5 para SCH y 5 para RS como se observa en la figura 6.3.

f) Un script en bash que integra los esfuerzos en toda la Grid como se muestra en el apéndice H, este script es el que resuelve los problemas de sincronización entre los diferentes procesos que están sintonizando a lo largo de la Grid, calcula la media de los resultados obtenidos, selecciona el mejor parámetro al final de las 30 pruebas y durante todo este proceso, escribe los resultados en archivos como se muestra en el apéndice C.

g) El directorio de trabajo donde el script escribirá los resultados de la sintonización.

h) El número de pruebas por cada variación de los parámetros a sintonizar que en esté caso esta fijado a 30.

La especificación de todos estos valores hace posible la sintonización distribuida automática aplicada en paralelo propuesta en la sección 5.2, para los algoritmos de sistema de colonia de hormigas, recocido simulado y el algoritmo genético tratados en esta tesis.

La forma en la cual se aplican y procesan estos parámetros sobre la Grid, se muestra

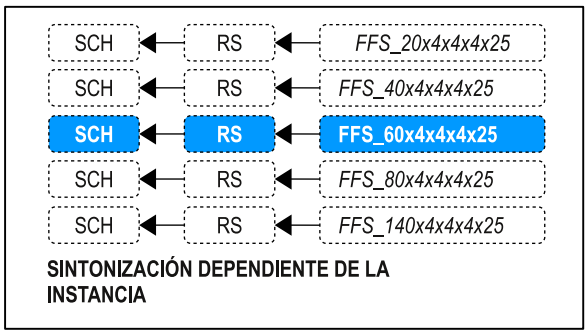

Figura 6.3. *Instancias de SCH y RS a sintonizar. El número de instancias de SCH y RS son 5, en total 10 sintonizaciones, las cuales se realizaron en forma independiente aplicando el método SDAAP desarrollado para este fin.*

en la figura 6.4, en la cual se muestra que primero se deposita, sobre el sistema de archivos distribuido, el archivo de configuración (.conf), el script maestro (.sh) y la aplicación ejecutable, que recibirá las combinaciones generadas por el script, una vez depositados, quedan disponibles para todos los nodos de la Grid.

La ejecución de los procesos sobre la Grid, corren a través de tunning.sh, que es el script mostrado en el apéndice H, que obtiene su comportamiento con base en los parámetros leídos del archivo .conf, como el mostrado en el apéndice E, después genera el conjunto de permutaciones, que es diferente para cada proceso p que corre en determinado *nodo:núcleo* en el tiempo i.

El conjunto de permutaciones es pasado a la aplicación ejecutable, configurada para tratar con determinada instancia uno a uno, los resultados obtenidos son escritos por el script maestro línea a línea en un archivo de salida .out, cuyo nombre permite identificar el parámetro, nodo y núcleo fuente donde fue procesado, al agotarse el conjunto de permutaciones, el proceso 1, calcula con base en la media el mejor parámetro y lo escribe en un archivo .best, que corresponde al nombre del parámetro que término de sintonizar en ese momento y lo deposita sobre el sistema de archivos distribuido en el tiempo $i + 1$.

Finalmente este mejor parámetro, es tomado por todos los demás procesos, para comenzar la sintonización del siguiente parámetro, hasta terminar con todos, en el tiempo $i + 2$. Un lanzador de procesos .sh, generado automáticamente a partir del archivo de configuración, como se muestra en el apéndice G, permite ejecutar en forma paralela los procesos sobre la Grid, teniendo especial cuidado, en que el balanceo de la carga de los procesos sea uniforme, como se muestra en el apéndice I, en donde cada proceso, tiene asignado el mismo tiempo.

Para el caso de RS, se volvió ha hacer este paso generando nuevos rangos acotados, p. ej. para la instancia FFS_20x4x4x25 en la primera refinación, con base en la serie de valores de los parámetros mostrados en la tabla 6.3, se obtienen los siguientes valores sintonizados: $t_o = 440$, $m = 75$, $\mu = 0.990$, $t_f = 0.001$, con estos valores se

crea una segunda serie de valores entre los valores $(inferior, superior)$, esto es que para temperatura la nueva serie de valores es $t_o = \{425, 430, 435, 445, 450, 455\}$, para la longitud de la cadena de markov $m = \{70, 71, 72, 73, 74, 76, 77, 78, 79, 80\}$ y así sucesivamente para los demás parámetros.

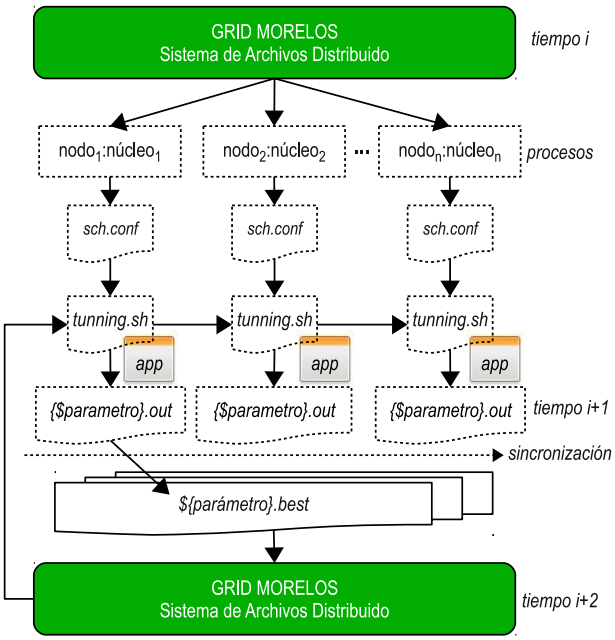

Figura 6.4. *Aplicación de la sintonización distribuida automática aplicada en paralelo (SDAAP). Este método permite sintonizar un grupo de parámetros sobre la Grid en forma automática, es capaz de distribuir los procesos de sintonización, coordinar e integrar los resultados en archivos para su análisis posterior.*

Todo el proceso es automático y concluye con la escritura de un número de archivos igual al número de parámetros a sintonizar, que contienen el valor óptimo, que por probabilidad obtuvo los mejores resultados, además de todas las evidencias de la experimentación, en archivos de resultados .out, tantos como procesos sobre la Grid se utilizaron. Solo en el caso de RS, se incluye una segunda refinación, creando una sub-serie acotada entre el valor inmediato inferior y superior del óptimo encontrado en la primera vuelta.

Finalmente este proceso se repite para las 4 instancias de pruebas restantes, tanto para SCH como RS, como se observa en la figura 6.3, previa creación de los archivos de configuración correspondientes y la compilación de los fuentes, para obtener el ejecutable de SCH y RS ajustado para trabajar con las nuevas instancias.

4. Definir los rangos para el algoritmo AGHCGrid. Los rangos de valores encontrados en la literatura [Ruiz and Maroto, 2006], consideran dos operadores de selección: ruleta y torneo. Para el operador de cruce consideran PMX (Partially Mapped Crossover), OP (One Point order crossover), TP (Two Point order crossover), OX (Order Crossover), entre otros. La tasa de cruzamiento está considerada en el

rango $[0.1, 0.5]$. El tamaño de la población se fija en el rango $[20, 50]$. Los rangos establecidos para el algoritmo los exponemos en la tabla 6.4.

PARÁMETRO	GAHCGrid - SERIE DE VALORES
Selección	S={ruleta}
Generaciones	G={10,12,14,16,18,20,22,24,26,28,30}
Tasa de cruce	T={0.10 0.20 0.30 0.40 0.50 0.60 0.70 0.8 0.9 1}
Población	P={20,25,30,35,40,45,50,55,60,65,70}

Tabla 6.4. *Los rangos utilizados para evaluar el AGHCGrid, estuvieron basados en la literatura, para la selección de los nuevos individuos, el algoritmo utiliza una selección de ruleta, el rango de número de generaciones que se consideran, están comprendidas en el rango* $[10, 30]$ *con incrementos de 0.10, la tasa de cruce esta en el rango* $[0.10, 1.0]$ *con incrementos de 0.10, finalmente el tamaño de la población esta en el rango* $[20, 75]$ *con incrementos de 5.*

5. Aplicar el método SDAAP para sintonizar AGHCGrid. Donde el AGHCGrid es un algoritmo genético (AG), cuyos parámetros se sintonizan después de sintonizar SCH y RS, para este caso, aplica una sintonización independiente de la instancia, como se observa en la figura 6.5, es decir, que solo se sintoniza para un tamaño representativo y se aplica al resto, esta manera contrasta con la sintonización dependiente de SCH y RS que se sintonizan para cada tamaño de instancia.

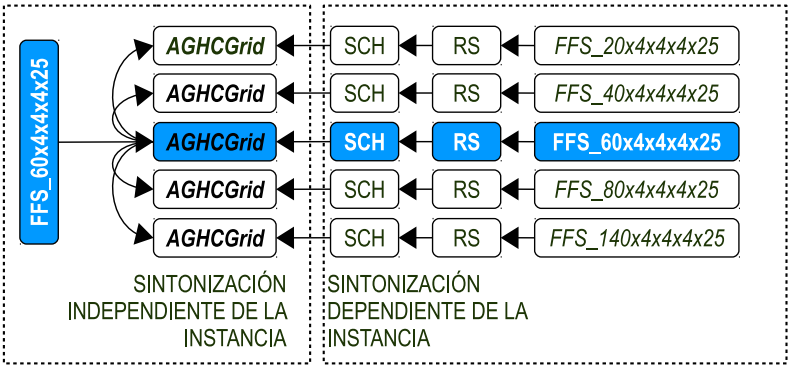

Figura 6.5. *Instancias a sintonizar AGHCGrid. El algoritmo genético es dependiente de la previa sintonización de SCH y RS, la misma instancia FFS_ 60x4x4x4x25, es usada para sintonizar las tres metaheurísticas. Se aplica una secuencia* $SCH \rightarrow RS \rightarrow AGHCGrid$*, el resultado de la sintonización del AG entonces es usado para los demás instancias, sin importar el tamaño, pero mantiene intactos los parámetros sintonizados de SCH y RS.*

Esta decisión fue motivada por los tiempos largos para las instancias grandes, ya que al combinarse el algoritmo genético con SCH y RS, los tiempos de sintonización se vuelven prohibitivos para las instancias grandes. Una vez establecidos los rangos en el archivo de configuración correspondiente, se aplica el método SDAAP, como lo indica el punto 3 descrito anteriormente, mientras mantiene los parámetros de SCH y RS como se explica en la figura 6.4.

6.3.2. Resultados del análisis de sensibilidad obtenidos vía SDAAP.

Los resultados para RS y SCH utilizaron dos clusters: Tarántula y Cuexcomate, de los cuales se utilizaron 14 núcleos del cluster Tarántula y 60 de los 66 núcleos del cluster Cuexcomate, dejando 6 núcleos para tareas administrativas en el nodo maestro. Antes de la existencia de la Grid Morelos como plataforma de producción solo existía el cluster Tarántula, los cálculos realizados en ese tiempo, para determinar las instancias manejables con base en la duración de sintonización que consumían, reflejan la selección de tres tamaños de instancias: 20, 40 y 60 trabajos.

Posteriormente con la puesta en marcha de la Grid Morelos se pudieron manejar instancias más grandes como resultado del uso del cluster Cuexcomate y la adición de dos nuevos tamaños de instancias: 80 y 140 trabajos, los dos clusters que integran la Grid Morelos (Cuexcomate y Texcal) estaban en proceso de integración, por lo que los trabajos de sintonización no hacen uso del cluster Texcal, hacia el final de la integración es cuando se utiliza por completo la Grid para el proceso de experimentación del algoritmo propuesto, ya con los parámetros adecuados para medir su eficacia y eficiencia en paralelo en la sección 6.4 y 6.5 respectivamente.

Los resultados encontrados aplicando el método SDAAP, se muestran en las tablas 6.5, 6.6 y 6.7 para sistema de colonia de hormigas, recocido simulado y AGHCGrid respectivamente, para RS y SCH se ejecutaron 30 pruebas para cada combinación de la instancia como se muestra en el paso 3 de la sección 6.3.1 y para AGHCGrid se ejecutaron 30 pruebas por la instancia representativa de tamaño medio como se muestra en el paso 5 de la sección 6.3.1.

Para las tres figuras: en la columna uno están los nombres de las instancias, en la columna dos los parámetros óptimos encontrados para esa instancia en particular, en la columna tres el cluster utilizado donde se llevaron a cabo las experimentaciones, en la cuarta columna el tiempo total computado para las 30 pruebas y en la quinta columna el tiempo calculado si se hubiera utilizado una técnica secuencial en solo un núcleo.

Los resultados de la sintonización de RS mostrados en la tabla 6.5, utilizan un tamaño diferente de núcleos: 14 y 60 para el cluster Tarántula y Cuexcomate respectivamente, en combinación con las diferentes velocidades y arquitecturas de los procesadores resultan en los tiempos computados de la sintonización para cada instancia. Podemos observar como impacta el número de procesadores en los tiempos de sintonizado. p. ej. para la instancia 60x4x4x25 al utilizar 14 núcleos del cluster Tarántula, se computa un tiempo de 17 horas 51 minutos y al compararlo con la siguiente instancia en tamaño 80x4x4x25 al utilizar el cluster Cuexcomate con 60 núcleos, se reduce el tiempo de sintonización en más de un 80 %, al computar un tiempo de 3 horas y 33 minutos, a pesar de ser una instancia mayor.

Finalmente con respecto a intentar utilizar un solo núcleo, vemos como el factor tiempo se multiplica por el número núcleos usados en paralelo y el tiempo calculado

INSTANCIA	RS(t_0,n,μ,t_f)	PLATAFORMA	NÚCLEOS	TIEMPO SDAAP	TIEMPO 1 NÚCLEO
20x4x4x25	(450,78,0.989,0.0015)	TARÁNTULA	14	01:18 HRS	18:12 HRS
40x4x4x25	(425,56,0.890,0.0001)	TARÁNTULA	14	10:59 HRS	153:46 HRS
60x4x4x25	(395,71,0.989,0.0001)	TARÁNTULA	14	17:51 HRS	249:54 HRS
80x4x4x25	(160,65,0.990,0.0001)	CUEXCOMATE	60	03:33 HRS	213:00 HRS
140x4x4x25	(40,70,0.990,0.0001)	CUEXCOMATE	60	10:09 HRS	609:00 HRS

Tabla 6.5. *Resultados de la sintonización recocido simulado. Las 5 instancias de prueba sintonizadas arrojan valores óptimos en temperatura inicial* $t_0 = \{450, 425, 395, 160, 40\}$*, la longitud de la cadena de markov* $m = \{78, 56, 71, 65, 70\}$*, el factor de descenso de temperatura* $\mu = \{0.989, 0.890, 0.989, 0.990, 0.990\}$ *y la temperatura final* $t_f = \{0.0015, 0.0001, 0.0001, 0.0001\}$*, los parámetros que más impactan en el tiempo son un factor de descenso muy cercano a uno y una temperatura final tendiendo a cero.*

ronda las 213 horas con respecto a la misma instancia. Concluimos entonces en que un número de núcleos mayor nos trae beneficios en la reducción de los tiempos de sintonizado y en la posibilidad de extender los rangos de los parámetros, al incrementar la serie de valores, esto por supuesto solo es posible si se implementa un método de sintonización distribuida automática como el SDAAP propuesto en esta tesis.

Los resultados de la sintonización de SCH mostrados en la tabla 6.6, utilizan los mismos recursos: el cluster Tarántula y Texcal con 14 y 60 núcleos respectivamente. Los tiempos computados son ligeramente menores que los utilizados por RS, esta metaheurística tiene una fuerte influencia dada por número de hormigas y el número de ciclos de la búsqueda local sobre el tiempo computado.

Los resultados de la sintonización de AGHCGrid mostrados en la tabla 6.7, a diferencia de la sintonización de RS y SCH, solo se utiliza el cluster Cuexcomate y solo se sintoniza una instancia representativa del conjunto FFS_60x4x4x25 como se muestra en el punto 5 de la sección 6.3.1, en esta tabla se aprecia un aumento considerable en los tiempos computados con respecto a los mostrados por RS y SCH, para esta instancia en particular, este aumento se debe a los tiempos computados por SCH y RS incluidos dentro del algoritmo AGHCGrid.

Para darnos una idea del tamaño de la instancias manejables por el método SDAAP para la sintonización, vamos a calcular sobre la base de los resultados teóricos, los tiempos aproximados computados para la sintonización final del algoritmo:

1. El algoritmo AGHCGrid ejecuta 30 pruebas por cada variación en los parámetros a sintonizar como se muestra en la tabla 6.4.

INSTANCIA	SCH(h,α,β,γ,δ,q,p)	PLATAFORMA	NÚCLEOS	TIEMPO SDAAP	TIEMPO 1 NÚCLEO
20x4x4x25	(18,1.0,0.2,0.2,0.3,0.2,1250)	TARÁNTULA	14	01:06 HRS	43:25 HRS
40x4x4x25	(20,1.0,0.2,0.6,0.9,0.2,2000)	TARÁNTULA	14	09:03 HRS	126:42 HRS
60x4x4x25	(14,1.0,0.1,0.1,0.9,0.1,2000)	TARÁNTULA	14	11:28 HRS	160:32 HRS
80x4x4x25	(18,1.0,0.3,0.3,0.9,0.1,1750)	CUEXCOMATE	60	02:15 HRS	135:00 HRS
140x4x4x25	(16,1.0,0.4,0.9,0.2,0.2,1500)	CUEXCOMATE	60	04:58 HRS	298:00 HRS

Tabla 6.6. *Resultados de la sintonización del sistema de colonia de hormigas. Las 5 instancias de prueba sintonizadas arrojan valores óptimos en el número de hormigas* $h = \{18, 20, 14, 18, 16\}$*, el factor de importancia* $\alpha = \{1\}$*, el factor de importancia* $\beta = \{0.2, 0.2, 0.1, 0.3, 0.4\}$*, el factor de evaporación local* $\gamma = \{0.2, 0.6, 0.1, 0.9, 0.9\}$*, el factor de evaporación global* $\delta = \{0.3, 0.9, 0.9, 0.9, 0.2\}$*, el factor de exploración/explotación* $p = \{0.2, 0.2, 0.2, 0.2, 0.2\}$ *y el número de ciclos de la búsqueda local* $q = \{1250, 2000, 2000, 1750, 1500\}$*, los parámetros que más impactan en el tiempo son el número de hormigas y el número de ciclos de la búsqueda local.*

INSTANCIA	AGHCGrid(G,C,P,(SCH),(RS))	PLATAFORMA	NÚCLEOS	TIEMPO SDAAP	TIEMPO 1 NÚCLEO
60x4x4x25	(30,0.5,70,(SCH),(RS))	CUEXCOMATE	60	170:00 HRS	10,200 HRS

Tabla 6.7. *Los parámetros sintonizados del algoritmo propuesto fueron: método de selección por ruleta, un número de generaciones G=30, una tasa de cruce C=0.6 y un tamaño de población P=70.*

2. Dado uno de los peores casos, donde el número de generaciones $G = 30$ y un tamaño de población $P = 60$, tenemos una repartición de procesos perfecta, ya que el número de núcleos que consideramos en el cluster Cuexcomate es 60 que equivale a 60 procesos, que es el tamaño de población, al repartir 60 procesos sobre 60 núcleos se ejecuta un proceso por núcleo, por tanto no existe sobrecarga.

3. El ciclo principal del algoritmo AGHCGrid, esta dado por el número de generaciones G, dentro del cual cada proceso aplica el operador de mutación, primero con SCH y después con RS.

4. Suponiendo que la ejecución de una sola prueba de SCH y RS para la instancia FFS_60x4x4x25 tarda 1 minuto cada uno, tenemos que en total ambos tardan 2 minutos.

Tomando en cuenta estos datos, el tiempo de la ejecución en paralelo para la sintonización esta dado por la ecuación 6.1, donde t_{sch} es en tiempo para SCH y t_{rs} es el tiempo para RS.

$$t_p = 30_{pruebas} * 30_{generaciones} * (t_{sch} + t_{rs}) \tag{6.1}$$

Sustituyendo tenemos que $t_p = 30_{pruebas} * 30_{generaciones} * (1_{sch} + 1_{rs}) = 1800$ minutos ó 30 horas, que es tiempo teórico computado para una sola variación en los parámetros a sintonizar, y dado que el número de variaciones o combinaciones total de los parámetros de la tabla 6.4 es de 33 por la ecuación 5.2, entonces tendríamos un tiempo de sintonización total de $30 * 33 = 990$ horas o ∼41 días, esto considerando 60 núcleos, ya que si solo consideramos uno tendríamos que multiplicarlo por 60, lo que nos daría un tiempo secuencial $t_s = t_p * 60 = 59{,}400$ horas ó 2,475 días, lo cual quiere decir que sin un cluster o Grid seria imposible llevar a cabo la sintonización.

Afortunadamente durante el proceso de sintonización y bajo la influencia que toman ciertos parámetros, no todas las pruebas computan el mismo tiempo, con generaciones $G < 30$ y tamaños de población $P < 60$, en la secuencia de valores a sintonizar, los tiempos son menores, por tanto el tiempo computado para la sintonización del algoritmo AGHCGrid de 170 horas, en la tabla 6.7 es conservador.

Los resultados de cada una de la instancias sintonizadas aplicando el método propuesto se muestran el la sección 6.4.

6.3.3. Convergencia del algoritmo AGHCGrid.

Para la instancia seleccionada FFS_60x4x4x25, la convergencia en el número de generaciones es mostrada en la figura 6.6, el número de generaciones posee un peso muy considerable en el consumo de tiempo, dado que a mayor número de generaciones, mayor es el tiempo que consume su ejecución, como se observó en el cálculo de los tiempos en la ecuación 6.1, los resultados muestran que partiendo de un número de generaciones $G = 10$ con incrementos de 2, se mejora constantemente el valor de la función objetivo hasta alcanzar la generación $G = 30$, a partir de ahí el valor de la función objetivo se degrada.

El parámetro para la tasa de cruce C esta acotado en el rango [0.1,1] con incrementos de 0.1, a futuro podría refinarse en incrementos más cortos con una segunda refinación, pero se consideró dar prioridad a trabajar con el tamaño de la población, que tiene que ver con la Grid y su número de núcleos. A la fecha en que se llevó a cabo esta sintonización solo se utilizó el cluster Cuexcomate y se considero un máximo de 70 procesos esclavos y un proceso maestro, con la finalidad de observar si el algoritmo convergía más allá de los límites del cluster que eran 60 núcleos, los resultados se observan en la figura 6.7 y muestra que la convergencia no se alcanza, ya que se observa una mejora constante, lo que indica que un número mayor de núcleos puede mejorar la eficacia del algoritmo, pero que al no contar con ellos, se cae en la sobrecarga de núcleos al asignar más de un proceso lo cual duplica los tiempos de sintonizado.

G	$\overline{C_{max}}$ 30 pruebas
10	481.233
12	480.300
14	479.900
16	479.733
18	479.633
20	479.233
22	478.967
24	478.267
26	478.667
28	478.100
30	478.067
31	478.667
32	479.233

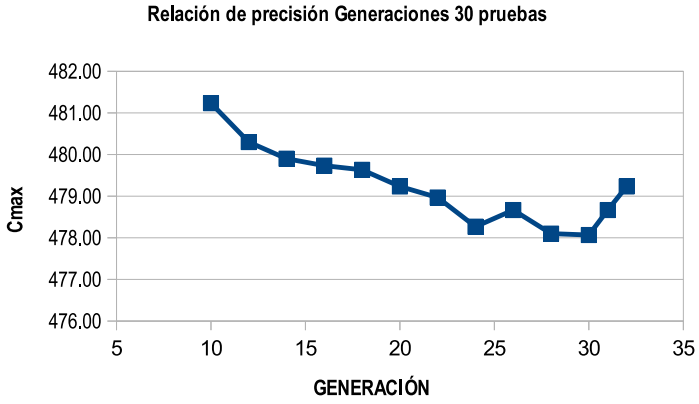

Figura 6.6. *Convergencia del número de generaciones. Dada la serie de valores del parámetro G={10,12,...,32}, en combinación con el resto de parámetros C y P, la ejecución de 30 pruebas en paralelo aplicando SDAAP, arroja una convergencia en la generación G=30, con una degradación en la calidad de las soluciones con G>30.*

Esta convergencia que llamaremos parcial debido a que la Grid Morelos estaba en proceso de integración, fue la limitante para comprobar si a mayor tamaño de población continuaba mejorando. A futuro, una vez que la Grid queda funcional se pudieron hacer las pruebas sobre los 120 núcleos, e incluso llevar a cabo una sobrecarga en los núcleos por un factor de 2 y 4 con un total de 240 y 480 procesos respectivamente.

6.4. Eficacia en la Grid.

Después del proceso de sintonización analizado en la sección 6.3 y una vez completado el proceso de integración de la Grid Morelos, en esta sección se analiza la eficacia del algoritmo AGHCGrid.

Pensar en paralelo, es pensar en la distribución de procesos, en agrupaciones de núcleos, en repartición de tareas, en sincronización de procesos, pero sobre todo en la emoción que nos brinda poder alcanzar y tratar problemas que de manera secuencial nunca podrían ser tratados. En esté caso, brinda la oportunidad de sentar las bases de la nueva experimentación, de la experimentación numérica aquí expuesta, para tratar instancias de problemas más grandes y más complejos, cuya característica principal es que no puede ser recreada sin una Grid similar a la Grid Morelos.

La distribución de procesos de manera uniforme sobre una Grid, no es una tarea trivial, dado que de manera explicita hay que indicar la capacidad de cada uno de los

P	$\overline{C_{max}}$ 30 pruebas
20	478.900
25	477.900
30	477.167
35	477.400
40	477.233
45	476.933
50	476.500
55	476.467
60	476.467
65	475.733
70	475.700

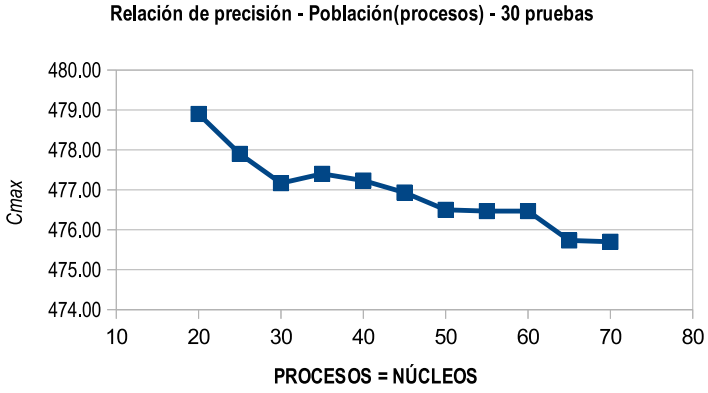

Figura 6.7. *Convergencia del tamaño de la población. La figura muestra el promedio de 30 pruebas por cada variación del tamaño de la población, que es directamente proporcional al número de procesos usados hasta un límite de 80 procesos y se observa que a medida que se aumenta el tamaño de la población, se mejora la calidad de la solución y no se observa convergencia a falta de utilizar un número más grande de procesos.*

nodos, en términos de núcleos como se explica en la sección 6.4.1. La distribución de procesos sobre la Grid debe ser uniforme en términos de tiempos de procesamiento, es decir, se debe cuidar la asignación de tiempos de procesamiento por cada proceso y comprobar que es uniforme en cada nodo como se explica en la sección 6.4.2.

La sintonización distribuida automática aplicada en paralelo (SDAAP), es funcional para la experimentación de la eficacia, con la característica de que solo hay una permutación que evaluar: *la óptima,* encontrada con el análisis de sensibilidad en la sección 6.3. La Grid tiene un impacto en la calidad de las soluciones al utilizar diferente número de procesos cooperativos distribuidos a lo largo de la Grid, en la sección 6.4, se analiza este comportamiento para instancias pequeñas, medianas y grandes, con el fin de encontrar el número óptimo de procesos que dan la mejor solución, finalmente en la sección 6.5, se analiza la eficacia de la paralelización con los métodos secuenciales SCH y RS.

6.4.1. Planeación del anclaje de procesos en núcleos de los procesadores.

El anclaje se refiere a la forma física en que se agrupan los núcleos de la Grid para asignarles tareas o procesos, dado que se tienen 66 núcleos en el cluster Cuexcomate y 66 núcleos en el cluster Texcal, en donde para cada nodo maestro, se tomaron 6 núcleos de 12, dejando 6 para tareas administrativas propias de cada cluster como mantenimiento, procesos de control, servidor web, entre otros, entonces se tiene en total 120 núcleos disponibles, de esta forma no afectamos el comportamiento del algoritmo en los nodos

maestros.

Formalmente podemos definir los recursos disponibles de la Grid Morelos como una Grid $G = \{C_1, C_2\}$ compuesta por dos clusters $C_1(Cuexcomate)$ y $C_2(Texcal)$, donde cada nodo n_i esta compuesto por un número de núcleos p_i asociados en la forma de *nodo:núcleos*$(n_i : p_i)$ para cada $n_i : p_i \in G$.

La tabla 6.8 muestra el anclaje de núcleos requerida para hacer agrupaciones de $n = \{120, 60, 30, 15, 10, 5\}$ sobre los cuales se asignaran procesos, p. ej. si se requiere agrupar 120 núcleos, solo se requiere una agrupación $D_1 = \{n_1 : 6, n_2 : 12, n_3 : 12, n_4 : 12, n_5 : 12, n_6 : 6, n_7 : 6, n_8 : 12, n_9 : 12, n_{10} : 12, n_{11} : 12, n_{12} : 6\}$ que pone a disposición 6 núcleos en el nodo n_1, n_6, n_7, n_{12} y 12 núcleos en los nodos $n_2, n_3, n_4, n_5, n_8, n_9, n_{10}, n_{11}$ tomados de cluster C_1 y C_2, esto es, un grupo $|D_1| = 120$ núcleos, que hace un total de 120 $(1 : 120 : 120)$.

En general se pueden hacer agrupaciones de distintas formas tomando núcleos de los dos clusters de forma arbitraria, sin embargo, la recomendación más acertada y la que utilizamos es seleccionar los más próximos uno del otro, así para una agrupación de 60 núcleos existen dos posibles distribuciones $D_1 = \{n_1 : 6, n_2 : 12, n_3 : 12, n_4 : 12, n_5 : 12, n_6 : 6\}$ y $D_2 = \{n_7 : 6, n_8 : 12, n_9 : 12, n_{10} : 12, n_{11} : 12, n_{12} : 6\}$, donde D_1 utiliza los núcleos del cluster C_1 y D_2 utiliza los núcleos del cluster C_2, evitando usar ambos a la vez para no pasar por el cuello de botella de la Grid que se da en la interconexión $C_1 \leftrightarrows C$, esto es, dos agrupaciones de 60 núcleos para un total $|D_{1,2}| = 120$ $(2 : 60 : 120)$.

Para agrupaciones de 30 se tienen 4 agrupaciones $D_1 = \{n_1 : 6, n_2 : 12, n_3 : 12\}$, $D_2 = \{n_4 : 12, n_5 : 12, n_6 : 6\}$, $D_3 = \{n_7 : 6, n_8 : 12, n_9 : 12\}$ y $D_3 = \{n_{10} : 12, n_{11} : 12, n_{12} : 6\}$, donde $D_1, D_2 \in C_1$ y $D_3, D_4 \in C_1$, esto es, 4 agrupaciones de 30 núcleos para un total $|D_{1,2,3,4}| = 120$ $(4 : 30 : 120)$.

Los subsecuentes agrupaciones son cada vez menores en número de núcleos, así para hacer agrupaciones de 10 tenemos 12 agrupaciones $D_1 = \{n_1 : 5, n_6 : 5\}$, $D_2 = \{n_2 : 10\}$, $D_3 = \{n_3 : 10\}$, $D_4 = \{n_4 : 10\}$, $D_5 = \{n_5 : 10\}$, $D_6 = \{n_1 : 1, n_2 : 2, n_3 : 2, n_4 : 2, n_5 : 2, n_6 : 1\}$, observe que la distribución D_1 utiliza 5 de 6 núcleos del nodo n_1 y 5 de 6 núcleos del nodo n_6 y sobran 2, después D_2, D_3, D_4, D_5 utilizan 10 de 12 núcleos de los nodos n_2, n_3, n_4, n_5 respectivamente, quedando 8 núcleos libres que corresponde a 2 en cada nodo, lo que da un total de 10, finalmente D_6 integra los núcleos sobrantes, todos correspondientes al cluster C_1, los otras 6 agrupaciones son idénticas y corresponden al cluster C_2, esto es, 12 agrupaciones de 10 núcleos para un total $|D_{1,2,\ldots,12}| = 120$ $(12: 10 : 120)$. Finalmente para hacer grupos de 5 núcleos resulta en 24 agrupaciones $(24: 5 : 120)$.

El anclaje de núcleos para la ejecución del algoritmo debe ser escrito en un archivo de configuración como se muestra en el apéndice F.

GRID MORELOS												
GRUPOS: NÚCLEOS: TOTAL	CLUSTER CUEXCOMATE (66 Núcleos)						CLUSTER TEXCAL (66 Núcleos)					
	Nodo01 12 núcleos	Nodo02 12 núcleos	Nodo03 12 núcleos	Nodo04 12 núcleos	Nodo05 12 núcleos	Nodo06 6 núcleos	Nodo07 12 núcleos	Nodo08 12 núcleos	Nodo09 12 núcleos	Nodo10 12 núcleos	Nodo11 12 núcleos	Nodo12 6 núcleos
1:120:120	6	12	12	12	12	6	6	12	12	12	12	6
2:60:120	6	12	12	12	12	6	6	12	12	12	12	6
4:30:120	6	12	12	12	12	6	6	12	12	12	12	6
8:15:120	3,3	12	12	12	12	3,3	3,3	12	12	12	12	3,3
12:10:120	5,1	10,2	10,2	10,2	10,2	5,1	5,1	10,2	10,2	10,2	10,2	5,1
24:5:120	5,1	5,5,2	5,5,2	5,5,2	5,5,2	5,1	5,1	5,5,2	5,5,2	5,5,2	5,5,2	5,1

Tabla 6.8. *Anclaje de núcleos en la Grid Morelos. La Grid provee una capacidad máxima de 120 núcleos, los cuales deben ser fijados a un proceso conocido como anclaje, para construir grupos de núcleos, este anclaje debe ser especificado en un archivo de configuración, que es usado al momento de la ejecución del algoritmo AGHCGrid, para hacer una distribución uniforme de procesos sobre un grupo en particular, múltiples configuraciones pueden ser especificadas, pero la experiencia nos dice que funciona mejor aquellos anclajes, donde los núcleos asignados son los más próximos uno del otro.*

6.4.2. Planeación de la distribución de procesos.

La distribución de procesos consiste en asignar de manera física, un conjunto de tareas o procesos sobre uno o más conjuntos o grupos de núcleos en la Grid, para llevar a cabo esta planeación se diseñan grupos de núcleos como se explica en la sección 6.4.1.

La planeación para la distribución de procesos en una Grid, debe conllevar a una distribución uniforme en tiempo y espacio, en tiempo porque todos los procesos deben aprovechar los núcleos de igual forma y en espacio porque todos los procesos deben procesar el mismo tamaño de datos, una mala distribución en al menos un proceso, provocará que los tiempos no sean uniformes y llevara al algoritmo a computar tiempos más largos.

Otro factor igualmente importante que afecta el comportamiento del algoritmo, es la heterogeneidad de la Grid, donde el uso procesadores con núcleos no uniformes, requiere de una nueva distribución de procesos para balancear las cargas de trabajo, se observó que este fenómeno no afecta la eficacia pero si la eficiencia, en el escenario de pruebas el cluster Cuexcomate y Texcal son exactamente idénticos en recursos, por lo que la eficiencia no se ve afectada por este fenómeno, reduciendo los problemas de una distribución de procesos.

La distribución de procesos se realiza de la siguiente manera: Dada una Grid $G = \{C_1, C_2\}$ compuesta por dos cluster $C_1(Cuexcomate)$ y $C_2(Texcal)$, donde cada nodo n_i esta compuesto por un número de núcleos p_i asociados en la forma de *nodo:núcleo*$(n_i : p_i)$ para cada $n_i : p_i \in G$, existe un conjunto de distribuciones uniformes D_i con $i > 0$, tal

que $\sum_{i=1}^{n} |D_i| = 120$, esto quiere decir que no importa de que tamaño sea la agrupación D_i, la cual puede ser de $n = \{5, 10, 15, 30, 60, 120\}$ o más dependiendo del tamaño de la Grid, lo que importa es que se utilice el total de los recursos que provee la Grid de manera uniforme.

Sobre la base de un número de pruebas de 30, estás deben ser repartidas entre el número de agrupaciones D_i, p. ej. para realizar una distribución de 120 procesos sobre 120 núcleos, el anclaje de núcleos resulta en una solo distribución D_1 mostrada en la sección anterior 6.4.1, la cual debe ejecutar 30 pruebas, igualmente para distribuir 60 procesos sobre 120 núcleos, resulta dos dos distribuciones D_1 y D_2, donde $D_1 \in C_1$ y $D_2 \in C_2$, por tanto se requieren lanzar 15 pruebas en el cluster C_1 y 15 pruebas en el cluster C_2 para obtener 30 pruebas como se indica en la tabla 6.9.

Para distribuir 30 procesos sobre 120 núcleos, se tendrían 4 distribuciones D_1, D_2, D_3 y D_4, por lo que se requieren 8 pruebas por cada distribución lo que daría 32 pruebas, para agrupaciones más pequeñas como el caso de utilizar 10 núcleos tendríamos 12 agrupaciones, por lo se necesita 3 pruebas por cada agrupación lo que daría 36 pruebas.

GRID MORELOS												
PROCESOS : NÚCLEOS	CLUSTER CUEXCOMATE (66 Núcleos)						CLUSTER TEXCAL (66 Núcleos)					
	Nodo01 12 núcleos	Nodo02 12 núcleos	Nodo03 12 núcleos	Nodo04 12 núcleos	Nodo05 12 núcleos	Nodo06 6 núcleos	Nodo07 12 núcleos	Nodo08 12 núcleos	Nodo09 12 núcleos	Nodo10 12 núcleos	Nodo11 12 núcleos	Nodo12 6 núcleos
480:120	4 procesos por núcleo, cuadriplica la capacidad de la Grid, 30 pruebas											
240:120	2 procesos por núcleo, duplica la capacidad de la Grid, 30 pruebas											
120:120	1 proceso por núcleo, utiliza la capacidad máxima de la Grid, 30 pruebas											
60:120	1 proceso por núcleo, capacidad máxima del Cluster, 15 pruebas						1 proceso por núcleo, capacidad máxima del Cluster, 15 pruebas					
30:120	½ Cluster, 8 pruebas			½ Cluster, 8 pruebas			½ Cluster, 8 pruebas			½ Cluster, 8 pruebas		
15:120	¼ Cluster, 4 pruebas	¼ Cluster, 4 pruebas	¼ Cluster, 4 pruebas	¼ Cluster, 4 pruebas			¼ Cluster, 4 pruebas	¼ Cluster, 4 pruebas	¼ Cluster, 4 pruebas	¼ Cluster, 4 pruebas		
10:120	|5,1|10,2|10,2|10,2|10,2|5,1| = 6 agrupaciones 3 pruebas por agrupación = 18 pruebas						|5,1|10,2|10,2|10,2|10,2|5,1| = 6 agrupaciones 3 pruebas por agrupación = 18 pruebas					
5:120	|5,1|5,5,2|5,5,2|5,5,2|5,5,2|5,1| = 12 agrupaciones 2 pruebas por agrupación = 24 pruebas						|5,1|5,5,2|5,5,2|5,5,2|5,5,2|5,1| = 12 agrupaciones 2 pruebas por agrupación = 24 pruebas					

Tabla 6.9. *Distribución de procesos en la Grid Morelos. Se da sobre la base de un conjunto de agrupaciones D_i que se reparten 120 núcleos y se dividen las 30 pruebas requeridas, se muestran conjuntos de 480, 240,120, 60, 30, 15, 10 y 15 procesos a distribuirse sobre agrupaciones de 120, 60, 30, 15, 10 y núcleos, las distribuciones de 480 y 240 muestran una sobrecarga cuádruple y doble respectivamente por cada núcleo, para algunas distribuciones de 10 y 5 procesos, el número de pruebas es mayor que 30.*

6.4.2.1. Sobrecarga de núcleos de los procesadores.

La distribución de procesos puede llevar a una sobrecarga de los núcleos en más de un proceso, así para poder distribuir 480 procesos, donde solo existen 120 núcleos reales, se tiene que sobrecargar cada núcleo con un factor de 4 procesos, para obtener una distribución sobrecargada $D_1 = \{n_1 : 24, n_2 : 48, n_3 : 48, n_4 : 48, n_5 : 48, n_6 : 24, n_7 : 24, n_8 : 48, n_9 : 48, n_{10} : 48, n_{11} : 48, n_{12} : 24\}$, como se muestra en la tabla 6.9, para el caso de 240 núcleos solo requiere una sobrecarga por un factor de 2 dando una distribución $D_1 = \{n_1 : 12, n_2 : 24, n_3 : 24, n_4 : 24, n_5 : 24, n_6 : 12, n_7 : 12, n_8 : 24, n_9 : 24, n_{10} : 24, n_{11} : 24, n_{12} : 12\}$, una distribución perfecta asigna un proceso por núcleo dando la distribución $D_1 = \{n_1 : 6, n_2 : 12, n_3 : 12, n_4 : 12, n_5 : 12, n_6 : 6, n_7 : 6, n_8 : 12, n_9 : 12, n_{10} : 12, n_{11} : 12, n_{12} : 6\}$.

6.4.3. Implementación de la Metodología SDAAP en Múltiples Instancias (SDAAP-MI).

El método propuesto SDAAP es capaz de sintonizar en forma automática, un conjunto de parámetros sobre una sola agrupación D_1, la variante para el método SDAAP es que además, el número de pruebas debe ser repartido entre el número de agrupaciones $30/D_i$, para este nuevo caso, donde la experimentación incluye los parámetros sintonizados previamente y la utilización completa de la Grid con 120 núcleos, el método SDAAP también puede ser utilizado para ejecutar 30 pruebas con la combinación óptima.

Dado que el método propuesto SDAAP solo puede llevar a cabo una experimentación automática para una solo distribución de procesos, por lo que los casos que resultan en una sola distribución D_1 están cubiertos, tal es el caso de utilizar 120, 240 y 480 como se muestra en la tabla 6.9. Para los casos de utilizar $60, 30, 15, 10$ y 5 núcleos, donde existen más de una agrupación D_i, se propone una extensión al método SDAAP para poder aplicarlo en forma paralela a múltiples instancias asociadas a cada agrupación D_i, este método SDAAP en múltiples instancias denominado SDAAP-MI consiste en lo siguiente:

1. Dada una serie de agrupaciones D_i con $i > 0$, tal que $\sum_{i=1}^{n} |D_i| = 120$ núcleos, se selecciona un nodo r_i con $r_i \in D_i$ como nodo maestro.

2. Se aplica un número i de métodos $SDAAP_i$ en cada nodo r_i en forma manual, preparando para cada agrupación D_i los archivos de configuración correspondientes como se explica en la sección 6.3.1.

3. Se aplica el método SDAAP en múltiples instancias (SDAAP-MI), que sustituye al punto 2, mediante un script que automatiza el proceso de ir a cada nodo r_i y ejecutar $SDAAP_i$ manualmente.

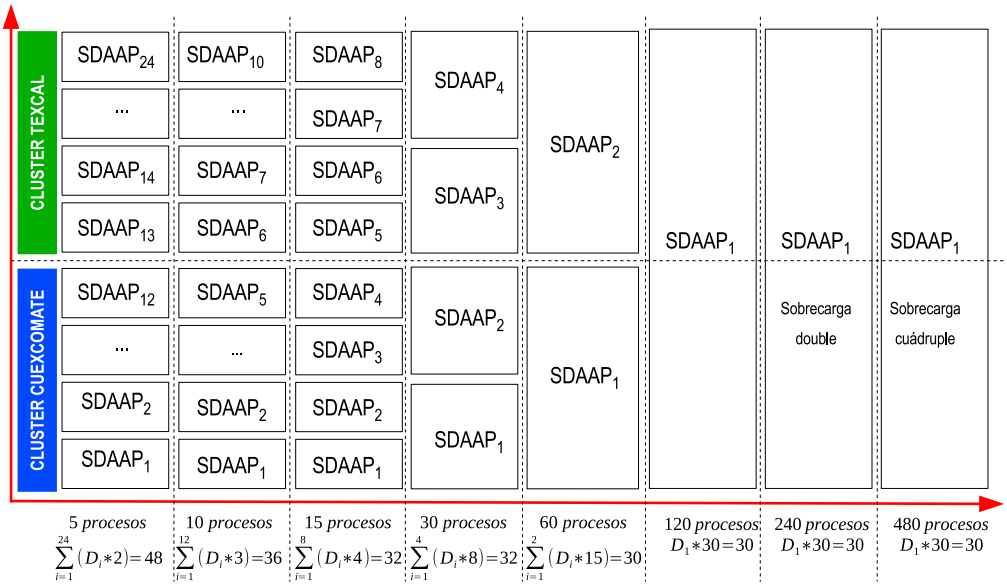

Figura 6.8. *Distribución de procesos con SDAAP-MI. El método propuesto permite reducir el tiempo de experimentación a t/D_i, donde t es el tiempo secuencial y D_i el número de instancias $SDAAP_i$ ejecutadas en paralelo, siempre y cuando el número de agrupaciones sea mayor a uno y no exista sobrecarga de núcleos.*

El método SDAAP-MI mostrado en la figura 6.8, ejecuta en forma paralela el total de instancias $SDAAP_i$ en cada nodo r_i, para cada agrupación de núcleos D_i, por tanto cada instancia $SDAAP_i$, es responsable de un número de pruebas $30/D_i$ y el tiempo requerido para llevar a cabo toda la experimentación se reduce a t/D_i, donde t es el tiempo secuencial que le llevaría a una sola instancia SDAAP realizar las 30 pruebas, esta ejecución en paralelo de las múltiples instancias es lo que permite ocupar al 100 % los recursos de la Grid y disminuir los tiempos de procesamiento como se observa en la figura 6.9.

6.4.4. Análisis de eficacia de la cooperación.

Para realizar el análisis de la cooperación de los procesos se utilizaron 5 tamaños de instancias de pruebas: $\{20, 40, 60, 80, 140\}$ generados como se indica en la sección 6.2, 6 agrupaciones de núcleos: $\{5, 10, 15, 30, 60, 120\}$ como se indica en la sección 6.4.1, y 8 tamaños de procesos: $\{5, 10, 15, 30, 60, 120, 240, 480\}$ para su ejecución sobre la Grid como se indica en la sección 6.4.2. Dos distribuciones de procesos presentan sobrecarga de núcleos 240 y 480, al tener que distribuirlos sobre una agrupación de 120 núcleos como se indica en la sección 6.4.2.1, finalmente para la ejecución automatizada se utiliza el método propuesto SDAAP-MI mostrado en la sección 6.4.3. Durante la fase de experimentación, la agrupación de 480 procesos requiere mucho tiempo para las instancias grandes, debido a la excesiva sobrecarga, por lo que no se incluyen resultados, pero si se mencionan sus cálculos teóricos.

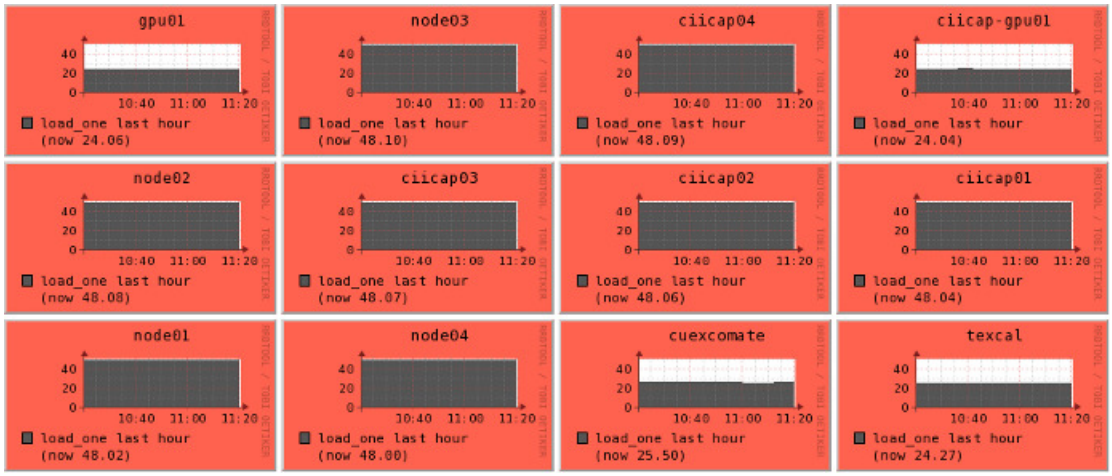

Figura 6.9. *Carga procesos sobre la Grid con SDAAP-MI. El método propuesto permite ocupar al 100 % todos los núcleos disponibles de la Grid Morelos, en la imagen se observa por el color del nodo (rojo) que se esta usando al máximo la capacidad de procesamiento de cada uno de los núcleos que componen cada nodo de la Grid.*

La tabla 6.10, muestra los 5 tamaños de instancias que se usan para ejecutar los 8 tamaños de procesos sobre las agrupaciones, en total 40 combinaciones, donde cada combinación requiere 30 pruebas, para cada agrupación de núcleos se requiere un número determinado de ejecuciones del método SDAAP en paralelo, para producir el número total de pruebas que se requiere, p. ej. para la instancia FFS_60x4x4x25 cuando se lleva a cabo la experimentación utilizando una agrupación de 5 núcleos, que albergan a 5 procesos, el número total de agrupaciones que se pueden distribuir sobre la Grid es de 24, entonces las 30 pruebas se pueden conseguir automatizando con un SDAAP, 2 pruebas por cada agrupación para un total de 48.

Para poder conseguir la ejecución en paralelo de los 24 SDAAP, se utiliza SDAAP-MI, finalmente podría pensarse que para conseguir las 30 pruebas solo se requiere una primera prueba de 24 y una segunda prueba solo de 6, pero realizar 1 a 24 conlleva el mismo tiempo por ser en paralelo.

Las cotas superiores que se presentan en esta tesis, son las mejores encontradas en todas las pruebas realizadas durante la fase de sintonización y experimentación, la siguiente serie de ejecuciones corresponden a un promedio de 30 pruebas, para cada uno de los 5 tamaños de instancia, que distribuyen 8 tamaños de procesos, sobre 8 tamaños de agrupaciones de núcleos. Las combinaciones de procesos y núcleos generadas para cada una de las instancias de prueba son: (5,5), (10,10), (20,20), (30,30), (60,60), (120,120), (240,120), (480,120), que corresponden a (*procesos* : *núcleos*), por las 5 tamaños de instancias nos dan el total de 56 combinaciones, para cada método aplicado existen criterios de paro diferentes como se observa en la tabla 6.11, los cuales son:

1. Para RS existen 5 sintonizaciones con criterios de paro diferentes, uno por cada tamaño de instancia: (450, 78, 0.989, 0.0015), (425, 56, 0.890, 0.0001), (395, 71,

PLATAFORMA DE PRODUCCIÓN GRID MORELOS – DISTRIBUCIÓN DE PROCESOS								
INSTANCIA	NÚCLEOS 5	NÚCLEOS 10	NÚCLEOS 15	NÚCLEOS 30	NÚCLEOS 60	NÚCLEOS 120	NÚCLEOS 120	NÚCLEOS 120
FFS_20x4x4x25	5 Procesos	10 Procesos	15 Procesos	30 Procesos	60 Procesos	120 Procesos	240 Procesos	480 Procesos
FFS_40x4x4x25								
FFS_60x4x4x25	24 SDAAP	12 SDAAP	8 SDAAP	4 SDAAP	2 SDAAP	1 SDAAP	1 SDAAP	1 SDDAP
FFS_80x4x4x25								
FFS_140x4x4x25	48 Pruebas	30 Pruebas	32 Pruebas	32 Pruebas	30 Pruebas	30 Pruebas	30 Pruebas	30 Pruebas

Tabla 6.10. *Distribución de procesos con SDAAP-MI. El método propuesto permite ejecutar en paralelo, grupos de procesos distribuidos a los largo de la Grid y dividir el total de pruebas requeridas, entre el número de agrupaciones para reducir los tiempos de la experimentación.*

0.989, 0.0001), (160, 65, 0.990, 0.0001) y (40, 70, 0.990, 0.0001) que corresponden a (t_o, m, μ, t_f), donde t_o es la temperatura inicial, m es el número de ciclos que repite la longitud de la cadena de markov, μ es el coeficiente de enfriamiento y t_f es la temperatura final.

2. Para SCH existen también 5 sintonizaciones con criterios de paro diferentes, uno por cada tamaño de instancia: (18, 1.0, 0.2, 0.2, 0.3, 0.2, 1250), (20, 1.0, 0.2, 0.6, 0.9, 0.2, 2000), (14, 1.0, 0.1, 0.1, 0.9, 0.1, 2000), (18, 1.0, 0.3, 0.3, 0.9, 0.1, 1750) y (16, 1.0, 0.4, 0.9, 0.2, 0.2, 1500), que corresponden a $SCH = (h, \alpha, \beta, \gamma, \delta, p, q)$, donde h es el número de hormigas, α es el coeficiente de importancia *alfa*, β es el coeficiente de importancia *beta*, γ es el coeficiente de evaporación *gamma* para la transición local, δ es el coeficiente de evaporación *delta* para una transición global, q es el coeficiente que divide la exploración/explotación y p es el criterio de paro.

3. Finalmente para el algoritmo genético AGHCGrid, el cual incluye los criterios de paro de SCH y RS, solo existe una sintonización con criterio de paro, para todas los tamaños de instancia: (30,0.5,T), que corresponden a $AG = (G, C, T)$, donde G el número de generaciones previamente sintonizado, C la tasa de cruce previamente sintonizado y T es el tamaño de la población que es directamente proporcional al número de procesos, donde $T = \{5, 10, 15, 30, 60, 120, 240\}$, que representa el número de procesos a usar para las pruebas de eficiencia y eficacia en esta tesis.

Estos valores de los parámetros y los criterios de paro incluidos se utilizan para evaluar el comportamiento del algoritmo sobre tres tipos de instancias: pequeñas, medianas y grandes analizadas a continuación.

6.4.4.1. Cooperación de procesos en instancias pequeñas.

La tabla 6.12, muestra los resultados de la experimentación para la instancia FFS_20x4x4x25, donde P el número de procesos, N el número de núcleos, N_p el núme-

INSTANCIA	RS(t_0,n,μ,t_f)	SCH(h,α,β,γ,δ,q,p)	GAHCGrid(G,C,P,(SCH),(RS))
20x4x4x25	(450,78,0.989,0.0015)	(18,1.0,0.2,0.2,0.3,0.2,1250)	(30,0.5,70,(SCH),(RS))
40x4x4x25	(425,56,0.989,0.0001)	(20,1.0,0.2,0.6,0.9,0.2,2000)	(30,0.5,70,(SCH),(RS))
60x4x4x25	(395,71,0.989,0.0001)	(14,1.0,0.1,0.1,0.9,0.1,2000)	(30,0.5,70,(SCH),(RS))
80x4x4x25	(160,65,0.990,0.0001)	(18,1.0,0.3,0.3,0.9,0.1,1750)	(30,0.5,70,(SCH),(RS))
140x4x4x25	(40,70,0.990,0.0001)	(16,1.0,0.4,0.9,0.2,0.2,1500)	(30,0.5,70,(SCH),(RS))

Tabla 6.11. *Criterio de paro para las tres metaheurísticas. En la tabla se observan 5 diferentes criterios de paro para RS, 5 criterios de paro para SCH y 1 criterio de paro para el AG, el algoritmo propuesto AGHCGrid esta compuesto por tres metaheurísticas(AG, SCH, RS), por lo que se usan 6 versiones del algoritmo AGHCGrid con criterios de paro diferentes para poder realizar el análisis de la eficacia, esto es, una por cada tamaño de instancia de prueba.*

ro de agrupaciones de tamaño N, #P el número de pruebas, $\overline{C_{max}}$ la media de las 30 pruebas, $\overline{T}$ la media en tiempo de las 30 pruebas, $Best$ el mejor makespan obtenido, UB la mejor cota encontrada, T_p el tiempo calculado de las 30 pruebas en paralelo y T_s el tiempo teórico calculado en forma secuencial, de estos datos se desprenden el siguiente análisis:

FFS_20x4x4x25									
P	N	Np	#P	$\overline{C_{max}}$ 30 pruebas	$\overline{T}$ 30 pruebas	Best	UB	Tp	Ts
5	5	24	2	481.43	361	478	469	723	54,220
10	10	12	3	479.67	363	472	469	1,090	108,980
15	15	8	4	479.17	364	476	469	1,456	163,785
30	30	4	8	477.70	364	473	469	2,915	327,990
60	60	2	15	476.23	366	473	469	5,488	658,500
120	120	1	30	475.10	415	471	469	12,435	1,492,200
240	120	1	30	473.80	1087	469	469	32,611	3,913,320

Tabla 6.12. *Resultados de la experimentación para FFS_ 20x4x4x25. La figura muestra los resultados de la ejecución de 30 pruebas para un número de procesos de 5, 10, 15, 30, 60, 120 y 240 procesos, se muestran la media del makespan ($\overline{C_{max}}$), la media en tiempo ($\overline{T}$), la mejor resultado obtenido de las 30 pruebas ($Best$), la mejor cota encontrada (UB), el tiempo calculado de las 30 pruebas en paralelo (T_p) y el tiempo calculado en forma secuencial (T_s).*

En la figura 6.10, muestra para la instancia FFS_20x4x4x25, un promedio de 30 ejecuciones por cada prueba, en donde cada prueba utiliza diferente número de procesos sobre agrupaciones de núcleos, en esta figura se observa que la cooperación da mejores resultados en el makespan conforme se aumenta el número de procesos, se observa que el peor promedio del makespan, es cuando se utilizan 5 procesos y el mejor promedio cuando se utilizan 240 procesos, para este problema se muestra un comportamiento

bien definido, en donde la mejora se da conforme se aumente el número de procesos hasta llegar a un máximo de 240 procesos sobre 120 núcleos, para este último existe una sobrecarga doble y no se observa aún convergencia.

En cuanto a los mejores resultados del makespan, en la tabla 6.12, se muestra que el mejor resultado se da al utilizar 240 procesos, el cual es la mejor cota encontrada para esta instancia de prueba propuesta, también se observa que la mejora se da a partir de los 120 núcleos, debido a que al utilizar 5, 10, 15 y 60 procesos el mejor makespan se da al utilizar 15 y solo se mejora al llegar a 120. En general el algoritmo muestra una mejora en la media del makespan al usar un creciente número de procesos en el orden: 5, 10, 15 ,30, 60, 120 y 240, pero solo puede encontrar un mejor valor al usar 120 núcleos o más en relación a usar un número de procesos inferior, sin embargo, tenemos la limitante de 120 núcleos, por lo que posiblemente para observar una convergencia, se requiere utilizar un número más grande de procesos, para lo que se necesita sobrecargar el número de procesos por núcleo en un factor $n > 2$, con lo cual tendríamos tiempos muy largos.

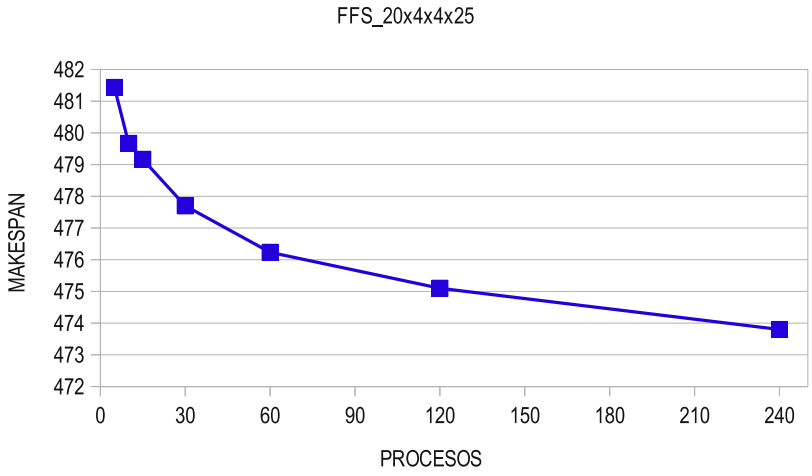

Figura 6.10. *Media de la eficacia de la cooperación de procesos para FFS_20x4x4x25. En la figura se observa que la mejora en el makespan, se da conforme se aumenta el número de procesos en la Grid, en el límite de 120 núcleos y 240 procesos, aun no se observa convergencia y al sobrecargar el número de procesos por núcleo, nos daría tiempos muy largos.*

La figura 6.11, se presentan la media de tiempos T_m computados para la instancia FFS_20x4x4x25, de un total de 30 ejecuciones por cada prueba, en donde cada prueba utiliza diferente número de procesos sobre agrupaciones de núcleos, en esta figura se observa que los tiempos al utilizar 5, 10, 15, 30 y 60 procesos se mantienen estables al utilizar casi el mismo tiempo en las ejecuciones de sus pruebas y solo se incrementan ligeramente debido a las comunicaciones entre los procesos, es decir que a medida que aumenta el número de procesos, las comunicaciones entre ellos tienden a incrementarse y con ello el tiempo del algoritmo.

Esta uniformidad en el tiempo se debe a que se mantiene un balance, al asignar un

proceso a cada núcleo, por lo tanto al aumentar el número de procesos en donde exista un igual número de núcleos, se mantiene el balance, los tiempos computados solo deben mostrar un ligero aumento, debido a una mayor comunicación entre ellos, que deriva en el paso de mensajes para la colaboración de soluciones encontradas y en los mensajes que se requieren para la sincronización de los procesos.

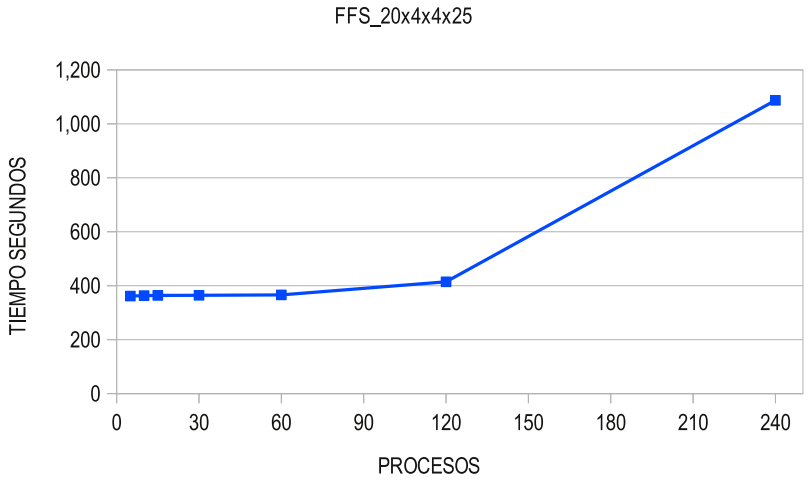

Figura 6.11. *Media de tiempos de la cooperación de procesos para FFS_ 20x4x4x25. La figura muestra un comportamiento ideal para 5, 10, 15, 30 y 60 procesos, al usar 120 procesos se muestra una aumento en el tiempo, ocasionado por la VPN que provee una alta latencia, y por ruido ocasionado por procesos ajenos al algoritmo, la utilización de 240 procesos duplica la carga de los núcleos y aumenta al doble el tiempo.*

Sin embargo, la figura también muestra que cuando se utilizan 120 procesos sobre 120 núcleos, lo cual involucra el uso de la Grid al utilizar los dos clusters, ya no se se mantiene el balance antes mencionado, a pesar de asignar un proceso por núcleo, el tiempo se incrementa más de lo normal, esto es debido a tres causas:

1. Ruido en los procesos. El ruido en al menos un proceso, se debe a se esta compartiendo tiempo de cómputo con algún otro proceso o grupo de procesos ajenos al algoritmo, en este caso, la sincronización de todos los procesos del algoritmo se ve afectado, al tener que esperar a este proceso que debe de compartir su tiempo con el proceso ajeno y que por tanto tiende a consumir más tiempo de lo normal.

2. Mal balanceo de las cargas de trabajo. Una planeación no uniforme, en la distribución de los procesos sobre los núcleos, conlleva a que algunos procesos tiendan a tardar más tiempo que otros.

3. Cambio en las comunicaciones. La Grid tuvo un cambio en las comunicaciones al pasar de utilizar una VPN a una VLAN, lo que llevó a una mejora del desempeño.

Por lo anterior expuesto, el incremento mostrado al utilizar 120 procesos, se debió en

parte a la alta latencia de la Grid y al uso del cluster por otro usuario al mismo tiempo en que se estaba realizando las pruebas, que afecto su desempeño, más adelante se vera que al mejorar la latencia y mantener un uso exclusivo de la Grid, se logran mejores desempeños del algoritmo. Finalmente al utilizar 240 procesos y contar con solo 120 núcleos, se duplican la carga por núcleo y por consiguiente se duplica el tiempo del algoritmo como se observa en la figura 6.11.

A partir de la media de los tiempos computados por las 30 pruebas, se puede determinar mediante la ecuación 6.2, el tiempo total requerido para la instancia al utilizar diferente número de procesos sobre agrupaciones de núcleos.

$$T_p = T_m * N_p \tag{6.2}$$

donde T_p es el tiempo que consume el método propuesto SDAAP-MI en la sección 6.4.3, al ejecutar las 30 pruebas en paralelo, T_m que es la media de las 30 pruebas para un número de procesos especifico y N_p es el número de agrupaciones formadas en paralelo para un número de procesos especifico, como se muestra en la sección 6.4.2, finalmente el tiempo secuencial requerido al utilizar un solo núcleo, se consigue con la ecuación 6.3, en donde se requiere multiplicar la media de las 30 pruebas T_m por el tiempo paralelo T_p, por las 30 pruebas requeridas.

$$T_s = T_m * T_p * 30 \tag{6.3}$$

La figura 6.12, muestra los tiempos en paralelo T_p de las 30 pruebas, al utilizar diferente número de procesos y los tiempos teóricos T_s secuenciales al utilizar 1 núcleo, la figura 6.12 a), muestra un comportamiento similar a la figura 6.11, debido a que los tiempos en paralelo son una escala de la media, al ser multiplicados por el número de agrupaciones que ejecutan SDAAP-MI en paralelo, la figura 6.12 b), de igual manera muestra un comportamiento similar en el cual se observa que los tiempos secuenciales se vuelven intratables.

Con base en los datos de la tabla 6.12, p. ej. al utilizar 120 procesos, se pasa de 415 segundos de la media por prueba mostrado en la figura 6.11, a un tiempo de 12,435 segundos de las 30 pruebas en paralelo mostrado en la figura 6.12 a), aplicando la ecuación 6.2, que al convertirse en tiempo secuencial aplicando la ecuación 6.3, se pasa a un tiempo de 1,492,200 segundos mostrado en la figura 6.12 b), que equivalen a $\sim$ 33.9 días para esta instancia de tamaño pequeño, además esta misma figura 6.12 b), muestra la relación que existe entre los tiempos paralelos y secuenciales, observándose la gran diferencia que existe entre ellos y la gran ventaja de hacer uso de la Grid para llevar la experimentación de esta tesis.

La tabla 6.13, muestra los resultados de la cooperación de procesos obtenidos para la instancia FFS_40x4x4x25, utilizando la misma configuración para la asignación de

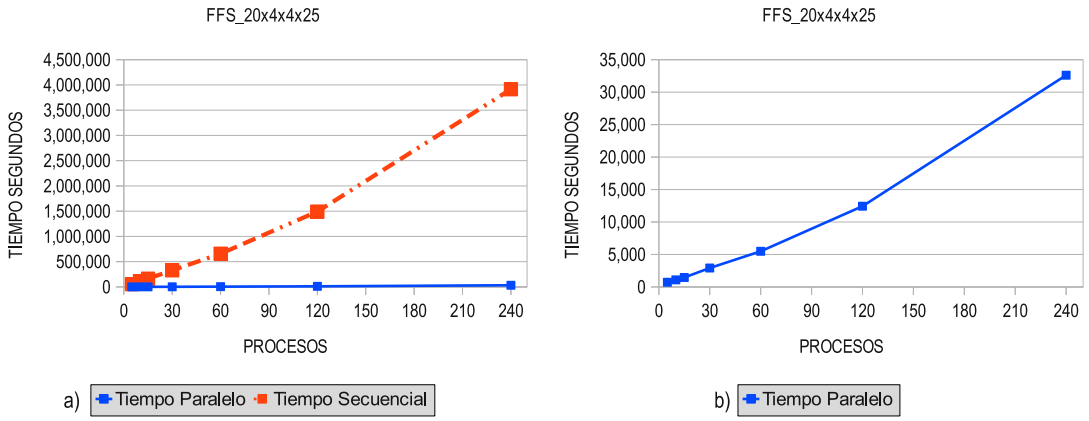

Figura 6.12. *Tiempos en paralelo y secuencial de las pruebas para FFS_ 20x4x4x25. La figura muestra la gran diferencia en tiempo necesario para realizar 30 pruebas en forma paralela utilizando el método propuesto SDAAP-MI y en forma secuencial utilizando un solo núcleo.*

procesos (P), sobre un total de agrupaciones (N_p), donde cada agrupación esta formado por un número de núcleos (N), que requieren ejecutar (#P) pruebas cada uno, para alcanzar las 30 requeridas utilizando el método SDAAP-MI propuesto.

La tabla muestra que el mejor makespan se da cuando se utiliza 240 procesos, que es la mejor cota encontrada $UB = 649$, la secuencia de mejora en la utilización de procesos se da en el orden 5, 10, 15, 30, 60, 120, 240, lo cual significa que para esta instancia, el algoritmo logra una mejora constante a partir de la utilización de 5 procesos o más, por tanto se observa una mejora constante, tanto en la media del makespan como en el mejor valor del makespan encontrado, de igual manera, aun no se observa convergencia en el número de procesos, en el límite de una doble sobrecarga por núcleo por lo que, para observarla, se requiere cuadruplicar la carga o escalar la Grid en número de procesadores.

La figura 6.13, muestra el comportamiento de la cooperación en la media de 30 pruebas, al utilizar diferente número de procesos, se observa que al inicio con 5 procesos se logra una media en el makespan de 678.77 y un mejor valor de 664, al utilizar 10 procesos aumenta la calidad de la media a 677.90 y un mejor valor del makespan de 659, esta mejora continúa hasta alcanzar los 240 procesos.

Este comportamiento observado es debido a que, el tamaño de la población aumenta en forma directa al número de núcleos utilizados, aumentando la exploración del espacio de soluciones, por lo que a partir de la utilización de un número creciente de procesos, a partir de 5, lo cual implica utilizar un tamaño de población más grande, el algoritmo empieza a generar mejores soluciones en un orden bien definido 5, 10, 15, 30, 60, 120, 240.

La ganancia en la calidad del makespan en las soluciones, es en el orden de ∼[1,2] al

FFS_40x4x4x25									
P	N	Np	#P	$\overline{C_{max}}$ 30 pruebas	$\overline{T}$ 30 pruebas	Best	UB	Tp	Ts
5	5	24	2	678.77	1,631	664	649	3,262	244,640
10	10	12	3	677.90	1,696	659	649	5,088	508,840
15	15	8	4	674.60	1,747	656	649	6,989	786,210
30	30	4	8	660.87	1,664	655	649	13,311	1,497,510
60	60	2	15	659.30	1,708	655	649	25,619	3,074,280
120	120	1	30	657.97	1,996	654	649	75,205	9,024,600
240	120	1	30	656.73	4,857	649	649	145,698	17,483,760

Tabla 6.13. *Resultados de la experimentación para FFS_40x4x4x25. La mejor cota se logra al utilizar 240 procesos con UB=649, se observa, que la mejora continua de la media de makespan, asi como del mejor valor encontrado, se da en el orden: 5, 10, 15, 30, 60, 120, 240, por lo que se observa una mejora constante directamente proporcional al número de procesos usados.*

duplicar la cantidad de procesos hasta llegar a 240, donde la ganancia de es ∼1, sabemos que duplicar el tamaño de la población que es directamente proporcional a duplicar el número de procesos sobre una base de 120 núcleos, se sobrecarga cada núcleo de la Grid por un factor r, con r=2,4,...,n, lo cual duplica los tiempos de procesamiento cada vez, por lo que para encontrar el número exacto donde el algoritmo para una instancia especifica ya no produce mejores soluciones, tendíamos que sobrecargar los núcleos o esperar a que la Grid cuente con un número mayor de procesadores.

La figura 6.14, muestra la media de tiempo de 30 pruebas en segundos al ejecutar el algoritmo para la instancia FFS_40x4x4x2 y utilizar 5, 10, 15, 30, 60, 120 y 240 procesos. Al utilizar 5, 10, 15, 30 y 60 procesos, la media de los tiempos fue de: 1,631, 1,696, 1,747, 1,664 y 1,708 respectivamente mostrados en la tabla 6.13, se observa un comportamiento uniforme en el uso del tiempo, pues conserva el balance de un proceso por núcleo, cuya tendencia sugiere que solo debería incrementarse el tiempo en proporción a la utilización de un número mayor de procesos, debido a las comunicaciones entre ellos y la latencia de la Grid, pero para el caso de utilizar 15 procesos, se observa que el tiempo se incrementa ligeramente debido al ruido de procesos ajenos que afecto el algoritmo, cabe mencionar, que este tipo de sucesos son difíciles de determinar, ya que potencialmente cualquier usuario que posea una cuenta, puede entrar a la Grid y ejecutar procesos.

Al utilizar 120 procesos e involucrar los dos clusters Cuexcomate y Texcal, se observa un aumento significativo en un poco más de 200 segundos con respecto a utilizar 60 procesos, al pasar de utilizar 1,708 a 1,996 segundos a pesar de seguir conservando el equilibrio de un proceso por núcleo, este incremento de tiempo se debe a la utilización de la VPN, el cual proporciona una alta latencia, finalmente al utilizar 240 procesos y sobrecargar el cada núcleo con dos procesos, se duplica el tiempo en relación a la utilización de 120 procesos lo que resulta en un tiempo de 4,857 segundos.

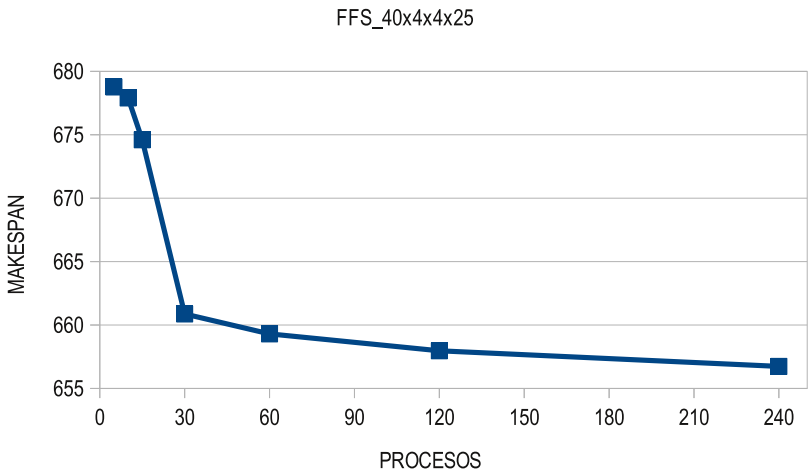

Figura 6.13. *Media de la eficacia de la cooperación de procesos para FFS_40x4x4x25. La figura muestra la eficacia al utilizar diferente número de procesos, con una tendencia bien definida, la cual le permite mejorar de forma constante, tanto la media, como el mejor valor del makespan, en el orden creciente: 5, 10, 15, 30, 60, 120, 240, así mismo, no se observa convergencia pues se requiere un mayor número de procesos, sin embargo, esto lleva a duplicar los tiempos de procesamiento, ya que solo se cuenta con 120 núcleos reales.*

La figura 6.15, muestra la media los tiempos paralelos T_p y los tiempos secuenciales T_s obtenidos a partir de la ecuaciones 6.2 y 6.3. La figura 6.15 a) muestra tiempos paralelos de 3,262, 5,088, 6,989, 13,311, 25,619, 75,205 y 145,698 al utilizar 5, 10, 15, 30, 60, 120 y 240 procesos respectivamente mostrado en la tabla 6.13, se muestra que, la utilización de un número pequeño de procesos como en el caso de 5, resulta en un número mayor de agrupaciones que pueden distribuirse sobre la Grid, en consecuencia al utilizar el método SDAAP-MI, permite reducir el número de pruebas por agrupación, conduciendo esto a reducir el tiempo requerido para obtener las 30 pruebas, a medida que aumenta el número de procesos, son menos las agrupaciones que pueden distribuirse y ejecutarse en forma paralela, lo que conlleva a un aumento en los tiempos en paralelo.

La figura muestra un comportamiento creciente, en los tiempos a medida que aumenta el número de procesos utilizados, una diferencia mayor se observa a partir de 120 procesos, que involucran el uso de la Grid, debido a la alta latencia y ruido en los procesos, seguido de 240 procesos, que duplican el tiempo por la sobrecarga de los núcleos.

La 6.15 b), muestra la relación de tiempo entre utilizar el método SDAAP-MI en paralelo y la utilización de un solo núcleo en forma secuencial, donde el mejor de los casos, muestra que para 5 procesos el tiempo, es de 3,262 segundos (∼1 hora) al utilizar 24 agrupaciones de 5 procesos, donde cada SDAAP se ocupa de dos pruebas, pero que al utilizar un solo núcleo el tiempo requerido es de 244,640 segundos (∼68 horas), para el peor de los casos, al utilizar 240 procesos, el tiempo el paralelo es de 145,698 segundos

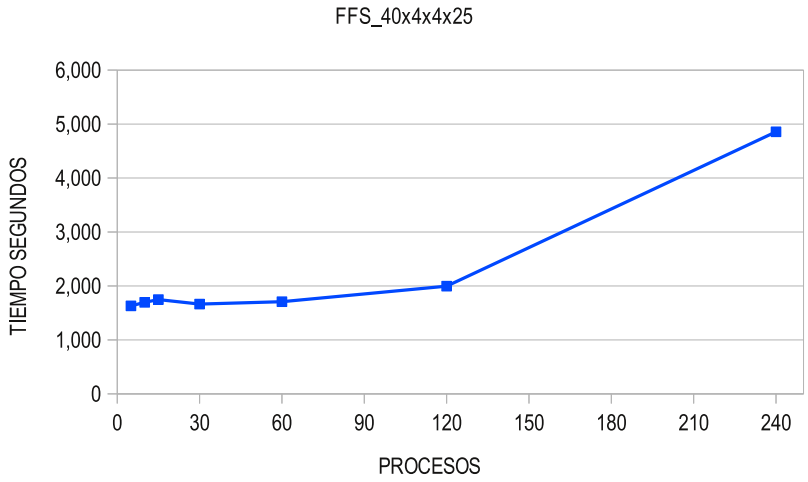

Figura 6.14. *Media de tiempos de la cooperación de procesos para FFS_40x4x4x25 con diferente número de procesos. La figura muestra un comportamiento ideal al utilizar 5, 10, 15, 30 y 60 procesos, para los cuales el tiempo se mantiene uniforme, al utilizar 120 procesos se muestra un aumento en el tiempo, debido a la alta latencia de la VPN, finalmente al utilizar 240 procesos se duplica la carga de los núcleos y aumenta al doble el tiempo con respecto a la utilización de 120.*

(∼40 horas) y el tiempo secuencial de 17,483,760 segundos (∼202 días).

6.4.4.2. Cooperación de procesos en instancias medianas.

La tabla 6.14, muestra los resultados de la cooperación obtenidos para la instancia FFS_60x4x4x25, al utilizar diferente número de procesos (P), sobre un total de agrupaciones (N_p), donde cada agrupación esta formado por un número de núcleos (N), que requieren ejecutar (#P) pruebas, cada uno para alcanzar las 30 requeridas utilizando el método SDAAP-MI propuesto, esta experimentación defiere de los problemas de tamaño chico analizados anteriormente, en la utilización de una VLAN que une los dos clusters Cuexcomate y Texcal y proporciona un ancho de banda de 15 Mb/s bidireccional a diferencia de la VPN que solo proporcionaba 3 Mb/s.

Para este instancia la tabla muestra, que el mejor makespan se obtiene al utilizar 240 procesos que es la mejor cota encontrada $UB = 927$ y muestra que tanto la media de las 30 pruebas del makespan como el mejor valor encontrado, mejora en forma constante a partir de utilizar 15 procesos en el orden 15, 30, 60, 120, 240.

La figura 6.16, muestra el comportamiento de la cooperación en la media de 30 pruebas al utilizar diferente número de procesos, muestra que para el caso de utilizar 5 y 10 procesos, se encuentran valores de 943 y 940.27 para la media del makespan y de 937

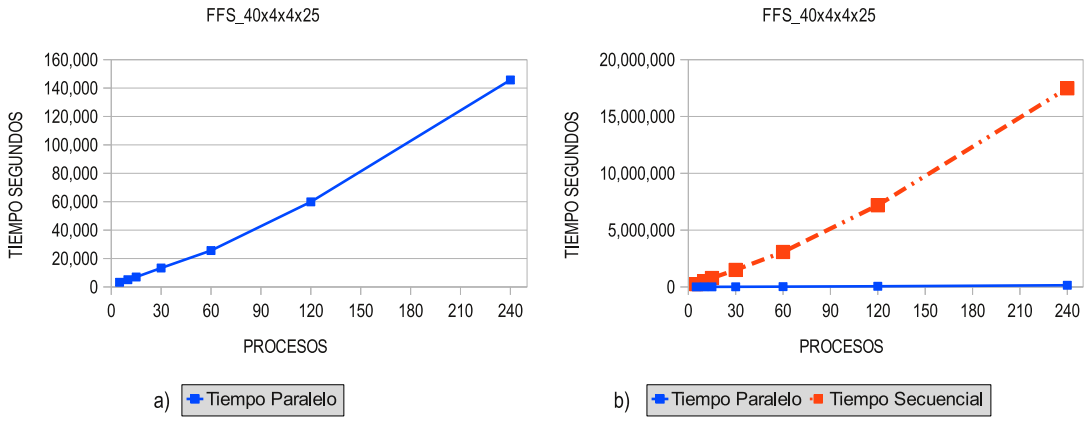

Figura 6.15. *Tiempos en paralelo y secuencial de las pruebas para FFS_40x4x4x25. La figura muestra la gran diferencia en tiempo necesario para realizar 30 pruebas en forma paralela, utilizando el método propuesto SDAAP-MI y en forma secuencial utilizando un solo núcleo.*

y 932 para el mejor makespan, estos valores solo son mejorados al utilizar 30 procesos para la media y 120 procesos para el mejor valor, para este problema en especifico, lo ideal es utilizar 120 o 240 procesos para encontrar tanto la mejor media como el mejor valor del makespan, ya que utilizar un número inferior de procesos lo ideal es utilizar 10, que encuentra el mismo makespan que utilizar 60, en este sentido la media mejora, pero no puede encontrar un mejor valor hasta llegar a los 120 procesos.

Para este problema no se observa una convergencia al utilizar 240 procesos, por lo que se requiere sobrecargar cada núcleo con un número mayor de procesos, pero para problemas de tamaño medio esté, repercute seriamente en el tiempo, por tanto podemos intuir una mejora constante a medida que utilizamos un mayor número de procesos.

La figura 6.17, muestra la media de los tiempos al utilizar diferente número de procesos en el orden 5, 10, 15, 30, 60, 120 y 240, el balance en la asignación de un proceso por núcleo se mantiene hasta alcanzar los 120 núcleos, en la figura podemos observar que la nueva configuración de la VLAN, proporciona una comunicación estable y suficientemente grande para mantener los tiempos uniformes, al utilizar procesos en el rango $[5, 240]$.

El incremento constante mostrado en la gráfica, representa el tiempo adicional que consume una mayor comunicación entre los procesos en el paso de mensajes y en los mensajes para la sincronización, observe que al pasar de 60 a 120 procesos, al utilizar la Grid y la VLAN que comunica los dos clusters Cuexcomate y Texcal, la latencia es mucho menor al tener un ancho de banda más grande, lo que beneficia los tiempos para mantenerlos uniformes, a diferencia de la comunicación de las instancias pequeñas, analizados anteriormente en las figuras 6.11 y 6.14, que utilizaban una VPN, donde el tiempo se incrementaba de manera considerable, finalmente al utilizar 240 procesos, el tiempo se duplica al doble al sobrecargar cada núcleo con una doble carga de trabajo

FFS_60x4x4x25									
P	N	Np	#P	$\overline{C_{max}}$ 30 pruebas	$\overline{T}$ 30 pruebas	Best	UB	Tp	Ts
5	5	24	2	943.00	3,889	937	927	7,778	583,330
10	10	12	3	940.27	3,890	932	927	11,670	1,167,020
15	15	8	4	941.13	3,988	934	927	15,950	1,794,405
30	30	4	8	939.37	3,968	933	927	31,745	3,571,260
60	60	2	15	937.53	4,047	932	927	60,707	7,284,780
120	120	1	30	935.73	4,076	930	927	122,276	14,673,120
240	120	1	30	934.37	7,953	927	927	238,578	28,629,360

Tabla 6.14. *Resultados de la experimentación para FFS_ 60x4x4x25. La figura muestra que el mejor makespan se da al utilizar 240 procesos UB=927, observándose una mejora constante a partir de utilizar 15 procesos o más en el orden 15, 30, 60, 120, 240, la utilización de 10 ó 60 procesos obtienen el mismo makespan y solo se mejora al utilizar 120 procesos o más.*

como se muestra en la figura.

Este comportamiento uniforme de los tiempos es el ideal, ya que para esta experimentación no se observan ruidos de procesos externos y la latencia es lo suficientemente baja para no afectar el comportamiento del algoritmo por lo que la ejecución de las pruebas se da en un entorno ideal.

La figura 6.18 a), muestra el comportamiento de los tiempos requeridos para las 30 pruebas al utilizar 5, 10, 15, 30, 60, 120 y 240 procesos en forma paralela, utilizando el método SDAAP-MI y en forma secuencial al utilizar un solo núcleo, la figura muestra un comportamiento casi lineal, debido a que los tiempos se duplican cada vez, los tiempos paralelos para conseguir las 30 pruebas fueron 7,778, 11,670, 15,950, 31,745, 60,707, 122,276, 238,578, mostrando una uniformidad en el tiempo en relación al número de procesos utilizados.

Cuando la relación es al doble, como el el caso de pasar de 15 a 30 procesos, la relación es aproximadamente 2 a 1 al pasar de 15,950 segundos a 31,754 segundos, esta relación se mantiene al pasar de 30 a 60 procesos en una relación de 31,754 segundos a 60,707 segundos, y así sucesivamente hasta llegar a los 240 núcleos, en entorno ideal no permite ver la diferencia al pasar de 60 a 120 procesos, que involucran el uso de la Grid, ya que las comunicaciones son suficientemente robustas para mostrar un comportamiento uniforme.

La diferencia se da con respecto a la utilización de un solo núcleo, la figura 6.18 b), muestra los tiempos calculados en paralelo y los tiempos calculados en forma secuencial con base en las ecuaciones 6.2 y 6.3 respectivamente, los tiempos calculados para el mejor de los casos, al utilizar 5 procesos, muestran un tiempo paralelo $T_p = 7,778$ segundos

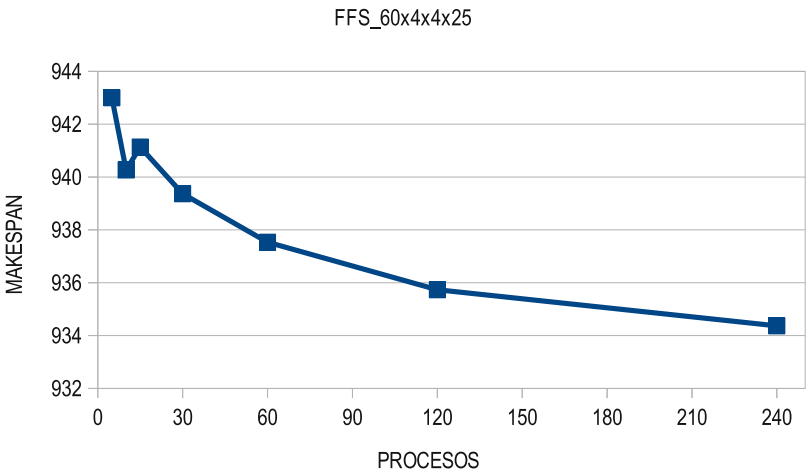

Figura 6.16. *Media de la eficacia de la cooperación de procesos para FFS_ 60x4x4x25. La figura muestra que la media de makespan, tiene una mejora constante a partir de la utilización de 30 procesos o más, para este problema la mejora real en la media se da al utilizar 60, 120 y 240 procesos en relación a un número inferior.*

($\sim$ 2 horas) y un tiempo secuencial $T_s = 583,330$ segundos ($\sim$162 horas) y en el peor de los casos, al utilizar 240 procesos muestra un tiempo paralelo $T_p = 238,578$ segundos ($\sim$66 horas) y un tiempo secuencial $T_s = 28,629,360$ segundos ($\sim$331 días).

La tabla 6.15, muestra los resultados obtenidos para la instancia FFS_80x4x4x25 al ejecutar 30 pruebas por cada diferente número de procesos, para esta instancia al igual que la instancia anterior, en donde al utilizar 5, 10, 15, 30 y 60 se utiliza la comunicación interna del cluster y al utilizar 120 y 240 procesos se utiliza una VLAN, con una capacidad de 15 Mb/s bidireccional para los dos clusters.

La tabla 6.15, muestra que la cooperación de procesos obtiene mejores resultados a medida que se aumenta en forma constante su número en el orden 5, 10, 15, 30, 60, 120 y 240. Para esta instancia, a diferencia del problema anterior mostrado la tabla 6.14, en donde la media y el mejor makespan solo mejoran en forma constante, a partir de 120 procesos, para esta instancia se observa una mejora constante a partir de los 5 procesos y hasta los 240, observándose que el mejor makespan, es cuando se utilizan 240 procesos $UB = 1263$, que es la mejor cota encontrada para este problema propuesto.

La figura 6.19, muestra el comportamiento de la cooperación en la media de 30 pruebas al utilizar diferente número de procesos, la figura muestra que, tanto la media como el mejor valor obtenido del makespan, se mejora en forma constante a partir de utilizar 5 procesos que dan una media de 1,282 y un mejor valor de 1273, para el caso de utilizar 10 procesos, la media mejora con un valor de 1,279 y un mejor valor de 1,272, este comportamiento continúa al encontrar valores en la media de 1,278, 1,275, 1,274, 1,272, 1,271 y los mejores valores del makespan de 1,270, 1,269 1,266, 1,264, 1,263 al utilizar

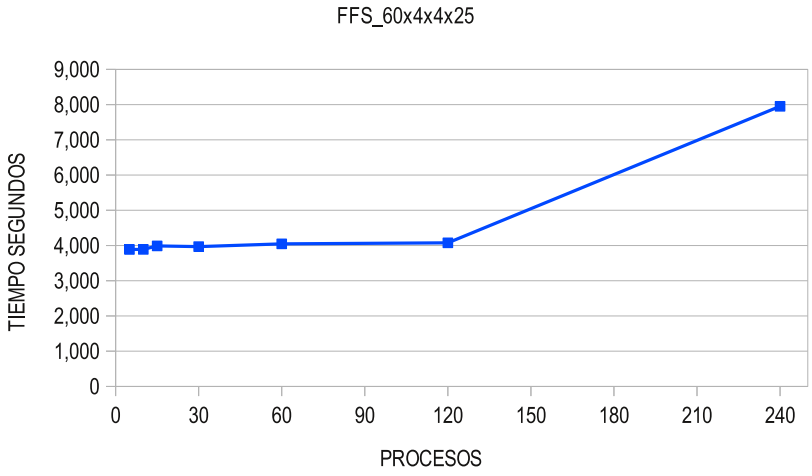

Figura 6.17. *Media de tiempos de la cooperación de procesos para FFS_ 60x4x4x25. La figura muestra que los tiempos mantienen una consistencia uniforme, al utilizar un número de procesos en el rango [5,120], en donde el balance se mantiene al asignar un proceso por núcleo, el punto critico de las comunicaciones, se da al utilizar los dos clusters Cuexcomate y Texcal, con la diferencia que ahora se utiliza una VLAN, lo que brinda un ancho de banda más grande y por tanto una latencia menor, esta mejora se refleja en que los tiempos que involucran la utilización de los dos clusters, se mantiene uniformes y solo se incrementan al doble, al sobrecargar los núcleos con una doble carga.*

15, 30, 60, 120, 240 procesos respectivamente, así mismo no se observa una convergencia en el tamaño de procesos por lo que se requiere utilizar un número más grande.

La figura 6.20, muestra la media de los tiempos al utilizar diferente número de procesos y al igual que la instancia anterior, mantiene un balance en la asignación de un proceso por núcleo al utilizar 5, 10, 15, 30, 60 y 120 procesos, la figura muestra que, en el rango [5,120] la latencia no afecta el tiempo de los procesos ya que los tiempos se mantienen casi iguales y el ligero incremento que se observa al aumentar el tamaño de los procesos, se debe al paso de mensajes y a los mensajes de sincronización de los procesos.

Podemos observar que la VLAN proporciona una buena comunicación, lo que ayuda mucho a no mostrar un incremento en los tiempos al pasar de 60 a 120 procesos, con el uso de los dos clusters, este comportamiento es parecido al mostrado en la figura 6.17, para la instancia FFS_60x4x425, ya que se mantienen los tiempos uniformes y la experimentación no muestra ruidos de procesos externos y la latencia es lo suficientemente baja para no afectar el comportamiento del algoritmo, por lo que la ejecución de las pruebas, se da igualmente en un entorno ideal, hacia el final al utilizar 240 procesos y sobrecargar cada núcleos con dos procesos, el tiempo utilizado para 120 procesos se duplica como se observa en la figura.

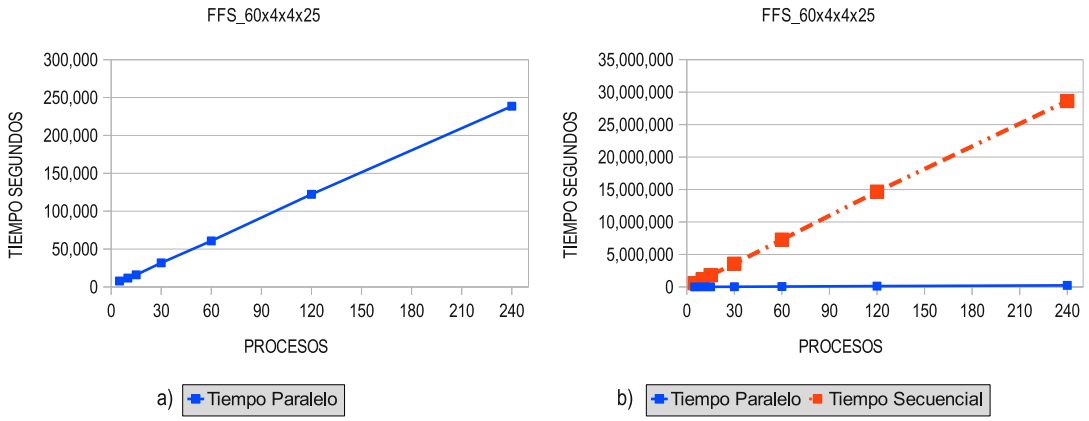

Figura 6.18. *Tiempos en paralelo y secuencial de las pruebas para FFS_ 60x4x4x25. La figura muestra un comportamiento lineal uniforme a medida que aumenta el número de procesos, no se observa diferencia al pasar de 60 a 120 procesos, entre los dos clusters debido a que la VLAN proporciona una comunicación suficientemente buena, para no afectar al algoritmo, la figura muestra para el mejor de los casos, al utilizar 5 procesos, un tiempo en paralelo de* $T_p = 7,778$ *segundos (∼ 2 horas) y un tiempo secuencial de* $T_s = 583,330$ *segundos (∼162 horas) y en el peor de los casos, al utilizar 240 procesos muestra un tiempo paralelo* $T_p = 238,578$ *segundos (∼66 horas) y un tiempo secuencial* $T_s = 28,629,360$ *segundos (∼331 días).*

La figura 6.21 a), muestra los tiempos utilizados por el algoritmo al llevar a cabo las 30 pruebas con diferente número de procesos sobre la Grid y la figura 6.21 b), muestra la misma cantidad de pruebas pero calculados a partir de la utilización de un solo núcleo, a partir de las formulas 6.2 y 6.3, podemos observar como los tiempos fluctúan en el mejor de los casos, al utilizar 5 procesos en un tiempo paralelo de 13,042 segundos (∼ 4 horas) y un tiempo secuencial de 978,155 segundos (∼11 días) y para el peor de los casos al utilizar 240 procesos, en un tiempo paralelo de 398.974 segundos (∼5 días) y un tiempo secuencial de 47,876,880 segundos (∼554) días. Para estas instancias de tamaño mediano podemos observar que los tiempos secuenciales se vuelven intratables sin el uso de una Grid.

6.4.4.3. Cooperación de procesos en instancias grandes.

Para el análisis de este tamaño de problemas se siguió el mismo plan de pruebas utilizado para las instancias pequeñas y medianas, los valores óptimos de los parámetros, para esta instancia son: $t_o = 40$, $m = 70$, $\mu = 0.990$, $t_f = 0.0001$ para RS, $h = 16$, $\alpha = 1.0$, $\beta = 0.4$, $\gamma = 0.9$, $\delta = 0.2$, $p = 0.2$, $q = 1500$ para SCH y $T = 30$, $C = 0.5$, $G = \{5, 10, 15, 30, 60, 120, 240\}$ para AG. La tabla 6.16, muestra los resultados de la experimentación para la instancia FFS_140x4x4x25 donde P el número de procesos, N el número de núcleos, N_p el número de agrupaciones de tamaño N, #P el número de pruebas, $\overline{C_{max}}$ la media de las 30 pruebas, $\bar{T}$ la media en tiempo de las 30 pruebas, $Best$ el mejor makespan obtenido, UB la mejor cota encontrada, T_p el tiempo

FFS_80x4x4x25									
P	N	Np	#P	$\overline{C_{max}}$ 30 pruebas	$\overline{T}$ 30 pruebas	Best	UB	Tp	Ts
5	5	24	2	1,282	1,282	1,273	1,263	13,042	978,155
10	10	12	3	1,279	1,279	1,272	1,263	19,441	1,944,070
15	15	8	4	1,278	1,278	1,270	1,263	26,424	2,972,700
30	30	4	8	1,275	1,275	1,269	1,263	52,940	5,955,780
60	60	2	15	1,274	1,274	1,266	1,263	100,721	12,086,520
120	120	1	30	1,272	1,272	1,264	1,263	204,082	24,489,840
240	120	1	30	1,271	1,271	1,263	1,263	398,974	47,876,880

Tabla 6.15. *Resultados de la experimentacion para FFS_ 80x4x4x25. La tabla muestra para la media de las 30 pruebas del makespan y para el mejor resultado de las 30 pruebas, una mejora constante mientras se utiliza un número de procesos cada vez mayor, en el orden 5, 10, 15, 30, 15, 60, 120 y 240, la figura también muestra que el mejor makespan, es cuando se utilizan 240 procesos para encontrar UB=1263 la mejor cota para este problema.*

calculado de las 30 pruebas en paralelo y T_s el tiempo calculado en forma secuencial.

La tabla muestra que el peor makespan de 2,022, se da al utilizar 5 procesos y que el mejor makespan de 2,013 se da al utilizar 240 procesos, siendo esta última $UB = 2,013$, la mejor cota conocida para esta instancia de prueba propuesta.

La figura 6.22, muestra el comportamiento de la cooperación al utilizar diferente número de procesos, se observa que la media mejora constantemente a medida que se aumenta el número de procesos en el orden 5, 10, 15, 30, 60, 120, 240, también se observa que el mejor resultado del makespan obtenido de las 30 pruebas ($Best$), mejora progresivamente pero no constantemente al utilizar 5, 15, 60, ya que utilizar 15 ó 30 procesos dan el mismo resultado de 216, así mismo al utilizar 60, 120 y 240 procesos dan el mismo resultado de 2013, pero que en ningún caso empeora.

Para este tamaño de instancias muy grandes, se observa la dureza del problema al tratar de mejorar la solución, utilizando un número de procesos mayor. Si bien las medias mejoran, lo que nos da una idea de que al utilizar 480 procesos, se pueda conseguir una mejor solución al cuadruplicar la carga de cada núcleo, también se puede observar, que el tiempo se duplica como lo muestra la tabla 6.16, que para 240 procesos la media por prueba es de 38,727 segundos (∼11 horas), es decir que para las 30 pruebas lleva un tiempo de 1,161,820 segundos en paralelo (∼13.5 días), entonces al utilizar 480 procesos el tiempo tendería a duplicarse a aproximadamente 2,232,640 segundos (∼27) días en paralelo.

Podemos decir por lo tanto, que para el tamaño y dureza de las instancias grandes, el algoritmo tiene un buen desempeño al mejorar la media en forma constante, pero que

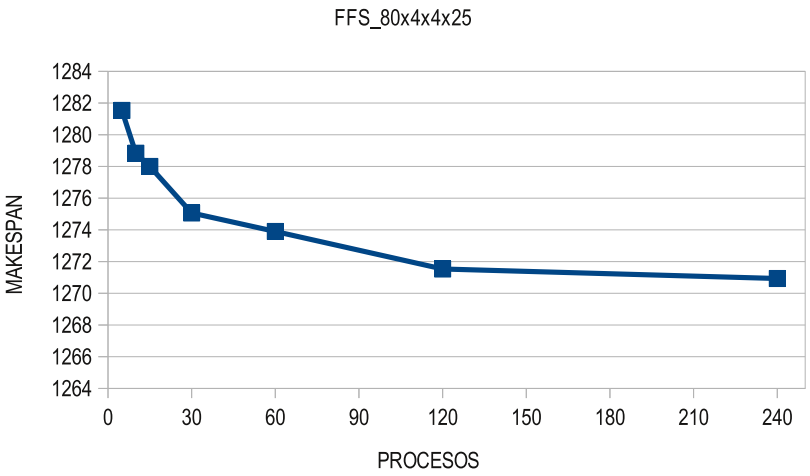

Figura 6.19. *Media de la eficacia de la cooperación de procesos para FFS_80x4x4x25. La figura muestra un aumento constante en la calidad de las soluciones a medida que se aumenta el número de procesos a partir de 5 y hasta 240, así mismo muestra que no existe convergencia en el número de procesos utilizados y que para observarla requiere de un mayor número de procesos.*

no puede mejorar la mejor solución encontrada con 60 procesos, al utilizar 120 y 240 procesos, sin embargo no empeora, lo que nos da la certeza de que con un número mayor se podría encontrar una mejor cota, por tanto no se observa convergencia aún para esta instancia de tamaño grande.

La figura 6.23, muestra la media del tiempo de las 30 pruebas sobre la Grid, de igual forma que en las gráficas de las instancias anteriores, si bien los tiempos se incrementan debido al aumento en el tamaño y dureza del problema, el balance de la asignación de un proceso por núcleo mantiene los tiempos uniformes desde 5 hasta 120 procesos, en donde la Grid alcanza su máxima capacidad, al pasar a 240 procesos, es donde se observa un aumento considerable en el tiempo debido a que se comienza la sobrecarga de los núcleos, para esta instancia grande, la media de tiempos en segundos fue de 18,921, 19,189, 19,621, 19,888, 20,159, 20,382 y 38,127 al utilizar 5, 10, 15, 30, 60, 120 y 240 procesos respectivamente, en la figura se observa un aumento en el tiempo entre un mayor número de procesos debido a la sobrecarga de las comunicaciones y al utilizar 240 procesos se observa la misma sobrecarga, más la sobrecarga doble de los núcleos.

La figura 6.24, muestra el comportamiento de los tiempos paralelos y los tiempos secuenciales calculados a partir de las ecuaciones 6.2 y 6.3, para el total de las 30 pruebas requeridas al utilizar 5, 10, 15, 30, 60, 120 y 240 procesos repartidos sobre la Grid, la figura 6.24 a), muestra que a medida que aumenta el número de procesos, disminuye el número de agrupaciones que pueden distribuirse a lo largo de la Grid, y es debido a esto, que los tiempos se incrementan en forma constante, también los tiempos secuenciales en la figura 6.24 b), muestran que los tiempos secuenciales aumentan en forma constante y

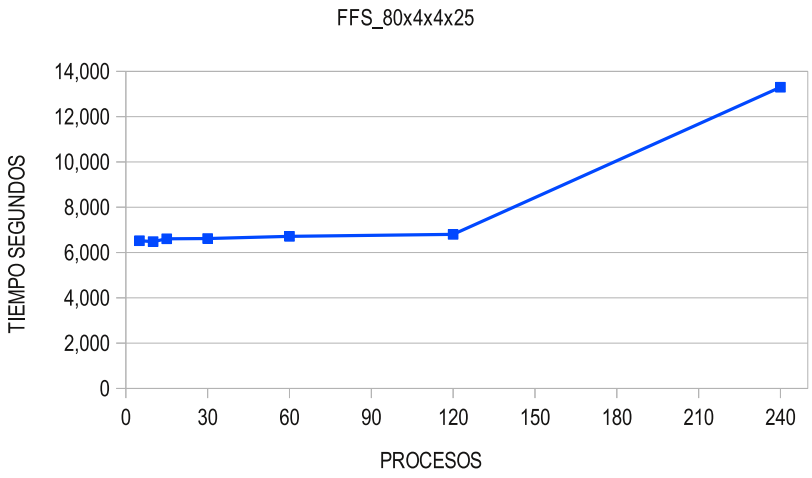

Figura 6.20. *Media de tiempos de la cooperación de procesos para FFS_ 80x4x4x25. La figura muestra tiempos uniformes en la media de los tiempos al utilizar de 5 a 120 procesos y solo se duplican los tiempos al sobrecargar cada núcleo con dos procesos al utilizar 240.*

sostenida, mientras que los tiempos paralelos, casi son imperceptibles, lo que nos da una idea del ahorro de tiempo que conlleva utilizar muchos núcleos para la experimentación y que sin ella tendríamos tiempos intratables, p. ej. para el mejor de los casos la ejecución de las 30 pruebas, al utilizar 5 núcleos y 24 agrupaciones, nos da un tiempo de 37,843 segundos (∼11 horas) y al utilizar 1 núcleo, se alcanzan los 2,838,195 segundos (∼33 días) y en el peor de los casos al utilizar 240 procesos en paralelo, se calcula un tiempo de 1,161,820 segundos (∼13.5 días) y en secuencial se consumiría un tiempo de 139,418,400 segundos (∼ 1,613 días), esta última instancia grande pone en perspectiva las bondades en eficiencia que nos trae el uso de la Grid, al utilizar un número de procesos grande sobre los tiempos secuenciales.

Finalmente, después de llevar a cabo el análisis de los resultados, podemos observar la reducción en tiempo para alcanzar la eficacia del algoritmo en la Grid, además de darnos una idea clara de los tiempos permisibles, que se tienen para diseñar algoritmos parecidos o mejores, conduciendo esto a una mayor exploración y explotación de espacio de soluciones y con ello encontrar mejores cotas en la experimentación.

6.5. Análisis de la acelaración (Speedup) y su eficiencia.

Durante la planeación del diseño de un algoritmo paralelo, es de mucha importancia poder predecir con exactitud el rendimiento que tendrá, esto con el fin de saber si los esfuerzos destinados a la programación, pruebas, depuración y experimentación será una buena inversión, así mismo poder medir el tiempo de ejecución de un programa paralelo, ayuda a comprender mejor las barreras físicas de un buen desempeño y ayudar a decidir,

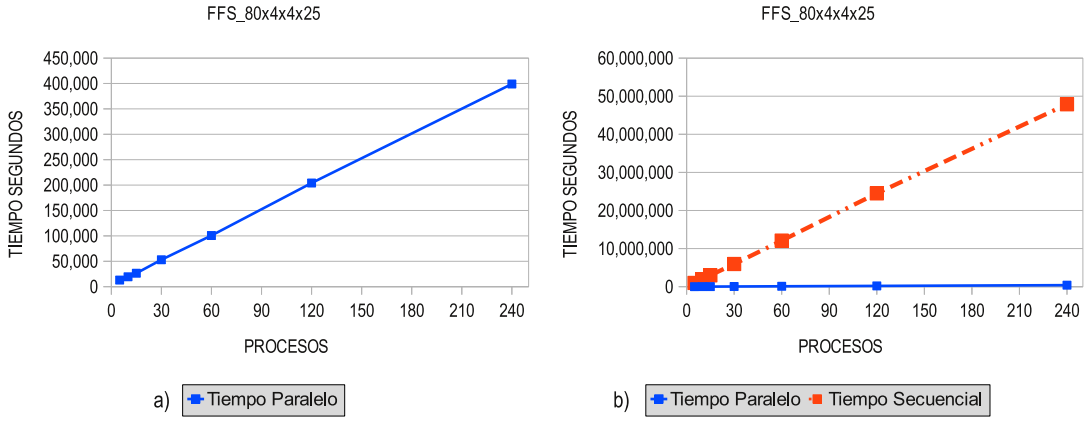

Figura 6.21. *Tiempos en paralelo y secuencial de las pruebas para FFS_ 80x4x4x25.*

si un mayor número de núcleos, ayudaran a obtener un mejoría en el rendimiento del algoritmo.

Los algoritmos paralelos se diseñan con el fin de que su ejecución sea mucho más rápida que su contraparte secuencial, en este sentido la aceleración (Speedup) es el radio diferencial entre los tiempos obtenidos de la ejecución paralela y secuencial, de esta manera la aceleración se calcula con la ecuación 6.4.

$$Speedup = \frac{tiempo\,ejecución\,secuencial}{tiempo\,ejecución\,paralela} \tag{6.4}$$

Para el caso de un cálculo más minucioso, se observa que las operaciones efectuadas por un algoritmo paralelo, se ven afectadas por tres de variables:

1. Operaciones que deben ser ejecutadas en orden secuencial $\sigma(n)$.

2. Operaciones que pueden ser ejecutadas en paralelo $\varphi(n)$.

3. Sobrecarga de las comunicaciones (latencia) necesarias para las operaciones $\kappa(n,p)$.

Con estas tres variables, entonces se puede definir un modelo simple para el cálculo de la aceleración (Speedup), donde $\psi(n,p)$ es la aceleración alcanzada por al algoritmo paralelo al resolver un problema de tamaño n utilizando p núcleos, $\sigma(n)$ denota la porción del algoritmo secuencial, es decir, la parte del algoritmo que no puede ser paralelizada, $\varphi(n)$ denota la porción que se puede ejecutar en paralelo, y finalmente $\kappa(n,p)$ denota el tiempo requerido por la sobrecarga de las comunicaciones al utilizar p núcleos, las cuales dependen de la alta o baja latencia, este tiempo es requerido para establecer la sincronización de los procesos distribuidos en la Grid y los mensajes de comunicación que se dan entre ellos.

FFS_140x4x4x25									
P	N	Np	#P	$\overline{C_{max}}$ 30 pruebas	$\overline{T}$ 30 prueba	Best	UB	Tp	Ts
5	5	24	2	2,033.2	18,921	2,022	2,013	37,843	2,838,195
10	10	12	3	2,028.1	19,189	2,015	2,013	57,567	5,756,740
15	15	8	4	2,028.0	19,621	2,016	2,013	78,485	8,829,570
30	30	4	8	2,024.9	19,888	2,016	2,013	159,107	17,899,500
60	60	2	15	2,022.4	20,159	2,013	2,013	302,380	36,285,540
120	120	1	30	2,010.9	20,382	2,013	2,013	611,454	73,374,480
240	120	1	30	2,018.6	38,727	2,013	2,013	1,161,820	139,418,400

Tabla 6.16. *Resultados de la experimentación para FFS_140x4x4x25. La figura muestra los resultados de la ejecución de 30 pruebas, para un número de procesos de 5, 10, 15, 30, 60, 120 y 240 procesos, se muestran la media del makespan ($\overline{C_{max}}$), la media en tiempo ($\overline{T}$), la mejor resultado obtenido de las 30 pruebas ($Best$), la mejor cota encontrada (UB) la mejor cota encontrada, el tiempo calculado de las 30 pruebas en paralelo (T_p) y el tiempo calculado en forma secuencial (T_s).*

La ejecución del algoritmo en un solo núcleo del procesador, solo puede realizar una operación a la vez, es decir una ejecución secuencial, entonces solo requiere un tiempo $\sigma(n) + \varphi(n)$ para ejecutar todas las operaciones, así mismo no requiere comunicaciones entre procesos ya que solo es uno, por tanto no hay sobrecarga y no requiere del parámetro $\kappa(n, p)$.

Ahora consideremos el tiempo de ejecución del algoritmo paralelo, para este caso, la parte secuencial dado por $\sigma(n)$ del algoritmo, no se beneficia de la paralelización sin importar que existan muchos núcleos disponibles, sin embargo, la parte paralelizable del algoritmo dado por $\varphi(n)$, si se beneficia de los núcleos disponibles, ya que puede ser perfectamente dividido entre el número de núcleos disponibles de la Grid, en este caso el tiempo requerido para la ejecución del algoritmo es de $\varphi(n)/p$, finalmente hay que considerar los tiempos de sincronización y paso de mensajes entre los procesos distribuidos a lo largo de la Grid dado por $\kappa(n, p)$, con estas premisas podemos determinar la ecuación 6.5, para el cálculo más preciso de la aceleración (Speedup).

$$\psi(n, p) = \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \frac{\varphi(n)}{p} + \kappa(n, p)} \tag{6.5}$$

Observe que la parte secuencial del algoritmo permanece constante, sin importar el número de núcleos que la Grid dispone, mientras que la parte paralelizable si se beneficia directamente de este número, por tanto, para obtener el mejor beneficio de la paralelización, se debe tener una parte secuencial pequeña y una parte paralelizable grande, en este sentido el tiempo requerido para ejecutar la parte paralela, disminuye

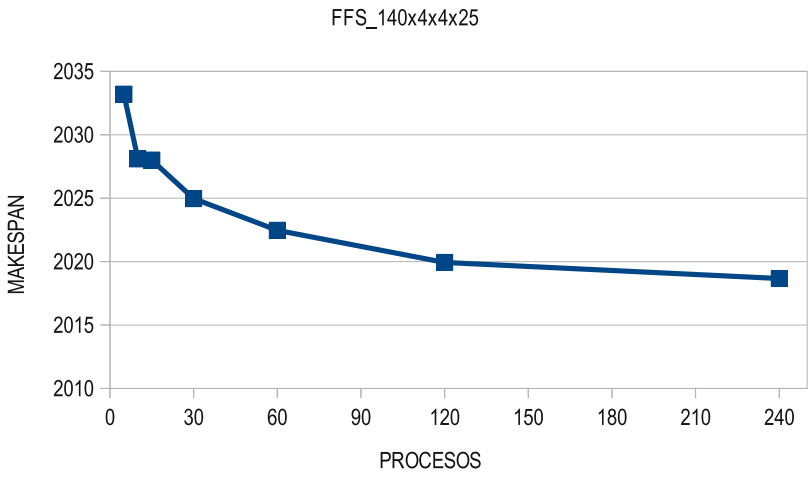

Figura 6.22. *Media de la eficacia de la cooperación de procesos para FFS_ 80x4x4x25. La figura muestra una mejora constante, para la media de 30 pruebas al utilizar 5, 10, 15, 30, 60, 120 procesos, muestra que al utilizar 10 y 15 procesos, la media casi es idéntica, pero al pasar a un número mayor de procesos mejora en forma constante y no se observa convergencia por lo que se requiere un número mayor de procesos.*

mientras más núcleos se utilicen, sin embargo, al aumentar el número de núcleos, también aumenta la sobrecarga de las comunicaciones, al tener un mayor número de procesos que sincronizar y comunicar, por lo que la tendencia del tiempo puede incrementarse en lugar de reducirse debido la sobrecarga de las comunicaciones.

La eficiencia del algoritmo se define como la medida de la utilización de los núcleos de procesamiento y se obtiene al dividir la aceleración (Speedup) entre el número de procesadores usados, como se observa en la ecuación 6.6, en donde $0 \leqslant \varepsilon(n,p) \leqslant 1$.

$$\varepsilon(n,p) = \frac{\sigma(n)+\varphi(n)}{p\left(\sigma(n)+\frac{\varphi(n)}{p}+\kappa(n,p)\right)} \tag{6.6}$$

6.5.1. Resultados del acelaración (Speedup) y eficiencia

Para llevar a cabo el cálculo de la aceleración del algoritmo propuesto, se utilizó una instancia representativa de tamaño n=FFS_80_4x4x4x25 y un número de núcleos $p = \{1, 10, 30, 60, 120\}$, donde al utilizar 1 núcleo representa el tiempo secuencial $\sigma(n) + \varphi(n)$ y al utilizar un número de núcleos mayor a uno, representa el tiempo en paralelo $\sigma(n) + \frac{\varphi(n)}{p}$, la planeación de la ejecución consiste en dividir el número de procesos, de tal forma que siempre se utilice la Grid, esto es, al utilizar 10 procesos, 5 se ejecutan en el cluster Cuexcomate y 5 en el cluster Texcal, el proceso maestro siempre se localiza

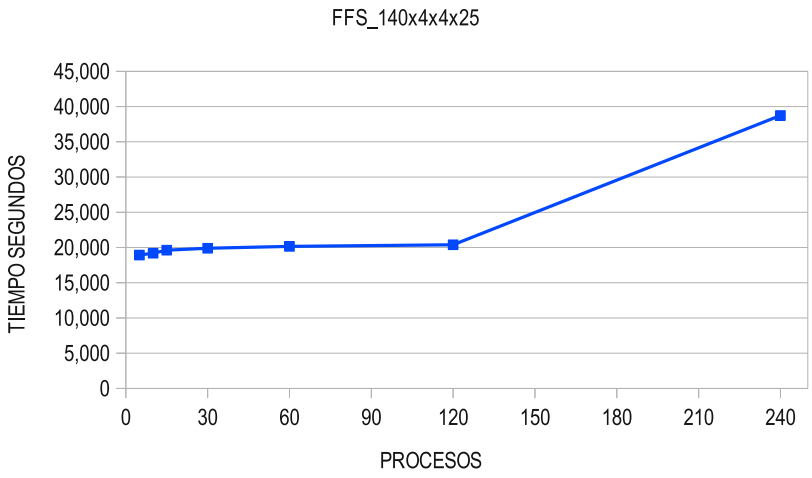

Figura 6.23. *Media de tiempos de la cooperación de procesos para FFS_140x4x4x25. La figura muestra un comportamiento uniforme, en la media de tiempos al utilizar 5, 10, 15, 30, 60 y 120 procesos donde se mantiene un balance de un proceso por núcleo, el ligero aumento en el tiempo, al pasar a un número más grande de procesos, es debido a la sobrecarga en las comunicaciones, finalmente al utilizar 240 procesos, se ven reflejados dos sobrecargas: la sobrecarga en las comunicaciones y la sobrecarga de dos procesos por núcleos.*

en el cluster Cuexcomate, al utilizar 30 procesos, 15 se ejecutan en Cuexcomate y 15 el Texcal, con 60 se ejecutan 30 y 30 en cada cluster, finalmente 120 corresponde a 60 y 60.

La parte más difícil de calcular es la sobrecarga $\kappa(n, p)$, dado que esta variable en las mediciones del algoritmo no se separa, si no que se incluye dentro del tiempo paralelo junto a otros parámetros, como los tiempos de sincronización y la carga propia del sistema y sus servicios, para cuestiones prácticas es un tanto difícil poder calcularlo, sin embargo, se ideó una forma aproximada para obtenerlo y consiste en los siguientes pasos:

1. Dados dos procesos ubicados en cada extremo de la Grid, uno en el cluster Cuexcomate y uno en el cluster Texcal, se llevaron a cabo mediciones para determinar el tiempo que lleva enviar un conjunto de bits de ida y regreso, considerando un ancho de banda de 15 Mb/s bidireccional.

2. Los conjuntos de datos a enviar y recibir se definen como una secuencia de bits de 2^n_{bits}, comenzando con $n = 6, 7, ..., 26$, que cubre el rango de datos que se pueden enviar en un segundo y que alcanzan y sobrepasan el límite de los 15 Mb/s.

3. Para la instancia de prueba, se calcula el tamaño de una solución (individuo) en bits, después se multiplica por el número de procesos utilizados (120) y se determina el tiempo que le toma ir y regresar, comparándolo con un tamaño similar

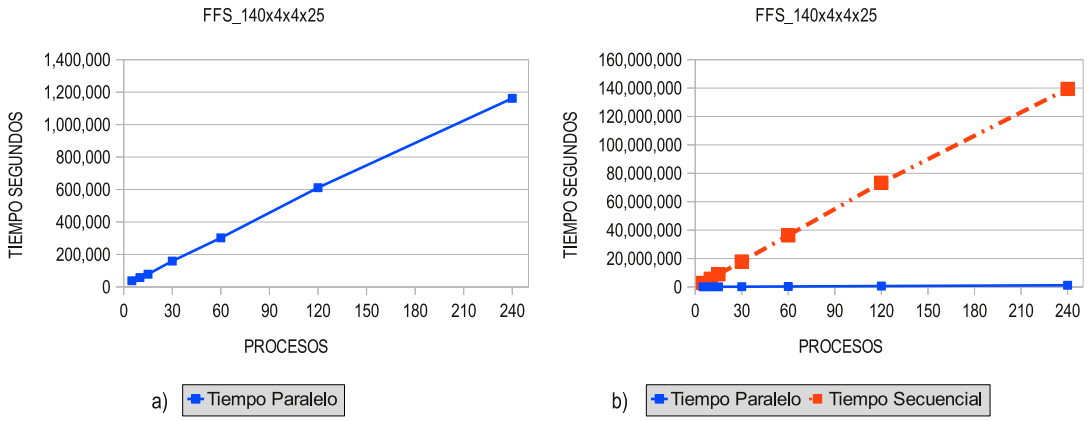

Figura 6.24. *Tiempos en paralelo y secuencial de las pruebas para FFS_ 140x4x4x25. La figura muestra las bondades de utilizar un número grande de procesos uniformemente distribuidos a lo largo de la Grid sobre un total de 120 procesos, como se observa al calcular los tiempos paralelos* $T_p = 1,161,820$ *segundos (∼13.5 días) al utilizar 240 procesos sobre 120 núcleos y su contraparte secuencial donde* $T_s = 139,418,400$ *segundos (∼ 1,613 días).*

usado en las mediciones.

4. El tiempo calculado se multiplica por el número de generaciones (30), que es el total de mensajes enviados y recibidos por el algoritmo, a fin de calcular el tiempo total de sobrecarga.

El proceso completo del cálculo de la sobrecarga, se describe a detalle en la sección 6.5.2, los resultados obtenidos con este método se muestran en la tabla 6.17, la figura muestra en la primera columna el número de procesos base utilizados $p = 120$ que equivalen a la máxima capacidad que provee nuestra Grid de 120 núcleos, la columna dos indica el número de núcleos utilizados para ejecutar los $p = 120$ procesos, comenzando con $p = 1$ que representa la ejecución secuencial y continuando con $p = \{10, 30, 60, 120\}$ para su ejecución en paralelo.

La columna tres muestra el tiempo de ejecución secuencial que es calculado en $722,215$ segundos, la columna cuatro muestra la media de tiempos de 30 ejecuciones al utilizar diferente número de procesos, la columna cinco muestra la sobrecarga aproximada generada en las comunicaciones al utilizar 120 procesos, calculada con base en el método propuesto en la sección 6.5.2, la columna seis muestra el cálculo de la aceleración (Speedup) con base en la ecuación 6.5, en la columna siete se muestra la eficiencia del uso de los procesadores, representados por núcleos con base en la ecuación 6.6, finalmente en la columna ocho se muestra la aceleración ideal.

La tabla 6.17, muestra que el tiempo secuencial $\sigma(n)+\varphi(n) = 722,210$ segundos, al ejecutar el algoritmo sobre un solo núcleo, con un peso de las comunicaciones $\kappa(n,p) = 5$ segundos calculado en la sección 6.5.2, la figura también muestra que al utilizar 10 núcleos

tomando la parte secuencial y dividiendo la parte paralela entre los 10 núcleos utilizados $\sigma(n)+\frac{\varphi(n)}{p}$, resulta en un tiempo de 78, 572 segundos, el cual representa una aceleración (Speedup) de 9.19 con respecto al ideal que es de 10, alcanzando una eficiencia del 91.9 % en la utilización de 10 núcleos, cuando se pasa a utilizar 30 núcleos, la aceleación alcanza los 27.44 con respecto al ideal de 30 que representa un 91.5 % de eficiencia, la utilización de 60 núcleos da una aceleración del 54.81 con respecto al ideal de 60 con una eficiencia del 91.3 %, finalmente al utilizar 120 núcleos alcanza una aceleración del 106.17 con respecto al ideal de 120 con una eficiencia del 88.5 %.

FFS_80x4x4x25							
#P	#N	SECUENCIAL	PARALELO	SOBRECARGA	SPEEDUP($\Psi(n,p)$)	EFICIENCIA($\varepsilon(n,p)$)	IDEAL
p	n	$\sigma(n)+\varphi(n)$	$\sigma(n)+\frac{\varphi(n)}{p}$	$\kappa(n,p)$	$\frac{\sigma(n)+\varphi(n)}{\sigma(n)+\frac{\varphi(n)}{p}+\kappa(n,p)}$	$\frac{\sigma(n)+\varphi(n)}{p(\sigma(n)+\frac{\varphi(n)}{p}+\kappa(n,p))}$	$\Psi(n,p)=p$
120	1	722,215	-	-	1.00	1.000	1
	10		78,572	5	9.19	0.919	10
	30		26,310	5	27.44	0.915	30
	60		13,170	5	54.81	0.913	60
	120		6,794	5	106.17	0.885	120

Tabla 6.17. *Resultados de la acelaración (Speedup) y eficacia del algoritmo para FFS_ 80x4x4x25. La tabla muestra el cálculo de la aceleración y eficacia de la utilización de los núcleos con base en las ecuaciones 6.5 y 6.6 respectivamente, así mismo se presentan los tiempos de la sobrecarga aproximados y la aceleración ideal esperada.*

La figura 6.25, muestra la acelaración (Speedup) ideal y el real obtenido a partir de la tabla 6.17, se muestra una aceleración sub-lineal con una eficiencia promedio calculada de la utilización de los procesadores del 0.93 %, es decir que la paralelización hace eficiente el uso de los núcleos en un promedio del 93 % al utilizar 10, 30, 60 y 120 procesos, el 7 % restante representa la parte que no se puede paralelizar y la sobrecarga de las comunicaciones, la sobrecarga calculada en esta tesis solo corresponde al tiempo utilizado por el envió y recepción de soluciones basadas en individuos, dejando fuera los tiempos de sincronización propias de MPI y de la sobrecarga de procesos ajenos y propios del sistema operativo Linux con sus servicios activos de comunicación.

El algoritmo propuesto presenta una buen desempeño y muestra una ligera caída al utilizar un número mayor de núcleos, p. ej. al utilizar 120 procesos sobre 10 núcleos tiene una caída del 8 % inicial, al utilizar 120 procesos sobre 30 núcleos presenta una caída del 9 %, conservando la misma eficiencia al utilizar 120 procesos sobre 60 núcleos, observándose un ligero descenso al decrementar la eficiencia en el orden 0.919, 0.915, 0.913, en donde estos breves descensos representa la sobrecarga de las comunicaciones.

Al ocupar la Grid y utilizar 120 procesos sobre 120 núcleos, muestra una caída aproximada del 12 %, es decir una caída adicional del 3 % con respecto a utilizar un número de núcleos menos a 120, este 3 % adicional en la caída de la eficiencia, se da al

utilizar en su totalidad los recursos de la Grid, en combinación con un ancho de banda menor pues se pasa de utilizar un ancho de banda de 1 Gb/s local del cluster, a un ancho de banda de 15 Mb/s de la interconexión global de los clusters Cuexcomate y Texcal.

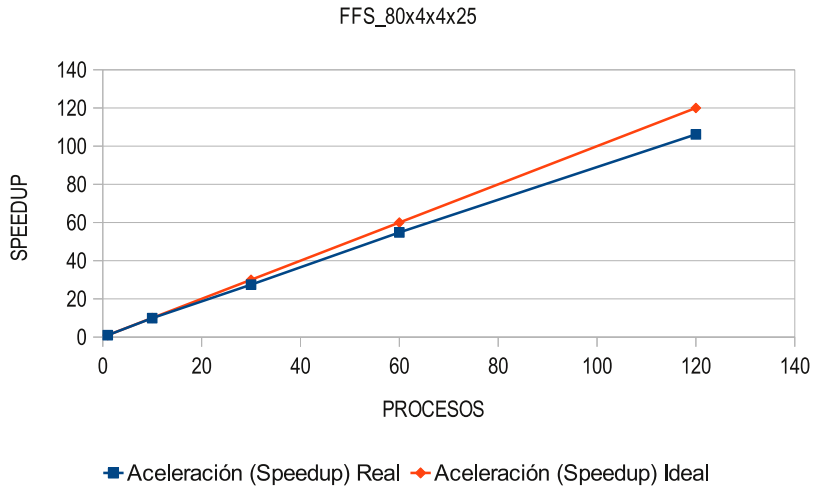

Figura 6.25. *Aceleración (Speedup) real vs aceleración ideal. La figura muestra un comportamiento sub-lineal de la aceleración del algoritmo en compraración con el ideal.*

Si el algoritmo conserva esta eficacia con una caída del 12 % con pequeñas disminuciones, debido a la sobrecarga y al no contar con un mayor número de núcleos de la Grid, se espera que el algoritmo pueda escalar linealmente en el orden de los miles de procesos a un ritmo sub-lineal, antes de comenzar a degradarse debido a que los tiempos, comenzaran a alargarse poco a poco, debido a que la sobrecarga en las comunicaciones que tendrá cada vez un mayor peso.

6.5.2. Sobrecarga

La sobrecarga se calcula sobre el ancho de banda máximo que proveen las comunicaciones entre los clusters Cuexcomate y Texcal, que es del orden de 15Mb/s, la infraestructura de la Grid Morelos permite una comunicación bidireccional, es decir, que se puede enviar y recibir al mismo tiempo en ambos sentidos (subida y bajada), por tanto el ancho de banda total bidireccional puede alcanzar los 30 Mb/s, las equivalencias usadas para medir el ancho de banda, están dadas por las ecuaciones 6.7 y 6.8.

$$15\,Mb/s = 15,360\,kb/s = 15,728,640\,bits/s \tag{6.7}$$

$$15,728,640\,bits/s = 1,966,080\,B/s = 1,920\,KB/s = 1.875\,MB/s \tag{6.8}$$

Con base en este ancho de banda, se requiere saber cual es la sobrecarga en las comunicaciones que se tiene para el problema FFS_80x4x4x25, al utilizar 120 procesos distribuidos sobre 120 núcleos que seria el peor de los casos, ya que requerirá de hacer uso de la VLAN que une los dos clusters.

6.5.2.1. Cálculo de Round Trip Time (RTT).

El RTT es el tiempo calculado que inicia, al enviar un paquete de n bits desde Cuexcomate, recibido en Texcal, reenviado de regreso desde Texcal y finalmente recibido en Cuexcomate, es lo que conoce como RTT, esta dado por la ecuación 6.9.

$$tiempo = send(n_{bits}) \rightarrow cuexcomate \rightarrow texcal \rightarrow cuexcomate \tag{6.9}$$

Los tiempos usados para calcular este transcurso de tiempo, se describen en microsegundos, esto es 10^{-6} ($\mu secs$), es decir en millonésimas de segundo, esto por los tiempos de latencia que son bajos y principalmente que la función MPI_Wtime() de la librería de paso de mensajes de MPI se adecúa a esta métrica.

La suma de bits enviados y recibidos en un paquete de tamaño n bits de ida y de regreso, esta dado por la ecuación 6.10.

$$bits_{rtt} = n_{bits} * 2 \tag{6.10}$$

El RTT se refiere al tiempo que se tarda en enviar un paquete de tamaño n_{bits}, primero de ida, más en tiempo de enviar ese mismo paquete de regreso, esta dado por la ecuación 6.11.

$$tiempo_{rtt} = tiempo_{envia}(n_{bits}) + tiempo_{regreso}(n_{bits}) \tag{6.11}$$

El ancho de banda, esta dado por el número de paquetes de tamaño $bits_{rtt}$ que se pueden enviar o recibir en un segundo, esta dado por la ecuación 6.12.

$$ancho_{banda} = \frac{\left(\frac{10^{-6}}{tiempo_{rtt}} * bits_{rtt}\right)}{1024 * 1024} \tag{6.12}$$

Observe que esta ecuación basada en el RTT, es equivalente a calcular el ancho de banda solo de ida o de regreso, ya que entonces tenemos que dividir el tiempo y el número de bits entre dos, por tanto las ecuaciones 6.12 y 6.13 son equivalentes.

$$ancho_{banda} = \frac{\left(\frac{10^{-6}}{(tiempo_{rtt}/2)} * (bits_{enviados}/2)\right)}{1024 * 1024} \tag{6.13}$$

6.5.2.2. Cálculo de la sobrecarga.

Para calcular la sobrecarga $\kappa(n, p)$ que el algoritmo AGHCGrid presenta al utilizar la instancia representativa, donde n =FFS_80x4x4x24 y $p = 120$ procesos distribuidos sobre 120 núcleos en la Grid, que es directamente proporcional al tamaño de la población, se tiene que calcular primero el tamaño en bits de una solución representada por un individuo.

Para calcular el tamaño de un individuo que representa una solución del problema, se tiene que multiplicar el número de trabajos a calendarizar, por el número de etapas, por dos, para identificar el par *trabajo:máquina,* por 4 bytes del tipo entero, por 8 bits que hay en cada Byte, esta dado por la ecuación 6.14.

$$individuo_{bits} = trabajos * etapas * 2 * 4 * 8 \tag{6.14}$$

El tamaño de la población se calcula multiplicando el tamaño de un individuo, por el número de procesos p que se ejecutan sobre la Grid, esta dado por la ecuación 6.15.

$$poblacion_{bits} = individuo_{bits} * procesos \tag{6.15}$$

El tamaño total en bits que el algoritmo envía y recibe, se calcula multiplicando el tamaño de la población, por el número de generaciones del algoritmo genético, que es 30, esta dado por la ecuación 6.16.

$$total_{bits} = poblacion_{bits} * 30 \tag{6.16}$$

La instancia representativa esta compuesta por 80 trabajos, que deben ejecutarse en serie sobre 4 etapas, en donde cada etapa tiene 4 máquinas paralelas idénticas, para ello se utiliza toda la Grid con 120 procesos, sobre 120 núcleos, con estos datos podemos calcular el tamaño en bits que ocupa el algoritmo para este problema, usando la ecuaciones 6.14, 6.15 y 6.16 se tiene que:

- $individuo_{bits} = 80(trabajos) * 4(etapas) * 2(trabajo : maquina) * 4(int) * 8(bits) = 20,478_{bits}$

- $poblacion_{bits} = 20,478_{bits}(poblacion_{bits}) * 120(procesos) = 2,457,600_{bits}$

  Observe que la repartición de los 120 procesos corresponder a 60 en el cluster Cuexcomate y 60 en el cluster Texcal, por lo que el cuello de botella, corresponde solo a la mitad de la población, es decir $1,228,800_{bits}$.

- $total_{bits} = 1,228,800_{bits}(poblacion_{bits}) * 30(generaciones) = 36,864,000_{bits}$

Ahora se necesita saber para este tamaño de bits, cuanto tiempo se consume utilizando paso de mensajes con MPI, para observar cual es el tamaño de bits ideal, para el cual MPI muestra la mejor tasa de transferencia por segundo, se toma la máxima tasa de transferencia que es de 15,728,640 *bits/s* por la ecuación 6.7, se tiene que los múltiplos de bits que se van a enviar serán hasta alcanzar y superar este tamaño, la secuencia de bits es de 2^n_{bits}, con $n = 6, 7, ..., 26$ como se muestra en la tabla 6.18, en esta tabla se observa que el ancho de banda entre Cuexcomate y Texcal se alcanza entre 2^{23} y 2^{25} bits.

2^n	bits
2^6	64
2^7	128
...	...
2^{23}	8,388,608
2^{24}	16,777,216
2^{25}	33,554,432
2^{26}	67,108,864

Tabla 6.18. *Cantidad de bits a transferir.*

Con estos datos y conversiones, se programó en C con MPI el siguiente programa:[5] mpi_ping.c para llevar a cabo la transferencia de paquetes como se muestra en el apéndice K.

Los resultados se muestran en la tabla 6.19, los datos se interpretan de la siguiente manera: El programa utiliza dos procesos uno que corre en el nodo Cuexcomate (maestro) y uno que corre en el nodo esclavo (Texcal), en la primera línea de resultados se observa que para enviar 64 bits = 2^6_{bits}, de ida y vuelta (RTT) utiliza 750 microsegundos ($\mu secs$), aplicando la ecuación 6.12, se obtiene una velocidad de transferencia de 0.163 Mb/s de ida y vuelta, de la observación de los datos arrojados por la aplicación se deduce:

1. Un crecimiento sostenido a medida que se aumenta el tamaño del mensaje hasta alcanzar los $14\,Mb/s = 524,288_{bits} = 2^{19}_{bits}$.

2. En el rango $[2^{19}_{bits}, 2^{23}_{bits}]$ se converge en una tasa de transferencia de 14 Mb/s.

[5] http://www.gridmorelos.uaem.mx:8080/~fjuarez/mpi_ping.html

3. En el rango $[2^{24}_{bits},2^{26}_{bits}]$ se degrada la tasa de transferencia por ser muy grande el mensaje 13 Mb/s.

4. La mejor tasa de transferencia se da al enviar $8,388,608_{bits} = 2^{23}_{bits} = 14.339\,Mb/sec.$

5. En los tamaños del mensaje $[2^{19}_{bits},2^{23}_{bits}]$ del punto 2, en el cual converge una tasa de transferencia de 14 Mb/s, la ganancia en la tasa de transferencia como resultado de duplicar el tamaño del paquete no es relevante, pero si es relevante en términos de bits enviados, ya que nos da un rango amplio en el tamaño del mensaje enviado $[524,288_{bits},8,388,608_{bits}]$ sin que con ello se degrade la tasa de transferencia.

6. No se alcanzan los 15 Mb/s del ancho de banda teórico manejado, esto es debido a que los paquetes al ser enviados, se le adicionan los encabezados para su transferencia, con lo que los bits enviados van encapsulados en paquetes más grandes, de tomarse en cuenta el tamaño real de los paquetes, se alcanzarián los 15 Mb/s.

Se puede deducir entonces que para el algoritmo AGHCGrid, se puede usar el rango óptimo de mensaje que comprende $[2^{19}_{bits}, 2^{23}_{bits}] = [524,288_{bits},8,388,608_{bits}]$ y considerar este mensaje como un tamaño idóneo para nuestra aplicación, es decir, que el tamaño de la población que debemos usar para optimizar el uso del ancho de banda, deben ser tamaños de población que estén en este rango.

```
[fjuarez@cuexcomate ~]$ mpiexec -ppn 1 -np 2 ./ping
Soy en proceso  0 of 2 corriendo en :cuexcomate
Soy en proceso  1 of 2 corriendo en :texcal
Tiempo más pequeño posible medible ~0.953674 usecs
        64 bytes(RTT)       750 microsegundos(usecs) (   0.163 Mb/sec)
       128 bytes(RTT)       717 microsegundos(usecs) (   0.341 Mb/sec)
       256 bytes(RTT)       768 microsegundos(usecs) (   0.636 Mb/sec)
       512 bytes(RTT)       809 microsegundos(usecs) (   1.207 Mb/sec)
      1024 bytes(RTT)       912 microsegundos(usecs) (   2.142 Mb/sec)
      2048 bytes(RTT)      1128 microsegundos(usecs) (   3.463 Mb/sec)
      4096 bytes(RTT)      1574 microsegundos(usecs) (   4.963 Mb/sec)
      8192 bytes(RTT)      2471 microsegundos(usecs) (   6.323 Mb/sec)
     16384 bytes(RTT)      3564 microsegundos(usecs) (   8.769 Mb/sec)
     32768 bytes(RTT)      5917 microsegundos(usecs) (  10.563 Mb/sec)
     65536 bytes(RTT)     10230 microsegundos(usecs) (  12.219 Mb/sec)
    131072 bytes(RTT)     18921 microsegundos(usecs) (  13.213 Mb/sec)
    262144 bytes(RTT)     36294 microsegundos(usecs) (  13.776 Mb/sec)
    524288 bytes(RTT)     71130 microsegundos(usecs) (  14.059 Mb/sec)
   1048576 bytes(RTT)    140704 microsegundos(usecs) (  14.214 Mb/sec)
   2097152 bytes(RTT)    279734 microsegundos(usecs) (  14.299 Mb/sec)
   4194304 bytes(RTT)    559525 microsegundos(usecs) (  14.298 Mb/sec)
   8388608 bytes(RTT)   1115870 microsegundos(usecs) (  14.339 Mb/sec)
  16777216 bytes(RTT)   2354889 microsegundos(usecs) (  13.589 Mb/sec)
  33554432 bytes(RTT)   4748298 microsegundos(usecs) (  13.479 Mb/sec)
  67108864 bytes(RTT)   9321278 microsegundos(usecs) (  13.732 Mb/sec)
[fjuarez@cuexcomate ~]$
```

Tabla 6.19. *Mediciones del ancho de banda de la Grid Morelos. La figura muestra en cálculo del RTT para una serie de bits de tamaño 2^n_{bits}, con $n = 6,7,...,26$, se muestra el tiempo en microsegundos y el ancho de banda alcanzado.*

Finalmente para calcular la sobrecarga $\kappa(n,p)$ se debe seleccionar el tiempo requerido para enviar el tamaño de la población de la instancia FFS_80x4x4x24, inicialmente calculada como $poblacion_{bits} = 2,457,600_{bits}$ por la ecuación 6.15, de la cual solo corresponde a la mitad pasar por el cuello de botella de la Grid, esto es $1,228,800_{bits}$. En la tabla 6.19, observamos que para este tamaño de bits, el paquete más próximo corresponde a un tamaño de $1,048,576_{bits}$ que requiere de un tiempo de $140,704\mu secs$, si aproximamos el tiempo requerido para enviar $1,228,800_{bits}$ del tamaño de población con estos datos, con una regla de tres, entonces tenemos un tiempo de $164,867.5\mu secs$ para el tamaño de población, que corresponde a una generación, entonces para calcular el tiempo total de sobrecarga, debemos multiplicar por 30 generaciones obteniendo un tiempo de $4,946,655\mu secs \sim 5$ segundos de sobrecarga, este tiempo corresponde al tiempo que el algoritmo invierte en enviar y recibir las soluciones encontradas por los procesos esclavos, durante todo el tiempo de ejecución del algoritmo, estos 5 segundos son los que se muestran en la figura 6.17 de la sección 6.5.1.

Capítulo 7

CONCLUSIONES Y TRABAJO FUTURO

En este capítulo se muestran las conclusiones, contribuciones y trabajos futuros como resultado de esta tesis doctoral. Hacia el final se hace especial énfasis, el diseño de interfaces gráficas para el algoritmo como un trabajo futuro.

7.1. Conclusiones.

Las conclusiones obtenidas en esta tesis doctoral se listan a continuación:

- Se comprobó mediante la experimentación, que el uso de la Grid, mejora la eficiencia del algoritmo AGHCGrid, con respecto a una ejecución secuencial para tratar el problema de FFS-SDST, así mismo, se comprobó que el uso de una Grid, que integra dos de cluster de alto rendimiento alejados geograficamente y que tiene la limitante del ancho de banda que los une de 15 Mb/s, mejora la eficiencia del algoritmo GHCGrid, al hacer una distribución de procesos alejados geograficamente, esta mejora es debido al diseño del algoritmo que minimiza el intercambio de mensajes en la comunicación con los procesos remotos, de tal forma que, para una instancia cualquiera de las propuestas, las tareas que se realizan en paralelo, son las que más consumen tiempo, que son las de construcción de la población, mutación cooperativa y mutación iterativa, mientras que en secuencial se realizan las tareas de selección y cruce.

- El diseño de algoritmos paralelos en ambiente Grid, para el FFS-SDST, permite la utilización de recursos que se encuentran geográficamente dispersos, en la forma de clusters de alto rendimiento, esta capacidad permite, a los algoritmos paralelos que están diseñados para ejecutarse bajo este ambiente, aprovechar los recursos

como la suma de todos sus núcleos y memoria que integran una Grid, para el procesamiento numérico. La utilización masiva de este conjunto de recursos, permite mejorar el tratamiento del problema FFS-SDST, que por su complejidad es difícil de tratar y de encontrar un óptimo global. La mejora se presenta en el diseño del algoritmo para proveerlo de la capacidad de ejecución en paralelo y de cooperación, la ejecución en paralelo, le permite ejecutarse en múltiples copias llamados procesos, sobre un conjunto de núcleos que provee la Grid y la cooperación le permite compartir los resultados encontrados con otros procesos, este proceso de cooperación, es el que permite la mejora en la calidad de las soluciones, mientras que la ejecución en paralelo, permite disminuir los tiempos de procesamiento del algoritmo, en este sentido, en tiempo que ocupa un algoritmo diseñado en forma secuencial que utiliza solo un núcleo, podría reducirse a la mitad si se cambia su diseño para utilizar dos núcleos y se le agrega la capacidad de poder ejecutarse en paralelo y cooperación, entonces a medida que utiliza un mayor número de núcleos de procesamiento su tiempo será menor y a medida que se perfeccionen sus técnicas de cooperación entre los procesos, se podrían obtener mejoras en la calidad de las soluciones con respecto al secuencial.

- La ejecución en paralelo requiere de una planeación detallada, en el sentido de que, hay que tener una planeación muy cuidada para proveer una distribución uniforme de procesos a núcleos, esta planeación, varía con base al tipo de librerías de MPI utilizadas (MPICH, OpenMPI, Intel), cada tipo presenta una forma particular de ejecutar programas en paralelo, en donde los compiladores de Intel, en combinación de las librerías de MPI de Intel, proveen el mejor desempeño. Existen situaciones en las cuales, a pesar de una distribución de procesos uniforme en la configuración del entorno, durante la ejecución del algoritmo, este tiene un comportamient ineficiente, por tal motivo, se hace necesario una comprobación en tiempo real, para ver si efectivamente, la carga de trabajo por proceso y núcleo en toda la Grid, es uniforme, ya que siempre existe ruido de otros procesos ajenos al algoritmo, entonces el observar ligeras diferencias en los tiempos asignados a cada proceso, es normal mientras se mantengan en cierto rango. Observar estas pequeñas diferencias y mantenerlas así, es lo que permite mejorar la eficiencia del algoritmo, ya que el tiempo de término esta dado por el último proceso en terminar, así un buen algoritmo que haya mostrado anteriormente una buena eficiencia, ahora con al menos un proceso que este mal distribuido o que sea afectado por un proceso externo, obliga a los demás procesos a esperar y por consiguiente obtener tiempos muy grandes empeorando nuestra eficiencia.

- Se comprobo que la cooperación de procesos en el algoritmo AGHCGrid, permite encontrar buenas soluciones, en el sentido de que cada proceso coopera aportando una solución en forma de individuo al proceso maestro, para la construcción de la población, selección y cruce, posteriormente, estas soluciones son devueltas a los procesos origen, para iniciar una mutación cooperativa con SCH y después una mutación iterativa con RS, este ciclo de mejora continua, es lo que permite encontrar buenas soluciones, al realizar una exploración con selección y cruce en el nodo maestro, y una explotación en los procesos esclavos con SCH y RS.

- La aplicación de la sintonización distribuida automática aplicada en paralelo (SDAAP) propuesta en el capítulo 5, para el FFS-SDST, constituye una excelente herramienta para acortar los tiempos del análisis de sensibilidad, ya que permite distribuir las combinaciones de los parámetros a sintonizar, sobre un conjunto de núcleos, que son tomados por procesos que se ejecutan en paralelo, esta sintonización distribuida, también permite analizar un mayor rango en cada uno de los parámetros, con el fin de hacer una sintonización más fina, esto quiere decir, que se pueden analizar una serie de 120 valores para un parámetro en una sola pasada, al asignar una combinación a un proceso por cada uno de los 120 núcleos disponibles en la Grid.

- La aplicación de la sintonización distribuida automática aplicada en paralelo en múltiples instancias (SDAAP-MI), aplicada en la sección 6.3.1, para el FFS-SDST, se basa en la herramienta propuesta SDAAP, que inicialmente fue concebida para la sintonización, pero que posteriormente también se uso en la experimentación para cada una de las cinco instancias de prueba propuestas, la limitante del método SDAAP, es que solo puede ejecutarse una sola agrupación de núcleos manteniendo al resto ociosos, el método SDAAP-MI se ocupa de ello, al permitir ejecutar en paralelo múltiples instancias de una experimentación, en donde cada instancia se ejecuta sobre una agrupación y en donde el conjunto de agrupaciones cubre la totalidad de los recursos de la Grid, este método permite reducir los tiempos de experimentación.

- El análisis de sensibilidad del algoritmo AGHCGrid, requirió de la sintonización de las tres metaheurísticas que integran el algoritmo (AG, SCH,RS), donde la secuencia de sintonización fue SCH, RS y AG. Para el caso de SCH y RS, se utilizó una sintonización dependiente de la instancia, es decir, se realizo una sintonización para cada de las cinco instancias de prueba propuestas, en total 10 sintonizaciones, cada una de tamaño diferente y con tiempos de sintonizado diferentes, posteriormente se realizo la sintonización con independencia de la instancia del AG, integrado por SCH y RS, es decir, solo se llevó a cabo la sintonización de una instancia representativa, utilizando los correspondientes parámetros sintonizados previamente de SCH y RS. Finalmente en la fase de experimentación, se configura el algoritmo AGHCGrid para utilizar los parámetros sintonizados del AG, con solo una instancia representativa y los parámetros sintonizados de SCH y RS de cada una de las cinco instancias.

- La aceleración (Speedup) del algoritmo en la Grid, muestra un comportamiento sub-lineal, ligeramente por debajo del ideal, con una tendencia cargada hacia el peso de las comunicaciones, a medida que aumenta el número de núcleos usados. La aceleración promedio al utilizar una secuencia de procesos asignados a núcleos de 10, 30, 60 y 120 es del 93 %, en donde la eficiencia ideal es del 100 % para una aceleración ideal, es decir que solo tenemos una perdida en eficiencia del 7 %, que representa la parte que no se puede paralelizar y la sobrecarga de las comunicaciones, la sobrecarga calculada en esta tesis, solo corresponde al tiempo utilizado por el envió y recepción de soluciones basadas en individuos, dejando fuera los tiempos

de sincronización propias de MPI y de la sobrecarga de procesos ajenos y propios del sistema operativo Linux, con sus servicios activos de comunicación.

- La eficiencia del algoritmo propuesto, muestra que para lograr una convergencia, es necesario llevar a cabo pruebas con un número mayor de núcleos a los usados en esta tesis doctoral, en el límite de los recursos de la Grid que es de 120 núcleos y de los tiempos de procesamiento para la instancia más grande, se llego a la utilización de los 120 núcleos y 240 procesos, observandose que es posible mejorar las soluciones más allá del límite de nuestra Grid, para lo cual se necesitan utilizar más procesadores o en su defecto, sobrecargar los existentes duplicando los tiempos de espera.

7.2. Contribuciones.

Las contribuciones de esta tesis en el diseño de algoritmos paralelos aplicando el FFS-SDST y en análisis de sensibilidad se listan a continuación:

- Diseño y programación del algoritmo híbrido AGHCGrid en ambiente Grid, compuesto por tres metaheurísticas (AG, SCH y RS), donde el uso del paralelismo, la variante del problema para el FFS con tiempos de inicio dependientes de la secuencia y su entorno de ejecución en ambiente Grid, no aparece en la literatura existente.

- La forma en la que se integra los procesos para su distribución, comunicación, cooperación, haciendo especial enfasis en la sincronización, así como la construcción de la población, los procesos de selección, cruzamiento y mutación sobre un ambiente Grid.

- Dos métodos propuestos para llevar a cabo el proceso de sintonización y experimentación del algoritmo GHCGrid: sintonización distribuida automática aplicada en paralelo (SDAAP) y SDAAP en múltiples instancias (SDAAP-MI), la primera se utiliza para la sintonización y la segunda para la experimentación, ambos métodos solo funcionan al utilizar cómputo de alto rendimiento como cluster y ambientes Grid.

7.3. Trabajos futuros.

Los trabajos futuros que de desprende de esta tesis doctoral se listan a continuación:

- Continuar con el diseño de algoritmos paralelos, haciendo énfasis en buscar nuevos métodos de cooperación para aumentar la eficacia de las soluciones, en esta tesis

se demostro que la eficiencia se mejora al utilizar un número mayor de núcleos de procesamiento, conservando la eficacia con respecto al secuencial, por tal razón el trabajo futuro consiste en investigar nuevas formas de cooperación que nos ayuden a mejorar tanto la eficiencia como la eficacia.

- Continuar la búsqueda de nuevas formas de integración, de técnicas para la búsqueda de soluciones, que pueden ser integradas por algoritmos genéticos híbridos, ya que se demostró que esta tipo de integración es factible a ser paralelizado.

- Buscar nuevas formas de exploración y explotación, que puedan ser paralelizadas, de explotación al mejorar las técnicas de selección y cruce, que sean aplicadas en paralelo, y de explotación al paralelizar las búsquedas locales, en esta tesis se presento una pseudo paralelismo que permite ejecutar muchos procesos con SCH y RS en paralelo, centralizados por un algoritmo genético, el reto ahora es poder paralelizar SCH y RS que puedan ejecutar sus múltiples fases sobre un conjunto de procesos repartidos en varios núcleos de procesamiento.

- La paralelización de técnicas secuenciales con resultados probados, constituye una de las primeras aproximaciones para tratar el paralelismo, pero el diseño de algoritmos puramente paralelos es un área poco explorada, ya que para un algoritmo puramente paralelo no existe algoritmo secuencial conocido.

- El algoritmo propuesto tiene un impacto directo a la industria de la manufactura, en la investigación sobre el uso de la Grid y el uso de herramientas de supercómputo, por este motivo, uno de los trabajos futuros es el diseño de interfaces gráficas, que permita a la industria hacer uso del algoritmo o de futuros algoritmos mediante la web y poder vincular de una forma más directa la investigación con el sector industrial.

A continuación se da una breve introducción a lo que seria el diseño de interfaces graficas.

7.3.1. Diseño de interfaz gráfica.

El diseño, desarrollo y programación de algoritmos paralelos, se demostró en esta investigación de tesis doctoral, que permiten explotar de manera adecuada los recursos de los equipos de computo de alto rendimiento conocidos como cluster o Grid, pero que las aplicaciones que se desprenden para su explotación, están limitadas a línea de comandos conocido como consola (Shell), esta limitación implica que solo el diseñador y programador de los algoritmos, puede llevar a cabo la experimentación, recuperación e interpretación de resultados del FFS-SDST.

La manera de acercar el uso de estos recursos hacia el sector de la industria, y puedan ellos solucionar problemas de calendarización con equipo de cómputo de alto

rendimiento, es convertir las aplicaciones basadas en línea de comandos, en una interfaz gráfica de usuario (GUI), permitiendo llevar a cabo la experimentación por otra persona distinta al diseñador del algoritmo.

Las interfaces gráficas deben permitir realizar cambios en la configuración del entorno del algoritmo, los tres aspectos básicos que una GUI debe proporcionar son: transferencia de datos, ejecución del algoritmo y recuperación de resultados, estos tres aspectos se descomponen en diferentes propiedades listadas a continuación.

- Benchmark e instancias de prueba. El sistema debe ser capaz de leer archivos proporcionados por el usuario final, que representen el problema a tratar, sin requerir la intervención del administrador del sistema.

- Parámetros del algoritmo. El sistema debe permitir realizar cambios en la sintonización de los parámetros del algoritmo, a fin de comprobar si otra sintonización ofrece mejores resultados sin tener que volver a recompilar el código fuente.

- Uso de recursos. El sistema debe poder seleccionar un conjunto de recursos disponibles (núcleos y memoria) para la ejecución del algoritmo.

- Ejecución. El sistema debe ser capaz de ejecutar el algoritmo con los recursos seleccionados.

- Supervisión. El sistema debe proporcionar información relativa el estado que guarda la ejecución del algoritmo, por ejemplo el tiempo transcurrido, el tiempo asignado a los procesos, la carga de los núcleos, porcentajes de avance global, tiempo restante de ejecución, resultados parciales, entre otros.

- Recuperación y almacenamiento de datos. El sistema debe ser capaz de permitir que los resultados obtenidos puedan ser recuperados en medios distintos a los locales.

- Interpretación. El sistema debe ser capaz de interpretar los resultados arrojados por el algoritmo, entre estos resultados mostrar la gráfica de Gantt de la mejor solución.

Actualmente todas las aplicaciones convergen en el uso de la Internet, por lo que la transformación del algoritmo hacia el uso de sistemas web es la mejor opción para proporcionar un valor agregado, al desarrollo de algoritmos para tratar problemas reales de la industria, la solución, aunque parece sencillo con el uso de herramientas de desarrollo web, presenta el problema de reducir la eficiencia del algoritmo, debido a que un servidor web para alojar nuestra aplicación, representa una sobrecarga.

Otro problema que se desprende es lograr la comunicación de los algoritmos en C con los servidores web, ya que cualquier interfaz implicaría que el algoritmo tuviera que hacer tareas adicionales de comunicación con la GUI, para solucionar este problema,

se ha tratado de generar procesos independientes que sean capaces de supervisar el desempeño del algoritmo y capturar resultados parciales sin interferir en su desempeño y sin tener que realizar cambios al código fuente, finalmente se hace la observación de que este tema es particularmente interesante, pero se escapa de los alcances de la presente investigación y se deja como trabajo futuro.

7.4. Resumen de publicaciones.

A continuación se listan los artículos realizados a partir de esta investigación doctoral.

Revistas indexadas.

- M. A. Cruz-Chávez, A. Rodríguez-León, E. Y. Ávila-Melgar, F. Juárez-Pérez, M. H. Cruz-Rosales, R. Rivera-López, Gridification of Genetic Algorithm with Reduced Communication for the Job Shop Scheduling Problem, International Journal of Grid and Distributed Computing, Science and Engineering Research Support soCiety, Australia, ISSN: 2005-4262, Vol. 3, No. 3, pp. 13-28, 2010.

- M. A. Cruz-Chávez, A. Rodríguez-León, E. Y. Ávila-Melgar, F. Juárez-Pérez, M. H. Cruz-Rosales, R. Rivera-López, Genetic-Annealing Algorithm in Grid Environment for Scheduling Problems, Communications in Computer and Information Science: Security-Enriched Urban Computing and Smart Grid, Springer Verlag Pub., Berlin Heidelberg, ISSN: 1865-0929, Vol. 78, pp.1-9, 2010.

Revistas de divulgación científica.

- M. A. Cruz-Chávez, F. Juárez-Pérez y P. Moreno Bernal, MiniGrid Morelos, una Sinergia Interinstitucional, Hypatia, Revista de Divulgación Científico-Tecnológica del Consejo de Ciencia y Tecnología del Estado de Morelos, Editor Responsable: Silvia Patricia Pérez Sabino, No. 40, Pág.32-33, Enero/Marzo de 2012.

Capítulos de libro.

- Logistics Management and Optimization through Hybrid Artificial Intelligence Systems, Editor Carlos Alberto Ochoa Ortiz Zezzatti et al., Chapter 3: Grid Platform Applied to the Vehicle Routing Problem with Time Windows for the Distribution of Products, Marco Antonio Cruz-Chávez, Abelardo Rodríguez-León, Rafael Rivera-López, Fredy Juárez-Pérez, Carmen Peralta-Abarca and Alina Martínez-Oropeza, ISBN13: 9781466602977, IGI Global, 422 pages, march 2012.

Congresos nacionales a internacionales.

- M. A. Cruz-Chávez, E.Y. Avila-Melgar, S. A. Serna Barquera, F. Juárez-Pérez,

General Methodology for Converting a Sequential Evolutionary Algorithm into Parallel Algorithm with MPI to Water Design Networks, Electronics, Robotics and Automotive Mechanics Conference, CERMA2010, IEEE-Computer Society, ISBN 978-0-7695-4204-1, pp 149 - 154, September 28 - October 1, México, 2010.

- A. Rodriguez-León, M. A. Cruz-Chávez, R. Rivera-López, E. Y. Ávila-Melgar, F. Juárez-Pérez, O. Díaz-Parra, A Communication Scheme for an Experimental Grid in the Resolution of VRPTW using an Evolutionary Algorithm, 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.

- M. A. Cruz-Chávez, A. Rodriguez-León, E. Y. ávila-Melgar, F. Juárez-Pérez, J. C. Zavala-Díaz, R. Rivera-López, Parallel Hybrid Evolutionary Algorithm in a Grid Environment for the Job Shop Scheduling Problem, Proceedings of the Second EELA-2 Conference, CIEMAT, ISBN 978-84-7834-627-1, pp 227 - 234, November 25-27, Choroní, 2009.

- M. A. Cruz-Chávez, E. Y. Ávila-Melgar, F. Juárez Pérez, W. G. Torres-Sánchez, Empirical Transformation of Job Shop Scheduling Problem to the Hydraulic Networks Problem in a Water Distribution System, Electronics, Robotics and Automotive Mechanics Conference, CERMA2009, IEEE-Computer Society, ISBN 978-0-7695-3799-3, pp 76-81, September 22 - 25, México, 2009.

- M. A. Cruz-Chávez, F. Juárez-Pérez, E. Y. Ávila-Melgar, A. Martínez-Oropeza, Simulated Annealing Algorithm for the Weighted Unrelated Parallel Machines Problem, Electronics, Robotics and Automotive Mechanics Conference, CERMA2009, IEEE-Computer Society, ISBN 978-0-7695-3799-3, pp 94-99, September 22 - 25, México, 2009.

- M. A. Cruz-Chávez, F. Juárez-Pérez, J. C. Zavala-Díaz, E. Y. ávila-Melgar, Algoritmo de Recocido Simulado Secuencial y Paralelizado con Memoria Distribuida para el Problema de Máquinas Paralelas no Relacionadas Ponderadas, CICos 2009, 7to Congreso Internacional de Cómputo en Optimización y Software, ISBN(e) 978-607-00-1970-8, pp 34-51, 17-20 Noviembre, México, 2009.

- M. A. Cruz Chávez, J. C. Zavala-Díaz, C. E. Mariano Romero, F. Juárez-Pérez, E. Y. Avila Melgar, Calendarización en Redes de Distribución de Agua, CICos 2009, 7to Congreso Internacional de Cómputo en Optimización y Software, ISBN(e) 978-607-00-1970-8, pp 52-66, 17-20 Noviembre, México, 2009.

Capítulo 7

REFERENCIAS BIBLIOGRÁFICAS

Aarts, E.H.L. and Van Laarhoven, P.J.M. 1985. Statistical cooling: A general approach to combinatorial optimization problems. Philips Journal of Research, 40(4), 193-226. 99

A. Agnetis, A. Pacifici, F. Rossi, M. Lucertini, S. Nicoletti, F. Nicolo, G. Oriolo, D. Pacciarelli, E. Pesaro, Scheduling of flexible flow lines in an automobile assembly plant, European Journal of Operational Research 97 (2) (1997) 348– 362. 38

A. Alem Tabriz, M. zandieh and Z. vaziri A Novel Simulated Annealing Algorithm to Hybrid Flow Shop Scheduling with Sequence-Dependent Setup Times Journal of Applied Sciences 9(10): 1943-1949, 2009. 74

A. Allahverdi, F.S. Al-Anzi, Scheduling multi-stage parallel-processor services to minimize average response time, Journal of the Operational Research Society 57 (1) (2006) 101–110. 46

A. Bolat, I. Al-Harkan, B. Al-Harbi, Flow-shop scheduling for three serial stations with the last two duplicate, Computers and Operations Research 32 (3) (2005) 647–667. 32

A. Janiak, E. Kozan, M. Lichtenstein, C. Oguz, Metaheuristic approaches to the hybrid flow shop scheduling problem with a cost-related criterion, International Journal of Production Economics 105 (2) (2007) 407–424. 46

A. Vepsalainen and T. Morton. Priority rules for job shops with weighted tardiness costs. Management Science, 33(8):10351047, 1985. 36

A.G.P. Guinet, M.M. Solomon, Scheduling hybrid flowshops to minimize maximum tardiness or maximum completion time, International Journal of Production Research 34 (6) (1996) 1643–1654. 19, 37

A.G.P. Guinet, Textile production systems: a succession of non-identical parallel processor shops, Journal of the Operational Research Society 42 (8) (1991) 655–671. 38

A. Rodriguez-León, M. A. Cruz-Chávez, R. Rivera-López, E. Y. Ávila-Melgar, F. Juárez-Pérez, O. Díaz-Parra, A Communication Scheme for an Experimental Grid in the Resolution of VRPTW using an Evolutionary Algorithm, 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. 52

A. Mártinez Oropeza. Solucion al Problema de Máquinas en Paralelo Mediante un Algoritmo de Colonia de Hormigas. Tesis de Maestria. Universidad Autónoma del Estado de Morelos, 2010. 3, 6, 34, 77, 97

Allahverdi A, Ng CT, Cheng TCE, Kovalyov MY. A survey of scheduling problems with setup times or costs. European Journal of Operational Research 2008;187(3):985–1032. 17

Allahverdi, A., Gupta, J. N. D., and Aldowaisan, A review of scheduling research involving setup considerations. Omega-International Journal of Management Science, 27(2):219–239, 1999. 3

B. Bullnheimer, R.F. Hartl, and C. Strauss, A new rank based version of the Ant System a computational study, Institute of Management Science, University of Vienna, Tech. Rep., 1997. 42

B. Naderi, M. Zandieh, A. Khaleghi Ghoshe Balagh, V. Roshanaei, An improved simulated annealing for hybrid flowshops with sequence-dependent setup and transportation times to minimize total completion time and total tardiness, Expert Systems with Applications 36 (6) (2009) 9625–9633. 46

B. Naderi, M. Zandieh, V. Roshanaei, Scheduling hybrid flowshops with sequence dependent setup times to minimize makespan and maximum tardiness, International Journal of Advanced Manufacturing Technology 41 (11–12) (2009) 1186–1198. 46

B. Naderi, Rubén Ruiz , M. Zandieh. Algorithms for a realistic variant of flowshop scheduling. Computers & Operations Research 37 (2010) 236-246 5

B. Roy and B. Sussmann. Les problèmes d'ordonnancement avec constraintes disjonctives. Note d.s. no. 9 bis, d6c, SEMA, Matrouge, Paris, 1964. 37

B.S. Mittal, P.C. Bagga, Two machine sequencing problem with parallel machines, OPSEARCH: Journal of the Operational Research Society of India 10 (1973) 50–61. 32

B. Wardono, Y. Fathi, A tabu search algorithm for the multi-stage parallel machine problem with limited buffer capacities, European Journal of Operational Research 155 (2004) 380–401. 72

B. Jia, Process Cooperation in Multiple Message Broadcast, Lecture Notes in Computer Science, 2007, Volume 4757/2007, 27-35, DOI: 10.1007/978-3-540-75416-9_11 62

Blickle, T. and Thiele, L. (1995). A comparison of selection schemes used in genetic algorithms. Technical Report 11, Computer Engineering and Comunnication Network Lab (TIK), Gloriastrasse 35, 8092 Zurich, Switzerland. 40

B. Pavez-Lazo, J. Soto-Cartes, Carlos Urrutia, Millaray Curilem. Selección Determinística y Cruce Anular en Algoritmos Genéticos: Aplicación a la Planificación de Unidades Térmicas de Generación. Ingeniare. Vol. 17 Nº 2, 2009, pp. 175-181. 66

C. Oguz, Y. Zinder, V. Do, A. Janiak, M. Lichtenstein, Hybrid flow-shop scheduling problems with multiprocessor task systems, European Journal of Operational Research 152 (1) (2004) 115–131. 47

C. Rajendran, D. Chaudhuri, A multistage parallel-processor flowshop problem with minimum flowtime, European Journal of Operational Research 57 (1) (1992) 111–122. 33

C. Rajendran, D. Chaudhuri, Scheduling in n-jobs m stage flowshop with parallel processors to minimize makespan, International Journal of Production Economics 27 (2) (1992) 137–143. 33

C. Sriskandarajah, S.P. Sethi, Scheduling algorithms for flexible flowshops: worst and average case performance, European Journal of Operational Research 43 (2) (1989) 143–160. 37

C.L. Chen, C.L. Chen, Bottleneck-based heuristics to minimize tardy jobs in a flexible flow line with unrelated parallel machines, International Journal of Production Research 46 (22) (2008) 6415–6430. 38

C.L. Chen, C.L. Chen, Bottleneck-based heuristics to minimize total tardiness for the flexible flow line with unrelated parallel machines, Computers and Industrial Engineering 56 (4) (2009) 1393–1401. 38

C.T. Tseng, C.J. Liao, A particle swarm optimization algorithm for hybrid flow- shop scheduling with multiprocessor tasks, International Journal of Production Research 46 (17) (2008) 4655–4670. 47

C.Y. Low, Simulated annealing heuristic for flow shop scheduling problems with unrelated parallel machines, Computers and Operations Research 32 (8) (2005) 2013–2025. 46

C. Kahraman, O. Engin, I. Kaya, R. Elif Öztürk, Multiprocessor task scheduling in multistage hybrid flow-shops: A parallel greedy algorithm approach, Applied Soft Computing 10 (2010) 1293–1300. 6

C. Lowa, C.-J. Hsub, Chwen-Tzeng Su, A two-stage hybrid flowshop scheduling problem with a function constraint and unrelated alternative machines, Computers & Operations Research 35 (2008) 845 – 853. 6

C.-Y. Lee, George L. Vairaktarakis. Minimizing makespan in hybrid flowshops. Operations Research Letters 16 (1994) 149 158. 5

Churchman, C. W., Ackoff, R. L., and Arnoff, E. L., Introduction to Operations Research. Wiley, New York, 1969. 1

D. L. Santos, J. L. Hunsucker, D. E. Dealt, an Evaluation of Sequencing Heuristic in Flow Shop with Multiple Processors, Computers ind. Engng Vol. 30, No. 4, pp. 681-692, 1996 Copyright © 1996 Elsevier. 2

D.L. Santos, J.L. Hunsucker, D.E. Deal, On makespan improvement in flow shops with multiple processors, Production Planning and Control 12 (3) (2001) 283–295. 38

David E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA ©1989 ISBN:0201157675. 40

Dudek, R. A., Panwalkar, S. S., and Smith, M. L., The lessons of flowshop scheduling research. Operations Research, 40(1):7–13, 1992. 2

Dudek, R. A., Smith, M. L., and Panwalkar, S. S., Use of a Case Study in Sequencing/Scheduling Research, Omega. Vol. 2, No. 2, 1974, pp. 253-261. 1

Ebbe G. Negenman, Theory and Methodology Local search algorithms for the multiprocessor Flow shop scheduling problem, European Journal of Operational Research 128 (2001) 147-158. 50

E. Figielska. A genetic algorithm and a simulated annealing algorithm combined with column generation technique for solving the problem of scheduling in the hybrid flowshop with additional resources. Computers & Industrial Engineering 56 (2009) 142–151. 52, 74

F. Choong, S. Phon-Amnuaisuk, M.Y. Alias, Metaheuristic methods in hybrid flow shop scheduling problem, Expert Systems with Applications, Elsevier (2011). 3, 38

F.Y. Ding, D. Kittichartphayak, Heuristics for scheduling flexible flow lines, Computers and Industrial Engineering 26 (1) (1994) 27–34. 38

Ford, F. N., Bradbard, D. A., Ledbetter, W. N., and Cox, J. F., Use of operations research in production management. Production and Inventory Management, 28(3):59–62. 2

G. C. Lee, Y. D. Kim, A branch-and-bound algorithm for a two-stage hybrid flowshop scheduling problem minimizing total tardiness, International Journal of Production Research 42 (22) (2004) 4731–4743. 32

G. Vairaktarakis, M. Elhafsi, The use of flowlines to simplify routing complexity in two-stage flowshops, IIE Transactions 32 (8) (2000) 687–699. 38

G.J. Kyparisis, C. Koulamas, A note on weighted completion time minimization in a flexible flow shop, Operations Research Letters 29 (1) (2001) 5–11. 37

G.J. Kyparisis, C. Koulamas, Flexible flow shop scheduling with uniform parallel machines, European Journal of Operational Research 168 (3) (2006) 985–997. 39

Gourgand, M., Grangeon, N., Norre, S., 1999. Metaheuristics for the deterministic hybrid flow shop problem. In: Pro- ceedings of the International Conference on Industrial Engineering and Production Management, IEPMÕ99, Glas- gow. FUCAM-INRIA, pp. 136–145. 16, 187

Grassé P. P. La reconstruction du nid et les coordinations inter-individuelles chez bellicositermes natalensis et cubitermes sp. La théorie de la Stigmergie: essai d interpretation du Comportement des Termites Constructeurs, pages 41-81, 1959. 52

Graves, S. C., A review of production scheduling. Operations Research, 29(4):646–675, 1981. 3

H. Morita, N. Shio, Hybrid branch and bound method with genetic algorithm for flexible flowshop scheduling problem, JSME International Journal Series C-Mechanical Systems Machine Elements and Manufacturing 48 (1) (2005) 46–52. 33

H. Tsubone, M. Ohba, T. Uetake, The impact of lot sizing and sequencing on manufacturing performance in a two-stage hybrid flow shop, International Journal of Production Research 34 (11) (1996) 3037–3053. 37

H.D. Sherali, S.C. Sarin, M.S. Kodialam, Models and algorithms for a two-stage production process, Production Planning and Control 1 (1) (1990) 27–39. 33

H.S. Choi, D.H. Lee, Scheduling algorithms to minimize the number of tardy jobs in two-stage hybrid flow shops, Computers and Industrial Engineering 56 (1) (2009) 113–120. 33

Holland, J. H. (1975). Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor. Republished by the MIT press, 1992. 39, 40, 41

I. Foster and C. Kesselman, The GRID2, Blueprint for a New Computing Infrastructure, Elsevier, 2004. ISBN 1-55860-933-4 49

I. Perberry, Parallel Complexity Theory, Withmore Laboratory, Pennsylvania State University, USA, 1987. 97

I. Ribas, R. Leisten, Jose M. Framinan. Review and classification of hybrid flow shop scheduling problems from a production system and a solutions procedure perspective. Computers & Operations Research 37 (2010) 1439–1454. 3, 112

J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. Managament Science, 34:391401, 1988. 37

J. Carlier, E. NTron, An exact method for solving the multi-processor flow- shop, RAIRO Recherche OpTrationnelle 34 (1) (2000) 1–25. 33

J. R. Jackson, Simulation Research on Jon Scheduling, Naval Research Logistincs Querterly, 4:287-295, 1957. 34

J.A. Hoogeveen, J.K. Lenstra, B. Veltman, Preemptive scheduling in a two-stage multiprocessor flow shop is np-hard, European Journal of Operational Research 89 (1) (1996) 172–175. 6

J.L. Cheng, Y. Karuno, H. Kise, A shifting bottleneck approach for a parallel- machine flowshop scheduling problem, Journal of the Operations Research Society of Japan 44 (2) (2001) 140–156. 37

J.N.D. Gupta, A.M.A. Hariri, C.N. Potts, Scheduling a two-stage hybrid flow shop with parallel machines at the first stage, Annals of Operations Research 69 (1997) 171–191. 33

J.N.D. Gupta, E.A. Tunc, Schedules for a two-stage hybrid flowshop with parallel machines at the second stage, International Journal of Production Research 29 (7) (1991) 1489–1502. 37

J.N.D. Gupta, Two-stage, hybrid flow shop scheduling problem, Journal of the Operational Research Society 39 (4) (1988) 359–364. 6, 32, 37

J. Jungwattanakit, M. Reodecha, P. Chaovalitwongse and F. Werner. Constructive and Tabu Search Algorithm for Hybrid Flow Shop Problems with Unrelated Parallel Machines Problem and Setup Times. International Journal of Computational Science, 2007, Vol. 1, No. 2, 204-214. 5, 16

K. Alaykyran, O. Engin, A. Doyen, Using ant colony optimization to solve hybrid flow shop scheduling problems, International Journal of Advanced Manufacturing Technology 35 (5–6) (2007) 541–550. 47

K. Belkadi, M. Gourgan, M. Benyettou and A. Aribi. Sequential and Parallel Genetics Algorithms for the Hibrid Flow Shop Scheduling Problem. Journal of Applied Sciences 6 (4): 775-778 2006. 53

K. Takaku, K. Yura, Online scheduling aiming to satisfy due date for flexible flow shops, JSME International Journal Series C-Mechanical Systems Machine Elements and Manufacturing 48 (1) (2005) 21–25. 37

K. C. Ying, S.W. Lin, Multiprocessor task scheduling in multistage hybrid flow- shops: an ant colony system approach, International Journal of Production Research 44 (16) (2006) 3161–3177. 47

Kuo-Ching Ying, Shih-Wei, Lin. multi-heuristic desirability ant colony system heuristic for non-permutation flowshop scheduling problems. Int J Adv Manuf Technol (2007) 33: 793–802 DOI 10.1007/s00170-006-0492-8. 17, 115

L.M. Gambardella and M. Dorigo, Ant-Q: A reinforcement learning approach to the traveling salesman problem, in Proc. Twelfth International Conference on Machine Learning (ML- 95), A. Prieditis and S. Russell, Eds., Morgan Kaufmann Publishers, pp. 252–260, 1995. 42

L.M. Gambardella and M. Dorigo, Solving symmetric and asymmetric TSPs by ant colonies, in Proc. 1996 IEEE International Conference on Evolutionary Computation (ICEC'96), T. Baeck et al., Eds. IEEE Press, Piscataway, NJ, pp. 622–627, 1996. 42, 44

Ledbetter, W. N. and Cox, J. F., Operations research in production management: An investigation of past and present utilization. Production and Inventory Management, 18(3):84–91, 1977. 2

Li Wang, Dawei Li, A Scheduling Algorithm for Flexible Flow Shop Problem, Proceedings of the 4to World Congress on Intelligent Control and Automation, June 10-14, 2002, Shanghai, P.R.China. 1

Linn, R. and Zhang, W., Hybrid flow shop scheduling: A survey. Computers and Industrial Engineering, 37(1-2):57–61, 1999. 3

L. Araujo, C. Cervigón. Algoritmos Evolutivos, editorial Alfa-Omega, 2009. 41, 50, 62, 63

L. Maria Gambardella, M. Dorigo, Solving Symmetric and Asymmetric TSPs by Ant Colonies - Evolutionary Computation, 1996., Proceedings of IEEE International Conference on 69

M. Dorigo, Optimization, learning and natural algorithms (in italian), Ph.D. dissertation, Dipartimento di Elettronica, Politecnico di Milano, Italy, 1992. 42

M. Dorigo, V. Maniezzo, and A. Colorni, Positive feedback as a search strategy, Dipartimento di Elettronica, Politecnico di Milano, Italy, Tech. Rep. 91-016, 1991. 42

M. Haouari, L. Hidri, On the hybrid flowshop scheduling problem, International Journal of Production Economics 113 (1) (2008) 495–497. 46

M. Pinedo. Planning and Scheduling in Manufacturing and Services. Springer, 2005. 35, 36

M.C. Portmann, A. Vignier, D. Dardilhac, D. Dezalay, Branch and bound crossed with GA to solve hybrid flowshops, European Journal of Operational Research 107 (2) (1998) 389–400. 33

M.E. Kurz, R.G. Askin, Comparing scheduling rules for flexible flow lines, International Journal of Production Economics 85 (3) (2003) 371–388. 47

M.E. Kurz, R.G. Askin, Scheduling flexible flow lines with sequence- dependent setup times, European Journal of Operational Research 159 (1) (2004) 66–82. 47

M.J. Acero-Domfnguez, C.D. Patermina-Arboleda, Scheduling jobs on a k-stage flexible flow shop using a toc-based (bottleneck) procedure, in: M.H. Jones, S.D. Patek, B.E. Tawney (Eds.), Proceedings of the 2004 Systems and Information Engineering Design Symposium, IEEE Press, 2004, pp. 295–298. 38

M.M. Dessouky, M.I. Dessouky, S.K. Verma, Flowshop scheduling with identical jobs and uniform parallel machines, European Journal of Operational Research 109 (3) (1998) 620–631. 33

M.R. Garey, D.S. Johnson, Computers and intractability: a guide to the theory of NP-completeness, A Series of Books in the Mathematical Sciences, W.H. Freeman, San Francisco, 1979. 3, 4, 6

M.S. Jayamohan, C. Rajendran, A comparative analysis of two different approaches to scheduling in flexible flow shops, Production Planning and Control 11 (6) (2000) 572–580. 37

M.S. Salvador, A solution to a special case of flow shop scheduling problems, in: S.E. Elmaghraby (Ed.), Symposium of the Theory of Scheduling and Applications, Springer-Verlag, 1973, pp. 83–91. 33

MacCarthy, B. L. and Liu, J. Y., Addressing the gap in scheduling research - a review of optimization and heuristic methods in production scheduling. International Journal of Production Research, 31(1):59–79, 1993. 2

M. A. Cruz Chávez, Cooperación de Procesos para el Problema de Job Shop Schedulinn

Aplicando Recocido Simulado, Tesis de Doctorado, ITESM Campus Cuernavaca, 2005. 32, 37

M. A. Cruz Chávez, J. C. Zavala-Díaz, C. E. Mariano Romero, Fredy Juárez-Pérez, Erika Yesenia Avila Melgar, Calendarización en Redes de Distribución de Agua, CICos 2009, 7to Congreso Internacional de Cómputo en Optimización y Software, ISBN(e) 978-607-00-1970-8, pp 52-66, 17-20 Noviembre, México, 2009. 51

M. Antonio Cruz-Chávez , F. Juárez-Pérez y P. Moreno Bernal, MiniGrid Morelos , una Sinergia Interinstitucional, Hypatia, Revista de Divulgación Cientifico-Tecnológica del Consejo de Ciencia y Tecnología del Estado de Morelos , Editor Responsable: Silvia Patricia Pérez Sabino, No. 40. Pág.32-33, Enero/Marzo de 2012. 109

M. A. Cruz-Chávez, A. Rodriguez-León, E. Y. ávila-Melgar, F. Juárez-Pérez, J. C. Zavala-Díaz, Rafael Rivera-López, Parallel Hybrid Evolutionary Algorithm in a Grid Environment for the Job Shop Scheduling Problem, Proceedings of the Second EELA-2 Conference, CIEMAT, ISBN 978-84-7834-627-1, pp 227 - 234, November 25-27, Choroní, 2009. 52

M. A. Cruz-Chávez, A. Rodríguez-León, E. Y. Ávila-Melgar, F. Juárez-Pérez, M. H. Cruz-Rosales, R. Rivera-López, Gridification of Genetic Algorithm with Reduced Communication for the Job Shop Scheduling Problem, International Journal of Grid and Distributed Computing, Science and Engineering Research Support soCiety, Australia, ISSN: 2005-4262, Vol. 3, No. 3, pp. 13-28, 2010. 52

M. A. Cruz-Chávez, A. Rodríguez-León, E. Y. Ávila-Melgar, F. Juárez-Pérez, Martín H. Cruz-Rosales, Rafael Rivera-López, Genetic-Annealing Algorithm in Grid Environment for Scheduling Problems, Communications in Computer and Information Science: Security-Enriched Urban Computing and Smart Grid, Springer Verlag Pub., Berlin Heidelberg, ISSN: 1865-0929, Vol. 78, pp.1-9, 2010. 52

M. A. Cruz-Chávez, E.Y. Avila-Melgar, S. A. Serna Barquera, F. Juárez-Pérez, General Methodology for Converting a Sequential Evolutionary Algorithm into Parallel Algorithm with MPI to Water Design Networks, Electronics, Robotics and Automotive Mechanics Conference, CERMA2010, IEEE-Computer Society, ISBN 978-0-7695-4204-1, pp 149 - 154, September 28 - October 1, México, 2010. 51

M. A. Cruz-Chávez, E. Y. Ávila-Melgar, F. Juárez Pérez, W. G. Torres-Sánchez, Empirical Transformation of Job Shop Scheduling Problem to the Hydraulic Networks Problem in a Water Distribution System, Electronics, Robotics and Automotive Mechanics Conference, CERMA2009, IEEE-Computer Society, ISBN 978-0-7695-3799-3, pp 76-81, September 22 - 25, México, 2009. 51

M. A. Cruz-Chávez, F. Juárez-Pérez, E. Y. Ávila-Melgar, A. Martínez-Oropeza, Simulated Annealing Algorithm for the Weighted Unrelated Parallel Machines Problem, Electronics, Robotics and Automotive Mechanics Conference, CERMA2009, IEEE-

Computer Society, ISBN 978-0-7695-3799-3, pp 94-99, September 22 - 25, México, 2009. 51

M. A. Cruz-Chávez, F. Juárez-Pérez, J. C. Zavala-Díaz, E. Y. ávila-Melgar, Algoritmo de Recocido Simulado Secuencial y Paralelizado con Memoria Distribuida para el Problema de Máquinas Paralelas no Relacionadas Ponderadas, CICos 2009, 7to Congreso Internacional de Cómputo en Optimización y Software, ISBN(e) 978-607-00-1970-8, pp 34-51, 17-20 Noviembre, México, 2009. 51

M. D., M. Birattari and T. Stutzle, IEEE Computational Intelligence Magazine, November, 2006. 69

M. Dorigo, M. Birattari, and T. Stutzle. Ant Colony Optimization Artificial Ants as a Computational Intelligence Technique. IEEE Computational Intelligence Magazine, November 2006. 50

M. Martinez Rangel. Algoritmo de recocido simulado paralelizado aplicado al problema de asignacion de recursos en un taller de manofactura flexible sujeto a disposiciones de tiempo, Tesis de doctorado pp42, UAEM 2008. 50

McKay, K. N., Pinedo, M., and Webster, S. (2002). Practice-focused research issues for scheduling systems. Production and Operations Management, 11(2):249–258. 2

McKay, K. N., Safayeni, F. R., and Buzacott, J. A., Job-shop scheduling theory - what is relevant. Interfaces, 18(4):84–90, 1988 2

Metrópoliss N., Rosenbluth A. W.,. Rosenbluth M, Teller A. H., and Teller. E. (1953), Equation of State Calculations by Fast Computing Machines. J. Chem. Phys. Vol. 21,pp. 1087-1092, 1953. 45

Michael J. Quinn. Parallel Programing in C with MPI and OpenMP. Editorial Mc Graw Hill, ISBN 0-07-282256-2, 2004. Página 51. 111

Michael L. Pinedo. Scheduling, Teory, Algorithms and Systems. Third edition, Editorial Springer, 2008. 4, 6, 7, 21

M. Á. González Fernández. Soluciones Metaheurísticas al Job-Shop Scheduling Problem with Sequence-Dependent Setup Times. Tesis Doctoral. Universidad de Oviedo, 2011. 31, 34, 37, 52

O. Cordón, I.F. de Viana, F. Herrera, and L. Moreno, A new ACO model integrating evolutionary computation concepts: The best-worst Ant System, in Proc. ANTS 2000, M. e Dorigo et al., Eds., IRIDIA, ULB, Belgium, pp. 22–29, 2000. 42

O. Holthaus and C. Rajendram. Ecient dispatching rules for scheduling in a job shop. International Journal of Production Economics, 48(1):87105, 1997. 35

Ocotlán Díaz Parra, Un Algoritmo Evolutivo Paralelizado para una Cadena de Suministros con Ventanas de Tiempo, Tesis de doctorado, UAEM 2008. 40

Olhager, J. and Rapp, B., Operations research techniques in manufactur- ing planning and control systems. International Transactions in Operational Research, 2(1):7–13, 1995.4 2

P. Bratley, M. Florian, P. Robillard, Scheduling with earliest start and due date constraints on multiple machines, Naval Research Logistics Quearterly 22 (1) (1975) 165–173. 33

Dejan S., Fred Douglis, Y. Paindaveine, S. Zhou, Process Migration, Journal ACM Computing Surveys (CSUR) Surveys Homepage archive Volume 32 Issue 3, Sept. 2000 Pages 241-299. 61

R. Ruiz, C. Maroto, A genetic algorithm for hybrid flowshops with sequence dependent setup times and machine eligibility, European Journal of Operational Research 169 (3) (2006) 781–800. 47

R.J. Paul, Production scheduling problem in the glass-container industry, Operations Research 27 (2) (1979) 290–302. 38

R.J. Wittrock, Scheduling algorithms for flexible flow lines, IBM Journal of Research and Development 29 (4) (1985) 401–412. 39

Rajeev Thakur, Rolf Rabenseifner, William Gropp, Optimization of Collective Communication Operations in MPICH International Journal of High Performance Computing Applications Spring 2005 19: 49-66. 61

Reisman, A., Kumar, A., and Motwani, J., Flowshop schedul- ing/sequencing research: A statistical review of the literature, 1952-1994. Ieee Transactions on Engineering Management, 44(3):316–329, 1997. 3

R. Tavakkoli-Moghaddam, N. Safaei, Farrokh Sassani. A memetic algorithm for the flexible flow line scheduling problem with processor blocking. Computers & Operations Research 36 (2009) 402 – 414. 52

Tavakkoli-Moghaddam and Safei, 2007, A New Mathematical Model for Flexible Flow Lines with Blocking Processor and Sequence-Dependent Setup Time, Source: Multiprocessor Scheduling: Theory and Applications, Book edited by Eugene Levner, ISBN 978-3-902613-02-8, pp.436, December 2007. 19

Richard P. Martin, Amin M. Vahdat, David E. Culler, Thomas E. Anderson, Effects of communication latency, overhead, and bandwidth in a cluster architecture, ISCA '97 Proceedings of the 24th annual international symposium on Computer architecture Pages 85 - 97 58

R. Ruiz a, T. Stutzle. An Iterated Greedy heuristic for the sequence dependent setup times flowshop problem with makespan and weighted tardiness objectives. European Journal of Operational Research 187 (2008) 1143–1159 111, 112

R. Ruiz, C. Maroto. A genetic algorithm for hybrid flowshops with sequence dependent setup times and machine eligibility . European Journal of Operational Research 169 (2006) 781–800. 119

R. Ruiz, J. Vázquez-Rodríguez. The hybrid flow shop scheduling problem. European Journal of Operational Research 205 (2010) 1–18. 3, 5, 6, 111, 112

S. Bertel, J.-C. Billaut, A genetic algorithm for an industrial multiprocessor flow shop scheduling problem with recirculation, European Journal of Operational Research 159 (3) (2004) 651–662. 47

S. Guirchoun, P. Martineau, J. C. Billaut, Total completion time minimization in a computer system with a server and two parallel processors, Computers and Operations Research 32 (3) (2005) 599–611. 32

S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by Simulated Annealing. Science Vol. 220(4598): pp. 671-680, 1983. 45, 73

S. Kochhar, R.J.T. Morris, Heuristic methods for flexible flow line scheduling, Journal of Manufacturing Systems 6 (4) (1987) 299–314. 37

S. Kochhar, R.J.T. Morris, W.S. Wong, The local search approach to flexible flow line scheduling, Engineering Costs and Production Economics 14 (1) (1988) 25–37. 37

S. Lawrence. Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (supplement). Technical report, Graduate School of Industrial Administration, Carnegie Mellon University, 1984. 35

S. Reza and S. Saghafian. Flowshop-scheduling problems with makespan criterion: a review. International Journal of Production Research, Vol. 43, No. 14, 15 July 2005, 2895–2929. 4

S.A. Brah, J.L. Hunsucker, Branch and bound algorithm for the flow-shop with multiple processors, European Journal of Operational Research 51 (1) (1991) 88–99. 33

S.A. Brah, Scheduling in a Flow Shop with Multiple Processors, Thesis/ Dissertation, University of Houston, 1988. 33

S.K. Verma, M.I. Dessouky, Multistage hybrid flowshop scheduling with identical jobs and uniform parallel machines, Journal of Scheduling 2 (1999) 135–150. 37

S.L. Narasimhan, P.M. Mangiameli, A comparison of sequencing rules for a two-stage hybrid flowshop, Decision Sciences 18 (2) (1987) 250–265. 38

S.L. Narasimhan, S.S. Panwalkar, Scheduling in a two-stage manufacturing process, International Journal of Production Research 22 (4) (1984) 555–564. 38

S.N. Kadipasaoglu, W. Xiang, B.M. Khumawala, A note on scheduling hybrid flow systems, International Journal of Production Research 35 (5) (1997) 1491–1494. 38

S.V. Sevastianov, Geometrical heuristics for multiprocessor flowshop scheduling with uniform machines at each stage, Journal of Scheduling 5 (3) (2002) 205–225. 39

Salveson, M. E., On a quantitative method in production planning and scheduling. Econometrica, 20(4):554–590,1952. 2

Shaukat A. Brah, Luan Luan Loo, Heuristics for scheduling in a flow shop with multiple processors, European Journal of Operational Research 113 (1999) 113-122. 3

T. Stützle and H.H. Hoos, Improving the Ant System: A detailed report on the MAX –MIN Ant System, FG Intellektik, FB Informatik, TU Darmstadt, Germany, Tech. Rep. AIDA–96–12, Aug. 1996. 42, 43

T.B.K. Rao, Sequencing in the order a, b, with multiplicity of machines for a single operation, OPSEARCH: Journal of the Operational Research Society of India 7 (1970) 135–144. 32

T.J. Sawik, A scheduling algorithm for flexible flow lines with limited intermediate buffers, Applied Stochastic Models and Data Analysis 9 (2) (1993) 127–138. 38

T.J. Sawik, Mixed integer programming for scheduling flexible flow lines with limited intermediate buffers, Mathematical and Computer Modelling 31 (13) (2000) 39–52. 33

T.J. Sawik, Mixed integer programming for scheduling surface mount technology lines, International Journal of Production Research 39 (14) (2001) 3219–3235. 34

T.J. Sawik, Scheduling flexible flow lines with no in-process buffers, International Journal of Production Research 33 (5) (1995) 1357–1367. 38

T.S. Arthanary, K.G. Ramaswamy, An extension of two machine sequencing problem, OPSEARCH: The Journal of the Operational Research Society of India 8 (4) (1971) 10–22. 32

Thijs Urlings, Heuristics and metaheuristics for heavily constrained hybrid flowshop problems, tesis de doctorado, Universidad Politécnica de Valencia, 2010. 1, 2

V. Suresh, A note on scheduling of two-stage flow shop with multiple processors, International Journal of Production Economics 49 (1) (1997) 77– 82. 38

Victor Yaurima, Larisa Burtseva, Andrei Tchernykh, Hybrid flowshop with unrelated machines, sequence-dependent setup time, availability constraints and limited buffers, Computers & Industrial Engineering, Elsevier (2008). 6

Vignier, A., Billaut, J. C., and Proust, C. (1999). Hybrid flowshop scheduling problems: State of the art. Rairo-Recherche Operationnelle-Operations Research, 33(2):117–183. 3, 6, 15, 33

W. Huang, S. Li, A Two-Stage Hybrid Flowshop with Uniform Machines and Setup Times, Mathl. Comput. Modellzng Vol. 27, No. 2, pp. 27-45, 1998 Copyright@1998 Elsevier Science Ltd. 6

W. Xiao, P. Hao, S. Zhang, X. Xu, Hybrid flow shop scheduling using genetic algorithms, in: Proceedings of the Third World Congress on Intelligent Control and Automation, IEEE Press, 2000, pp. 537–541. 47

W.B. Gooding, J.F. Pekny, P.S. Mccroskey, Enumerative approaches to parallel flowshop scheduling via problem transformation, Computers and Chemical Engineering 18 (10) (1994) 909–927. 33

W.L. Pearn, S.H. Chung, M.H. Yang, C.Y. Chen, The integrated circuit packaging scheduling problem (icpsp): a case study, International Journal of Industrial Engineering-Theory Applications and Practice 12 (3) (2005) 296–307. 34

Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Panda D.K., Gropp W., Thakur R, High performance MPI-2 one-sided communication over InfiniBand, Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium, Page(s): 531 - 538. 59

Wetzel. A. Evaluation of the Effectiveness of Genetic Algorithms in Combinatorial Optimization. University of Pittsburgh, Pittsburgh (unpublished). 1983. 34

Y. Yang, Optimization and Heuristic Algorithms for Flexible Flow Shop Scheduling, Ph.D. Thesis, Columbia University, 1998. 38

Y.-D. Lee, G.-C. Kim, S.-W. Choi, Bottleneck-focused scheduling for a hybrid flowshop, International Journal of Production Research 42 (1) (2004) 165–181. 38

Z.H. Jin, Z. Yang, T. Ito, Metaheuristic algorithms for the multistage hybrid flowshop scheduling problem, International Journal of Production Economics 100 (2) (2006) 322–334.

3, 46

Apéndice A

CÁLCULO DE LA COMPLEJIDAD TEMPORAL POR PASOS

Para realizar el cálculo de la complejidad temporal, se toman en cuenta las siguientes consideraciones:

1. El costo de las declaraciones de variables, constantes y comentarios es cero (0) pasos.

2. Para el caso de asignaciones el costo es de un paso, si son asignaciones múltiples, se consideran tantos pasos como asignaciones.

3. Para las llamadas a funciones el costo es de un paso, más los pasos que componen la función.

4. Para las estructuras de control como *for*, *do*, *while*, se múltiplica el número de ciclos que contiene, por los pasos contenidos dentro de la estructura de control.

5. Para sentencias condicionales *if {...} else* {...}, se toma el bloque de sentencias con el mayor número de pasos.

6. Para el caso de estructuras anidadas, se calculan los pasos de adentro hacia afuera, multiplicando el número de pasos de la estructura interior por el número de ciclos de la estructura que la contiene.

Consideremos el siguiente fragmento de código fuente.

```
1   void Metaheuristica_SCH( void )
2   {
3       int iCriterio_Paro, int iHormiga;
4       int iTotal = 0;
5       for ( iCriterio_Paro = 0; iCriterio_Paro < CFG_CRITERIO; iCriterio_Paro++ )
6       {
7           for ( iHormiga = 0; iHormiga < CFG_HORMIGAS; iHormiga++ )
8           {
9               Construye_Solucion( iHormiga );
10              Hormigas_CMAX( iHormiga );
11          }
12          Acciones_Daemon();
13          Inicializa_Soluciones();
14      }
15  }
```

Primero calculamos el número de pasos del ciclo más interno: linea 7 a 10.

- El ciclo *for* tiene un número de pasos de *iHormiga + 2,* es decir, se realiza *iHormiga* veces, más una unica asignación al inicio *(iHormiga = 0)* y la última comparación *(iHormiga < CFG_ HORMIGAS)* que se realiza pero que no entra el bloque for.

- Las líneas 9 y 10 se hacen en dos pasos.

- En total el número de pasos es de 2*(iHormiga) + 2.*

Después calculamos el ciclo *for* exterior: líneas 5, 6, 12, 13 y 14.

- El ciclo *for* tiene un número de pasos de *iCriterio_ Paro + 2,* es decir, se realiza *iCriterio_ Paro* veces, más una unica asignación al inicio *(iCriterio_ Paro = 0)* y la última comparación *(iCriterio_ Paro < CFG_ CRITERIO)* que se realiza pero que no entra el bloque for.

- Las líneas 12 y 13 se hacen en dos pasos.

- El número de pasos del ciclo externo es de 2*(iCriterio_ Paro) + 2.*

- Dado que el ciclo interno depende del externo, entonces el total de pasos de ambos ciclos es de 2*(iCriterio_ Paro) + 2 + iCriterio_ Paro(*2*(iHormiga) + 2)*, es decir, que el ciclo externo se hace 2*(iCriterio_ Paro) + 2*, sin tomar en cuenta el ciclo interno, para incluirlo entonces hay que multiplicar el número de veces que el ciclo externo entra al bloque que es de *iCriterio_ Paro* veces por el número de pasos del ciclo interno que es 2*(iHormiga) + 2.*

Finalmente calculamos el resto de pasos.

- La última línea que nos hace falta es la 4, que se realiza en un paso, asi tenemos que el número de pasos es de 2*(iCriterio_Paro) + 2 + iCriterio_Paro(*2*(iHormiga) + 2)* $+ 1$, por tanto la complejidad temporal de nuestra función es: *T(n)=4(iCriterio_Paro) + 2 (iHormiga)(iCriterio_Paro) + 3*.

Los que nos da un polinomio de primer orden, por tanto nuestra complejidad asintótica es $O(iHormiga\, iCriterio\_paro)$.

Por otra parte, sobre la base de [Laarhoven 1985], la complejidad computacional del algoritmo de recocido simulado, esta dada pro la expresión A.1.

$$O(\tau\, L\, ln\, |R|) \tag{A.1}$$

donde τ es el tiempo para generar y evaluar una posible solución, en el algoritmo propuesto esta dado por las funciones *Nueva_Solucion()* y *RS_MAX()*, en donde la primera genera una nueva solución y la segunda evalua la función objetivo como se muestra en la tabla 4.2, las cuales tienen una complejidad temporal en conjunto de $T(n) = 30 + 3m + 2m_k + 5n + 26mn + 7m\, m_k n$.

La variable L representa la longuitud de la cadena de Markov, que para el algoritmo esta dada por $m\, n(m\, n - 1)$, finalmente R representa el espacio de soluciones del FFS-SDST, que es calculado como $n! \left(\prod_{i=1}^{m} m_i\right)^m$ [Gourgand et al., 1999].

Sustituyendo estos valores en la expresión A.1, tenemos que la complejidad asintótica de RS es $O\left(30 + 3m + 2m_k + 5n + 26mn + 7m^2 m k_k n^2(-1 + mn)\left(n!\left(\prod_{i=1}^{m} m_i\right)^m\right)\right)$.

Apéndice B

EJEMPLO INSTANCIA GENERADA ALEATORIAMENTE

```
# Center for Research in Engineering and Applied Sciences
# Electrical && Computer Sciences Deparment
# jobs(n)=20, stages(k)=4, machines per stage(m)=4
20
4
4
56 90 25 63
84 8  84 36
84 68 92 97
70 59 73 45
64 55 62 22
28 41 10 42
29 65 57 77
78 81 37 35
71 61 97 56
69 81 91 54
50 84 51 20
43 24 65 8
78 28 29 7
68 39 48 97
4  6  74 82
86 12 18 58
72 15 15 42
96 7  95 46
90 47 66 34
70 31 42 49
# stage 1
15  18  14  7   3   2   7   4   3   25  24  25  3   4   14  21  8   18  7   7
19  5   18  17  17  10  1   10  18  11  22  7   4   11  14  6   12  20  9   14
20  8   14  22  11  2   18  18  19  24  24  13  3   17  5   20  1   5   4   19
16  1   25  19  11  14  24  23  8   8   12  3   15  25  24  1   1   16  19  20
14  17  8   17  9   13  12  10  17  15  3   7   16  3   25  1   17  24  23  24
6   9   2   21  8   25  21  9   16  14  4   5   6   12  21  15  24  8   24  16
22  2   22  13  5   22  13  21  21  11  20  2   20  21  22  3   21  18  11  11
6   15  15  12  2   11  1   25  18  25  15  15  1   12  2   6   9   15  1   5
25  20  6   19  16  2   21  11  19  7   21  25  22  11  11  23  21  12  23  14
11  13  4   12  24  5   17  8   20  17  12  19  12  18  13  2   19  9   12  13
16  8   12  13  18  23  10  14  9   8   3   20  20  6   6   19  11  23  2   5
14  13  24  1   6   11  3   24  20  15  12  10  22  24  22  15  21  7   4   5
14  6   24  9   11  5   2   22  2   3   1   16  16  24  16  21  10  18  20  4
7   6   14  4   5   11  18  25  17  21  5   6   2   3   14  13  8   16  9   9
18  9   24  9   8   14  4   18  7   24  22  14  5   10  18  9   21  11  9   13
7   13  18  8   16  7   20  23  22  3   6   15  12  5   24  20  18  3   12  25
1   8   14  5   17  6   13  13  16  22  25  23  9   17  5   25  24  25  22  21
3   3   10  14  7   8   8   25  10  19  25  11  2   13  15  19  19  3   6   10
```

```
24  5   7   8   22  12  7   20  11  4   15  13  6   25  2   13  8   10  12  17
3   12  3   5   24  18  23  17  20  3   1   19  7   8   2   4   19  8   23  4
# stage 2
11  13  17  17  12  18  5   19  2   17  11  5   3   13  10  1   5   7   18  25
9   19  19  16  1   20  19  20  3   17  24  14  4   15  6   16  8   10  9   10
1   20  15  4   7   24  5   12  5   22  12  14  16  6   5   17  25  24  11  3
15  10  16  19  25  21  9   7   6   18  17  7   12  6   10  19  5   14  6   10
11  18  24  1   23  3   18  23  1   4   25  16  13  15  10  12  11  19  19  16
11  10  22  23  16  7   17  20  20  22  5   5   14  3   6   11  5   24  8   6
2   8   22  14  22  7   25  8   25  18  24  11  3   20  8   18  1   24  13  21
21  17  1   10  19  7   21  24  5   4   5   6   11  1   19  8   7   19  15  7
12  13  17  14  8   25  7   9   24  19  4   19  11  5   4   4   11  24  3   15
2   7   20  13  8   13  20  15  7   10  21  18  23  13  6   6   12  13  14  10
6   18  4   16  22  7   20  7   6   22  21  7   4   16  20  12  3   14  1   10
24  22  2   21  9   8   2   21  20  15  5   1   7   9   16  4   15  11  10  21
8   6   3   11  21  22  22  24  11  23  8   9   19  10  5   3   18  6   23  13
21  3   13  3   11  4   6   1   15  16  21  22  21  23  8   17  19  5   16  4
2   24  13  21  8   17  23  1   23  21  14  18  23  2   20  8   6   25  8   20
16  4   16  12  1   23  3   20  2   19  24  4   17  12  24  25  4   22  25  1
17  14  18  14  15  13  21  20  13  4   14  3   7   5   15  8   3   17  3   5
10  1   8   2   12  7   1   15  3   1   16  20  14  9   8   3   21  4   23  9
7   11  12  14  16  1   21  18  18  23  22  3   24  5   4   11  12  4   1   14
5   16  8   18  24  16  21  20  19  18  4   1   3   15  14  19  15  9   11  8
# stage 3
7   8   10  6   13  13  16  24  17  16  13  22  6   21  14  5   11  9   25  4
1   3   4   4   17  18  23  7   2   8   14  8   16  24  13  3   12  3   2   4
19  14  25  25  9   14  4   19  22  4   23  23  6   2   2   23  19  24  5   20
7   19  2   22  18  15  25  4   18  1   7   11  14  7   11  23  20  14  16  17
18  14  14  24  15  16  21  8   15  1   2   21  20  4   18  12  19  18  15  11
19  22  22  8   3   7   5   22  21  21  14  14  9   2   12  23  18  7   5   7
8   7   3   2   11  21  13  4   13  3   15  6   24  12  14  2   19  18  24  14
13  12  3   21  14  14  18  6   20  23  13  3   5   16  4   16  11  16  20  24
18  9   5   17  21  18  18  14  10  16  3   23  2   5   18  16  18  11  21  13
9   9   15  14  24  19  4   10  10  23  8   3   7   12  19  2   5   12  15  15
2   17  12  4   21  4   19  14  15  15  1   23  23  16  11  22  9   15  6   18
12  13  20  18  25  13  19  4   24  8   19  1   25  5   4   21  9   23  9   23
13  10  20  10  25  6   6   8   20  12  25  7   24  19  24  24  7   18  3   5
1   21  6   25  25  10  21  8   7   5   5   19  14  25  4   14  6   10  21  1
21  21  7   20  15  6   18  21  23  20  1   23  15  6   23  15  15  19  22  22
23  2   16  12  1   20  25  7   4   21  7   25  16  13  19  5   18  12  25  16
7   1   14  21  6   11  10  21  4   7   17  2   8   8   13  9   2   12  15  6
7   22  5   23  9   24  2   2   11  2   18  17  2   6   13  8   16  23  3   20
4   20  21  12  2   9   21  4   20  10  9   2   7   14  24  16  13  1   17  23
2   10  15  4   15  2   11  6   24  14  25  3   8   21  14  10  4   10  13  24
# stage 4
20  22  1   1   10  25  16  23  25  8   21  1   17  10  5   6   12  15  12  10
4   12  13  12  7   2   21  11  11  9   10  5   5   10  5   15  9   21  12  9
3   7   10  19  17  14  25  3   4   11  12  7   22  25  18  4   1   14  15  12
22  24  16  2   9   21  16  18  16  3   1   18  9   10  12  1   23  12  3   1
23  14  8   20  14  25  23  14  13  13  25  10  11  16  11  20  11  2   12  1
5   13  19  14  22  5   14  20  16  16  21  13  5   3   8   18  3   5   7   16
17  7   25  3   22  11  23  8   13  9   9   17  21  2   5   18  7   18  13  23
8   8   11  13  10  18  5   13  23  11  3   15  18  3   17  15  14  14  22  2
23  5   18  19  7   23  12  14  16  24  11  24  6   22  11  16  15  16  4   12
1   6   1   19  9   18  8   22  7   4   23  5   9   16  24  16  14  10  4   4
9   15  2   14  12  13  5   1   3   9   12  4   14  13  22  23  5   4   20  11
8   18  16  17  8   14  7   21  23  11  25  6   25  2   20  12  14  25  12  16
8   23  20  22  10  16  19  15  20  14  1   2   6   16  18  14  4   25  10  2
10  9   7   10  10  2   21  24  2   7   15  9   5   9   6   15  25  25  4   19
13  4   21  18  19  14  6   23  13  15  24  23  24  6   8   9   8   3   7   9
10  21  18  15  5   23  4   4   22  7   23  10  11  18  2   5   6   8   2   19
23  25  17  21  6   24  5   13  1   11  21  11  7   13  25  12  11  3   15  8
10  13  17  20  5   19  24  11  2   1   5   24  1   21  20  6   19  24  18  20
10  14  5   16  2   5   3   13  8   17  20  17  5   12  12  9   6   11  20  7
11  25  6   12  20  1   17  14  24  10  8   9   23  13  25  25  17  2   12  24
```

Apéndice C

EJEMPLO SALIDA GENERADA POR SDAAP

```
# Centro de Investigaciones en Ingenieria y Ciencias Aplicadas
# archivo generado automaticamente por tunningrscluster
# listado de ejecuciones generadas
#
#
# Inicia ejecucion el Thu May 12 22:05:50 CDT 2011 en el nodo gridmorelos.uaem.mx
[1] /home/fjuarez/Sintoniza/RS/FFS_SDST_20x4x4x25_1/FFS-RS-Sintoniza(1,50,0.986,0.8)
        1. en 5.990 Segundos semilla 1305255950 Makespan=902
        2. en 5.970 Segundos semilla 1305255956 Makespan=917
        3. en 5.960 Segundos semilla 1305255962 Makespan=906
        4. en 5.950 Segundos semilla 1305255968 Makespan=933
        5. en 5.970 Segundos semilla 1305255974 Makespan=938
       ...
        25. en 5.950 Segundos semilla 1305256096 Makespan=925
        26. en 5.970 Segundos semilla 1305256102 Makespan=902
        27. en 5.950 Segundos semilla 1305256108 Makespan=884
        28. en 5.990 Segundos semilla 1305256114 Makespan=914
        29. en 5.980 Segundos semilla 1305256120 Makespan=900
        30. en 6.060 Segundos semilla 1305256127 Makespan=912
[2] /home/fjuarez/Sintoniza/RS/FFS_SDST_20x4x4x25_1/FFS-RS-Sintoniza(2,50,0.986,0.8)
        1. en 0.340 Segundos semilla 1305255950 Makespan=929
        2. en 0.350 Segundos semilla 1305255950 Makespan=929
        3. en 0.340 Segundos semilla 1305255951 Makespan=959
        4. en 0.340 Segundos semilla 1305255951 Makespan=959
        5. en 0.340 Segundos semilla 1305255951 Makespan=959
        ...
        25. en 0.340 Segundos semilla 1305255959 Makespan=987
        26. en 0.370 Segundos semilla 1305255959 Makespan=987
        27. en 0.350 Segundos semilla 1305255959 Makespan=987
        28. en 0.350 Segundos semilla 1305255960 Makespan=944
        29. en 0.340 Segundos semilla 1305255960 Makespan=944
        30. en 0.340 Segundos semilla 1305255960 Makespan=944
#  Total ejecuciones = 60 finalizado el Thu May 12 22:15:15 CDT 2011
```

Apéndice D

PROGRAMA PARA GENERAR INSTANCIAS DE PRUEBA

```
/*
 *  Centro de Investigaciones en Ingenieria y Ciencias Aplicadas
 *  GNU GPL 2011 - juarezfredy@uaem.mx
 *
 *  Generador de instancias para el HIBRID FLOW SHOP con tiempos de iniciado
 *  dependiente de la secuencia y con número de máquinas identicas en cada etapa
 *
 *  Tomado en base al articulo: Algorithm for a realistic variant of flowsop scheduling,
 *  B. Naderi, Ruben Ruiz, M. Zandieh, ELSEVIER 2010.
 *
 *  Para compilar este programa use GNU gcc Version 4.x
 */
#define HEAD1 "# Center for Research in Engineering and Applied Sciences\n"
#define HEAD2 "# Electrical && Computer Sciences Deparment\n"
#define NewLine "\n"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* funcion inline fast */
inline int Rand( int iMin, int iMax) { return (iMin + (int) (((float)iMax)*rand()/(RAND_MAX+(float)iMin))); }
/* numero de trabajos */
#define TRABAJOS 6
int N[TRABAJOS] = {20,40,50,60,80,150};
/* numero de etapas */
#define ETAPAS 3
int K[ETAPAS] = {2,4,8};
/* numero de máquinas por etapa es estatico */
#define MAQUINAS 4
int M[MAQUINAS] = {2,3,4,5};
/* tiempos de procesamiento uniformemente distribuido entre [1,max] */
int Pij = 99;
/* tiempos de iniciado dependientes de la secuencia uniformemente distribuido entre [1,max] */
#define SETUPS 4
int Sij[SETUPS] = {25,50,99,125};
/* numero de instancias generados por cada combinacion */
#define INSTANCIAS 5
/* p. ej. el numero de combinaciones es KxMxNxSijn en total para este programa = 196 combinaciones */
/* directorio base de trabajo y archivo */
char *Directorio = "/home/fjuarez/pruebas";
/* manejador de archivos */
FILE* File;
/* justificador para datos tabulados */
char* Tabulador = "   ";
/* funcion para crear el archivo de un nombre de una instancia */
int Crear_Archivo( char* Nombre )
{
        /* crear archivo de prueba para la instancia con el nombre adecuado */
        if ((File = fopen( Nombre, "w" )) == NULL )
```

```
        {
                printf("Error el archivo :%s no pudo ser abierto\n",Nombre);
                return ( EXIT_FAILURE );
        }
        else
                fprintf(File,"%s%s",HEAD1,HEAD2);
        return ( EXIT_SUCCESS );
}
/* escribe los jobs, stages and machines por etapa */
void Escribe_Pij( int Trabajos, int Etapas )
{
    char Tiempo[10];
        int N, K;
        for ( N = 0; N < Trabajos; N++ )
        {
                for ( K = 0; K < Etapas; K++)
                {
                        sprintf(Tiempo,"%d",Rand(1,Pij));
                        strncat(Tiempo, Tabulador, strlen(Tabulador) - strlen(Tiempo) );
                        fprintf(File, "%s",Tiempo);
                }
                fprintf(File,"%s", NewLine);
        }
}
/* escribe los tiempos de iniciado dependientes de la secuencia */
void Escribe_Sij( int Trabajos, int Etapas, int Setups )
{
    char Tiempo[10];
        int N1,N2, K;
        for ( K = 0; K < Etapas; K++)
        {
                fprintf(File,"# stage%d \n", K+1);
                for ( N1 = 0; N1 < Trabajos; N1++ )
                {
                        for ( N2 = 0; N2 < Trabajos; N2++ )
                        {
                                sprintf(Tiempo,"%d",Rand(1,Setups));
                                strncat(Tiempo, Tabulador, strlen(Tabulador) - strlen(Tiempo) );
                                fprintf(File, "%s ",Tiempo);
                        }
                        fprintf(File,"%s", NewLine);
                }
        }
}
int Crear_Instancias( void )
{
        /* nombre del benchmarck :HFS_STSD_nxkxmxmag,
         * los parametros n,m,k,mag seran sustituidos para dar lugar al nombre adecuado
        */
        char Instancia[100] = "";
        int Instancias, Trabajos, Etapas, Maquinas, Setups;
        for ( Trabajos = 0; Trabajos < TRABAJOS; Trabajos++)
            for ( Etapas = 0; Etapas < ETAPAS; Etapas++)
                for ( Maquinas = 0; Maquinas < MAQUINAS; Maquinas++)
                    for ( Setups = 0; Setups < SETUPS; Setups++)
                        for ( Instancias = 0; Instancias < INSTANCIAS; Instancias++)
                        {
                            /* construye nombre de la instancia */
                            sprintf(Instancia, "%s/FFS_STSD_%dx%dx%dx%d_%d",Directorio,N[Trabajos],K[Etapas],
                                               M[Maquinas],Sij[Setups],Instancias+1);
                           /* crea archivo con nombre de la instancia y escribe lso encabezados */
                           if ( Crear_Archivo(Instancia) != EXIT_SUCCESS )
                           {
                               printf("Eror al crear el Benchmark:%s\n",Instancia);
                               return ( EXIT_FAILURE );
```

```
                        }
                        /* escribe el tamaño de la instancia*/
                        fprintf(File, "# jobs(n)=%d, stages(k)=%d, machines per stage(m)=%d\n",N[Trabajos],
                                      K[Etapas],M[Maquinas]);
                        /* escribe los jobs, stages and machines por etapa */
                        fprintf(File, "%d\n%d\n%d\n",N[Trabajos],K[Etapas],M[Maquinas]);
                        /* escribe los tiempos de procesamiento Pij */
                        Escribe_Pij( N[Trabajos], K[Etapas] );
                        /* escribe los tiempos de iniciado dependientes de la secuencia */
                        Escribe_Sij( N[Trabajos], K[Etapas], Sij[Setups] );
                        /* cierra archivo */
                        fclose(File);
                        printf("Benchmark:%s creado!!\n",Instancia);
                    }
        return ( EXIT_SUCCESS );
}
int main(void)
{
        Crear_Instancias();
        return EXIT_SUCCESS;
}
```

Apéndice E

EJEMPLO ARCHIVO DE CONFIGURACIÓN DE PARÁMETROS

```
cuexcomate 1
ciicap-01 1
ciicap-02 1
ciicap-03 1
ciicap-04 1
ciicap-05 1
ciicap-gpu01 1
HORMIGAS: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
BETA: 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0
FEROMONAL: 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
FEROMONAG: 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
QO: 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
CICLOS: 500 750 1000 1250 1500 1750 2000
UB: 660
EJECUTABLE: /home/fjuarez/Sintoniza/SCH/FFS_SDST_40x4x4x25_1/FFS-SCH-Sintoniza
SCRIPT: /home/fjuarez/Sintoniza/SCH/FFS_SDST_40x4x4x25_1/tunningrscluster.sh
DIRECTORIO: /home/fjuarez/Sintoniza/SCH/FFS_SDST_40x4x4x25_1/
PRUEBAS: 30
SINTONIZAR: HORMIGAS BETA FEROMONAL FEROMONAG QO CICLOS/*
```

Apéndice F

EJEMPLO ANCLAJE DE NÚCLEOS

::MPI Intel::

```
cuexcomate:25
ciicap01:48
ciicap02:48
ciicap03:48
ciicap04:48
ciicap-gpu01:24
texcal:24
node01:48
node02:48
node03:48
node04:48
gpu01:24
```

::MPICH2::

```
192.168.1.253:6  binding=user:-1,-1,-1,-1,-1,-1,6,7,8,9,10,11
ciicap01:12  binding=user:0,1,2,3,4,5,6,7,8,9,10,11
ciicap02:12 binding=user:0,1,2,3,4,5,6,7,8,9,10,11
ciicap03:12 binding=user:0,1,2,3,4,5,6,7,8,9,10,11
ciicap04:12 binding=user:0,1,2,3,4,5,6,7,8,9,10,11
ciicap-gpu01:6 binding=user:0,1,2,3,4,5
192.168.100.250:6 binding=user:-1,-1,-1,-1,-1,-1,6,7,8,9,10,11
node01:12 binding=user:0,1,2,3,4,5,6,7,8,9,10,11
node02:12 binding=user:0,1,2,3,4,5,6,7,8,9,10,11
node03:12 binding=user:0,1,2,3,4,5,6,7,8,9,10,11
node04:12 binding=user:0,1,2,3,4,5,6,7,8,9,10,11
gpu01:6 binding=user:0,1,2,3,4,5
```

Apéndice G

EJEMPLO LANZADOR DE PROCESOS

```
#!/bin/sh
#
# Centro de Investigaciones en Ingenieria y Ciencias Aplicadas
# archivo generado automaticamente por tunningrscluster
# lanzador maestro para las ejecucion de las instancias
# creado el lun feb 13 13:50:05 CST 2012 en cuexcomate
# www.gridmorelos.uaem.mx:8080
#
ssh ciicap01 /home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/tunningrscluster.sh
/home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/SCH.conf 1 &
ssh ciicap01 /home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/tunningrscluster.sh
/home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/SCH.conf 2 &
ssh ciicap02 /home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/tunningrscluster.sh
/home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/SCH.conf 3 &
ssh ciicap02 /home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/tunningrscluster.sh
/home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/SCH.conf 4 &
ssh ciicap03 /home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/tunningrscluster.sh
/home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/SCH.conf 5 &
ssh ciicap03 /home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/tunningrscluster.sh
/home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/SCH.conf 6 &
ssh ciicap04 /home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/tunningrscluster.sh
/home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/SCH.conf 7 &
ssh ciicap04 /home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/tunningrscluster.sh
/home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/SCH.conf 8 &
ssh ciicap-gpu01 /home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/tunningrscluster.sh
/home/fjuarez/Sintoniza/SCH/FFS_SDST_80x4x4x25_1/SCH.conf 9 &
```

Apéndice H

SHELL SCRIPT PARA SDAAP

```
#!/bin/sh
#
# Centro de Investigaciones en Ingenieria y Ciencias Aplicadas
# GNU GPL 2011 - juarezfredy@uaem.mx
#
# Shell script para la Sintonización Distribuida Automática Aplicada en Paralelo - SDAAP
# realiza una distribución uniforme de conbinaciones de series de cada parámetro y las
# distribuye sobre los nodos de un Cluster o Grid, integra y determina el mejor parámetro
# y lo fija hasta completar el total de parámetros.
Nodos=()
Procesadores=()
Generacion=()
Tasacruce=()
Poblacion=()
EJECUTABLE=""
SCRIPT=""
PRUEBAS=0
PARAMETROS=()
DIRECTORIO=""
SINTONIZAR=""
# funcion que realiza un parser al arcivo de configuracion
function Leer_Parametros()
{
  local TokensFile=($(cat $1))  # tokenizar archivo de configuracion
  local Len=${#TokensFile[@]}   # numero de tokens
  local Indice=0                # indice de los arreglos
  local Selector=1              # selector de parametros
  local IndiceToken=0           # indice de los tokens
  while [ $IndiceToken -lt $Len ]; do
    Token=${TokensFile[$IndiceToken]} # obtiene token a analizar
    case $Token in # selecciona tipo de token
      "GENERACION:")  Selector=2; Indice=-1;
      "TASACRUCE:")  Selector=3; Indice=-1;
      "POBLACION:")  Selector=4; Indice=-1;
      "EJECUTABLE:") Selector=5; Indice=-1;
      "SCRIPT:") Selector=6; Indice=-1;
      "DIRECTORIO:") Selector=7; Indice=-1;
      "PRUEBAS:") Selector=8; Indice=-1;
      "SINTONIZAR:") Selector=9; Indice=-1;
    esac
    if [ $Indice -gt -1 ]; then # agrega token al arreglo correspondiente
      case $Selector in
        1) Nodos[$Indice]=$Token
           IndiceToken=$(( IndiceToken + 1 ))
           Token=${TokensFile[$IndiceToken]}
           Procesadores[Indice]=$Token;;
        2) Generacion[$Indice]=$Token;;
        3) Tasacruce[$Indice]=$Token;;
        4) Poblacion[$Indice]=$Token;;
        5) EJECUTABLE=$Token;;
        6) SCRIPT=$Token;;
```

```
        7) DIRECTORIO=$Token;;
        8) PRUEBAS=$Token;;
        9) PARAMETROS[$Indice]=$Token;;
      esac
    fi
    Indice=$(( Indice + 1 ))
    IndiceToken=$(( IndiceToken + 1 ))
  done
}
# funcion que selecciona parametro a sintonizar
function Selecciona_Parametro()
{
  SINTONIZAR=${PARAMETROS[$INDICE_PARAMETRO]}
  if [ $SINTONIZAR == "GENERACION" ]; then
       GENERACION=${#Generacion[@]}
  else
       GENERACION=1
  fi
  if [ $SINTONIZAR == "TASACRUCE" ]; then
       TASACRUCE=${#Tasacruce[@]}
       if [ $FIJA_PARAMETRO == "SI" ]; then Generacion[0]=$MEJOR_PARAMETRO; fi
  else
       TASACRUCE=1
  fi
  if [ $SINTONIZAR == "POBLACION" ]; then
       POBLACION=${#Poblacion[@]}
       if [ $FIJA_PARAMETRO == "SI" ]; then Tasacruce[0]=$MEJOR_PARAMETRO; fi
  else
       POBLACION=1
  fi
}
# configura serie de archivos para todo el cluster en base al parametro a configurar
function Prepara_Archivos()
{
for (( INodo=0; INodo < ${#Nodos[@]}; INodo++ ))
  do
    for (( IProcesador=0; IProcesador < ${Procesadores[$INodo]}; IProcesador++ ))
    do
       PROCESADOR_MAQUINA[$PROCESADORES]="${Nodos[$INodo]}"
       PROCESADOR_ARCHIVO[$PROCESADORES]="$DIRECTORIO${Nodos[$INodo]}_$(( PROCESADORES + 1 ))_$SINTONIZAR.out"
       if [ ! -f ${PROCESADOR_ARCHIVO[$PROCESADORES]} ]; then
          touch ${PROCESADOR_ARCHIVO[$PROCESADORES]}
          echo "# Centro de Investigaciones en Ingenieria y Ciencias Aplicadas" >> ${PROCESADOR_ARCHIVO[$PROCESADORES]}
          echo "# archivo generado automaticamente por tunningrscluster" >> ${PROCESADOR_ARCHIVO[$PROCESADORES]}
          echo "# listado de ejecuciones generadas" >> ${PROCESADOR_ARCHIVO[$PROCESADORES]}
          echo "#" >> ${PROCESADOR_ARCHIVO[$PROCESADORES]}
       fi
       PROCESADORES=$(( PROCESADORES + 1 ))
     done
  done
}
# verifica los parametros
if [ $# -eq 0 ]; then
     echo "Usar TuningRS <option> archivo_configuracion idproceso"
     exit -1
fi
if [ $# -eq 1 ]; then
     if [ $1 == "--ayuda" ]; then
          echo "Usar: TuningRS [opciones] idproceso"
          echo "opciones:"
          echo "  --ayuda         muestra la ayuda a cerca del programa"
          echo "  --simula        simula la ejecucion de los procesos en el cluster correspondientes a idproceso"
          echo "  --simulatodo    simula toda la distribucion de los proesos en el cluster correspondientes a idproceso"
          echo "  --construye     construye lanzador en shell script para la sintonizacion en el cluster"
          echo "-------------------------------------------------------------------------------------------------------"
```

```
         echo "NOTAS: idproceso no puede ser mayor que el total de procesos definidos por NODOS*PROCESADORES"
         echo "Cuerpo academico de optimizacion y sofware, 2010, http://gridmorelos.uaem.mx:8080"
         exit 0
      else
         echo "opcion $1 desconocida"
         exit -1
      fi
fi
# determina si el numero de parametros es el adecuado y si el archivo existe
if [ $# -eq 2  ]; then
     if [ -f $1 ]; then ARCHIVO=$1; IDPROCESO=$2; else echo "el archivo $1 no existe"; exit -1; fi
fi
if [ $# -eq 3  ]; then
     if [ -f $2 ]; then ARCHIVO=$2; IDPROCESO=$3; else echo "el archivo $1 no existe"; exit -1; fi
fi
if [ $# -gt 3  ]; then echo "parametros incorrectos, verifique."; exit -1; fi
Leer_Parametros $ARCHIVO
# numero de nodos disponibles
NODOS=${#Nodos[@]}
# numero total de procesadores disponibles en el cluster
PROCESADORES=0
# arreglo de correspondencia procesador->maquina
PROCESADOR_MAQUINA=( )
#arreglo correspondencia procesador->archivo de escritura
PROCESADOR_ARCHIVO=( )
# numero de paramatros
NUMERO_PARAMETROS=${#PARAMETROS[@]}
# solo construye el lanzador maestro y sale
if [ $1 == "--construye" ]; then
     echo -e "#!/bin/sh\n#" > $3
     echo "# Centro de Investigaciones en Ingenieria y Ciencias Aplicadas" >> $3
     echo "# archivo generado automaticamente por tunningrscluster" >> $3
     echo "# lanzador maestro para las ejecucion de las instancias" >> $3
     echo "# creado el $(date) en $(hostname)" >> $3
     echo "# www.gridmorelos.uaem.mx:8080" >> $3
     echo "#" >> $3
     for (( INodo=0; INodo < ${#Nodos[@]}; INodo++ ))
     do
       for (( IProcesador=0; IProcesador < ${Procesadores[$INodo]}; IProcesador++ ))
       do
         PROCESADOR_MAQUINA[$PROCESADORES]="${Nodos[$INodo]}"
         echo "ssh ${PROCESADOR_MAQUINA[$PROCESADORES]} $SCRIPT $DIRECTORIO$ARCHIVO $(( PROCESADORES + 1 )) &" >> $3
         PROCESADORES=$(( PROCESADORES + 1 ))
       done
     done
     exit 0
fi
MEJOR_PARAMETRO=0
INDICE_PARAMETRO=0
FIJA_PARAMETRO="NO"
# selecciona paramatro a sintonizar
Selecciona_Parametro
# configura serie de archivos para todo el cluster en base al parametro a configurar
Prepara_Archivos
# determina si hay mas procesadores que variaciones de los parametros
MAX_FILES=( 0 $PROCESADORES $PROCESADORES $PROCESADORES )
if [ $PROCESADORES -gt ${#Generacion[@]} ]; then MAX_FILES[1]=${#Generacion[@]}; fi
if [ $PROCESADORES -gt ${#Tasacruce[@]} ]; then MAX_FILES[2]=${#Tasacruce[@]}; fi
if [ $PROCESADORES -gt ${#Poblacion[@]} ]; then MAX_FILES[3]=${#Poblacion[@]}; fi
echo "Total procesadores=$PROCESADORES, a mi $IDPROCESO me toca ${PROCESADOR_ARCHIVO[$IDPROCESO - 1]}"
if [ $IDPROCESO -gt $PROCESADORES ]; then
   echo "el proceso $IDPROCESO >> $PROCESADORES(numero total de procesadores) especificado en $ARCHIVO";
   exit -1;
fi
echo "${Nodos[@]}=$NODOS"
```

```
echo "${Procesadores[@]}=$PROCESADORES"
echo "${Generacion[@]}=$GENERACION"
echo "${Tasacruce[@]}=$TASACRUCE"
echo "${Poblacion[@]}=$POBLACION"
echo "EJECUTABLE=$EJECUTABLE"
echo "ASINTONIZAR=$SINTONIZAR de ${PARAMETROS[@]}"
echo "Numero de parametros : $NUMERO_PARAMETROS"
echo "PRUEBAS=$PRUEBAS"
echo "MAX_FILES=${MAX_FILES[@]}"
Total=$(( ${#Generacion[@]} + ${#Tasacruce[@]} + ${#Poblacion[@]} ))
Totalpruebas=$(( Total * 30 ))
echo "Total combinaciones = $Total * 30 pruebas = $Totalpruebas"
function Genera()
{
  echo "Inicia Sintonizado de $SINTONIZAR"
  if [ $# -eq 2 ]; then
     echo -e "#\n# Inicia ejecucion el $(date) en el nodo $(hostname)" >> ${PROCESADOR_ARCHIVO[$IDPROCESO - 1]};
  fi
  for (( G = 0; G < GENERACION; G++ ))
  do
    for (( T = 0; T < TASACRUCE; T++ ))
    do
      for (( PO = 0; PO < POBLACION; PO++ ))
      do
        ID=$(( (PROCESO_ACTUAL - 1) % PROCESADORES ))
        if [ "$1" == "--simulatodo" ]; then
           NODO_NOMBRE=${PROCESADOR_MAQUINA[$ID]}
           NODO_ARCHIVO=${PROCESADOR_ARCHIVO[$ID]}
           cat $NODO_ARCHIVO | grep "(${Generacion[$G]},${Tasacruce[$T]},${Poblacion[$PO]})" >
               /dev/null && (( EXISTE=1 )) || (( EXISTE=0 ))
           if [ $PROCESO_SIGUIENTE -eq $PROCESO_ACTUAL ]; then
              echo "[*$PROCESO_ACTUAL] mpirun $EJECUTABLE parametros(${Generacion[$G]},${Tasacruce[$T]},${Poblacion[$PO]}) en
                    $NODO_NOMBRE, estado=$EXISTE"
              PROCESO_SIGUIENTE=$(( PROCESO_ACTUAL + PROCESADORES ))
           else
              echo "[ $PROCESO_ACTUAL] mpirun $EJECUTABLE parametros(${Generacion[$G]},${Tasacruce[$T]},${Poblacion[$PO]}) en
                    $NODO_NOMBRE, estado=$EXISTE"
           fi
         fi
         if [ "$1" == "--simula" ]; then
            if [ $PROCESO_SIGUIENTE -eq $PROCESO_ACTUAL ]; then
               NODO_NOMBRE=${PROCESADOR_MAQUINA[$ID]}
               NODO_ARCHIVO=${PROCESADOR_ARCHIVO[$ID]}
               cat $NODO_ARCHIVO | grep "(${Generacion[$G]},${Tasacruce[$T]},${Poblacion[$PO]}" >
                   /dev/null && (( EXISTE=1 )) || (( EXISTE=0 ))
               echo "[$PROCESO_ACTUAL] mpirun $EJECUTABLE parametros(${Generacion[$G]},${Tasacruce[$T]},${Poblacion[$PO]}) en
                     $NODO_NOMBRE, estado=$EXISTE"
               PROCESO_SIGUIENTE=$(( PROCESO_ACTUAL + PROCESADORES ))
             fi
           fi
           if [ $# -eq 2 ] && [ $PROCESO_SIGUIENTE -eq $PROCESO_ACTUAL ]; then
              NODO_NOMBRE=${PROCESADOR_MAQUINA[$ID]}
              NODO_ARCHIVO=${PROCESADOR_ARCHIVO[$ID]}
              cat $NODO_ARCHIVO | grep "(${Generacion[$G]},${Tasacruce[$T]},${Poblacion[$PO]})" >
                  /dev/null && (( EXISTE=1 )) || (( EXISTE=0 ))
              if [ $EXISTE -eq 0 ]; then
                 echo "[$PROCESO_ACTUAL] $EJECUTABLE(${Generacion[$G]},${Tasacruce[$T]},${Poblacion[$PO]})
                       ...ejecutandose en $NODO_NOMBRE"
                 echo "[$PROCESO_ACTUAL] $EJECUTABLE(${Generacion[$G]},${Tasacruce[$T]},${Poblacion[$PO]})"  >> $NODO_ARCHIVO
                 for (( P = 1;  P <= PRUEBAS; P++ ))
                 do
                     mpirun -f nodes -np $(( Poblacion[PO] + 1 )) $EJECUTABLE $P ${Generacion[$G]} ${Tasacruce[$T]}
                             ${Poblacion[$PO]} < /dev/null >> $NODO_ARCHIVO
                 done
                 echo "[$PROCESO_ACTUAL] proceso en $NODO_NOMBRE Terminado!"
```

```
                EJECUCIONES=$(( EJECUCIONES + 1 ))
              else
                echo "[$PROCESO_ACTUAL] mpirun $EJECUTABLE parametros(${Generacion[$G]},${Tasacruce[$T]},${Poblacion[$PO]})
                      se ejecuto en $NODO_NOMBRE siguiente"
              fi
              PROCESO_SIGUIENTE=$(( PROCESO_ACTUAL + PROCESADORES ))
            fi
            PROCESO_ACTUAL=$(( PROCESO_ACTUAL + 1 ))
          done
      done
  done
  echo "Termina Sintonizado de $SINTONIZAR"
  if [ $# -eq 2 ]; then
     echo -e "#  Total ejecuciones = $((EJECUCIONES*30)) finalizado el $(date)\n" >> "$NODO_ARCHIVO"
     touch "$NODO_ARCHIVO.ctrl"
  fi
}
INDICE_PARAMETRO=0
for (( Parametro = 1; Parametro <= NUMERO_PARAMETROS; Parametro++ ))
do
  PROCESO_ACTUAL=1
  PROCESO_SIGUIENTE=$IDPROCESO
  EXISTE=0
  EJECUCIONES=0
  PROCESADORES=0
  Selecciona_Parametro $#
  Prepara_Archivos
  Genera $1 $2 $3
  INDICE_PARAMETRO=$(( INDICE_PARAMETRO + 1 ))
  FIJA_PARAMETRO="NO"
  if [ $# -eq 2 ]; then
     if (( IDPROCESO == 1 )); then
        i="$(ls ${DIRECTORIO}*${SINTONIZAR}*.ctrl | wc -l)"
        while (( i < MAX_FILES[Parametro] ))
        do
            i="$(ls ${DIRECTORIO}*${SINTONIZAR}*.ctrl | wc -l)"
            sleep 1
        done
        MEJOR_PARAMETRO=( echo $(awk -v Variable=$Parametro -f ${DIRECTORIO}mejormedia.awk ${DIRECTORIO}*${SINTONIZAR}*.out
                        | grep Correspondiente))
        echo "parametro seleccionado :${MEJOR_PARAMETRO[3]}"
        rm ${DIRECTORIO}*${SINTONIZAR}*.ctrl
        echo ${MEJOR_PARAMETRO[3]} > ${DIRECTORIO}${SINTONIZAR}.best
        else
              if [ ! -f ${DIRECTORIO}${SINTONIZAR}.best ]; then
                 (( EXISTE=0 ))
              else
                 (( EXISTE=1 ))
              fi
              while (( EXISTE == 0 ))
              do
                  echo "Proceso $IDPROCESO esperando los demas procesos para leer ${DIRECTORIO}${SINTONIZAR}.best"
                  if [ ! -f ${DIRECTORIO}${SINTONIZAR}.best ]; then
                     (( EXISTE=0 ))
                  else
                     (( EXISTE=1 ))
                  fi
                  sleep 1
              done
        fi
        MEJOR_PARAMETRO=$(cat ${DIRECTORIO}${SINTONIZAR}.best)
        FIJA_PARAMETRO="SI"
  fi
  if [ "$1" == "--simulatodo" ] || [ "$1" == "--simula" ]; then
     MEJOR_PARAMETRO=( echo $(awk -v Variable=$Parametro -f ${DIRECTORIO}mejormedia.awk ${DIRECTORIO}*${SINTONIZAR}*.out
```

```
                    | grep Correspondiente))
      MEJOR_PARAMETRO=${MEJOR_PARAMETRO[3]}
      if [ $MEJOR_PARAMETRO != "0" ]; then
         FIJA_PARAMETRO="SI"
         echo "parametro seleccionado :$MEJOR_PARAMETRO"
      fi
   fi
done
```

Apéndice I

BALANCEO UNIFORME DE PROCESOS

```
USER     PID  %CPU     %MEM      VSZ    RSS   TTY    STAT   START      TIME COMMAND
fjuarez  11332 25.0    0.4       140772 105900 ?      R      20:58      11:45 ./AG-SCH-RS-Sintoniza
fjuarez  11333 25.3    0.4       140772 105880 ?      R      20:58      11:52 ./AG-SCH-RS-Sintoniza
fjuarez  11334 24.9    0.4       140772 105896 ?      R      20:58      11:40 ./AG-SCH-RS-Sintoniza
fjuarez  11335 24.9    0.4       140772 105880 ?      R      20:58      11:41 ./AG-SCH-RS-Sintoniza
fjuarez  11336 24.9    0.4       140772 105896 ?      R      20:58      11:41 ./AG-SCH-RS-Sintoniza
fjuarez  11337 24.8    0.4       140772 105880 ?      R      20:58      11:38 ./AG-SCH-RS-Sintoniza
fjuarez  11338 25.3    0.4       140772 105896 ?      R      20:58      11:54 ./AG-SCH-RS-Sintoniza
fjuarez  11339 24.8    0.4       140772 105880 ?      R      20:58      11:40 ./AG-SCH-RS-Sintoniza
fjuarez  11340 24.9    0.4       140772 105900 ?      R      20:58      11:40 ./AG-SCH-RS-Sintoniza
fjuarez  11341 24.8    0.4       140772 105880 ?      R      20:58      11:40 ./AG-SCH-RS-Sintoniza
fjuarez  11342 24.7    0.4       140772 105896 ?      R      20:58      11:35 ./AG-SCH-RS-Sintoniza
fjuarez  11343 24.6    0.4       140772 105880 ?      R      20:58      11:33 ./AG-SCH-RS-Sintoniza
fjuarez  11344 24.8    0.4       140772 105896 ?      R      20:58      11:40 ./AG-SCH-RS-Sintoniza
fjuarez  11345 24.8    0.4       140772 105880 ?      R      20:58      11:37 ./AG-SCH-RS-Sintoniza
fjuarez  11346 24.9    0.4       140772 105892 ?      R      20:58      11:42 ./AG-SCH-RS-Sintoniza
fjuarez  11347 24.9    0.4       140772 105880 ?      R      20:58      11:42 ./AG-SCH-RS-Sintoniza
fjuarez  11348 24.9    0.4       140772 105896 ?      R      20:58      11:42 ./AG-SCH-RS-Sintoniza
fjuarez  11349 24.8    0.4       140772 105880 ?      R      20:58      11:40 ./AG-SCH-RS-Sintoniza
fjuarez  11350 24.6    0.4       140772 105900 ?      R      20:58      11:34 ./AG-SCH-RS-Sintoniza
fjuarez  11351 25.0    0.4       140772 105880 ?      R      20:58      11:44 ./AG-SCH-RS-Sintoniza
fjuarez  11352 24.8    0.4       140772 105900 ?      R      20:58      11:40 ./AG-SCH-RS-Sintoniza
fjuarez  11353 24.8    0.4       140772 105880 ?      R      20:58      11:40 ./AG-SCH-RS-Sintoniza
fjuarez  11354 24.7    0.4       140772 105896 ?      R      20:58      11:35 ./AG-SCH-RS-Sintoniza
fjuarez  11355 24.6    0.4       140772 105880 ?      R      20:58      11:32 ./AG-SCH-RS-Sintoniza
fjuarez  11356 24.8    0.4       140772 105900 ?      R      20:58      11:39 ./AG-SCH-RS-Sintoniza
fjuarez  11357 24.7    0.4       140772 105880 ?      R      20:58      11:36 ./AG-SCH-RS-Sintoniza
fjuarez  11358 24.9    0.4       140772 105900 ?      R      20:58      11:41 ./AG-SCH-RS-Sintoniza
fjuarez  11359 24.9    0.4       140772 105880 ?      R      20:58      11:41 ./AG-SCH-RS-Sintoniza
fjuarez  11360 24.9    0.4       140772 105900 ?      R      20:58      11:41 ./AG-SCH-RS-Sintoniza
fjuarez  11361 25.1    0.4       140772 105880 ?      R      20:58      11:48 ./AG-SCH-RS-Sintoniza
fjuarez  11362 24.7    0.4       140772 105896 ?      R      20:58      11:36 ./AG-SCH-RS-Sintoniza
fjuarez  11363 24.8    0.4       140772 105880 ?      R      20:58      11:40 ./AG-SCH-RS-Sintoniza
fjuarez  11364 24.9    0.4       140772 105900 ?      R      20:58      11:40 ./AG-SCH-RS-Sintoniza
fjuarez  11365 24.7    0.4       140772 105900 ?      R      20:58      11:37 ./AG-SCH-RS-Sintoniza
fjuarez  11366 25.8    0.4       140772 105880 ?      R      20:58      12:06 ./AG-SCH-RS-Sintoniza
fjuarez  11367 24.8    0.4       140772 105900 ?      R      20:58      11:40 ./AG-SCH-RS-Sintoniza
fjuarez  11368 24.8    0.4       140772 105880 ?      R      20:58      11:40 ./AG-SCH-RS-Sintoniza
fjuarez  11369 24.8    0.4       140772 105880 ?      R      20:58      11:38 ./AG-SCH-RS-Sintoniza
fjuarez  11370 24.9    0.4       140772 105904 ?      R      20:58      11:41 ./AG-SCH-RS-Sintoniza
fjuarez  11371 24.9    0.4       140772 105880 ?      R      20:58      11:41 ./AG-SCH-RS-Sintoniza
fjuarez  11372 24.8    0.4       140772 105896 ?      R      20:58      11:40 ./AG-SCH-RS-Sintoniza
fjuarez  11373 24.8    0.4       140772 105880 ?      R      20:58      11:39 ./AG-SCH-RS-Sintoniza
fjuarez  11374 24.9    0.4       140772 105900 ?      R      20:58      11:40 ./AG-SCH-RS-Sintoniza
fjuarez  11375 24.9    0.4       140772 105880 ?      R      20:58      11:40 ./AG-SCH-RS-Sintoniza
```

Apéndice J

Código fuente AGHCGrid

```
/*
Centro de Investigaciones en Ingenieria y Ciencias Aplicadas
GNU GPL 2010 - juarezfredy@uaem.mx
Este programa es software libre, puede redistribuirlo y / o modificarlo bajo
los términos de la Licencia Pública General de GNU publicada por la
Free Software Foundation, bien de la versión 2 de la Licencia, o (a su
elección) cualquier versión posterior.
Use  compilador de C (gcc, icc) y librerias de MPI (MPICH, MPICH2, OpenMPI, Intel MPI).
www.gridmorelos.uaem.mx:8080
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <mpi.h>
/* definiciones */
#define CFG_ETAPAS      4
#define CFG_MAQUINAS    4
#define CFG_TRABAJOS    140
#define MAESTRO 0
#define MAX_POBLACION 500
#define CFG_HORMIGAS    16
#define CFG_BETA        0.4
#define CFG_FEROMONAL   0.9
#define CFG_FEROMONAG   0.2
#define CFG_Qo          0.2
#define CFG_UB          2035
#define CFG_CRITERIO 1500
#define EXPLORA         1
#define EXPLOTA         2
#define ETAPAS (CFG_ETAPAS + 2)
#define OPERACIONES_MAQUINAS ((CFG_ETAPAS * CFG_TRABAJOS * CFG_MAQUINAS) + 2)
#define OPERACIONES_TRABAJOS ((CFG_ETAPAS * CFG_TRABAJOS) + 2)
#define ARCOS_N (CFG_TRABAJOS * CFG_MAQUINAS)
#define ARCOS_MAQUINA (((CFG_TRABAJOS * CFG_MAQUINAS) * 2) - CFG_MAQUINAS)
#define ARCOS_ETAPA (ARCOS_MAQUINA) * (CFG_TRABAJOS * CFG_MAQUINAS)
#define ARCOS_ETAPAS ((ARCOS_ETAPA) * (CFG_ETAPAS - 1))
#define ARCOS_MAQUINA_F (((CFG_TRABAJOS * CFG_MAQUINAS) - CFG_MAQUINAS) + 1)
#define ARCOS_F (ARCOS_MAQUINA_F) * (CFG_TRABAJOS * CFG_MAQUINAS)
#define TOTAL_ARCOS (ARCOS_N) + (ARCOS_ETAPAS) + (ARCOS_F) + 1
#define PAR_COMENTARIO    5
#define CFG_TEMPERATURA   40
#define CFG_FACTOR        0.990
#define CFG_FROZEN        0.0001
#define CFG_CICLOS        (CFG_TRABAJOS * CFG_ETAPAS * 2) * 70;
#define CFG_EVDATA         CFG_TRABAJOS * CFG_ETAPAS
/* todos los arcos desde la etapa N hasta la F */
typedef struct arco
{
    int    ID;
```

```
    float  Feromona;
    float  Costo;
    float  ProductoAB;
    float  Probabilidad;
    int    Origeni;
    int    Origenj;
    int    Origenk;
    int    Destinoi;
    int    Destinoj;
    int    Destinok;
    int    l;
} ARCO;
/* operaciones trabajo/maquina/etapa todas las posibles */
typedef struct maquina
{
    int ID;
    int iAInicial;
    int iAFinal;
    int i;
    int j;
    int k;
    int Costo;
} MAQUINA;
/* operaciones trabajo/etapa con disjuntiva maquina */
typedef struct trabajo
{
    int ID;
    int iVisitado;
    int iMInicial;
    int iMFinal;
    int j;
    int k;
} TRABAJO;
/* etapas del sistema */
typedef struct etapa
{
    int k;
    int iTInicial;
    int iTFinal;
} ETAPA;
/* sistema de colonia de hormigas */
typedef struct sistema
{
    ARCO Arco[TOTAL_ARCOS];
    MAQUINA Maquina[OPERACIONES_MAQUINAS];
    TRABAJO Trabajo[OPERACIONES_TRABAJOS];
    ETAPA Etapa[ETAPAS];
}SISTEMA;
/* estructura para las soluciones de las hormigas */
typedef struct Hormigas
{
    int Arco[OPERACIONES_TRABAJOS];
    int Disponible[OPERACIONES_TRABAJOS];
    int Makespan;
} HORMIGAS;
/* estructura para recocido simulado */
typedef struct rs
{
    float  Temperatura;
    float  Factor;
    float Frozen;
    int  Ciclos;
} RS;
/* estructura de vecindad para el RS */
typedef struct ev
```

```
{
    int iTrabajo;
    int iMaquina;
} EV;
/* estcutura para el intercambio */
typedef struct intercambio
{
    int Trabajo1;
    int Trabajo2;
    int Maquina1;
    int Maquina2;
} INTERCAMBIO;
/* estructura del AG */
typedef struct INDIVIDUO
{
    int Numero;
    int Adaptacion;
    float Inversa;
    float Probabilidad;
    float Acumulado;
    EV Individuo[CFG_EVDATA];
} INDIVIDUO;
inline int Rand( int iMin, int iMax) { return (iMin + (int) (((float)iMax)*rand()/(RAND_MAX+(float)iMin))); }
inline float Random01() { return drand48(); }
void Inicializa_SCH();                    /* inicializa grafo, arcos, distancias y feromonas */
void Configura_Etapas();                  /* configura numero de etapas */
void Configura_Trabajos();                /* configura numero de trabajos */
void Configura_Maquinas();                /* configura numero de maquinas */
void Configura_Arcos();                   /* configura numero de arcos */
void Agregar_To();                        /* agrega la cantidad de feromona inicial */
void Inicializa_Soluciones();             /* limpia variables de las hormigas */
int  Agregar_Tiempos(char* aFuente);      /* leer parametros de SCH */
void Metaheuristica_SCH();                /* optimizacion por sistema de hormigas - ANTSystem */
void Construye_Solucion ( int iHormiga ); /* construye la solucion para la hormiga k */
void Acciones_Daemon();                   /* acciones globales del daemon */
void Agrega_Feromona();                   /* actualiza feromona de los arcos que pertenecen a la mejor hormiga */
int  Hormigas_CMAX( int Hormiga );        /* evalua las soluciones encontradas por las hormigas */
int  Siguiente_Explota( MAQUINA* Trabajo, int Disponible, HORMIGAS* Solucion  );
/* selecciona operacion en base a la funcion de transicion para explotar */
int  Siguiente_Explora( MAQUINA* Trabajo, int Disponible, HORMIGAS* Solucion  );
/* selecciona operacion en base a la funcion de transicion para explorar */
void Inicializa_RS(); /* inicializa paametros de RS */
void Metaheuristica_RS(); /* metaheuristica de RS */
int  RS_CMAX();             /* evalua las soluciones encontradas por la EV */
void Nueva_Solucion();      /* genera las soluciones para la EV */
void Regresa_Solucion();    /* regresa a solucion anterior si no mejora la nueva */
void Guarda_Solucion();     /* guarda la mejor solucion obtenida */
void AG_SCH_RS_Procesos( void );
/* funcion de los procesos esclavos que realizan SCH+RS - mutacion */
void Individuo_SCH( void );      /* pasa una solucion individuo a una de SCH */
void AG_SCH_RS_Maestro( void );         /* funcion del proceso maestro que controla el AG sobre el cluster */
void Genera_Poblacion_Inicial( void );  /* funcion que genera la poblacion inicial con los mejores soluciones de los procesos es
void Distribuye_Individuos( void );     /* envia los individuos de la poblacion a los procesos esclavos para que reinicien */
void Seleccion( void );                 /* seleccion por el metodo de la ruleta */
void Cruzamiento( void );               /* realiza el cruzamiento */
void Selecciona_Mejor();                /* selecciona el mejor Global de todo el proceso */
void Evaluacion( void );                /* realiza la evaluacion */
void Cruzamiento_PMX( int PadreA, int PadreB, int Etapa );  /* realiza el cruzamiento PBX*/
int Busca_Elemento_PMX( int Elemento, EV* Padre, EV* Hijo, int PuntoCruceA, int PuntoCruceB );     /* busca elementos dentro de
int IDPROCESO;
int PROCESOS;
int ESCLAVO;
int DESTINO;
int Bandera = 0;
MPI_Status Estado ;
```

```
int iGlobalRS = 999999;
int iGlobalSCH = 999999;
int iGlobal = 999999;
int iMejorRSLocal = 999999;
time_t TiempoInicial;
time_t TiempoFinal;
long int Semilla;
float To;
SISTEMA SCH;
int SDST[CFG_ETAPAS][CFG_TRABAJOS+1][CFG_TRABAJOS+1];
HORMIGAS SOLUCION[CFG_HORMIGAS];
int MejorArco[CFG_EVDATA + 1];
RS Recocido;
EV RSVecindario[ CFG_EVDATA ];
EV RSMejor[ CFG_EVDATA ], RSMejorLocal[ CFG_EVDATA ], RSGlobal[ CFG_EVDATA ];
INTERCAMBIO SW;
INDIVIDUO AGPoblacion[MAX_POBLACION];
INDIVIDUO AGSeleccion[MAX_POBLACION];
int AGPosSeleccion[MAX_POBLACION];
EV HijoA[CFG_EVDATA], HijoB[CFG_EVDATA];
int AGCRUCE = 0;
int Generacion;

/* configura numero de etapas */
void Configura_Etapas()
{
    int iEtapa;
    int iTrabajos = 0;
    for ( iEtapa = 0; iEtapa < ETAPAS; iEtapa++)
    {
        SCH.Etapa[iEtapa].iTInicial = iTrabajos;
        SCH.Etapa[iEtapa].k = iEtapa;
        if ( iEtapa == 0 || iEtapa == ETAPAS - 1 )      iTrabajos += 1;
        else iTrabajos += CFG_TRABAJOS;
        SCH.Etapa[iEtapa].iTFinal = iTrabajos - 1;
    }
}

/* configura numero de trabajos */
void Configura_Trabajos()
{
    int iEtapa, iTrabajo;
    int iMaquina = 0;
    for ( iEtapa = 0; iEtapa < ETAPAS; iEtapa++)
    {
        for ( iTrabajo = SCH.Etapa[iEtapa].iTInicial; iTrabajo <= SCH.Etapa[iEtapa].iTFinal; iTrabajo++)
        {
            SCH.Trabajo[iTrabajo].iMInicial = iMaquina;
            SCH.Trabajo[iTrabajo].k = iEtapa;
            SCH.Trabajo[iTrabajo].j = iTrabajo;
            if ( iEtapa == 0 || iEtapa == ETAPAS - 1 )      SCH.Trabajo[iTrabajo].ID = 1;
            else SCH.Trabajo[iTrabajo].ID = ((iTrabajo - 1)% CFG_TRABAJOS) + 1;
            if ( iEtapa == 0 || iEtapa == ETAPAS - 1 )      iMaquina += 1;
            else iMaquina += CFG_MAQUINAS;
            SCH.Trabajo[iTrabajo].iMFinal = iMaquina - 1;
        }
    }
}

/* configura numero de maquinas */
void Configura_Maquinas()
{
    int iEtapa, iTrabajo, iMaquina;
    int iArco = 0;
    int iNumeroArcos;
    for ( iEtapa = 0; iEtapa < ETAPAS; iEtapa++)
    {
```

```
        for ( iTrabajo = SCH.Etapa[iEtapa].iTInicial; iTrabajo <= SCH.Etapa[iEtapa].iTFinal; iTrabajo++)
        {
            for ( iMaquina = SCH.Trabajo[iTrabajo].iMInicial; iMaquina <= SCH.Trabajo[iTrabajo].iMFinal; iMaquina++)
            {
                SCH.Maquina[iMaquina].iAInicial = iArco;
                SCH.Maquina[iMaquina].k = iEtapa;
                SCH.Maquina[iMaquina].i = iMaquina;
                SCH.Maquina[iMaquina].j = iTrabajo;
                if ( iEtapa == 0 || iEtapa == ETAPAS - 1 )  SCH.Maquina[iMaquina].ID = 1;
                else SCH.Maquina[iMaquina].ID = ((iMaquina - 1)% CFG_MAQUINAS) + 1;
                iNumeroArcos = ARCOS_MAQUINA;
                if ( iEtapa == 0 )         iNumeroArcos = ARCOS_N;
                if ( iEtapa == ETAPAS - 2 ) iNumeroArcos = ARCOS_MAQUINA_F;
                if ( iEtapa == ETAPAS - 1 ) iNumeroArcos = 1;
                iArco += iNumeroArcos;
                SCH.Maquina[iMaquina].iAFinal = iArco - 1;
            }
        }
    }
}

/* configura numero de arcos */
void Configura_Arcos()
{
    int iEtapa, iTrabajo, iMaquina, iArco;
    int iMaquinaDestino;
    for ( iEtapa = 0; iEtapa < ETAPAS; iEtapa++)
    {
        for ( iTrabajo = SCH.Etapa[iEtapa].iTInicial; iTrabajo <= SCH.Etapa[iEtapa].iTFinal; iTrabajo++)
        {
            for ( iMaquina = SCH.Trabajo[iTrabajo].iMInicial; iMaquina <= SCH.Trabajo[iTrabajo].iMFinal; iMaquina++)
            {
                iMaquinaDestino = SCH.Trabajo[SCH.Etapa[iEtapa].iTInicial].iMInicial;
                for ( iArco = SCH.Maquina[iMaquina].iAInicial; iArco <= SCH.Maquina[iMaquina].iAFinal; iArco++)
                {
                    SCH.Arco[iArco].ID = (iArco - SCH.Maquina[iMaquina].iAInicial) + 1;
                    SCH.Arco[iArco].Origenk = iEtapa;
                    SCH.Arco[iArco].Origeni = iMaquina;
                    SCH.Arco[iArco].Origenj = iTrabajo;
                    SCH.Arco[iArco].l       = iArco;
                    if ( iEtapa != ETAPAS - 1 )
                        while ( SCH.Maquina[iMaquina].j == SCH.Maquina[iMaquinaDestino].j) iMaquinaDestino++;
                    SCH.Arco[iArco].Destinok = SCH.Maquina[iMaquinaDestino].k;
                    SCH.Arco[iArco].Destinoi = iMaquinaDestino;
                    SCH.Arco[iArco].Destinoj = SCH.Maquina[iMaquinaDestino].j;
                    iMaquinaDestino++;
                }
            }
        }
    }
}

/* agregar feromona inicial To */
void Agregar_To( void )
{
    int iEtapa, iTrabajo, iMaquina, iArco, K, L;
    To = 1.0  / ( CFG_UB * CFG_TRABAJOS*CFG_ETAPAS*CFG_MAQUINAS);
    for ( iEtapa = 0; iEtapa < ETAPAS; iEtapa++ )
    {
        for ( iTrabajo = SCH.Etapa[iEtapa].iTInicial; iTrabajo <= SCH.Etapa[iEtapa].iTFinal; iTrabajo++ )
        {
            for ( iMaquina = SCH.Trabajo[iTrabajo].iMInicial; iMaquina <= SCH.Trabajo[iTrabajo].iMFinal; iMaquina++ )
            {
                for ( iArco = SCH.Maquina[iMaquina].iAInicial; iArco <= SCH.Maquina[iMaquina].iAFinal; iArco++ )
                {
                    SCH.Arco[iArco].Costo = SCH.Arco[iArco].Feromona = SCH.Arco[iArco].ProductoAB = 0.0;
```

```
                    if ( SCH.Arco[iArco].Destinok < ETAPAS-1 )
                    {
                        SCH.Arco[iArco].Feromona = To;
                        SCH.Arco[iArco].ProductoAB = 0.0;
                        K = SCH.Etapa[ SCH.Arco[iArco].Destinok ].k - 1;
                        L = ( SCH.Etapa[ SCH.Arco[iArco].Origenk ].k != SCH.Etapa[ SCH.Arco[iArco].Destinok ].k) ? 0 :
                                                                    SCH.Trabajo[ SCH.Arco[iArco].Origenj ].ID;
                        SCH.Arco[iArco].Costo = 1.0 / ( SDST[K][ L ][ SCH.Trabajo[ SCH.Arco[iArco].Destinoj].ID] +
                                                SCH.Maquina[SCH.Arco[iArco].Destinoi].Costo);
                    }
                }
            }
        }
    }
}

/* selecciona operacion en base a la funcion de transicion para explotar */
int Siguiente_Explota( MAQUINA* Maquina, int iDisponible, HORMIGAS* Solucion )
{
    int     iArcoF = Maquina->iAInicial + (CFG_TRABAJOS * CFG_MAQUINAS) - ( CFG_MAQUINAS + 1);
    int     iArcoI = Maquina->iAInicial;
    int     Indice;          /* indice de busqueda */
    int     iMayor = 0;   /* indice del mayor producto A.B */
    float   Mayor = 0.0; /* Mayor valor del producto A.B */
    if ( iDisponible == 0 )
    {
        if ( Maquina->k == CFG_ETAPAS ) return ( Maquina->iAFinal );
        else if ( Maquina->k == 0 )
        {
            iArcoI = Maquina->iAInicial;
            iArcoF = Maquina->iAFinal;
        }
        else
        {
            iArcoI = iArcoF + 1;
            iArcoF = Maquina->iAFinal;
        }
    }
    for ( Indice = iArcoI; Indice <= iArcoF; Indice++)
    if ( Solucion->Disponible[SCH.Arco[Indice].Destinoj] != 1 )
    {
        SCH.Arco[Indice].ProductoAB =   SCH.Arco[Indice].Feromona * pow(SCH.Arco[Indice].Costo,CFG_BETA);
        if ( SCH.Arco[Indice].ProductoAB > Mayor )
        {
         Mayor   = SCH.Arco[Indice].ProductoAB;
            iMayor  = Indice;
        }
    }
    return ( iMayor );
}

/* selecciona operacion en base a la funcion de transicion para explorar */
int Siguiente_Explora( MAQUINA* Maquina, int iDisponible, HORMIGAS* Solucion )
{
    int     iArcoF = Maquina->iAInicial + (CFG_TRABAJOS * CFG_MAQUINAS) - ( CFG_MAQUINAS + 1);
    int     iArcoI = Maquina->iAInicial;
    int     Indice;                         /* indice de busqueda */
    float   Sumatoria;                    /* sumatoria de los productos */
    float   Aleatorio = Random01(); /* aleatorio entre 0 y 1 */
    float   Acumulado;                  /* acumulado de las probabilidades */
    int     iUltimo;                        /* ultimo indice libre disponible */
    if ( iDisponible == 0 )
    {
        if ( Maquina->k == CFG_ETAPAS ) return ( Maquina->iAFinal );
        else if ( Maquina->k == 0 )
        {
```

```
            iArcoI = Maquina->iAInicial;
            iArcoF = Maquina->iAFinal;
        }
        else
        {
            iArcoI = iArcoF + 1;
            iArcoF = Maquina->iAFinal;
        }
    }
    Sumatoria = Acumulado = 0.0;
    for ( Indice = iArcoI; Indice <= iArcoF; Indice++)
    if ( Solucion->Disponible[SCH.Arco[Indice].Destinoj] != 1 )
    {
        SCH.Arco[Indice].ProductoAB = SCH.Arco[Indice].Feromona*
                                      pow(SCH.Arco[Indice].Costo,CFG_BETA);
        Sumatoria += SCH.Arco[Indice].ProductoAB;
    }
    iUltimo = 0;
    for ( Indice = iArcoI; Indice <= iArcoF; Indice++)
    if ( Solucion->Disponible[SCH.Arco[Indice].Destinoj] != 1 )
    {
        Acumulado += ( SCH.Arco[Indice].ProductoAB / Sumatoria );
        iUltimo = Indice;
        if ( Acumulado >= Aleatorio ) return ( Indice );
    }
    if ( iUltimo > 0 ) return (iUltimo);
    printf("ERROR: los calculos andan mal%.32f,%.32f,%.32f\n",Acumulado,Aleatorio,Sumatoria);
    exit(1);
}

/* construye la solucion para la hormiga k */
void Construye_Solucion ( int iHormiga )
{
    MAQUINA*  Maquina = &SCH.Maquina[0];     /* primera maquina del grafo que es el nodo Dummy(N) */
    HORMIGAS* Solucion =&SOLUCION[iHormiga]; /* solucion de la hormiga k  */
    ARCO* Arco;                              /* arco seleccionado */
    int iIndiceArco = 0;                     /* indice del Arco */
    int iDisponible = 0;                     /* Arcos disponibles disjuntos, en la opeacion Dummy(N) no hay */
    int iSiguiente;                          /* indice del siguiente arco */
    int iAccion = (Random01() > CFG_Qo ? EXPLORA : EXPLOTA);
    Solucion->Disponible[0] = 1;
    do
    {
        if ( iAccion == EXPLOTA )
            iSiguiente = Siguiente_Explota( Maquina, iDisponible, &SOLUCION[iHormiga] );
        else
            iSiguiente = Siguiente_Explora( Maquina, iDisponible, &SOLUCION[iHormiga] );
        Arco = &SCH.Arco[iSiguiente];
        Arco->Feromona = ( (1.0 - CFG_FEROMONAL) * Arco->Feromona ) +  (CFG_FEROMONAL * To);
        Maquina = &SCH.Maquina[Arco->Destinoi];
        Solucion->Arco[iIndiceArco++] = Arco->l;
        Solucion->Disponible[Arco->Destinoj] = 1;
        if ( iDisponible == 0 ) iDisponible = CFG_TRABAJOS - 1;
        else                    iDisponible--;
    } while ( Maquina->k < ETAPAS - 1 );
}

/* acciones globales del daemon */
void Acciones_Daemon( void )
{
    int Indice = -1;
    int iMakespan = 0;
    int iHormiga, iMejor;
    for ( iHormiga = 0; iHormiga < CFG_HORMIGAS; iHormiga++)
    {
        iMakespan = SOLUCION[iHormiga].Makespan;
```

```
        if ( iGlobalSCH > iMakespan )
        {
            iGlobalSCH = iMakespan;
            Indice = iHormiga;
        }
    }
    if ( Indice != -1 )
    {
        for ( iMejor = 0; iMejor < CFG_EVDATA + 1; iMejor++ )
        MejorArco[iMejor] = SOLUCION[Indice].Arco[iMejor];
    }
      Agrega_Feromona();
}

/* actualiza feromona de los arcos que pertenecen a la mejor hormiga */
void Agrega_Feromona( void )
{
    int iHormiga, iArcoGlobal, iArcoHormiga;
    for ( iArcoGlobal = 0; iArcoGlobal < CFG_EVDATA; iArcoGlobal++ )
    {
        for ( iArcoHormiga = 0; iArcoHormiga < CFG_EVDATA; iArcoHormiga++ )
        {
            for ( iHormiga = 0; iHormiga < CFG_HORMIGAS; iHormiga++)
            {
                ARCO* Arco = &SCH.Arco[ SOLUCION[iHormiga].Arco[iArcoHormiga] ];
                if ( MejorArco[iArcoGlobal] == SOLUCION[iHormiga].Arco[iArcoHormiga] )
                    Arco->Feromona = ( (1.0 - CFG_FEROMONAG) * Arco->Feromona ) +  ( CFG_FEROMONAG * (1.0/iGlobalSCH) );
            }
        }
    }
}

/* evalua las soluciones encontradas por las hormigas */
int Hormigas_CMAX( int iHormiga )
{
    const int TOTAL_MAQUINAS = CFG_MAQUINAS * CFG_TRABAJOS * CFG_ETAPAS;
    int iArco, iMaquina, iTrabajo, iPij, iMakespan, iEtapa, Skij;
    ARCO* Arco = NULL;
    int Tiempo_Total[TOTAL_MAQUINAS];
    int Tiempo_Trabajo[CFG_TRABAJOS];
    int Anterior[CFG_ETAPAS];
    for ( iMaquina = 0; iMaquina < TOTAL_MAQUINAS; iMaquina++) Tiempo_Total[iMaquina] = 0;
    for ( iTrabajo = 0; iTrabajo < CFG_TRABAJOS; iTrabajo++) Tiempo_Trabajo[iTrabajo] = 0;
    for ( iEtapa = 0; iEtapa < CFG_ETAPAS; iEtapa++) Anterior[iEtapa] = 0;
    for ( iArco = 0; iArco < CFG_EVDATA; iArco++ )
    {
        iEtapa = ( iArco >= CFG_TRABAJOS ) ? iArco/CFG_TRABAJOS : 0;
        Arco = &SCH.Arco[ SOLUCION[iHormiga].Arco[iArco] ];
        iMaquina = ((SCH.Maquina[ Arco->Destinoi ].k - 1) * CFG_MAQUINAS) + (SCH.Maquina[ Arco->Destinoi ].ID - 1);
        iTrabajo = SCH.Trabajo[ Arco->Destinoj ].ID - 1;
        iPij     = SCH.Maquina[ Arco->Destinoi ].Costo;
        Skij = SDST[iEtapa][Anterior[iEtapa]][iTrabajo+1];
        Anterior[iEtapa] = iTrabajo+1;
        if ( Tiempo_Total[iMaquina] < Tiempo_Trabajo[iTrabajo] ) Tiempo_Total[iMaquina] = Tiempo_Trabajo[iTrabajo];
        Tiempo_Total[ iMaquina ] += (Skij + iPij);
        Tiempo_Trabajo[ iTrabajo ] = Tiempo_Total[ iMaquina ];
    }
    iMakespan = 0;
    for ( iMaquina =0; iMaquina < TOTAL_MAQUINAS; iMaquina++)
     if ( Tiempo_Total[iMaquina] > iMakespan ) iMakespan = Tiempo_Total[iMaquina];
    SOLUCION[ iHormiga ].Makespan = iMakespan;
    return ( iMakespan );
}

/* leer parametros de SCH */
int Agregar_Tiempos(char* aFuente)
{
```

```
    FILE *File;  /* apuntador a archivo de entrada */
    int iEtapa, iTrabajo, iMaquina, K, I, J;  /* indices */
    int IndiceT = 0;       /* indice calculado para los trabajos */
    char Linea[1000];   /* buffer para lectura de archivo de entrada */
    int Skij,Pij;
    if ((File = fopen( aFuente, "r" )) == NULL )
    {
        printf("Error el archivo :%s no pudo ser abierto\n",aFuente);
        return ( 0 );
    }
    fgets(Linea, sizeof(Linea),File);
    fgets(Linea, sizeof(Linea),File);
    fgets(Linea, sizeof(Linea),File);
    fscanf(File, "%d%d%d", &iTrabajo,&iEtapa,&iMaquina);
    for ( iTrabajo = 0; iTrabajo < CFG_TRABAJOS; iTrabajo++)
    {
        for ( iEtapa = 1; iEtapa < ETAPAS - 1; iEtapa++)
        {
            fscanf(File,"%d",&Pij);
            IndiceT = SCH.Etapa[iEtapa].iTInicial + iTrabajo;
            for ( iMaquina = SCH.Trabajo[IndiceT].iMInicial; iMaquina <= SCH.Trabajo[IndiceT].iMFinal; iMaquina++)
                SCH.Maquina[iMaquina].Costo = Pij;
        }
    }
    fgets(Linea, sizeof(Linea),File);
    for ( K = 0; K < CFG_ETAPAS; K++)
    {
        fgets(Linea, sizeof(Linea),File);
        for ( I = 0; I <= CFG_TRABAJOS; I++ )
        {
            for ( J = 0;  J <= CFG_TRABAJOS; J++ )
            {
                if ( J == 0 ) SDST[K][I][J] = 0;
                else
                {
                    fscanf(File,"%d",&Skij);
                    if ( J == I && J > 0 )
                    {
                        SDST[K][I][J] = 0;
                        SDST[K][I+1][J] = Skij;
                    }
                    else
                    if ( I > J )    SDST[K][I+1][J] = Skij;
                    else           SDST[K][I][J]   = Skij;
                }
            }
        }
    }
    fclose(File);
    return ( 1 );
}

/* optimizacion por sistema de hormigas - Ant Colony System */
void Metaheuristica_SCH( void )
{
    int iCriterio_Paro;
    int iHormiga;
    for ( iCriterio_Paro = 0; iCriterio_Paro < CFG_CRITERIO; iCriterio_Paro++ )
    {
        for ( iHormiga = 0; iHormiga < CFG_HORMIGAS; iHormiga++ )
        {
            Construye_Solucion( iHormiga );
            Hormigas_CMAX( iHormiga );
         }
        Acciones_Daemon();
        Inicializa_Soluciones();
```

```
    }
}

/* limpia variables de las hormigas */
void Inicializa_Soluciones()
{
    int iHormiga, iDisponible;
    for ( iHormiga = 0; iHormiga < CFG_HORMIGAS; iHormiga++ )
    {
        for ( iDisponible = 0; iDisponible < CFG_EVDATA + 1; iDisponible++)
        SOLUCION[iHormiga].Disponible[iDisponible] = SOLUCION[iHormiga].Arco[iDisponible] = -1;
        SOLUCION[iHormiga].Makespan = 0;
    }
}

/* inicializa grafo, arcos, distancias y feromonas */
void Inicializa_SCH( void )
{
    Configura_Etapas();          /* prepara arreglo de etapas */
    Configura_Trabajos();       /* prepara arreglo de trabajos */
    Configura_Maquinas();     /* prepara arreglo de maquinas */
    Configura_Arcos();          /* enlaza arcos */
    Inicializa_Soluciones();    /* inicializa estructura de soluciones */
    if ( !Agregar_Tiempos("/home/fjuarez/pruebas/FFS_STSD_140x4x4x25_1")) exit(1);   /* lee Pij y actualiza grafo */
    Agregar_To(); /* agrega la cantidad de feromona inicial */
}

/* inicializa parametros de Recocido Simulado */
void Inicializa_RS()
{
    ARCO* Arco;
    int iTrabajo, iMaquina, iArco;
    Recocido.Temperatura    = CFG_TEMPERATURA;
    Recocido.Factor         = CFG_FACTOR;
    Recocido.Frozen         = CFG_FROZEN;
    Recocido.Ciclos         = CFG_CICLOS;
    for ( iArco = 0; iArco < CFG_EVDATA; iArco++ )
    {
        Arco = &SCH.Arco[ MejorArco[iArco] ];
        iMaquina = SCH.Maquina[ Arco->Destinoi ].ID;
        iTrabajo = SCH.Trabajo[ Arco->Destinoj ].ID;
        RSVecindario[iArco].iMaquina = iMaquina;
        RSVecindario[iArco].iTrabajo = iTrabajo;
        RSMejor[iArco].iMaquina = iMaquina;
        RSMejor[iArco].iTrabajo = iTrabajo;
    }
}

/* evalua las soluciones encontradas por EV */
int RS_CMAX( void )
{
    const int TOTAL_MAQUINAS = CFG_MAQUINAS * CFG_TRABAJOS * CFG_ETAPAS;
    int Indice, iEtapa, iMaquina, iTrabajo, iPij, Skij, iMakespan, iPosicion;
    int Tiempo_Total[TOTAL_MAQUINAS];
    int Tiempo_Trabajo[CFG_TRABAJOS];
    int Anterior[CFG_ETAPAS];
    for ( iMaquina = 0; iMaquina < TOTAL_MAQUINAS; iMaquina++) Tiempo_Total[iMaquina] = 0;
    for ( iTrabajo = 0; iTrabajo < CFG_TRABAJOS; iTrabajo++) Tiempo_Trabajo[iTrabajo] = 0;
    for ( iEtapa = 0; iEtapa < CFG_ETAPAS; iEtapa++) Anterior[iEtapa] = 0;
    for ( Indice = 0; Indice < CFG_EVDATA; Indice++ )
    {
        iEtapa = ( Indice >= CFG_TRABAJOS ) ? Indice/CFG_TRABAJOS : 0;
        iTrabajo = RSVecindario[Indice].iTrabajo - 1;
        iMaquina = ( iEtapa * CFG_MAQUINAS ) + RSVecindario[Indice].iMaquina - 1;
        iPosicion = ( iEtapa * CFG_TRABAJOS * CFG_MAQUINAS ) +
                    ( iTrabajo * CFG_MAQUINAS ) +
                    ( RSVecindario[Indice].iMaquina );
```

```
        iPij = SCH.Maquina[ iPosicion ].Costo;
        Skij = SDST[iEtapa][Anterior[iEtapa]][iTrabajo+1];
        Anterior[iEtapa] = iTrabajo+1;
        if ( Tiempo_Total[iMaquina] < Tiempo_Trabajo[iTrabajo] ) Tiempo_Total[iMaquina] = Tiempo_Trabajo[iTrabajo];
        Tiempo_Total[ iMaquina ] += (Skij + iPij);
        Tiempo_Trabajo[ iTrabajo ] = Tiempo_Total[ iMaquina ];
    }
    iMakespan = 0;
    for ( iMaquina =0; iMaquina < TOTAL_MAQUINAS; iMaquina++)
     if ( Tiempo_Total[iMaquina] > iMakespan ) iMakespan = Tiempo_Total[iMaquina];
    return ( iMakespan );
}

/* genera las soluciones para la EV */
void Nueva_Solucion( void )
{
    int iTrabajo;
    int Etapa = rand()% CFG_ETAPAS;
    int Trabajo1, Trabajo2;
    int Maquina1, Maquina2;
    Trabajo1 = Trabajo2 = rand()% CFG_TRABAJOS;
    while ( Trabajo1 == Trabajo2 ) Trabajo2 = rand()% CFG_TRABAJOS;
    Maquina1 = Maquina2 = ( rand()% CFG_MAQUINAS ) + 1;
    while ( Maquina1 == Maquina2 ) Maquina2 = ( rand()% CFG_MAQUINAS ) + 1;
    SW.Trabajo1 = Etapa * CFG_TRABAJOS + Trabajo1;
    SW.Trabajo2 = Etapa * CFG_TRABAJOS + Trabajo2;
    SW.Maquina1 = RSVecindario[ SW.Trabajo1 ].iMaquina;
    SW.Maquina2 = RSVecindario[ SW.Trabajo2 ].iMaquina;
    iTrabajo = RSVecindario[ SW.Trabajo1 ].iTrabajo;
    RSVecindario[ SW.Trabajo1 ].iTrabajo = RSVecindario[ SW.Trabajo2 ].iTrabajo;
    RSVecindario[ SW.Trabajo1 ].iMaquina = Maquina1;
    RSVecindario[ SW.Trabajo2 ].iTrabajo = iTrabajo;
    RSVecindario[ SW.Trabajo2 ].iMaquina = Maquina2;
}

/* regresa a solucion anterior si no mejora la nueva */
void Regresa_Solucion( void )
{
    int iTrabajo;
    iTrabajo = RSVecindario[ SW.Trabajo1 ].iTrabajo;
    RSVecindario[ SW.Trabajo1 ].iTrabajo = RSVecindario[ SW.Trabajo2 ].iTrabajo;
    RSVecindario[ SW.Trabajo1 ].iMaquina = SW.Maquina1;
    RSVecindario[ SW.Trabajo2 ].iTrabajo = iTrabajo;
    RSVecindario[ SW.Trabajo2 ].iMaquina = SW.Maquina2;
}

/* guarda los cambios que resultan en la mejor solucion obtenida por RS */
void Guarda_Solucion()
{
    RSMejor[ SW.Trabajo1 ].iTrabajo = RSVecindario[ SW.Trabajo1 ].iTrabajo;
    RSMejor[ SW.Trabajo2 ].iTrabajo = RSVecindario[ SW.Trabajo2 ].iTrabajo;
    RSMejor[ SW.Trabajo1 ].iMaquina = RSVecindario[ SW.Trabajo1 ].iMaquina;
    RSMejor[ SW.Trabajo2 ].iMaquina = RSVecindario[ SW.Trabajo2 ].iMaquina;
}

/* guarda la mejor solucion global a todos los RS */
void Guarda_Solucion_Local()
{
    int iMejor;
    for ( iMejor=0; iMejor < CFG_EVDATA; iMejor++)
    {
        RSMejorLocal[ iMejor ].iTrabajo = RSMejor[ iMejor ].iTrabajo;
        RSMejorLocal[ iMejor ].iMaquina = RSMejor[ iMejor ].iMaquina;
    }
}

/* guarda la mejor solucion global de RS */
```

```
void Guarda_Solucion_Global()
{
    int iMejor;
    for ( iMejor=0; iMejor < CFG_EVDATA; iMejor++)
    {
        RSGlobal[ iMejor ].iTrabajo = RSMejorLocal[ iMejor ].iTrabajo;
        RSGlobal[ iMejor ].iMaquina = RSMejorLocal[ iMejor ].iMaquina;
    }
}

/* aplica metaheuristica de RS */
void Metaheuristica_RS()
{
    int Ciclos, iMakespan;
    int iMejorRS = 999999;
    Inicializa_RS();
    iMejorRSLocal = iGlobalSCH;
    Guarda_Solucion_Local();
    if ( iGlobalRS > iGlobalSCH )
    {
        iGlobalRS = iGlobalSCH;
        Guarda_Solucion_Global();
    }
    do
    {
        for ( Ciclos = 0; Ciclos < Recocido.Ciclos; Ciclos++)
        {
            Nueva_Solucion();
            iMakespan = RS_CMAX();
            if ( iMejorRS > iMakespan )
            {
                iMejorRS = iMakespan;
                Guarda_Solucion();
                if ( iMejorRSLocal > iMejorRS )
                {
                 iMejorRSLocal = iMejorRS;
                    Guarda_Solucion_Local();
                }
            }
            else
            {
                double iDiferencia = iMakespan - iMejorRS;
                double Prob =  exp((-1)*(iDiferencia/Recocido.Temperatura));
                if ( Prob > Rand(0,2))
                {
                    iMejorRS = iMakespan;
                    Guarda_Solucion();
                }
                else
                Regresa_Solucion();
             }
        }
    Recocido.Temperatura = Recocido.Factor * Recocido.Temperatura;
    } while ( Recocido.Temperatura > Recocido.Frozen);
    if ( iGlobalRS > iMejorRSLocal )
    {
        iGlobalRS = iMejorRSLocal;
        Guarda_Solucion_Global();
    }
}

/* evalua las soluciones encontradas por EV */
int AG_CMAX( int Individuo, int Fuente )
{
    const int TOTAL_MAQUINAS = CFG_MAQUINAS * CFG_TRABAJOS * CFG_ETAPAS;
    int Indice, iEtapa, iMaquina, iTrabajo, iPij, Skij, iMakespan, iPosicion;
```

```
    int Tiempo_Total[TOTAL_MAQUINAS];
    int Tiempo_Trabajo[CFG_TRABAJOS];
    int Anterior[CFG_ETAPAS];
    for ( iMaquina = 0; iMaquina < TOTAL_MAQUINAS; iMaquina++) Tiempo_Total[iMaquina] = 0;
    for ( iTrabajo = 0; iTrabajo < CFG_TRABAJOS; iTrabajo++) Tiempo_Trabajo[iTrabajo] = 0;
    for ( iEtapa = 0; iEtapa < CFG_ETAPAS; iEtapa++) Anterior[iEtapa] = 0;
    for ( Indice = 0; Indice < CFG_EVDATA; Indice++ )
    {
        iEtapa = ( Indice >= CFG_TRABAJOS ) ? Indice/CFG_TRABAJOS : 0;
        if ( Fuente == 1 ) iTrabajo = AGPoblacion[Individuo].Individuo[Indice].iTrabajo - 1;
        else               iTrabajo = AGSeleccion[Individuo].Individuo[Indice].iTrabajo - 1;
        if ( Fuente == 1 ) iMaquina = ( iEtapa * CFG_MAQUINAS ) + AGPoblacion[Individuo].Individuo[Indice].iMaquina - 1;
        else                iMaquina = ( iEtapa * CFG_MAQUINAS ) + AGSeleccion[Individuo].Individuo[Indice].iMaquina - 1;
        if ( Fuente == 1 )
            iPosicion = ( iEtapa * CFG_TRABAJOS * CFG_MAQUINAS ) +
                        ( iTrabajo * CFG_MAQUINAS ) +
                        ( AGPoblacion[Individuo].Individuo[Indice].iMaquina );
        else
            iPosicion = ( iEtapa * CFG_TRABAJOS * CFG_MAQUINAS ) +
                        ( iTrabajo * CFG_MAQUINAS ) +
                        ( AGSeleccion[Individuo].Individuo[Indice].iMaquina );
        iPij = SCH.Maquina[ iPosicion ].Costo;
        Skij = SDST[iEtapa][Anterior[iEtapa]][iTrabajo+1];
        Anterior[iEtapa] = iTrabajo+1;
        if ( Tiempo_Total[iMaquina] < Tiempo_Trabajo[iTrabajo] ) Tiempo_Total[iMaquina] = Tiempo_Trabajo[iTrabajo];
        Tiempo_Total[ iMaquina ] += (Skij + iPij);
        Tiempo_Trabajo[ iTrabajo ] = Tiempo_Total[ iMaquina ];
    }
    iMakespan = 0;
    for ( iMaquina =0; iMaquina < TOTAL_MAQUINAS; iMaquina++)
        if ( Tiempo_Total[iMaquina] > iMakespan ) iMakespan = Tiempo_Total[iMaquina];
    return ( iMakespan );
}

/* realiza la evaluacion */
void Evaluacion( void )
{
    int Individuo;
    for ( Individuo = 0; Individuo < CFG_POBLACION; Individuo++)
          AGSeleccion[Individuo].Adaptacion = AG_CMAX(Individuo,2);
}

/* selecciona el mejor Global de todo el proceso */
void Selecciona_Mejor()
{
    for (ESCLAVO = 1; ESCLAVO < PROCESOS; ESCLAVO++)
        if ( iGlobal > AGPoblacion[ESCLAVO - 1].Adaptacion ) iGlobal = AGPoblacion[ESCLAVO - 1].Adaptacion;
}

/* busca elementos dentro de los padres */
int Busca_Elemento_PMX( int Elemento, EV* Padre, EV* Hijo, int PuntoCruceA, int PuntoCruceB )
{
    int Indice;
    Indice = PuntoCruceA;
    while ( Indice <= PuntoCruceB )
    {
        if ( Padre[Elemento].iTrabajo == Hijo[Indice].iTrabajo ) return ( Indice );
        Indice++;
    }
    return ( -1 );
}

/* realiza el cruzamiento PBX*/
void Cruzamiento_PMX( int PadreA, int PadreB, int Etapa )
{
    int Inicio  = ( Etapa * CFG_TRABAJOS );
    int Fin     = ( Etapa * CFG_TRABAJOS ) + CFG_TRABAJOS;
```

```
    int PuntoCruceA = Inicio + Rand(0,CFG_TRABAJOS/2);
    int PuntoCruceB = PuntoCruceA + CFG_TRABAJOS/3;
    int Indice, Elemento, Nuevo_Elemento;
    for ( Indice = Inicio; Indice < Fin; Indice++ )
    {
        if ( Indice >= PuntoCruceA && Indice <= PuntoCruceB )
        {
            HijoA[Indice].iMaquina = AGSeleccion[PadreB].Individuo[Indice].iMaquina;
            HijoA[Indice].iTrabajo = AGSeleccion[PadreB].Individuo[Indice].iTrabajo;
            HijoB[Indice].iMaquina = AGSeleccion[PadreA].Individuo[Indice].iMaquina;
            HijoB[Indice].iTrabajo = AGSeleccion[PadreA].Individuo[Indice].iTrabajo;
        }
        else
            HijoA[Indice].iMaquina = HijoA[Indice].iTrabajo = HijoB[Indice].iMaquina = HijoB[Indice].iTrabajo = -1;
    }
    for ( Indice = Inicio; Indice < Fin; Indice++ )
    {
        if ( HijoA[Indice].iTrabajo == -1 )
        {
            Elemento = Nuevo_Elemento = Indice;
            while ( Nuevo_Elemento != -1 )
            {
                Nuevo_Elemento = Busca_Elemento_PMX( Elemento, AGSeleccion[PadreA].Individuo, HijoA, PuntoCruceA, PuntoCruceB );
                if ( Nuevo_Elemento != -1 ) Elemento = Nuevo_Elemento;
            }
            HijoA[Indice].iMaquina = AGSeleccion[PadreA].Individuo[Elemento].iMaquina;
            HijoA[Indice].iTrabajo = AGSeleccion[PadreA].Individuo[Elemento].iTrabajo;
            Elemento = Nuevo_Elemento = Indice;
            while ( Nuevo_Elemento != -1 )
            {
                Nuevo_Elemento = Busca_Elemento_PMX( Elemento, AGSeleccion[PadreB].Individuo, HijoB, PuntoCruceA, PuntoCruceB );
                if ( Nuevo_Elemento != -1) Elemento = Nuevo_Elemento;
            }
            HijoB[Indice].iMaquina = AGSeleccion[PadreB].Individuo[Elemento].iMaquina;
            HijoB[Indice].iTrabajo = AGSeleccion[PadreB].Individuo[Elemento].iTrabajo;
         }
    }
    for ( Indice = Inicio; Indice < Fin; Indice++ )
    {
        AGSeleccion[PadreA].Individuo[Indice].iMaquina = HijoA[Indice].iMaquina;
        AGSeleccion[PadreA].Individuo[Indice].iTrabajo = HijoA[Indice].iTrabajo;
        AGSeleccion[PadreB].Individuo[Indice].iMaquina = HijoB[Indice].iMaquina;
        AGSeleccion[PadreB].Individuo[Indice].iTrabajo = HijoB[Indice].iTrabajo;
    }
}

/* realiza el cruzamiento */
void Cruzamiento( void )
{
    int PadreA, PadreB;
    int Individuo, Etapa;
    /* realiza el cruce entre los elementos faltantes */
    for ( Individuo=0; Individuo < AGCRUCE; Individuo+=2)
    {
        PadreA = AGPosSeleccion[Individuo];
        PadreB = AGPosSeleccion[Individuo+1];
        for ( Etapa=0; Etapa < CFG_ETAPAS; Etapa++ )
         Cruzamiento_PMX( PadreA, PadreB, Etapa );
    }
}

/* seleccion por el metodo de la ruleta */
void Seleccion( void )
{
    int Individuo, Indice, Secuencia;
    float Aleatorio;
```

```
    AGCRUCE = 0;
    for ( Indice=0; Indice < CFG_POBLACION; Indice++ )
    {
        Aleatorio = Random01();
        Individuo = 0;
        while ( AGPoblacion[Individuo].Acumulado < Aleatorio && Individuo < CFG_POBLACION - 1) Individuo++;
        AGSeleccion[Indice].Numero = AGPoblacion[Individuo].Numero;
        AGSeleccion[Indice].Adaptacion = AGPoblacion[Individuo].Adaptacion;
        AGSeleccion[Indice].Acumulado = AGSeleccion[Individuo].Probabilidad = 0.0;
        for ( Secuencia=0; Secuencia < CFG_EVDATA; Secuencia++ )
        {
            AGSeleccion[Indice].Individuo[Secuencia].iMaquina = AGPoblacion[Individuo].Individuo[Secuencia].iMaquina;
            AGSeleccion[Indice].Individuo[Secuencia].iTrabajo = AGPoblacion[Individuo].Individuo[Secuencia].iTrabajo;
        }
        if ( Aleatorio <= CFG_TASACRUCE )
        {
            AGPosSeleccion[AGCRUCE] = Indice;
            AGCRUCE++;
        }
    }
    if ( AGCRUCE > 0)
    if ( (AGCRUCE% 2) == 1 ) AGCRUCE--;
}

/* envia los individuos de la poblacion a los procesos esclavos para que reinicien */
void Distribuye_Individuos( void )
{
    int ESCLAVO;
    for (ESCLAVO = 1; ESCLAVO < PROCESOS; ESCLAVO++)
    {
        MPI_Send( AGSeleccion[ESCLAVO - 1].Individuo,  2*CFG_EVDATA, MPI_INT, ESCLAVO, Bandera, MPI_COMM_WORLD);
    }
}

/* genera población inicial */
void Genera_Poblacion_Inicial( void )
{
    int Indice;
    float SumaAdaptacion = 0.0;
    float SumaProbabilidad = 0.0;
    for (ESCLAVO = 1; ESCLAVO < PROCESOS; ESCLAVO++)
    {
        AGPoblacion[ESCLAVO - 1].Numero = ESCLAVO;
        MPI_Recv( &AGPoblacion[ESCLAVO - 1].Individuo, 2*CFG_EVDATA, MPI_INT, ESCLAVO, Bandera ,MPI_COMM_WORLD, &Estado);
        MPI_Recv( &AGPoblacion[ESCLAVO - 1].Adaptacion, 1, MPI_DOUBLE, ESCLAVO, Bandera ,MPI_COMM_WORLD, &Estado);
        AGPoblacion[ESCLAVO - 1].Inversa = 1.0 / AGPoblacion[ESCLAVO - 1].Adaptacion;
        SumaAdaptacion += AGPoblacion[ESCLAVO - 1].Inversa;
    }
    for ( Indice=0; Indice < CFG_POBLACION; Indice++)
    {
        if ( Indice < (PROCESOS - 1) )
        {
            AGPoblacion[Indice].Probabilidad = AGPoblacion[Indice].Inversa / SumaAdaptacion;
            SumaProbabilidad += AGPoblacion[Indice].Probabilidad;
            AGPoblacion[Indice].Acumulado = SumaProbabilidad;
        }
        else
        {
            AGPoblacion[Indice].Adaptacion = 0;
            AGPoblacion[Indice].Acumulado = AGPoblacion[Indice].Probabilidad = 0.0;
        }
    }
}

/* funcion del proceso maestro que controla el AG sobre el cluster */
void AG_SCH_RS_Maestro( void )
{
```

```
    Inicializa_SCH();
    for ( Generacion=0; Generacion < CFG_GENERACIONES; Generacion++ )
    {
       MPI_Barrier(MPI_COMM_WORLD);
        Genera_Poblacion_Inicial();
        if ( Generacion < CFG_GENERACIONES-1 )
        {
            Seleccion();
            Cruzamiento();
            Evaluacion();
            Distribuye_Individuos();
        }
    }
    Selecciona_Mejor();
}

/* pasa una solucion individuo a una de SCH */
void Individuo_SCH( void )
{
    int iGen;
    /* arco inicial y final */
    int iArcoI = SCH.Maquina[0].iAInicial;
    int iArcoF = SCH.Maquina[0].iAFinal;
    int iArco, iArcoSel, iEtapa, iPosicion, iMaquina, iTrabajo;
   for ( iGen = 0; iGen < CFG_EVDATA; iGen++ )
    {
        iMaquina = RSMejor[iGen].iMaquina;
        iTrabajo = RSMejor[iGen].iTrabajo;
        iEtapa = ( iGen >= CFG_TRABAJOS ) ? iGen/CFG_TRABAJOS : 0;
        iArcoSel = 0;
        for ( iArco=iArcoI; iArco <= iArcoF; iArco++ )
        {
            if ( SCH.Maquina[ SCH.Arco[iArco].Destinoi ].ID == iMaquina &&
                 SCH.Trabajo[ SCH.Arco[iArco].Destinoj ].ID == iTrabajo &&
                 SCH.Arco[iArco].Destinok == iEtapa + 1)
            {
                iArcoSel = iArco;
                iArco = iArcoF;
            }
        }
        iPosicion = ( iEtapa * CFG_TRABAJOS * CFG_MAQUINAS ) +
                    ( (iTrabajo - 1) * CFG_MAQUINAS ) +
                    ( iMaquina );
        SOLUCION[0].Arco[iGen] = MejorArco[iGen] = iArcoSel;
        iArcoI = SCH.Maquina[ SCH.Arco[iArcoSel].Destinoi ].iAInicial; // SCH.Maquina[iPosicion].iAInicial;
        iArcoF = SCH.Maquina[ SCH.Arco[iArcoSel].Destinoi ].iAFinal; // SCH.Maquina[iPosicion].iAFinal;
    }
    SOLUCION[0].Arco[iGen] = MejorArco[iGen] = SCH.Maquina[iPosicion].iAFinal;
    iGlobalSCH = Hormigas_CMAX(0);
}

/* funcion de los procesos esclavos que realizan SCH+RS - mutacion */
void AG_SCH_RS_Procesos( void )
{
    Inicializa_SCH();
    /* Contenedor principal es el algoritmo Genetico */
    for ( Generacion=0; Generacion < CFG_GENERACIONES; Generacion++ )
    {
        Metaheuristica_SCH();
        Metaheuristica_RS();
        MPI_Barrier(MPI_COMM_WORLD);
        if ( Generacion < CFG_GENERACIONES-1 )
        {
            MPI_Send(RSMejorLocal, 2*CFG_EVDATA, MPI_INT, MAESTRO, Bandera, MPI_COMM_WORLD);
            MPI_Send(&iMejorRSLocal, 1, MPI_INT, MAESTRO, Bandera, MPI_COMM_WORLD);
            MPI_Recv( RSMejor, 2*CFG_EVDATA, MPI_INT, MAESTRO, Bandera ,MPI_COMM_WORLD, &Estado);
```

```
            Agregar_To();
            Inicializa_Soluciones();
            Individuo_SCH();
            Agrega_Feromona();
        }
        MPI_Send(RSGlobal, 2*CFG_EVDATA, MPI_INT, MAESTRO, Bandera, MPI_COMM_WORLD);
        MPI_Send(&iGlobalRS, 1, MPI_INT, MAESTRO, Bandera, MPI_COMM_WORLD);
    }
}

/* función principal */
int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &IDPROCESO);
    MPI_Comm_size(MPI_COMM_WORLD, &PROCESOS);
    if ( IDPROCESO == 0) Semilla = (time(NULL));
    MPI_Bcast(&Semilla,1,MPI_LONG,0,MPI_COMM_WORLD);
    Semilla = (Semilla*(IDPROCESO+1));
    srand48(Semilla);
    srand(Semilla);
    time( &TiempoInicial );
    CFG_GENERACIONES= atoi(argv[2]);  /* numero de generaciones */
    CFG_TASACRUCE   = atof(argv[3]);  /* tasa de cruce fraccion */
    CFG_POBLACION   = atoi(argv[4]);  /* tamaño de la poblacion */
    if ( IDPROCESO == MAESTRO ) AG_SCH_RS_Maestro();
    else AG_SCH_RS_Procesos();
    time ( &TiempoFinal );
    MPI_Finalize();
    if ( IDPROCESO == MAESTRO )
        printf("\t%s. en%.2lf Segundos semilla%ld Makespan=%d\n",argv[1],difftime (TiempoFinal,TiempoInicial),Semilla,iGlobal);
    return EXIT_SUCCESS;
}
```

Apéndice K

Código fuente mpi_ping.c

```
/*
Centro de Investigaciones en Ingenieria y Ciencias Aplicadas
Este programa mide las tasas de transferencia entre dos nodos
usando MPI, fuente: http://www.scl.ameslab.gov/Projects/mpi_introduction/para_pingpong.html
se efectuaron correciones a calculos para lo siguiente:
    - se cambio el factor size*8 debido a que el tipo de dato enviado es MPI_DOUBLE
    - se cambio al calculo del ancho de banda a ((1.e6/time)*(2.0*size*8))/(1024*1024)
donde anteriormente se calculaba como size/time para corregir el factor a Mb/s
    - se adecuarón los formatos de salidas.
fjuarez, 2012
www.gridmorelos.uaem.mx:8080
*/
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define sizebuff 264000*4
#define sizebits 2097152*4
int main (int argc, char **argv)
{
   int myproc, size, other_proc, nprocs, i, last;
   double t0, t1, time;
   double *a, *b;
   double max_rate = 0.0, min_latency = 10e6;
   char hostname[40];
   MPI_Request request, request_a, request_b;
   MPI_Status status;
#if defined (_CRAYT3E)
   a = (double *) shmalloc (sizebufff (double));
   b = (double *) shmalloc (sizebuff * sizeof (double));
#else
   a = (double *) malloc (sizebuff * sizeof (double));
   b = (double *) malloc (sizebuff * sizeof (double));
#endif
   for (i = 0; i < 132000; i++) {
      a[i] = (double) i;
      b[i] = 0.0;
   }
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myproc);
   if (nprocs != 2) exit (1);
   other_proc = (myproc + 1)% 2;
   gethostname(hostname, 40);
   printf("Soy en proceso %d of%d corriendo en :%s\n", myproc, nprocs,hostname);
   MPI_Barrier(MPI_COMM_WORLD);
   /* Timer accuracy test */
   t0 = MPI_Wtime();
   t1 = MPI_Wtime();
   while (t1 == t0) t1 = MPI_Wtime();
   if (myproc == 0)
```

```
      printf("Tiempo más pequeño posible medible ~%f usecs\n\n", (t1 - t0) * 1000000);
   for (size = 8; size <= sizebits; size *= 2) {
      for (i = 0; i < size / 8; i++) {
         a[i] = (double) i;
         b[i] = 0.0;
      }
      last = size / 8 - 1;
      MPI_Barrier(MPI_COMM_WORLD);
      t0 = MPI_Wtime();
      if (myproc == 0) {
         MPI_Send(a, size/8, MPI_DOUBLE, other_proc, 0, MPI_COMM_WORLD);
         MPI_Recv(b, size/8, MPI_DOUBLE, other_proc, 0, MPI_COMM_WORLD, &status);
      } else {
         MPI_Recv(b, size/8, MPI_DOUBLE, other_proc, 0, MPI_COMM_WORLD, &status);
         b[0] += 1.0;
         if (last != 0)
         b[last] += 1.0;
         MPI_Send(b, size/8, MPI_DOUBLE, other_proc, 0, MPI_COMM_WORLD);
      }
      t1 = MPI_Wtime();
      time = 1.e6 * (t1 - t0);
      MPI_Barrier(MPI_COMM_WORLD);
      if ((b[0] != 1.0 || b[last] != last + 1)) {
         printf("ERROR - b[0] =%f b[%d] =%f\n", b[0], last, b[last]);
         exit (1);
      }
      for (i = 1; i < last - 1; i++)
         if (b[i] != (double) i)
            printf("ERROR - b[%d] =%f\n", i, b[i]);
      if (myproc == 0 && time > 0.000001) {
         printf("%9d bytes(RTT)%9.0f microsegundos(usecs) (%8.3f Mb/sec) \n",
                     size*8,time, ((1.e6/time)*(2.0*size*8))/(1024*1024));
         if (2 * size * 8 / time > max_rate) max_rate = 2 * size * 8 / time;
         if (time / 2 < min_latency) min_latency = time / 2;
      } else if (myproc == 0) {
         printf("%7d bytes toma menos tiempo que el minimo posible medible en usec\n", size);
      }
   }
   if (myproc == 0)
      printf("\n Mejor tasa de transferencia =%f Mb/sec  latencia minima =%f usec\n",
               max_rate, min_latency);
   MPI_Finalize();
   return 0;
}
```