Benemérita Universidad Autónoma de Puebla
Facultad de Ciencias de la Computación

"Implementación paralela de un Algoritmo basado en la Colonia de Hormigas sobre una Plataforma desarrollada en Java"

Tesis que presenta

Karina Meneses Flores

Para obtener el título de
Ingeniero en Ciencias de la Computación

Asesora
Dra. Darnes Vilariño Ayala

Coasesora
Mtra. Mireya Tovar Vidal

Septiembre 2008

Dedicatoria

A mi Madre por todo su amor y por enseñarme a ser independiente.

A mi Padre que me ha enseñado que todo esfuerzo tiene su recompensa.

A mi hermana por ser mi conciencia, compañera de estudio y por desvelarse conmigo.

A mi hermano por siempre animarme y no dejar que me rinda.

Agradecimientos

Primero y antes que nada, agradezco a ***Dios****, por estar conmigo en cada paso que doy, por fortalecer mi corazón e iluminar mi mente y por haber puesto en mi camino a aquellas personas que han sido mi soporte y compañía durante todo el periodo de estudio.*

Agradezco hoy y siempre a mi familia ya que por ellos y su apoyo en todo momento, soy lo que soy, muchas gracias, los Amo demasiado.

También agradezco a ese amor que me complementa, el cual me ayuda hacer mejor las cosas y aunque no esté aquí, sigue inspirando.

Muchos son los momentos que agradezco, en los que mejore como estudiante, como ser humano y ahora como profesionista, pues son las personas, con las que comparto esos instante los que hacen que sean especiales, llenos de sabiduría, a veces con disertación, otras de aventura, también de melancolía y por supuesto de ocio, ellos son mis amigos, los de siempre Augusto, Laura, Roberto, Felipe, Jorge, Angélica, Antonia y los que ayudan a no perder la fe y a tener los pies en la tierra Lezhaida, Adriana, Cecilia, Elizabeth, Víctor, Edmundo, René bueno a todos los litúrgicos y por supuesto los amigos BUAP Zelideth, Francisco, Gustavo, Raúl, Fátima, Ángel, Viviana, gracias a todos.

Y de manera especial agradezco a la Dra. Darnes Vilariño A. por ser mi Asesora, por permitirme trabajar con ella y aprender de ella, a la Mtra. Mireya Tovar V. por ser mi Coasesora, y al Consejo de Ciencia y Tecnología del Estado de Puebla por otorgarme la Beca-Tesis CONCYTEP 2008.

En general agradezco a todas las personas que de forma directa e indirecta han vivido conmigo la realización de esta tesis.

MUCHAS GRACIAS

Índice

1. Introducción

Hoy en día existen múltiples problemas de optimización para los que no existen algoritmos exactos cuyo orden de ejecución sea polinomial, por eso los investigadores han buscado resolver esta problemática para grandes dimensiones. El Problema de Agente Viajero (Traveling Salesman Problem - TSP), catalogado como un problema NP completo debido a que el esfuerzo computacional que se debe realizar para encontrar una solución óptima que crece de forma exponencial con el tamaño del problema. Existen diversas aplicaciones de TSP en la industria, por ejemplo, en la cristalización de rayos X, reparación de turbinas de gas, el orden de recolección de un almacén, estableciendo el orden en que una máquina taladra agujeros en una tabla de circuitos u otros objetos, donde los agujeros serían las ciudades y el costo del viaje es el tiempo que toma la cabeza para realizar otro agujero, el secuenciamiento de trabajos en una máquina, en la asignación de rutas a flotas aéreas o vehiculares, entrega de comida a domicilio y rutas de camiones de entrega de paquetes.

Además existen otras heurísticas para la solución del TSP como los antecedentes más recientes de este trabajo, se cuenta con el desarrollo de P. Jog, J. Suh, y D. Van Gucht [1] quienes en 1991 proponen la aplicación de un Algoritmo Genético Paralelo para el TSP, posteriormente en 1992 P. P.Mutalik, L. R. Knight, J. L. Blanton y R. L.Wainwright [2] plantean un enfoque bastante interesante y novedoso al introducir el concepto de un algoritmo basado en Anhelo Simulado Paralelo y Algoritmos Genéticos Paralelos para la resolución de problemas de Optimización Combinatoria. Este mismo año S. Tschoke, R. Luling, M. Racke y B. Monien [3] proponen un algoritmo de Ramificar y Acotar paralelo en una red de 1024 procesadores, finalmente un trabajo reciente de Dorigo M., Maniezzo V. y Colorni A. [4] propusieron una técnica conocida como Ant system para el conocido paradigma del TSP que presentaba tres variantes que se diferencian simplemente en el momento y la manera de actualizar una matriz que representa las feromonas de los sistemas biológicos, de los cuales para efectos de este trabajo se considera la relación que existe entre la cantidad de feromonas depositadas por arco con la distancia que arco posea.

Estas soluciones surgen como una herramienta poderosa para hallar soluciones muy cercanas a las óptimas en problemas que anteriormente se habían considerados complejos y en la actualidad la observación de la naturaleza, ha contribuido en la creación de nuevos paradigmas computacionales, los cuales se han aplicado con éxito a la búsqueda de las soluciones óptimas de estos tipos de problemas, tal es el caso de los Sistemas Basados en Colonias de Hormigas, que es el tema central de este trabajo de investigación, que son el estudio de sistemas artificiales de hormigas, inspirado en la conducta colectiva de hormigas reales, considerando que las hormigas son agentes autónomos y dada la complejidad computacional del algoritmo secuencial que impide su aplicación a problemas de gran envergadura. Se han estado proponiendo varias estrategias de paralelización como lo son la implementación paralela síncrona y asíncrona, de las cuales la asíncrona es más apropiada cuando lo que se busca es la paralelización y distribución entre los diversos procesadores de una red de computadoras. En efecto, el método utiliza la interacción de muchas hormigas (agentes), que son básicamente independientes entre sí en cada tour y cooperan intercambiando información para el logro de un objetivo común.

La idea general de esta metaheurística es la de construir soluciones con hormigas artificiales para un determinado problema de optimización combinatoria. Típicamente una hormiga construye una solución con una secuencia de decisiones probabilísticas donde la mayoría de las decisiones se extienden parcialmente para dar una nueva solución. La secuencia de decisiones para la construcción de una solución puede ser vistas como una dirección a través de un grafo de decisión. Esto se hace en un proceso iterativo donde las buenas soluciones encontradas deben guiar a las hormigas siguientes. Por lo tanto, las hormigas que han encontrado buenas soluciones van dejando un rastro en el camino. Estas feromonas guían a las hormigas siguientes en la próxima iteración de la búsqueda, con el fin de que las feromonas con más tiempo no influyan en las siguientes iteraciones, debe realizarse una actualización en el valor del porcentaje de cada feromona. Y así el proceso continúa hasta que se determina algún criterio de parada se cumpla.

El método consiste en simular computacionalmente un conjunto de agentes cooperativos que intercambian información de manera indirecta, el paralelismo es inherente al funcionamiento del algoritmo, es decir, el comportamiento de una hormiga es independiente de todas las demás durante la misma iteración.

Se han propuesto varias estrategias de paralelización [5], como la implementación paralela síncrona y otra parcialmente asíncrona que se resumen a continuación.

a) En la implementación paralela síncrona, un proceso inicial (master) levanta a un conjunto de procesos hijos, uno para cada hormiga. Después de distribuir la información inicial acerca del problema, cada proceso inicia la construcción del camino y calcula la longitud del tour encontrado. Después de terminar este procedimiento, los resultados son enviados al master, quien se encarga de actualizar el nivel de feromonas y calcular el mejor tour encontrado hasta ahora. Se inicia una nueva iteración con el envío de la nueva matriz de feromonas.

b) En la implementación parcialmente asíncrona, se propone reducir la frecuencia de la comunicación. Para esto, cada hormiga realiza un cierto número de iteraciones del algoritmo secuencial, independientemente de las otras hormigas. Solo después de estas iteraciones locales, se realiza una sincronización global entre las hormigas. Entonces el master actualiza el nivel de feromonas y se inicia el proceso de nuevo.

Como una extensión de éste trabajo, Tomas Stützle presenta otra estrategia de paralelización [6], propone corridas independientes y paralelas de un mismo algoritmo, (ACO) evitando de éste modo el overhead de comunicación. Cabe destacar que en este modelo de paralelización no existe comunicación entre los procesos que corren en diferentes máquinas. El método funciona solo si el fundamento del algoritmo es aleatorio como en el caso de ACO. La mejor solución de k corridas es elegida al final de la siguiente forma:

a) Una implementación síncrona al tener un proceso master que es utilizado para actualizar las estructuras de datos y construir soluciones iníciales que son enviadas a los procesadores hijos que se encargan de aplicar búsqueda local sobre ellos. El master recoge las soluciones

óptimas locales y de encontrarse un número suficiente de tales soluciones se actualiza la matriz de feromonas antes de construir más soluciones.

b) Una implementación asíncrona en la que un procesador guarda la matriz de feromonas y la actualiza, mientras que uno o varios procesadores pueden usar la matriz de feromonas para construir soluciones y enviar a aquellos procesadores que aplicaran búsqueda local sobre éstos resultados.

Esta técnica comienza a tener la madurez tecnológica adecuada para su utilización en problemas reales, por el paralelismo inherente que en nuestro caso se ha trabajará con asincronismo que elimina los tiempos muertos producidos por la espera en la sincronización de la comunicación, los cuales son extremadamente perjudiciales cuando se trabaja con una red de computadoras cuyo tráfico no se puede controlar totalmente, es por eso que los algoritmo paralelo se ha desarrollado en un lenguaje orientado a objetos como lo es Java, el cual está ganando aceptación en diversos ámbitos, desde comerciales hasta en investigaciones. El lenguaje Java ofrece ciertas características tales como portabilidad, mecanismos de concurrencia, tecnologías para el desarrollo de aplicaciones distribuidas y mecanismos para la ejecución de código nativo a cada arquitectura. Esto hace de Java un candidato para el desarrollo de aplicaciones paralelas.

El presente trabajo propone implementar un Algoritmo paralelo basado en la Colonia de Hormigas que ofrece una solución óptima para el problema del Agente Viajero (TSP) sobre la Plataforma Paralela desarrollada en Java por el Mtro. Erick Pinacho R ex alumno de Maestría en Ciencias de la Computación de la FCC-BUAP, siendo el objetivo principal el comprobar la fiabilidad de la Plataforma en sistemas paralelos bajo un ambiente totalmente asíncrono con una red de PCs.

Esto es, agentes computacionales muy simples llamados hormigas trabajan en cada procesador de una red de computadoras, explorando el espacio de soluciones, comunicando en forma asíncrona a los demás procesadores los resultados más alentadores, consolidando la información recogida en matrices de feromonas propias de cada procesador, que servirán para guiar la búsqueda de mejores soluciones.

El trabajo está dividido en 6 Capítulos. El Capítulo 2 es un breve estudio bibliográfico del Problema del Agente Viajero, Algoritmos relacionados, heurísticas de programación y un breve estudio sobre el comportamiento de las hormigas y sus procesos de optimización para implementar el Algoritmo Secuencial y un estudio sobre la teoría del Paralelismo para determinar el algoritmo Paralelo. Estos algoritmos serán implementados en la Plataforma desarrollada en Java la cual es tratada en el Capítulo 3, para que posteriormente en el Capítulo 4 se presente el Análisis y Diseño del Algoritmo con las herramientas UML (*Unified Modeling Language*) y JUDE (*Java and UML Developers' Environment*) que nos proporcionaran información necesaria para la implementación y pruebas del sistema que se abordan en el Capitulo 5 y en el Capitulo 6 se dan las conclusiones de este trabajo.

2. Marco Teórico

A pesar de los continuos estudios existentes del problema del agente viajero (*TSP*), sigue siendo un reto y despertando curiosidad, de allí que frecuentemente se estén estudiando sus variaciones y posibles heurísticas que aproximan una solución optima para el TSP como lo es el Sistema de Hormigas *(AS – Ant System)* el cual se basa en el comportamiento colectivo de las hormigas en la búsqueda de alimentos para su subsistencia. Resulta fascinante entender como animales casi ciegos, moviéndose aproximadamente al azar, pueden encontrar el camino más corto desde su nido hasta la fuente de alimentos sin pasar más de una vez por el mismo lugar y regresar, tal como el agente viajero. Para esto, cuando una hormiga se mueve, deja una señal odorífera, depositando una substancia denominada feromona, para que las demás puedan seguirla. En principio, una hormiga aislada se mueve esencialmente al azar, pero las siguientes deciden con una buena probabilidad seguir el camino con mayor cantidad de feromonas.

2.1 Problema del Agente Viajero

El problema del agente viajero o TSP como se le conoce en la literatura, consiste en un agente de ventas que tiene que visitar ***n*** ciudades, comenzando y terminando en una misma ciudad, visitando solamente una vez cada ciudad, y haciendo el recorrido de costo mínimo, este costo de recorrido puede estar expresado en términos de tiempo o distancia, es decir, recorrer el mínimo de kilómetros o llevar a cabo un tour en el menor tiempo posible.

El problema del agente viajero se puede modelar fácilmente mediante un grafo completo dirigido, en donde los vértices del grafo son las ciudades y los arcos son los caminos, dichos arcos deben tener un peso, y este peso representa la distancia que hay entre dos vértices que están conectados por medio de dicho arco. Una solución del problema del agente viajero, se puede representar como una secuencia de ***n+1*** ciudades, en donde un tour comienza y termina con la misma ciudad.

2.1.2 Agente Viajero Simétrico y Asimétrico

Existen muchos variantes del problema del *Agente Viajero*, pero para nuestro propósito solo citaremos dos, de los cuales nos interesa conocer sus diferencias.

2.1.2.1 Problema del Agente Viajero Simétrico

Conocido como el *Symmetric Traveling Salesman Problem*, consiste en que dado un conjunto de n ciudades y distancias para cada par de ciudades, se requiere encontrar una ruta de longitud total mínima, visitando cada uno de los nodos exactamente una vez. Siendo la distancia de la ciudad i a la ciudad j la misma que de la ciudad j a la ciudad, para ilustrar

este problema veamos la figura 1.1 que muestra un ejemplo para cuatro ciudades de un Agente Viajero Simétrico, considerando este grafo No dirigido.

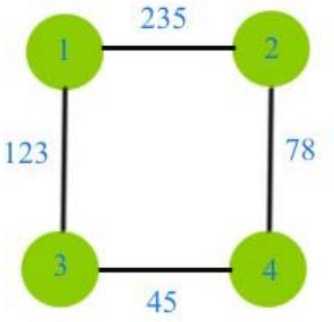


Figura 1.1 Representación del TSP Simétrico, mediante un grafo No dirigido.

Como podemos observar la distancia entre la ciudad 1 y 2 es de 235, tal como de la ciudad 2 a la 1 también es de 235, por lo tanto es la misma distancia.

2.1.3.2 Problema del Agente Viajero Asimétrico

Conocido como *Asymmetric Traveling Salesman Problem* que consiste en dado un conjunto de ciudades y distancias para cada par de ciudades, se requiere encontrar una ruta de longitud total mínima, visitando cada uno de las ciudades exactamente una vez. En este caso, la distancia de la ciudad i a la ciudad j es diferente de la distancia de la ciudad j a la ciudad i, como las distancias son diferentes entre las ciudades, la mejor forma de representar este caso es a través de un grafo dirigido, como se ilustra en la figura 1.2, donde se muestra un ejemplo con 4 ciudades.

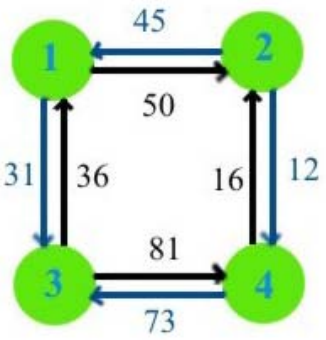


Figura 1.2 Representación del Agente Viajero Asimétrico con un grafo dirigido

La diferencia de distancias entre las ciudades se puede representar en la siguiente matriz, donde podemos notar que la distancia entre la ciudad 1 y 2 es de 50, que no es igual a la distancia de la ciudad 2 a la 1 que es de 45:

Ciudades	1	2	3	4
1	∞	50	31	∞
2	45	∞	∞	12
3	36	∞	∞	81
4	∞	16	73	∞

Tabla 1.1 Representación Matricial del Agente Viajero Asimétrico

Estos problemas de optimización son útiles en el caso de este proyecto, el cual pretende solucionar el problema del Agente Viajero Asimétrico.

2.2 Metaheurística

En la naturaleza los individuos de una población compiten entre sí en la búsqueda de recursos tales como comida, agua y refugio. Incluso los miembros de una misma especie compiten a menudo en la búsqueda de un compañero. Aquellos individuos que tienen más éxito en sobrevivir y en atraer compañeros tienen mayor probabilidad de generar un gran número de descendientes. Por el contrario individuos poco dotados producirán un menor número de descendientes. Esto significa que los genes de los individuos mejor adaptados se propagarán en sucesivas generaciones hacia un número de individuos creciente. La combinación de buenas características provenientes de diferentes ancestros, puede a veces producir descendientes súper individuos", cuya adaptación es mucho mayor que la de cualquiera de sus ancestros. De esta manera, las especies evolucionan logrando unas características cada vez mejor adaptadas al entorno en el que viven.

2.2.1 Algoritmos Genéticos Paralelos

Los *Algoritmos Genéticos* usan una análoga directa con el comportamiento natural. Trabajan con una población de individuos, cada uno de los cuales representa una solución factible a un problema dado. A cada individuo se le asigna un valor o puntuación, relacionado con la bondad de dicha solución. En la naturaleza esto equivale al grado de efectividad de un organismo para competir por unos determinados recursos.

El poder de los Algoritmos Genéticos proviene del hecho de que se trata de una técnica robusta, y pueden tratar con éxito una gran variedad de problemas provenientes de diferentes áreas, incluyendo aquellos en los que otros métodos encuentran dificultades. Si bien no se garantiza que el Algoritmo Genético encuentre la solución óptima del problema, existe evidencia empírica de que se encuentran soluciones de un nivel aceptable, en un tiempo competitivo con el resto de algoritmos de optimización combinatoria. En el caso de que existan técnicas especializadas para resolver un determinado problema, lo más probable es que superen al Algoritmo Genético, tanto en rapidez como en eficacia. El gran campo de

aplicación de los Algoritmos Genéticos se relaciona con aquellos problemas para los cuales no existen técnicas especializadas. Incluso en el caso en que dichas técnicas existan, y funcionen bien, surge otra técnica, que ante la necesidad de cómputo requerida por problemas de extrema complejidad, cuyo tiempo de ejecución utilizando los tradicionales algoritmos genéticos secuenciales es prohibitivo. Es por eso que se buscó la manera de poder adaptar este tipo de heurísticas a distintas configuraciones de cómputo paralelo, lo que dio lugar a tres grandes modelos de Algoritmos Genéticos Paralelos: (1) algoritmos maestro-esclavo [7].

2.2.2 Algoritmo maestro-esclavo

Este algoritmo trabaja con una única población de individuos que será gestionada por el nodo maestro. La evaluación de la adecuación de los individuos y/o la aplicación de los operadores genéticos puede ser realizada por los nodos esclavos. A cada nodo esclavo le corresponderá una parte de la población total, sobre la cual realizará las operaciones antes citadas. Una vez terminado este proceso, devolverán el resultado al nodo maestro, que realizará la selección de individuos. En la figura 2.1 podemos encontrar un pequeño esquema de la arquitectura de este tipo de algoritmos.

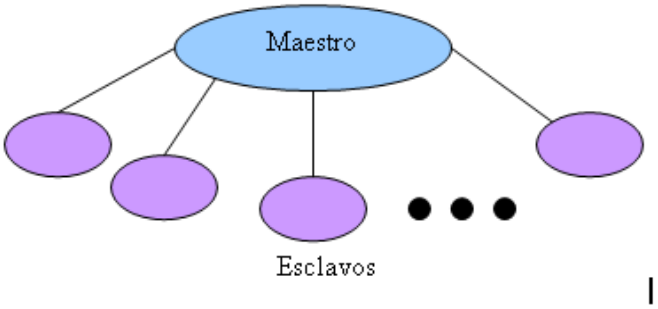


Figura 2.1 Representación esquemática de un AGP con arquitectura maestro-esclavo

Normalmente, la operación que se suele implementar en paralelo es la evaluación de la adecuación de los individuos, porque suele ser la más compleja. Además, este valor es independiente del resto de la población, por lo que su implementación es muy sencilla.

El intercambio de información entre los nodos es sencillo: el nodo maestro envía el subconjunto de individuos que corresponde a cada nodo, y éstos le devuelven los valores de adecuación para cada uno de ellos. La comunicación puede ser implementada de dos maneras: síncrona o asíncrona. En la primera de ellas, el nodo maestro espera a recibir los valores de adecuación de todos los individuos para generar la siguiente generación. En la segunda, el algoritmo no espera a que los nodos más lentos envíen sus valores de *fitness*. De este modo conseguimos agilizar el proceso, pero el comportamiento no es exactamente el mismo que el de un algoritmo genético secuencial. Es por eso que este Algoritmo es a

base de la implementación del Algoritmo Paralelo basado en la Colonia de Hormigas sobre la Plataforma Paralela, ya que cubre con los requerimientos específicos de optimización.

Siendo que este tipo de algoritmos no especifican nada acerca de la arquitectura subyacente, por lo que pueden ser implementados sin problemas en computadores tanto de memoria compartida como de memoria distribuida. En el caso de los computadores de memoria compartida, la población podría estar almacenada en memoria, y cada procesador esclavo podría leer directamente los individuos que le hubieran sido asignados. En caso de usar un computador de memoria distribuida, sería el nodo maestro el encargado de asignar y distribuir individuos entre los nodos esclavos y de recoger, una vez evaluada, la adecuación de cada uno de ellos.

Varios estudios han sido realizados para evaluar el rendimiento de este tipo de algoritmos. Algunos resultados interesantes pueden verse, por ejemplo, en el que realizaron Abramson, Mills y Perkins [8], que usando dos computadores de memoria compartida observaron un incremento de prestaciones razonablemente alto en comparación con los algoritmos genéticos tradicionales usando hasta 16 procesadores. A partir de ese número, el comportamiento se degradaba, dato éste que justificaban por el aumento del coste de las comunicaciones.

2.2.3 Sincronismo y asincronismo en los AGP

Cuando se pretende implementar un algoritmo genético paralelo, del tipo que sea, una duda surge inmediatamente: ¿deben las migraciones efectuarse de forma síncrona o asíncrona? En el caso de las implementaciones síncronas, cada nodo del sistema distribuido espera, llegado el momento de las migraciones, a recibir todos los individuos de sus nodos adyacentes. Esto no es así en el caso de las implementaciones asíncronas, en las que todo individuo recibido será insertado en la población nada más llegar, independientemente del resto. Esto hace que las poblaciones evolucionen de distinta manera, según la velocidad de cómputo de cada nodo, y que se difumine el concepto de generación.

Un experimento interesante a este respecto es el efectuado por Alba y Troya [12], para el que usaron varias implementaciones paralelas de algoritmos genéticos, tanto síncronas como asíncronas, distintas frecuencias de migraciones y número de nodos. Probaron los algoritmos con varios test de diferente complejidad, aunque podemos avanzar que, por regla general, las implementaciones asíncronas obtengan soluciones de igual calidad que las síncronas, evaluando un número similar de individuos, pero en un tiempo inferior. Además, observaron que esta mejora dependía también de la complejidad del problema a resolver y del número de nodos utilizados, ya que si el problema resultaba demasiado simple, o el número de nodos era excesivo, la disminución del tiempo para encontrar una solución óptima era despreciable.

Un hallazgo curioso relacionado con el uso de una política de comunicaciones asíncrona es el hecho de que estos algoritmos presentan una mayor resistencia al uso de frecuencias de migración erróneas. Es decir, si para un problema se eligen unas frecuencias inapropiadas,

tanto si éstas son excesivas como si son insuficientes, la versión síncrona del algoritmo tendrá problemas para encontrar soluciones en un tiempo razonable, llegando en algunos casos a no ser capaces de hacerlo. Por el contrario, parece que esto afecta en menor medida a los algoritmos asíncronos.

En resumen, la calidad de las soluciones encontradas no se ve afectada por el uso de algoritmos síncronos o asíncronos, aunque, en general, los tiempos de búsqueda serán inferiores en el caso de estos últimos.

2.2.4 Colonia de Hormigas

Las hormigas son insectos sociales que, en conjunto, solucionan problemas complejos que no podrían resolver en forma individual. De esta manera, como colonia y mediante la comunicación que existente entre ellas, pueden guiar su comportamiento para la solución de sus problemas, como por ejemplo, encontrar el camino más corto entre el hormiguero y la fuente de comida.

Cuando las hormigas salen del hormiguero en busca de comida sin algún conocimiento sobre las posibles fuentes de alimento, estas se guían por el olor de feromonas que dejan las primeras hormigas “buscadoras” o ellas mismas. Las hormigas hacen la búsqueda caminando al azar, que es similar a la humana "rastreando". Solo que cada hormiga deja un rastro de feromonas en su búsqueda, como se muestra en la Figura 2.4 donde se muestra el rastro de cada hormiga en color rojo.

Figura 24. Las hormigas buscan su comida, sin importar la ruta

Cuando alguna de estas hormigas encuentra comida, regresa a su nido a través de su rastro, haciendo más fuerte el olor

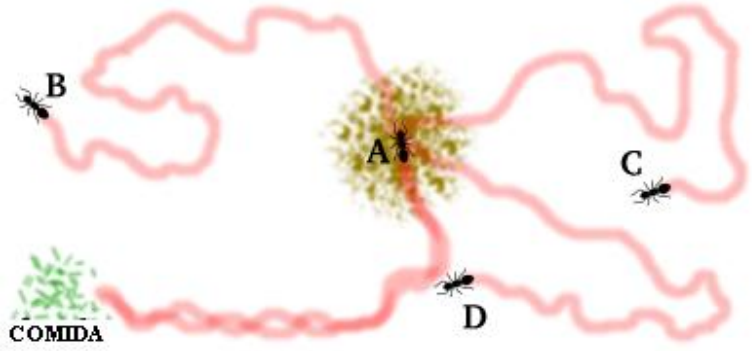


Figura 2.5 Cuando encuentra su comida, regresan por la misma ruta, gracias al rastro de feromonas

Cuando existe algún rastro de feromona, las hormigas tienden a escoger el camino que este indica. Así, por ejemplo, en un cruce de caminos la hormiga tiene mayor probabilidad de seguir por la ruta cuyo rastro de feromona sea más fuerte. Cuando no existe ningún rastro, las hormigas se mueven aleatoriamente.

Cuando se tienen dos caminos que llevan a la comida, uno largo y otro corto, las hormigas que encontraron el camino corto van a demorarse menos en ir y volver desde el hormiguero hasta la fuente de comida, dejando feromonas por la ruta recorrida y haciendo el rastro de feromona más fuerte que el de las que recorrieron el camino más largo. Así, las que encontraron el sendero más largo, cuando vuelvan a salir se van a encontrar con un cruce de caminos, uno con un rastro de feromona más fuerte que el otro, y van a tener mayor probabilidad de recorrer el camino más corto esta vez, figura 2.6.

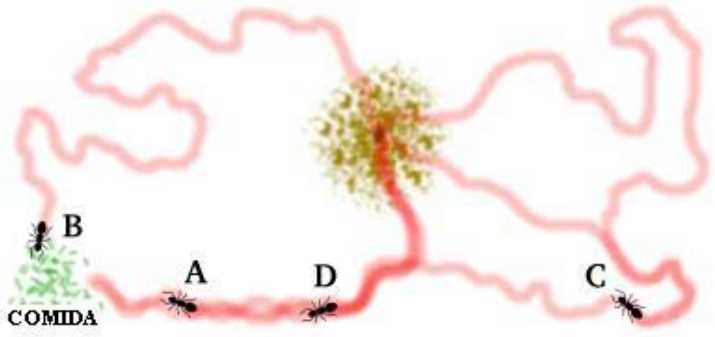


Figura 2.6 Como el olor de las feromonas es más fuerte en el camino corto, todas las hormigas siguen ese rastro

El rastro de feromona se evapora con el tiempo, haciendo que cada vez se pierda más el rastro del camino más largo, ya que por la ruta más corta han pasado más hormigas, renovando el rastro en este camino y haciendo que el rastro sea tan intenso con respecto a los demás, que obliga a todas las hormigas a escoger esta vía.

2.2.4.1 Condiciones de las Hormigas

La hormiga ha sido enfoque de la investigación ya que su forma diferente de navegar es muy diferente a la nuestra. El estudio nos muestra que lo importante es encontrar el mejor camino y no el tiempo que lleve el recorrerlo. Por supuesto, la hormiga en su método tiene ventajas de alto nivel sobre la nuestra. Por ejemplo, el método de la hormiga funciona bien en la oscuridad. Cuando se trata de navegar sin señales visuales, los seres humanos somos relativamente indefensos.

El método de la hormiga puede ser determinado en unas simples reglas, que pueden seguir todos sus miembros:

a) Si la hormiga no transporta comida y no deja feromonas entonces debe caminar al azar y dejar rastro de feromona.
b) Si no transporta comida y percibe feromonas entonces debe seguir el rastro y dejar más feromonas.

2.2.4.2 Procesos de Optimización Mediante Colonias de Hormigas

Los algoritmos de las Colonias de Hormigas (*ACH)* son modelos inspirados en el comportamiento de colonias de hormigas reales. Estudios realizados explican cómo animales casi ciegos, como son las hormigas, son capaces de seguir la ruta más corta en su camino de ida y vuelta entre la colonia y una fuente de abastecimiento.

El problema surge cuando los caminos se cruzan y entonces se mezclan los olores de feromona de dos o más caminos, en este caso las hormigas eligen el camino a seguir con una decisión probabilística sesgada por la cantidad de feromona: *cuanto más fuerte es el rastro de feromona, mayor es la probabilidad de elegirlo.* Puesto que las hormigas depositan feromona en el camino que siguen, este comportamiento lleva a un proceso de auto-refuerzo que concluye con la formación de rastros señalados por una concentración de feromona elevada. Este comportamiento permite además a las hormigas encontrar los caminos más cortos entre su hormiguero y la fuente del alimento.

Al paso del tiempo la feromona se evapora, es por esto que si un camino no ha sido visitado por muchas hormigas, es decir, que no es una buena solución ya que no es un camino corto a la comida, el olor tiende a desaparecer; en cambio si es una buena solución su olor es fuerte y persistente ya que muchas hormigas han decidido pasar por este camino.

Debido a la gran persistencia de feromona, es difícil que las hormigas “olviden” un camino que tiene un alto nivel de feromona aunque hayan encontrado un camino aún más corto. Esto en lenguaje computacional quiere decir que en un algoritmo de búsqueda nos quedemos estancados en un óptimo local.

Los algoritmos de colonia de hormiga son *algoritmos constructivos*: en cada iteración del algoritmo, cada hormiga construye una solución al problema recorriendo un grafo de construcción. Cada arista del grafo, que representa los posibles pasos que la hormiga puede dar, tiene asociada dos tipos de información que guían el movimiento de la hormiga:

- *Información heurística*, que mide la preferencia heurística de moverse desde el nodo *r* hasta el nodo *s*. Las hormigas no modifican esta información durante la ejecución del algoritmo.
- *Información de los rastros de feromona artificiales*, que mide la *"deseabilidad aprendida"* del movimiento de un lugar a otro. Imita a la feromona real que depositan las hormigas naturales. Esta información se modifica durante la ejecución del algoritmo dependiendo de las soluciones encontradas por las hormigas.

2.2.4.3 Características de Optimización

- Busca soluciones válidas de costo mínimo para el problema a solucionar.
- Tiene una memoria *Tabú* que almacena información sobre el camino seguido hasta el momento, esto es, *Tabú* almacena la secuencia generada. Esta memoria puede usarse para construir soluciones válidas, evaluar la solución generada, y reconstruir el camino que ha seguido la hormiga.
- Comienza en el estado inicial y se mueve siguiendo estados válidos, construyendo la solución asociada incrementalmente.
- El movimiento se lleva a cabo aplicando una función de los rastros de feromona que están disponibles localmente, de los valores heurísticos de la memoria privada de la hormiga y de las restricciones del problema.
- Durante el procedimiento de construcción una hormiga se mueve desde el nodo *r* hasta el *s,* puede actualizar el rastro de feromona. Este proceso se llama *actualización en línea de los rastros de feromona paso a paso*.
- El procedimiento de construcción acaba cuando se satisface alguna condición de parada, normalmente cuando se alcanza un estado objetivo.
- Una vez que la hormiga ha construido la solución puede reconstruir el camino recorrido y actualizar los rastros de feromona de los lugares visitados/as utilizando un proceso llamado *actualización en línea a posteriori*. Este es el único mecanismo de comunicación entre las hormigas, utilizando la estructura de datos que almacena los niveles de memoria compartida.
- Las acciones del demonio son acciones opcionales que por ejemplo permite observar la calidad de todas las soluciones generadas y depositar una nueva cantidad de feromona adicional sólo en las transiciones asociadas a algunas soluciones, o aplicar un procedimiento de búsqueda local a las soluciones generadas por las hormigas antes de actualizar los rastros de feromona. En ambos casos, el demonio reemplaza la actualización en línea a posteriori de feromona y el proceso pasa a llamarse actualización fuera de línea de rastros de feromona.

2.3 Paralelismo

Una batería de satélites en el espacio exterior que recoge un rango de 10^{10} bits por segundo. Los datos representan información del tiempo meteorológico, la polución, la agricultura y recursos naturales. Para que esta información sea útil necesita ser procesada con una velocidad de al menos 10^{15} operaciones por segundo. Por otra parte, un equipo de cirujanos desea ver, en un dispositivo especial una imagen tridimensional del cuerpo del paciente en espera para cirugía. Necesitan rotar la imagen para obtener una sección transversal de un órgano, observándolo en detalle y ejecutar una cirugía simulada, todo ello sin tocar al paciente. La velocidad de procesamiento mínima requerida debe ser de 1015 operaciones por segundo para realizar eficazmente los pasos descritos.

Los dos ejemplos anteriores son representativos de aplicaciones donde se necesitan ordenadores extremadamente rápidos para procesar grandes cantidades de datos o para permitir un gran número de cálculos muy rápidos (o al menos, dentro de un período de tiempo razonable). Y gracias a la evolución de la tecnología computacional es posible tener hoy en día un precio bajo de una computadora de escritorio con más de un procesador, por otro lado la tecnología de comunicación ha permitido comunicar múltiples computadoras uniprocesador y tener una red de cómputo distribuido extremadamente eficiente. Lo cual permite resolver este tipo de problemas en tiempos razonables con menor ejecución de procesos.

Aunque existen otras clases de aplicaciones como el testeo de aeroplanos, el desarrollo de nuevos medicamentos, exploración de petróleo, modelado de reactores de fusión, planificación económica, criptoanálisis, gestión de grandes bases de datos, astronomía, análisis biomédico, reconocimiento del habla en tiempo real, robótica y la solución de grandes sistemas de ecuaciones diferenciales parciales que surgen de simulaciones numéricas en disciplinas tan diversas como sismología, aerodinámica y física nuclear, atómica y del plasma. Para las cuales aun no existe algún computador que pueda desarrollar la velocidad de procesamiento requerida para estas aplicaciones.

Por que el factor límite es una simple ley física que da la velocidad de la luz en el vacío. Esta velocidad es aproximadamente igual a 3*108 m/s. Donde se asume que un periférico electrónico permite 1012 operaciones por segundo. Entonces tarda más una señal en viajar entre dos periféricos similares medio milímetro que lo que tardan en su proceso. En otras palabras, toda la ganancia en velocidad obtenida construyendo componentes electrónicos super-rápidos se pierde cuando un componente está esperando a recibir una entrada de otro. Pero, la física dice que la reducción en distancia entre dos periféricos electrónicos alcanza un punto a partir del cual dichos periféricos empiezan a interaccionar, de esta forma se reduce no sólo su velocidad sino también su fiabilidad.

La única manera de resolver este problema es usar una nueva técnica avanzada denominada *paralelismo*. La idea es que si una tarea se descompone en una serie de operaciones, y estas operaciones se realizan simultáneamente, el tiempo que tarda en realizarse dicha tarea puede reducirse significativamente. Esta es una noción intuitiva y es a lo que se está acostumbrado en una sociedad organizada. Desde el correo hasta la vendimia y desde una

oficina hasta el trabajo en una fábrica se ofrecen numerosos ejemplos de paralelismo a través de tareas compartidas.

Incluso en el campo de la computación la idea del paralelismo no es completamente nueva y ha adoptado muchas formas. Desde los primeros momentos del procesamiento de la información las personas se dan cuenta que es mucho más ventajoso tener muchos componentes de un computador que hagan diferentes cosas al mismo tiempo. Por ejemplo, mientras la CPU realiza cálculos, la entrada de datos puede ser leída de un cassette y la salida llevada a una impresora de líneas. En máquinas más avanzadas hay varios procesadores simples, cada uno de ellos realizando una tarea distinta. En nuestros días algunos de los más poderosos computadores contienen dos o más unidades de proceso que comparten los trabajos solicitados. En cada uno de los ejemplos anteriormente mencionados el paralelismo se explota rentablemente. Estrictamente hablando ninguna de las máquinas mencionadas es verdaderamente un computador paralelo. En el moderno paradigma que queremos describir, sin embargo, la computación paralela se puede realizar en toda su potencia. La meta de estos nuevos diseños sería la siguiente: "Una gran colección de procesadores que puedan comunicar y cooperar para resolver grandes problemas rápidamente" [19].

2.3.1 Cómputo Paralelo

Los sistemas de cómputo con procesamiento en paralelo surgen de la necesidad de resolver problemas complejos en un tiempo razonable, utilizando las ventajas de memoria, velocidad de los procesadores, formas de interconexión de estos y distribución de la tarea, a los que en su conjunto denominamos arquitectura en paralelo. Entenderemos por una arquitectura en paralelo a un conjunto de procesadores interconectados capaces de cooperar en la solución de un problema.

Así, para resolver un problema en particular, se usa una o combinación de múltiples arquitecturas (topologías), ya que cada una ofrece ventajas y desventajas que tienen que ser sopesadas antes de implementar la solución del problema en una arquitectura en particular. También es necesario conocer los problemas a los que se enfrenta un desarrollador de programas que se desean correr en paralelo, como son: el partir eficientemente un problema en múltiples tareas y como distribuir estas según la arquitectura en particular con que se trabaje.

En el cómputo paralelo los sistemas se extienden con más procesadores para obtener una reducción en el tiempo de ejecución. Otras veces, el problema a resolver puede ser modelado naturalmente en forma paralela. La eficiencia y la efectividad del paralelismo dependen por mucho de los problemas a resolver con algoritmos selectos y de las arquitecturas de hardware dedicadas.

Una computadora paralela es un conjunto de procesadores que pueden trabajar cooperativamente para resolver un problema computacional. Esta definición es lo suficientemente amplia para incluir supercomputadoras que tienen cientos o miles de

procesadores, redes de estaciones de trabajo, estaciones de trabajo con multiprocesadores y sistemas embebidos.

El paralelismo a veces ha sido visto como un área compleja e interesante de la computación, pero, de poca relevancia para el programador promedio. A pesar de que las aplicaciones científicas siguen siendo las principales motivaciones detrás del desarrollo de tecnologías de cómputo paralelo se está convirtiendo en la columna de la programación empresarial.

2.3.2 Categorías de Computadoras Paralelas

Clasificación moderna que hace alusión única y exclusivamente a los sistemas que tienen más de un procesador. Existen dos tipos de sistemas teniendo en cuenta su acoplamiento:

- Los sistemas fuertemente acoplados son aquellos en los que los procesadores dependen unos de otros.
- Los sistemas débilmente acoplados son aquellos en los que existe poca interacción entre los diferentes procesadores que forman el sistema.

Atendiendo a esta y a otras características, la clasificación moderna divide a los sistemas en dos tipos: Sistemas multiprocesador (fuertemente acoplados) y sistemas multicomputadoras (débilmente acoplados).

2.3.3 Clasificación de los ordenadores

Cualquier computador, ya sea secuencial o paralelo, opera por la ejecución de instrucciones sobre datos. Un flujo de instrucciones (el algoritmo) indica al computador qué es lo que tiene que hacer en cada paso. Un flujo de datos (la entrada del algoritmo) es modificado por estas instrucciones.

Para lograr una completa paralelización en la implementación del Algoritmo paralelo basado en la Colonia de Hormigas se procuro utilizar ordenadores de tipo:

Múltiple flujo de instrucción, múltiple flujo de datos (MIMD). Ya que esta arquitectura paralela consiste en N procesadores idénticos, cada uno con su propia memoria local donde guarda los datos. Todos los procesadores operan bajo el control de un flujo de instrucciones simple proporcionado por una unidad central de control. Los procesadores operan síncronamente: en cada paso, todos los procesadores ejecutan la misma instrucción sobre diferentes datos. La instrucción puede ser simple (sumar o comparar dos números) o compleja (la mezcla de dos listas de números). De igual forma, los datos pueden ser simples (un número) o complejos (un conjunto de números). Los procesadores que están

inactivos durante la ejecución de una instrucción o que completan la ejecución de la instrucción antes que los demás están desocupados hasta que se da la próxima instrucción.

En ocasiones, si los nodos que conforman esta máquina MIMD de memoria distribuida, no son sino nodos de propósito general interconectados por alguna interfaz de 12 red de uso general, tal como una red de estaciones de trabajo, se le conoce como SPMD (Same Program, Multiple Data).
Recientemente, la programación multihilos sobre arquitecturas SMP(Symmetrical MultiProcessor) y la programación con base en el Paso de Mensajes en sistemas de memoria distribuida (o incluso en sistemas tipo cluster) se vuelven más y más populares.

2.3.4 Programación en Paralelo

Son tareas casi independientes, las cuales ejecutan secciones del algoritmo de solución del problema (y por lo tanto no son idénticas), los datos y los resultados de cómputo son pasados entre las tareas como lo dicta el algoritmo.

Se trata de inyectar paralelismo al algoritmo que va a ser utilizado para resolver el problema. El algoritmo puede ya existir de manera secuencial o ser totalmente nuevo. El paralelismo se puede introducir considerando cómo puede el algoritmo separarse en secciones casi independientes. Cada sección puede ejecutarse en paralelo con un flujo de datos entre las secciones si es necesario.

Cada sección puede ejecutar algún cómputo con los datos y después pasarlos a otra sección si es necesario. El paralelismo inherente es frecuentemente encontrado en un ciclo o iteración. Considere una búsqueda lineal por ejemplo. En el caso secuencial, cada objeto en la lista es comparado uno a la vez hasta encontrarlo o bien llegar al final de la lista. Esta comparación, sin embargo, puede ser desarrollada más eficientemente en paralelo, donde cada comparación puede desarrollarse al mismo tiempo.

La aproximación algorítmica es modelada por procesos paralelos, y cada proceso es responsable de la ejecución de una sección del algoritmo, usando la sincronización y la comunicación a través de los canales para transferir datos entre procesos. Los costos indirectos en tiempo debidos a la comunicación entre los procesos paralelos pueden llegar a ser bastante significantes.

Un ejemplo común de aproximación algorítmica es el pipeline; cada unidad del pipeline contribuye a la ejecución de una sección del algoritmo. Las unidades independientes pueden operar en porciones separadas de información dando una corriente de datos que alimenta a las unidades siguientes del pipeline. Pero sobre todo, el efecto de esta superposición de operaciones es la realización de una ejecución en paralelo.

Esta organización no está limitada a casos unidimensionales. Por ejemplo, un arreglo sistólico es efectivamente un pipeline bidimensional que puede ser usado con gran efecto en ejecuciones paralelas de operaciones sobre matrices.

2.3.5 Desarrollo de Algoritmos en Paralelo.

Aunque el mundo en el que nos movemos es paralelo, y nuestra misma forma de vivir también es así, el arte de la programación tradicionalmente ha sido secuencial.

El desarrollo de la programación en paralelo no se ha debido a un paso natural en el desarrollo del software, sino debido a una necesidad de aumento de potencia en los ordenadores.

El desarrollo de las nuevas arquitecturas que se han descrito con anterior ha provocado la necesidad de disponer de unas herramientas para el desarrollo de programas en esas máquinas. Al igual que sucedió al inicio de los años 60 con los primeros ordenadores, nos encontramos con un vacío entre las arquitecturas desarrolladas y los lenguajes de programación para usar esas arquitecturas. De tal forma que podemos decir que *estamos programando las máquinas paralelas en el equivalente al lenguaje máquina*.

Si ya de por sí esto complica el diseño y desarrollo de algoritmos paralelos, hay otro factor aún más importante que lo dificulta, y es el comportamiento de un programa paralelo. A diferencia de un programa secuencial, un programa paralelo puede tener un comportamiento impredecible "no determinista". Como dicen McGraw y Axelrod en [17], *el hecho de que algún programa paralelo funcione correctamente una vez, o incluso cien veces, con algún conjunto de entradas determinado, no garantiza que no fallará mañana con las mismas entradas*.

El problema radica en que lo único que se busca es el aumento de la capacidad de procesamiento que ofrecen las máquinas con múltiples procesadores, pero no los efectos de la concurrencia, considerando el no determinismo de estas máquinas como un efecto lateral indeseable, más que como una propiedad que tiene que ser explotada. Ello conduce a una ineficiencia por parte del programador, pues sigue planteándose la resolución de problemas con una mentalidad secuencial. Si distinguimos cuatro pasos en el desarrollo de un programa:

- Planteamiento del problema y elección de una estrategia para su resolución.
- Desarrollo de un algoritmo que resuelva el problema anterior.
- Implementación del algoritmo usando un lenguaje de programación.
- Compilación, ejecución y verificación del programa.

se puede advertir que la resolución paralela se plantea habitualmente en la tercera fase, continuando el planteamiento del problema y el desarrollo del algoritmo de una forma secuencial.

Debido a esto ha habido una proliferación de pseudo-lenguajes paralelos, consistentes en lenguajes secuenciales a los que se les ha añadido algunas extensiones (a modo de directivas de compilación o macros) para manejar el paralelismo. Ciertamente, esto permite seguir usando todo el software ya desarrollado con muy pocos cambios, además de no tener

necesidad de aprender otro lenguaje de programación, pero empobrece la idea de la programación en paralelo.

Desde hace varios años se ha vuelto más importante el desarrollo de herramientas de software que permitan sacar ventaja de las posibilidades de hardware.
En cuanto a hardware, existen diversas arquitecturas paralelas hoy en día, y la librería de funciones más usada para el desarrollo de aplicaciones paralelas es MPI. Esta librería posee limitantes acerca del manejo de memoria, y es responsabilidad del desarrollador construir los mecanismos necesarios si se desea emplear medios de sincronización complejos, tal como una memoria compartida.

El lenguaje Java cada día gana mayor aceptación en la comunidad científica e industrial, extendiendo no sólo su uso, sino también las herramientas diseñadas para este lenguaje. Java ofrece mecanismos de concurrencia, manejo de funciones de bajo nivel y tecnologías para desarrollar aplicaciones distribuidas. Estas y otras características que posee Java, lo hacen un excelente candidato para desarrollar software paralelo.

Es por eso que este trabajo hace uso una Plataforma paralela recién creada bajo el lenguaje Java en la cual se puede desarrollar aplicaciones orientadas a Clústers en redes LAN.

2.4 Optimización de la Colonia de Hormigas

Los algoritmos basados en la colonia de hormigas son método que resuelven problemas que están inspirados en el comportamiento real de las hormigas reales. Un aspecto interesante de este comportamiento es que en estas colonias de hormigas individuos simples realizan tareas complicadas. Por ejemplo su comportamiento colectivo: i) la conducta de recolección que guía a las hormigas por caminos cortos a su fuente de alimento, ii) el transporte colectivo de comida donde un grupo de hormigas pueden transportar partículas de comida mucho más pesados, que la suma de cada miembro trabajando individualmente y iii) las crías se clasifican en huevo y larva las cuales son depositados en espacios en un nido con las mejores condiciones ambientales.

En este capítulo nos concentraremos en la meta-heurística de Optimización de la Colonia de Hormigas (*ACO* por sus siglas en ingles) para dar solución a los problemas de optimización combinatoria. La *ACO* está inspirado en la conducta de recolección de las hormigas. Un aspecto esencial es la comunicación indirecta de las hormigas por vía de las feromonas. La explicación de ese comportamiento tiene que ver con el hecho de que las hormigas depositan feromonas a lo largo de su camino. Es muy probable que las hormigas elijan un camino corto que las acerque más a fuente de comida. Cuando regresan a su nido el olor de las feromonas están en el camino más corto que se va acumulando mas y mas que en un camino más largo, ya que su concentración de feromonas es más alto.

Siendo el trabajo realizado por Dorigo [4] nuestra fuente de inspiración ya que el propone una heurística para resolver el TSP y una optimización en el ambiente de la Colonia de Hormigas.

2.4.1 Búsqueda Tabú

La Búsqueda Tabú (Tabú Search - TS) es un procedimiento meta heurístico introducido y desarrollado por *Fred Glover* [13] el cual se utiliza con gran éxito para resolver problemas de optimización cuya característica principal es la de escapar de la optimización local.

TS es una técnica para resolver problemas combinatorios de gran dificultad que está basada en principios generales de Inteligencia Artificial (*IA*). En esencia es una metaheurística que puede ser utilizada para guiar cualquier procedimiento de búsqueda local en la búsqueda agresiva del óptimo del problema. Por agresiva nos referimos a la estrategia de evitar que la búsqueda quede "atrapada" en un óptimo local que no sea global. A tal efecto, TS toma de la IA el concepto de memoria y lo implementa mediante estructuras simples con el objetivo de dirigir la búsqueda teniendo en cuenta la historia de ésta. Es decir, el procedimiento trata de extraer información de lo sucedido y actuar en consecuencia. En este sentido puede decirse que hay un cierto aprendizaje y que la búsqueda es inteligente.

TS comienza de la misma forma que cualquier procedimiento de búsqueda local, procediendo iterativamente de una solución *x* a otra *y* en el entorno de la primera: ***N(x).*** Sin embargo, en lugar de considerar todo el entorno de una solución, TS define el entorno reducido ***N*(x)*** como aquellas soluciones disponibles del entorno de ***x***. Así, se considera que a partir de *x,* sólo las soluciones del entorno reducido son alcanzables.

Existen muchas maneras de definir el entorno reducido de una solución. La más sencilla consiste en etiquetar como *tabú* las soluciones previamente visitadas en un pasado cercano. Esta forma se conoce como memoria a corto plazo *(short term memory)* y está basada en guardar en una lista *tabú T* las soluciones visitadas recientemente. Así en una iteración determinada, el entorno reducido de una solución se obtendría como el entorno usual eliminando las soluciones etiquetadas como tabú.

El objetivo principal de etiquetar las soluciones visitadas como tabú es el de evitar que la búsqueda se cicle. Por ello se considera que tras un cierto número de iteraciones la búsqueda está en una región distinta y puede liberarse del status tabú (pertenencia a T) a las soluciones antiguas. De esta forma se reduce el esfuerzo computacional de calcular el entorno reducido en cada iteración. En los orígenes de *TS* se sugerían listas de tamaño pequeño, actualmente se considera que las listas pueden ajustarse dinámicamente según la estrategia que se esté utilizando.

Se define un nivel de aspiración como aquellas condiciones que, de satisfacerse, permitirían alcanzar una solución aunque tenga status tabú. Una implementación sencilla consiste en permitir alcanzar una solución siempre que mejore a la mejor almacenada, aunque esté etiquetada tabú. De esta forma se introduce cierta flexibilidad en la búsqueda y se mantiene su carácter agresivo.

Es importante considerar que los métodos basados en búsqueda local requieren de la exploración de un gran número de soluciones en poco tiempo, por ello es crítico el reducir

al mínimo el esfuerzo computacional de las operaciones que se realizan a menudo. En ese sentido, la memoria a corto plazo de *TS* está basada en atributos en lugar de ser explícita; esto es, en lugar de almacenar las soluciones completas (como ocurre en los procedimientos de búsqueda exhaustiva) se almacenan únicamente algunas características de éstas. La memoria mediante atributos produce un efecto más sutil y beneficioso en la búsqueda, ya que un atributo o grupo de atributos identifica a un conjunto de soluciones. Así, un atributo que fue etiquetado como tabú por pertenecer a una solución visitada que hace *n* iteraciones, puede impedir en la iteración actual, el alcanzar una solución por contenerlo, aunque ésta sea muy diferente de la que provocó el que el atributo fuese etiquetado. Esto provoca, a largo plazo, el que se identifiquen y mantengan aquellos atributos que inducen una cierta estructura beneficiosa en las soluciones visitadas.

Un algoritmo *TS* está basado en la interacción entre la memoria a corto y la memoria a largo plazo. Ambos tipos de memoria llevan asociadas sus propias estrategias y atributos, y actúan en ámbitos diferentes. Como ya hemos mencionado la memoria a corto plazo suele almacenar atributos de soluciones recientemente visitadas, y su objetivo es explorar a fondo una región dada del espacio de soluciones. En ocasiones se utilizan estrategias de listas de candidatos para restringir el número de soluciones examinadas en una iteración dada o para mantener un carácter agresivo en la búsqueda.

La memoria a largo plazo almacena las frecuencias u ocurrencias de atributos en las soluciones visitadas tratando de identificar o diferenciar regiones. La memoria a largo plazo tiene dos estrategias asociadas: Intensificar y Diversificar la búsqueda. La intensificación consiste en regresar a regiones ya exploradas para estudiarlas más a fondo. Para ello se favorece la aparición de aquellos atributos asociados a buenas soluciones encontradas. La Diversificación consiste en visitar nuevas áreas no exploradas del espacio de soluciones. Para ello se modifican las reglas de elección para incorporar a las soluciones atributos que no han sido usados frecuentemente. Existen otros elementos más sofisticados dentro de *TS* que, aunque poco probados, han dado muy buenos resultados en algunos problemas.

2.4.2 Paralelismo inherente en la Colonia de Hormigas

La complejidad computacional del algoritmo secuencial impide su aplicación a problemas de gran envergadura. Por otra parte, *ACO* tiene características que lo hacen especialmente apropiado para su paralelización y distribución entre los diversos procesadores de una red de computadoras en un contexto asíncrono. En efecto, el método utiliza la interacción de muchos agentes relativamente simples llamados *hormigas*, pero básicamente independientes entre sí en cada tour, que cooperan intercambiando información para el logro de un objetivo común.

Cada hormiga va construyendo su propio tour, con la única restricción de no viajar a una ciudad ya visitada con anterioridad. En consecuencia, el paralelismo está implícito en el mismo algoritmo. Consecuentemente, el presente trabajo propone que cada procesador realice la computación del problema para un cierto número de hormigas, obteniendo resultados parciales que podrán ser transmitidos a los otros procesadores en forma

asíncrona, colaborando todos en la solución del problema global, pero sin necesidad de perder tiempo en sincronizar los procesadores, dado que la nueva información (útil para actualizar la matriz de feromonas) solo se usa si está disponible.

Con esta propuesta, el asincronismo elimina los tiempos muertos producidos por la espera en la sincronización de la comunicación, extremadamente perjudiciales cuando se trabaja con una red de computadoras cuyo tráfico no se puede controlar totalmente.

Podemos decir que los algoritmos genéticos como el basado en la Colonia de Hormigas tiene una estructura que se adapta perfectamente a la paralelización. De hecho la evolución natural es en si un proceso paralelo ya que evoluciona utilizando varios individuos. Los principales métodos de paralelización de AGs consisten en la división de la población en varias sub poblaciones. El tamaño y distribución de la población entre los distintos procesadores será uno de los factores fundamentales a la hora de paralelizar el algoritmo.

Existen varias formas de paralelizar un algoritmo genéticos como los ACH. La primera y más intuitiva es la global, que consiste básicamente en paralelizar la evaluación de los individuos manteniendo una población. Otra forma de paralelización global consiste en realizar una ejecución de distintos AGs secuenciales simultáneamente (estas dos formas de paralelización no cambian la estructura del algoritmo utilizado). El resto de aproximaciones sí cambian la estructura del algoritmo y dividen la población en sub poblaciones que evolucionan por separado e intercambian individuos cada cierto número de generaciones.

2.4.3 Algoritmo Paralelo

La idea general de la meta heurística de la ACO es construir de forma artificial una solución para los problemas de optimación combinatoria, el cual típicamente construye una hormiga por una secuencia de decisiones probabilísticas que se extiende a partir de soluciones que se van agregando a una nueva solución, por eso estas soluciones son derivadas de otras pequeñas. La secuencia de decisiones para construir una solución que puede ser visualizada como el camino que corresponde a un grafo de decisión. El objetivo es que una hormiga artificial busque un camino en un grafo de decisión que corresponda una buena solución.

Esto se hace con procesos interactivos donde las mejores soluciones encontradas por las hormigas pueden guiar a las hormigas siguientes en sus iteraciones. Por lo tanto las hormigas que tienen que buscar la mejor solución permiten dejar una marca en el camino correspondiente en el grafo de decisión con feromonas artificiales. El fin de las feromonas con mayor antigüedad no influyen en las iteraciones siguientes en los caminos más largos, ya que durante la actualización de las feromonas su porcentaje se va evaporando. Así de forma general el algoritmo de ACO es un proceso iterativo donde la información de las feromonas se traslado de una iteración a la siguiente. El procesos continua hasta cumplir con una condición de paro, por ejemplo, un cierto número de iteraciones o al encontrar alguna solución.

Ahora mostrare cual es el esquema general de ACO para poder dar solución al TSP. Recordando el *TSP* es un problema donde un agente debe visitar *n* ciudades con distancias d_{ij} entra cada par de ciudades $i, j \in [1:n]$, en este caso las ciudades son nodos donde las hormigas encuentra fuentes de alimento y el hormiguero será nuestro nodo/ciudad origen al cual deben retornar para cerrar el ciclo. Para el algoritmo secuencial he considerando que el esquema de la ACO tiene tres elementos importantes, el uso de las feromonas que sirven de información, que las soluciones se construyen por procesos iterativos y que debe haber un método que actualice las feromonas.

Para que la restricción del *TSP* se satisfaga en cuanto a que cada hormiga debe visitar solo una vez cada ciudad, asociamos a cada *hormiga k* una estructura de datos llamada lista $tabu_k$, la cual guarda información relativa de las ciudades ya visitadas por alguna hormiga k. Una vez que el tour haya sido completado la hormiga regresara al nodo/ciudad origen. La ya mencionada *lista tabú* se vacía o limpia para que la hormiga k este libre y pueda iniciar un nuevo tour. En este contexto se define como $tabu_k\ (n)$, al elemento n-esimo de la lista tabú de la hormiga *k* y como j_k (i) al conjunto de ciudades que aun no visito la *hormiga k* ubicada en la ciudad *i*.

De las *n* ciudades que las hormigas recorrerán, denotare como d_{ij} la longitud del camino entre las ciudades i, j; para el caso del problema simétrico del agente viajero, es la matriz de distancias $D = \{d_{ij},\ distancias\ entre\ la\ ciudad\ i\ y\ la\ ciudad\ j\}$. A la matriz de feromonas la denotare con $\tau = \{\tau(i,j)\}$ la cual será utilizada para consolidar la información que va siendo recogida por las hormigas. En este caso $\tau(i,j)$ representa la cantidad de feromonas que se van almacenando entre cada par de ciudades (i,j). Es típico que la matriz de feromonas sea inicializada con un porcentaje de feromonas en proporción a un tour típico corto el cual se obtiene con alguna otra heurística como lo es Dijkstra [14] para nuestro trabajo. Entonces definimos a τ_0 como:

$$\tau_0 = (n\ x\ L_{nn}) \qquad (1)$$

Donde L_{nn} es la longitud del tour típico obtenido inicialmente con Dijkstra, es necesario mencionar que el algoritmo no es sensible a este parámetro.

La intensidad de las feromonas del arco (i,j), es actualizada localmente mientras las hormigas construyen su propio tour, el cual es construido al moverse de una ciudad i a una ciudad j dejando a su paso un rastro de feromonas el cual se calcula de la siguiente manera:

$$\tau(i,j) = (1-\rho) \times\ \tau(i,j) + \rho \times\ \tau_0 \qquad (2)$$

donde ρ es un parámetro que puede entenderse como un coeficiente de evaporización de las feromonas en cada arco (*i,j*), este parámetro lo podemos tomar de $0 < \rho < 1$ el cual para este trabajo es de 0.1 una vez que las hormigas terminen su tour, se debe proceder a actualizar de forma global su recorrido con la siguiente ecuación:

$$\tau(i,j) = (1-\alpha) \times \tau(i,j) + \alpha \times \Delta\tau(i,j) \qquad (3)$$

donde $\alpha \in (0,1)$ y es el coeficiente de evaporización de las feromonas que determinan el grado de influencia de una buena solución en cada iteración donde se actualiza la matriz de feromonas.

Durante el proceso de búsqueda del mejor tour, cada hormiga posicionada en la ciudad *i* debe elegir el siguiente nodo (ciudad) *j* a visitar, este proceso se debe realizar respetando la condición del TSP que es la de no visitar una ciudad más de una vez, entonces las hormigas eligen la siguiente ciudad de las no visitas de su *lista tabú*, las hormigas elige con una probabilidad P_{ij} la ciudad más conveniente, la cual se determina por la relación directa que existe entre la cantidad de feromonas depositada en los arcos (*i, j*) y la distancia de los mismos, esto es:

$$P_{ij} = {\tau_{ij}}^{\alpha} \cdot {\eta_{ij}}^{\beta}$$

Donde $\eta_{ij} = {}^{1}/_{d_{ij}}$ proporciona información del nodo/ciudad que este más próxima al nodo/ciudad origen actual, los parámetros de α y β son usados para determinar la influencia relativa de las feromona.

Una descripción general del algoritmo paralelo basado en la ACO se inicia ubicando *m* hormigas en n ciudades de acuerdo a la regla de inicialización. Para nuestros experimentos se eligen las ciudades origen de manera aleatoria. Cada hormiga construye su propio tour, eligiendo la próxima ciudad a visitar aplicando la ecuación (4). Mientras construyen su tour, las hormigas actualizan feromonas al moverse de la ciudad *i* a la *j* según ecuación (2).

Cuando las hormigas volvieron a su nodo/ciudad origen, se calcula la mejor distancia hallada en el ciclo, y nuevamente se realiza la actualización de las feromonas de manera global considerando solo la mejor solución encontrada por cualquiera de las hormigas.
Como en general, el número de hormigas que conforman una colonia es bastante superior al número de computadoras personales disponibles, el presente trabajo propone que cada procesador realice la computación del problema para cierto número de hormigas, de la misma forma que se lleva a cabo en un contexto secuencial, obteniendo resultados parciales que al cabo de cierto número de ciclos serán trasmitidos a los otros procesadores (sin ninguna sincronización), con el fin de colaborar todos en la solución del problema global.

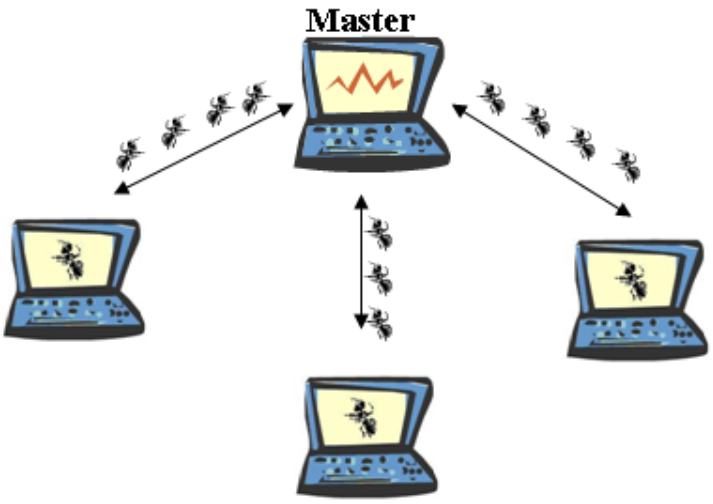


Figura 2.7 Estrategia de Paralelización.

El asincronismo aquí propuesto elimina los tiempos muertos producidos por la espera en la sincronización de la comunicación debido al importante cuello de botella que representa la sincronización en las implementaciones paralelas del Pseudocódigo 2, en especial, debido a las necesidades de sincronización. En consecuencia, el presente trabajo propone que en cada procesador las hormigas encuentren el mejor tour y los envíen al master para ser comparado con los resultados de los otros procesadores y asi el master elegirá el mejor tour encontrado.

Para levantar y controlar todos los procesos hijos donde realmente se hace el trabajo de optimización, se propone un master que comunique los parámetros iníciales y el número de ciclos que deben realizar cada uno de los procesos hijos, y espera de estos la comunicación de que han atendido al criterio de finalización (ver Figura 2.7) para dar por terminada la corrida.

El esquema del algoritmo paralelo basado en la optimización de la Colonia de Hormigas para el TSP puede expresarse como:

1. Inicio
   Levantar procesos
   Enviar parámetros a cada hijo
   Fin=Falso

2. Repetir mientras no sea Fin
   Recibir solución de los hijos
   Guardar mejor solución
   Si suficientes procesos terminaron
   Fin = Verdadero

3. Eliminar procesos hijos

Pseudocódigo 1: Proceso Master

1. Inicio
   Recibir parámetros
   Inicializar variables

2. Repetir mientras (Ciclo_actual < NCMAX)
   Mover hormigas hasta completar tour
   Calcular distancia recorrida para cada hormiga
   Actualizar la matriz de feromonas
   Guardar camino más corto
   Ciclo_actual = Ciclo_actual + 1

3. Enviar mejor solución al master

Pseudocódigo 2: Proceso Hijo

3. Plataforma Paralela Desarrollada en Java

La búsqueda de soluciones a problemas reales generalmente requiere realizar una gran cantidad de cálculos. Esto ocasiona que la obtención de resultados sea demasiado tardada. La única solución para este problema se ofrece por el Paralelismo, que permite realizar varias operaciones a la vez, favoreciendo la reducción del tiempo que se requiere para ejecutar una tarea.

Desde hace varios años el hardware no es más una limitante. Se ha vuelto más importante el desarrollo de herramientas de software que permitan sacar ventaja de las posibilidades de hardware.

En cuanto a hardware, existen diversas arquitecturas paralelas hoy en día, y la librería de funciones más usada para el desarrollo de aplicaciones paralelas es MPI. Esta librería posee limitantes acerca del manejo de memoria, y es responsabilidad del desarrollador construir los mecanismos necesarios si desea emplear medios de sincronización complejos, tal como una memoria compartida.

El lenguaje Java cada día gana mayor aceptación en la comunidad científica e industrial, extendiendo no sólo su uso, sino también las herramientas diseñadas para este lenguaje. Java ofrece mecanismos de concurrencia, manejo de funciones de bajo nivel y tecnologías para desarrollar aplicaciones distribuidas. Estas y otras características que posee Java, lo hacen un excelente candidato para desarrollar software paralelo.

3.1. Paralelismo en Java

El lenguaje Java está ampliamente considerado como inadecuado para tareas de cómputo intensivo. La razón obvia de esto recae en el pobre desempeño de los programas de Java, que corren más lento comparado contra sus contrapartes de C y Fortran [18]; pero a pesar de que Java no es tan eficiente como Fortran optimizado o C, la velocidad de Java es mejor de lo que su reputación sugiere [19]. Existen varias razones por las cuales Java es el lenguaje ideal para algunas de las aplicaciones a desarrollar [5]Erick PR.

Su popularidad y amplia aceptación han llamado la atención de la comunidad científica e ingenieril, y ha permitido el desarrollo de librerías y herramientas adaptadas al cómputo paralelo de alto desempeño.

Lo más importante, es que la independencia de Java a la arquitectura o al sistema operativo, permiten la distribución de los programas en plataformas heterogéneas. Esto tiene el potencial de llevar las aplicaciones paralelas o distribuidas más allá de clústers especializados de máquinas homogéneas, usadas tradicionalmente en cómputo de alto desempeño.

Finalmente, compiladores JIT (*Just In Time*) que traducen bytecode de Java a código nativo de la plataforma, han tenido avances significativos para mejorar el desempeño, y algunos

de estos compiladores han logrado alcanzar un 2/3 de la velocidad de código en C. El proyecto Ninja de IBM ha demostrado que cuando se compila código específicamente para arquitecturas paralelas, se puede alcanzar entre un 80% y un 100% del desempeño logrado por código en Fortran optimizado. Combinado con técnicas cuasi estáticas, el código de Java puede ser tan veloz, como el de C o Fortran [18].

Por otra parte, existe un paralelismo implícito en la arquitectura de la Máquina Virtual de Java. Esto se da en la implementación de la Máquina Virtual, la cual hace uso intensivo y natural de los hilos. Si la plataforma de hardware posee las capacidades (como tener instalado más de un procesador, o capacidades de administración multihilos), al igual que el sistema operativo (usando núcleos optimizados para ejecutar tareas en paralelo), la Máquina Virtual podrá hacer uso de estos recursos, logrando cierto paralelismo dentro de la misma computadora.

3.2. Modularidad en Java

Java es un lenguaje por naturaleza orientado a objetos puro [16]. Los paquetes de Java son un mecanismo mediante el cual se organizan las clases de Java en Espacios de Nombre (Namespaces). Esta es la forma en que Java ofrece la modularidad. Los paquetes en Java son identificados de manera única mediante nombres completamente calificados.

Las clases y las interfaces son los únicos tipos de datos que se exportan en los paquetes. El CLASSPATH de Java existe por una razón muy práctica, ya que ayuda a localizar módulos. En Java, cuando se exporta un paquete, por ejemplo java.net, puede surgir la duda de dónde se localiza físicamente el paquete. Técnicamente, el CLASSPATH de Java juega un rol importante para encontrar clases y paquetes. Todas las clases que se pueden localizar directamente por el CLASSPATH se dice que pertenecen a un paquete llamado default. Es por esta razón que se dice que el CLASSPATH de Java sirve para localizar paquetes.

El CLASSPATH de Java puede interpretarse como una combinación de espacios de nombres. Cada entrada del CLASSPATH define un espacio de nombres con múltiples módulos definidos, y todo el CLASSPATH combina todos los nombres de espacio, con todos los módulos incluidos. Si dos módulos tienen el mismo nombre, ambos se mezclan en un módulo único, y las características de ambos se mezclan.

Es erróneo creer que el CLASSPATH es usado solamente para la búsqueda estática de clases, y que es usado solamente para la compilación estática. Esto no es cierto, ya que las clases se cargan tardíamente. En tiempo de ejecución, Java necesita una manera de determinar la ubicación de las clases. Java mantiene, de hecho, dos CLASSPATH, uno usado para la compilación estática y el otro para la búsqueda en tiempo de ejecución de la máquina virtual. Los dos CLASSPATH no tienen que ser necesariamente los mismos.

3.3 Escalabilidad en la Máquina Virtual de Java

La Máquina virtual de Java toma los recursos que le ofrece el hardware host, parte de esos recursos los emplea en el funcionamiento de la misma Máquina Virtual y el resto los pone a disposición de las aplicaciones que ejecuta. Disponer de una mayor cantidad de recursos en el hardware host, es decir, una estalación vertical, ofrece una mayor cantidad de recursos para las aplicaciones que ejecuta la Máquina Virtual y a ella misma. Sin embargo, algunos elementos de la Máquina Virtual de Java se pueden mejorar para aprovechar aún más esos recursos. Uno de esos elementos, que puede acelerar los tiempos de ejecución, es el Garbage Collector.

3.4. Localidad en Java

La creciente disparidad en cuanto a velocidades entre el procesador y la memoria, está motivando una fuerte necesidad de optimizar la memoria de programas. Numerosos investigadores han dedicado esfuerzos a estudiar la localidad de la información, su importancia y su efecto en el desempeño, y han investigado la interacción entre el programa y varias arquitecturas de procesadores y de memorias. A pesar de que el trabajo en optimizar las aplicaciones que usan arreglos ha tenido mejoras, las aplicaciones que hacen uso intensivo de apuntadores y manejo de memoria dinámica, todavía representan un reto serio. A pesar de que mucho trabajo se ha realizado en esta última área, los investigadores indican que aún hay mucho por hacer para llegar a niveles deseables de desempeño [17].

La creciente popularidad de lenguajes como Java en una amplia variedad de plataformas, que van desde sistemas embebidos hasta servidores, presenta un particular reto desde dos puntos de vista:

Los programas de Java tienden a hacer uso intensivo de la reserva de memoria, lo que presiona en el subsistema de memoria.

La naturaleza dinámica de Java, debido a características como la carga dinámica de clases, hace impráctico aplicar optimizaciones, haciendo uso de análisis intensivo del programa y las clases que usa.

Existen algunas técnicas que han mostrado ser eficientes, que recaen sobre la visión macroscópica del sistema, en vez de la particularidad de las aplicaciones. Una de estas técnicas se basa en la identificación de las clases que más frecuentemente se instancian, que se conocen como tipos prolíficos, de un programa dado, y tratan de colocar todas las instancias de esos tipos prolíficos juntas, al momento de la reserva de memoria. Otras técnicas, trabajan al momento que se activa el Garbage Collector, haciendo más efectivo el trabajo de este.

3.5 Java y MPI

Con el evidente éxito de Java como lenguaje de programación y su inevitable conexión con el cómputo paralelo y distribuido, la ausencia de una librería bien diseñada y específica para Java para el paso de mensajes, conllevaría a aplicaciones divergentes, no portables. El "Message-Passing Working Group" del foro Java Grande se formó en otoño de 1998 como respuesta a la aparición de varias APIs para el paso de mensajes.

Algunas de esas implementaciones se encuentran disponibles desde 1997 trabajando sobre plataformas Linux, Solaris, Windows NT, Irix, AIX, HP-UX y MacOS, al igual que en plataformas paralelas tales como IBM SP-2 y SP-3, Sun E4000, SGI Origin-2000, Fujitsu AP3000, Hitachi SR2201 y otras[VLA01]. Una meta inmediata para el grupo fue la discusión y acuerdo de un API común para librerías MPI, para el paso de mensajes en Java. El API que se definió como parte del trabajo del grupo se nombró MPJ (Message Passing for Java). Una de las implementaciones más completas de esta API se llama mpiJava [16]

El estándar de MPI es explícitamente orientado a objetos. Las librerías de C y Fortran hacen uso de objetos opacos que pueden ser manipulados por manejadores de objetos que se obtienen con los constructores de tales objetos, y pasando esos manejadores a las funciones indicadas. La librería de C++, según se específica en el estándar de MPI-2, agrupa esas clases en una jerarquía y define muchas de las funciones de la librería como funciones miembro de las clases. La especificación de MPJ sigue este modelo, tomando la estructura de las clases directamente de C++. El propósito de esta acción es proveer una estandarización inmediata y natural para programas que hagan el uso de paso de mensajes en Java, al mismo tiempo que provee una base para la conversión de código entre C, C++, Fortran y Java.

La funcionalidad de mpiJava descansa sobre envoltorios (funciones que hacen llamado a otras funciones) a las implementaciones nativas de MPI para la plataforma objetivo. Mientras esta parece ser una solución razonable para muchos casos, tiene algunas desventajas:

Una instalación de dos fases. Primero se debe encontrar instalada una versión de MPI en la computadora antes de poder instalar mpiJava. Durante el desarrollo de mpiJava, se encontraron conflictos entre el comportamiento de la Máquina Virtual de Java y la librera de MPI en ejecución . La estrategia de crear envoltorios va en contra, en cierto sentido, con la ideología de Java de escribir una vez, ejecutar donde sea, ya que depende totalmente de la implementación de MPI propia de la plataforma.

De forma ideal, los dos primeros problemas podrían ser resueltos por los proveedores de las librerías nativas de MPI. Ellos podrían ofrecer una interfaz de Java junto con sus implementaciones de MPI de C y Fortran. Esto tendría la ventaja de que ellos serían los más indicados para resolver las disparidades que existen entre sus implementaciones y el comportamiento de la Máquina Virtual de Java. Al final de cuentas, las implementaciones más veloces del mercado de MPJ, serían las que ellos mismos podrían ofrecer.

En C y Fortran, cualquier desplazamiento dentro de un arreglo se puede manipular como otro arreglo. Esto no sucede en Java, por ello, los métodos incluyen un parámetro para indicar el desplazamiento que se quiere emplear como inicio del arreglo.
En las implementaciones de C y Fortran, las funciones regresan valores de errores. En mpiJava se emplea el mecanismo de las excepciones para reportar errores. En las funciones de mpiJava generalmente se omite el paso del parámetro que indique el tamaño de los arreglos, ya que eso se puede obtener directamente del arreglo con su propiedad length.

3.6 Implementación de la Plataforma

Se mencionan las razones por las cuales se decidieron ciertas formas de implementar algunos mecanismos, y en la mayoría de los casos se da una explicación sobre cómo funcionan esos mecanismos.

3.6.1 Comunicación

La plataforma se conforma por un conjunto de computadoras, en las cuales se estará ejecutando el Demonio Esclavo, el Demonio Maestro, o ambos. Al iniciar el Demonio en una computadora determinada, el Demonio se encarga de anunciar al resto de la Plataforma que el nodo se está agregando. Recuerde que para la Plataforma, un nodo es la instancia de un Demonio Esclavo.

Desde el punto de vista del usuario, los nombres de los nodos no es de relevancia, ya que el usuario no sabe en qua nodo se ejecutará alguna de sus clases. Sin embargo, para el envío de mensajes, es necesario conocer el destinatario, y en ocasiones, el origen. Para ello, cada uno de los nodos debe tener, para efectos de la plataforma, un nombre único. Inicialmente, al arrancar el Demonio, éste intentará tomar su nombre directamente del nombre de la computadora, para evitar duplicidades. Estas duplicidades en ocasiones no pueden salvaguardarse, y puede deberse a que diversas causas, siendo algunas de ellas, a que no estén bien configurados los nombres de las computadoras o bien que se levante más de un Demonio en la misma computadora.

Para resolver este problema, al momento de iniciar algún Demonio, el Demonio intenta obtener el nombre de la computadora. Si el nombre que obtiene es localhost, automáticamente el Demonio renombra (solo a nivel de la aplicación) el nodo, asignándole como nombre la cadena NODE XYZ, donde XYZ es un número aleatorio.

3.6.2 Paso de Mensajes

La plataforma usa un sólo punto de comunicaciones, que sea empleado por todas las funcionalidades de la plataforma: la memoria, la administración de la plataforma y los mensajes de usuario. Para ello, es necesario que el mensaje que se envíe, ofrezca una estructura que permita distinguir el origen, el destino y la finalidad del mensaje.

En la plataforma, los mensajes se representan por la clase Message, quien realiza el envío y recepción de bufferes con la siguiente estructura:

- Source Node, Nodo Origen (50 bytes): Almacena el nombre del nodo origen en texto.
- Destination Node, Nodo Destino (50 bytes): Almacena el nombre del nodo destino en texto.
- Source Task, Tarea Origen (4 bytes, un entero): Almacena el identificador de la tarea que envía el mensaje. Tiene el valor de 0 si quien genera el mensaje es algún elemento de la plataforma.
- Destination Task, Tarea Destino (4 bytes, un entero): Almacena el identificador de la tarea que debe recibir el mensaje. Tiene el valor de 0 si quien debe recibir el mensaje es algún elemento de la plataforma.
- Platform Tag,Bandera Plataforma: (1 byte): Tendrá un valor de 1 si el mensaje es exclusivamente para la plataforma, y un valor de 0 si el mensaje es para el usuario. Si el mensaje es para la plataforma, ésta se encargará de que el mensaje nunca le llegue al usuario.
- Message, Mensaje (4 bytes, un entero): Es un entero que identifica el contenido del mensaje. Si el mensaje es del usuario, ´este puede emplear este campo como desee.

3.6.3 Sockets para Comunicación

Para realizar la comunicación a través de la plataforma, se emplea un socket UDP en multicast. El uso del socket UDP implica que la entrega de paquetes no se garantiza, así como tampoco el orden. Sin embargo, la plataforma está orientada a un cluster o una red de computadoras LAN. Esto supone que las computadoras se encuentran en el mismo segmento de subred, y de hecho se deben encontrar de esta manera, de lo contrario, el socket UDP en multicast no funcionaría. Si las computadoras no se encuentran separadas por un enrutador, la probabilidad de perder un paquete o de recibirlos en desorden, es prácticamente despreciable.

El empleo de sockets UDP trae como ventaja la simplicidad y la rapidez del protocolo, además de que en TCP no existe una implementación en multicast. Si se hubieran empleado sockets TCP, en una red con N nodos, cada nodo debería crear N -1 sockets para tener comunicación con cada uno de los nodos restantes de la plataforma. Usando sockets UDP podría suceder algo similar, pero se tiene a disposición los sockets UDP en multicast.

Las direcciones IP de clase D, que se encuentran entre 224.0.0.0 y 239.255.255.255 inclusive, se emplean para multicast. Independientemente de la dirección asignada a cualquier interfaz de red, se puede emplear alguna dirección en el rango anteriormente indicado en alguna aplicación. La dirección en el rango indicado, más un número de puerto estándar UDP, y se pueden recibir y enviar mensajes en multicast.
En multicast, más de un proceso, o nodo, pueden emplear la misma dirección IP. Con ello, cualquier envío que se realice a tal dirección en multicast, será recibido por todos aquellos que estén empleando esa dirección.

3.7 Sincronización de la Plataforma

Las diversas plataformas de hardware paralelas ofrecen mecanismos de sincronización. Según el tipo de plataforma, los procesadores pueden estar sincronizados mediante una unidad de control a nivel de instrucción, o bien pueden sincronizarse a intervalos regulares en ciertos puntos de ejecución.

Los puntos de sincronización son una construcción común en las herramientas de software para el desarrollo de aplicaciones distribuidas y paralelas. A nivel de software, un punto de sincronización consiste en lograr que los diferentes procesos en ejecución alcancen un punto en la ejecución en el cual uno a uno de los procesos se detendrán hasta que todos ellos lleguen ahí, momento en que su ejecución continúa.

En el diseño de esta plataforma, se ha incluido una construcción para crear puntos de sincronización, bajo el nombre de Barriers. Estos puntos de sincronización ocurren en dos etapas. En la primera etapa, se busca la sincronización a nivel de nodo. En cada nodo puede ejecutarse más de un proceso de usuario, por lo que el nodo llegará al punto de sincronización cuando todos los procesos que está ejecutando se sincronicen.

En la segunda etapa, toda la plataforma logrará la sincronización cuando todos los nodos hayan completado la primera etapa de sincronización. El nodo maestro será el responsable de recibir las notificaciones de cada nodo sobre su sincronización. Tras haber recibido las notificaciones de sincronización de todos los nodos, el nodo maestro enviará un mensaje de liberación a todos los nodos y la ejecución en cada uno de ellos continuará.

4. Análisis y diseño

En este capítulo se presenta la descripción del problema a resolver. Partiendo de la descripción del problema, se identifican los primeros Requerimientos de Uso, que se emplean como base para el resto del análisis. Todo el análisis está compuesto por los Requerimientos de Uso más los diagramas UML de Casos de Uso, Actividades, Clases y Secuencia.

El Algoritmo Paralelo Asíncrono basado en la Colonia de Hormigas se ejecuta en un cluster de computadoras Linux, aunque por la naturaleza de Java, debe poder migrar con mínimo o ningún esfuerzo a otras plataformas. Como se ha desarrollado en Java sigue el paradigma orientado a objetos y el uso de mensajes en la Plataforma nos permite olvidarnos del protocolo de los mismos, sólo nos preocupamos por decidir cual tipo de mensaje usaremos y en qué momento se usará cada uno, es decir, comunicación síncrona y asíncrona.

El análisis y Diseño del Algoritmo paralelo basado en la Colonia de Hormigas se realizo en UML y JUDE, siguiendo las siguientes etapas:

- **Requerimientos de uso**. Se partirá de los requerimientos de uso, que son la descripción de las funcionalidades que se espera del sistema.

- **Casos de uso**. Se generarán los diagramas de casos de uso a partir de los requerimientos de uso. Los diagramas de Casos de uso son el “contrato gráfico” que denotará las funcionalidades mínimas que debe cumplir el sistema.

- **Clases.** Se generarán los diagramas de clases. Partiendo de los diagramas de actividades, se pueden generar de manera más precisa los diagramas de clases. Desde los diagramas de Casos de uso, se podrán detectar las primeras clases, que pueden surgir directamente de los actores. Con los diagramas de actividades pueden surgir clases directamente de alguna actividad, aunque es más común que sean los diagramas de actividades las que arrojen los métodos que deben cumplir las clases.

- **Secuencia.** Una vez comenzados los diagramas de clases, se puede comenzar a generar los diagramas de secuencia. En los diagramas de secuencia se termina de refinar el comportamiento que debe cumplir cada clase.

- **Componentes.** Se describirá cuales son los elementos físicos del sistema y las relaciones que se requieran para el funcionamiento.

En las siguientes secciones se muestra el análisis que se realizó para el Algoritmo Paralelo, con las etapas anteriormente descritas.

4.2 Diseño del Algoritmo Paralelo

4.2.1 Requerimientos de Uso

Los requerimientos de uso son la descripción de las funcionalidades que se espera que cumpla el Algoritmo Paralelo Asíncrono basado en la Colonia de Hormigas. Esta lista de funcionalidades que el usuario del sistema espera que cumpla, servirá como base para detectar las tareas adicionales que se tienen que realizar para completar tales funcionalidades.

Requerimientos de Uso:

- El sistema debe permitir la lectura de archivos, para obtener la información de los grafos de los que se desea obtener un recorrido.
- El sistema debe permitir la distribución de clases Java para su ejecución en cualquier nodo que forme parte de la red de la Plataforma.
- El sistema debe permitir conformar una red de ejecución sobre un cluster, utilizando la Plataforma.
- El sistema debe contar con un elemento centralizado que permita el control de la plataforma completa, encargado de tareas centrales, como la distribución de procesos en los nodos para encontrar el camino mas corto.
- El sistema debe usar clases de Java que permitan ejecutarse en forma paralela en diferentes nodos.
- Debe ser capaz de encontrar una solución para Grafos Conexos con las heurísticas de la Colonia de Hormigas.
- Debe ser capaz de encontrar el camino mas corto de cualquier grafo por cada proceso hijo que lance.
- Cuando los ciclos de ejecución se terminen, el Máster debe ser capaz de elegir la mejor solución encontrada por los Slaves, y mostrar esa única solución

4.2.2 Casos de Uso

Partiendo de los Requerimientos de uso, generamos el diagrama de Casos de uso, para el cual se han detectado 3 actores:

- **Usuario**: el primer actor que se detectó es el usuario que interactúa con el sistema solicitando la ejecución de las clases de Java y de la Plataforma.

- **Demonio Maestro:** dentro del sistema, se detectó que se necesitaba un elemento que se encargue de vigilar y administrar la Plataforma. Este elemento, que es una pieza de código, se modeló como un actor por conveniencia, ya que sus actividades las realizará sobre la Plataforma.

- **Demonio Esclavo:** al igual que con el demonio maestro, es necesario un elemento que se encargue de iniciar la ejecución de las clases del usuario. Por conveniencia, este elemento es una pieza de código que se modela como actor.

El diagrama de Casos de Uso para el Algoritmo Paralelo:

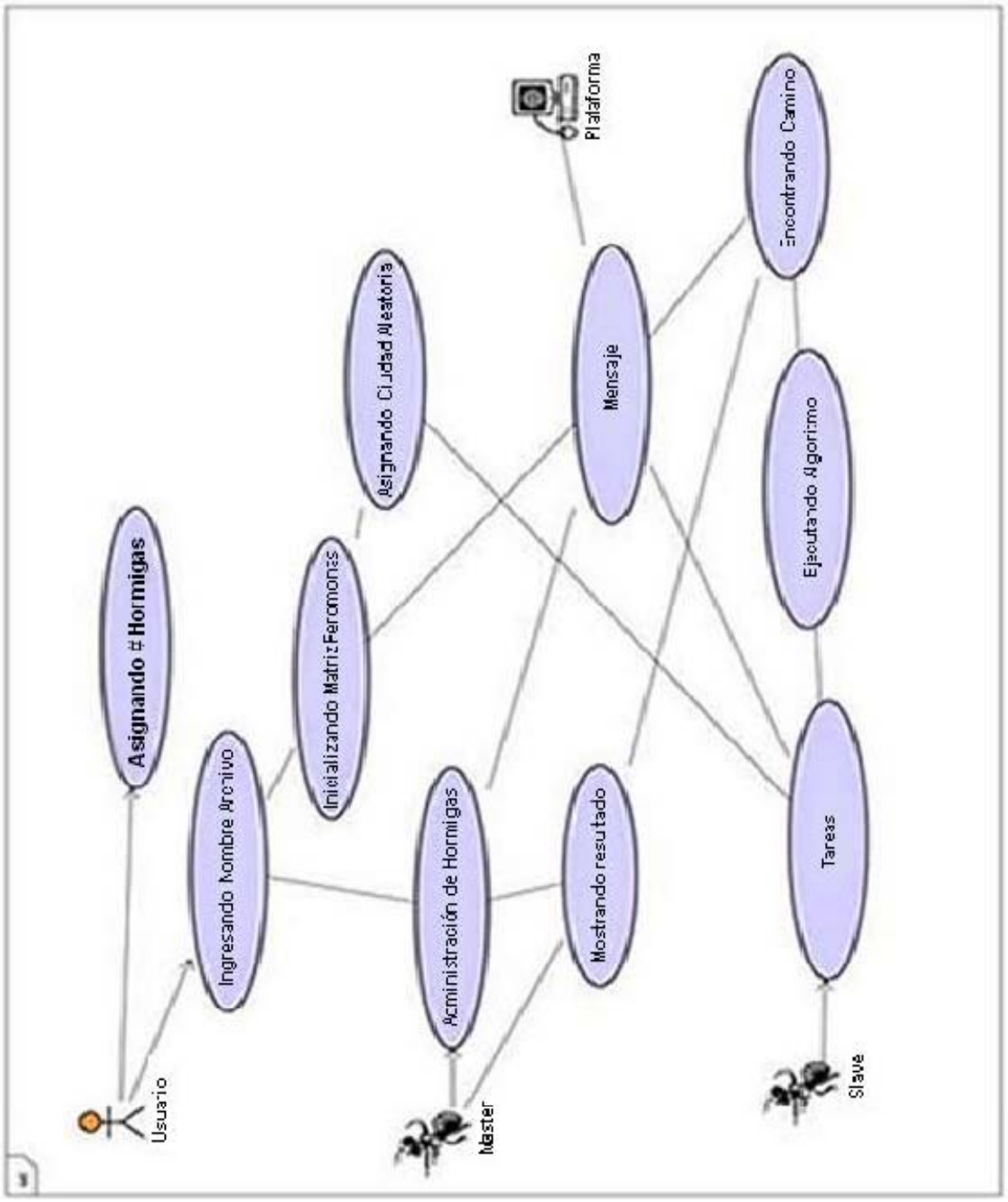


Figura D7. Diagrama de Casos de Uso del Algoritmo Paralelo

En este diagrama se puede observar que a través del actor *Usuario* se proporciona la asignación de los nodos iniciales ya que al ingresar el nombre del archivo donde se

encuentra la matriz de distancias el algoritmo da comienzo con la comunicación entre el actor *Slave* Hormiga y el actor S*istema*, que trabajan en conjunto para encontrar el mejor tour, para posteriormente enviarle los resultados al actor *Master* hormiga, el cual determina la mejor solución para el grafo en cuestión.

4.2.3 Diagrama de Clases del Algoritmo Paralelo

Las diferentes clases implementadas para la solución del problema del agente viajero a través de la heurística de la colonia de hormigas, deben de permitir la comunicación entre hormigas de forma paralela, para que la clase MasterHormiga elija el mejor tour encontrado, este proceso lo realiza la clase *SlaveHormiga* que se ejecuta en diferentes procesadores, gracias a la clase *definicionDColonia* donde se determina el número de esclavos y por lo tanto el número de procesadores a utilizar.

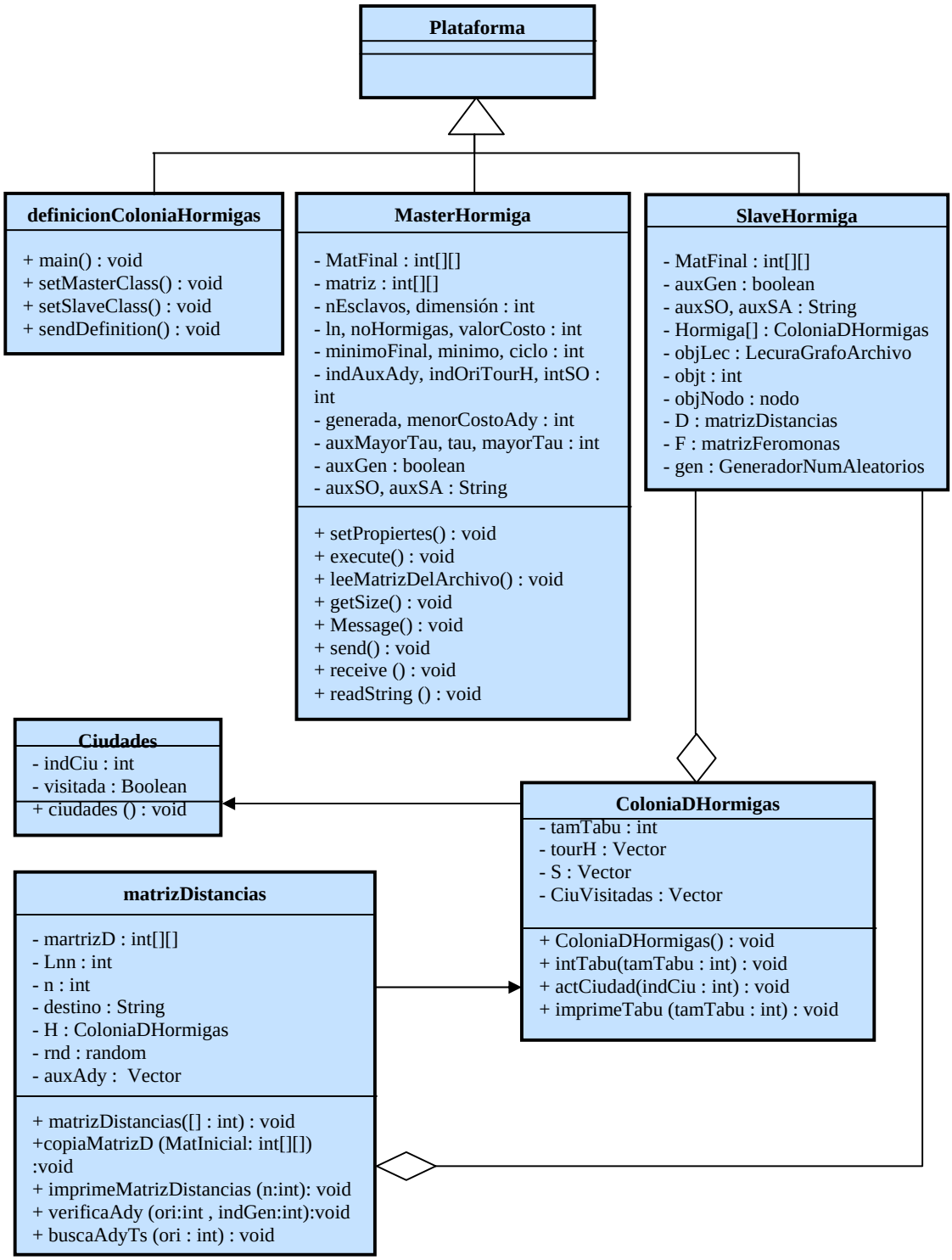
La clase *SlaveHormiga* recibe una copia de la matriz de distancias de la clase *MasterHormiga* la cual se obtiene al leer el grafo del archivo con la clase *lecturaGrafoArchivo* y así los procesos esclavos comienzan a buscar el mejor tour.

La actualización de la matriz de feromonas se hace en cada procesador, a través de los métodos de la clase *matrizFeromonas*, esta matriz es actualizada por las hormigas, las cuales son determinadas por la clase *ColoniaDHormigas* esta clase asigna el número de hormigas por procesador, así como la *matrizDistancias* que también debe existir por procesador para ser utilizada por las hormigas, para encontrar el mejor tour.

El algoritmo paralelo basado en la colonia de hormigas, necesita de un parámetro inicial, este parámetro debe ser un camino corto que visite todas las ciudades, y se debe obtener con otra heurística, que como en el algoritmo secuencial es Dijsktra [14], en este caso también. El manejo del grafo para ir de un nodo a otro se logra a través de las clases *Nodo*, *Cola* y *Ciudades*, que también son clases usadas por la clase *tourTipico* para la implementación del algoritmo Dijsktra.

El siguiente diagrama muestra todas las clases que hacen funcionar el sistema que resuelve los grafos para BR17, OLIVER30 y EILON50, que se mostrarán más adelante.

El diagrama de Clases para el Algoritmo Paralelo se muestra en la siguiente figura:


Plataforma
definicionColoniaHormigas
+ main() : void
+ setMasterClass() : void
+ setSlaveClass() : void
+ sendDefinition() : void
MasterHormiga
- MatFinal : int[][]
- matriz : int[][]
- nEsclavos, dimensión : int
- ln, noHormigas, valorCosto : int
- minimoFinal, minimo, ciclo : int
- indAuxAdy, indOriTourH, intSO : int
- generada, menorCostoAdy : int
- auxMayorTau, tau, mayorTau : int
- auxGen : boolean
- auxSO, auxSA : String
+ setPropiertes() : void
+ execute() : void
+ leeMatrizDelArchivo() : void
+ getSize() : void
+ Message() : void
+ send() : void
+ receive () : void
+ readString () : void
SlaveHormiga
- MatFinal : int[][]
- auxGen : boolean
- auxSO, auxSA : String
- Hormiga[] : ColoniaDHormigas
- objLec : LecuraGrafoArchivo
- objt : int
- objNodo : nodo
- D : matrizDistancias
- F : matrizFeromonas
- gen : GeneradorNumAleatorios
Ciudades
- indCiu : int
- visitada : Boolean
+ ciudades () : void
ColoniaDHormigas
- tamTabu : int
- tourH : Vector
- S : Vector
- CiuVisitadas : Vector
+ ColoniaDHormigas() : void
+ intTabu(tamTabu : int) : void
+ actCiudad(indCiu : int) : void
+ imprimeTabu (tamTabu : int) : void
matrizDistancias
- martrizD : int[][]
- Lnn : int
- n : int
- destino : String
- H : ColoniaDHormigas
- rnd : random
- auxAdy : Vector
+ matrizDistancias([] : int) : void
+copiaMatrizD (MatInicial: int[][])
:void
+ imprimeMatrizDistancias (n:int): void
+ verificaAdy (ori:int , indGen:int):void
+ buscaAdyTs (ori : int) : void

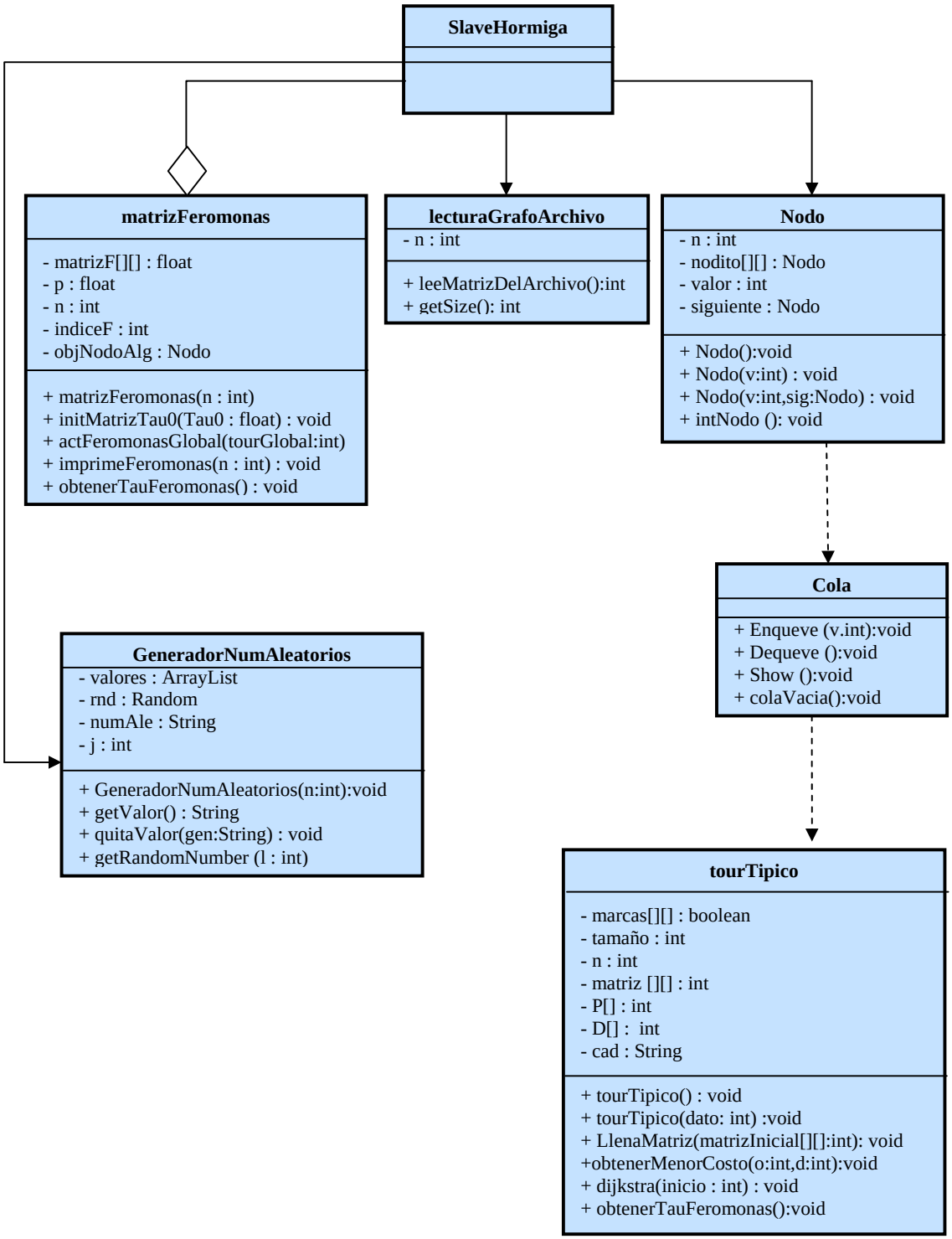


Figura D8. Diagrama de Clases

4.2.4 Diagrama de Secuencia

El algoritmo paralelo se inicia con la *deficionColoniaDhormigas* que es donde se determinan el número de Slave hormigas. El método *setMasterClass*() declara un sólo Master Hormiga que enviará la información inicial y recibirá la solución final. Para determinar cuántos Slave hormiga serán se utiliza de método *setSlaveClass*(). Toda esta información es enviada a la plataforma con *sendDefinition*(). Estas secuencias se muestran en la figura D9.

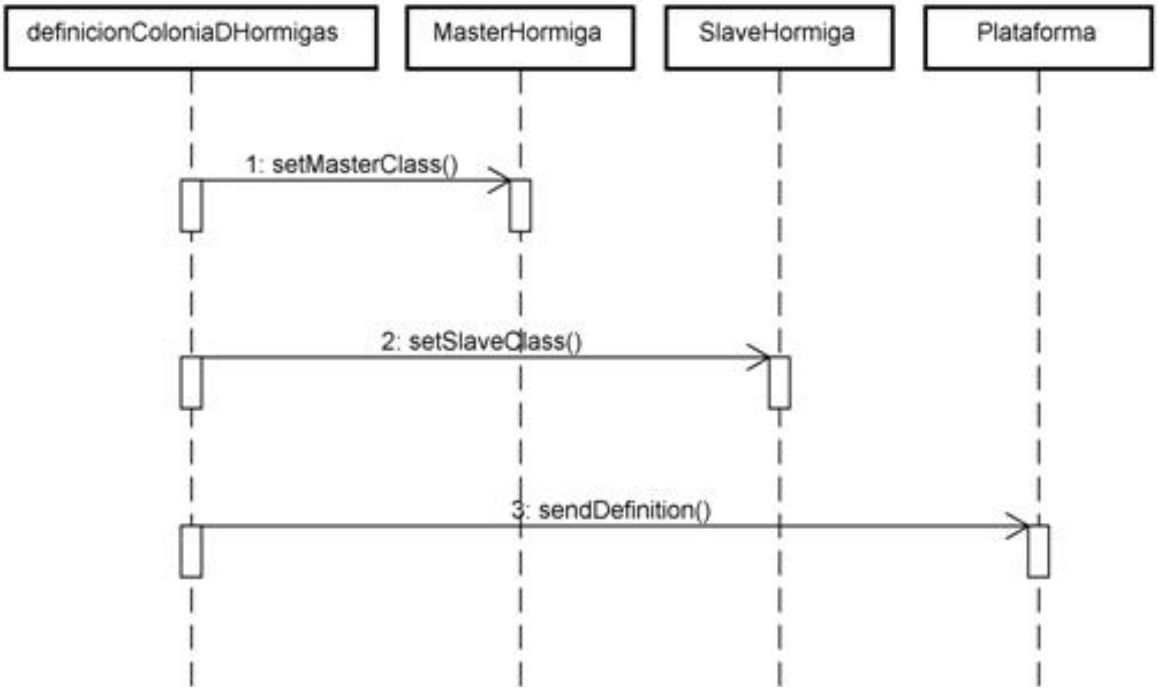


Figura D9. Diagrama de Secuencia del Algoritmo Paralelo demonizado de arranque

Una vez enviada la definición de clases a la Plataforma, esta pone a disposición sus métodos para comunicación paralela, siendo *setProperties()* el primer método que permite crear un objeto de la clase *PlataformBind* la cual permite toda la comunicación.

Esta comunicación se manipula através del método *execute()* donde se implementa el método *Message(),* con el cual se envía la copia de la matriz de distancias a todos los procesadores y ponemos al Master Hormiga en espera de la mejor solución por procesador.

Esta secuencia de comunicación inicial entre slave hormigas, master hormiga y Plataforma se puede visualizar en la siguiente figura:

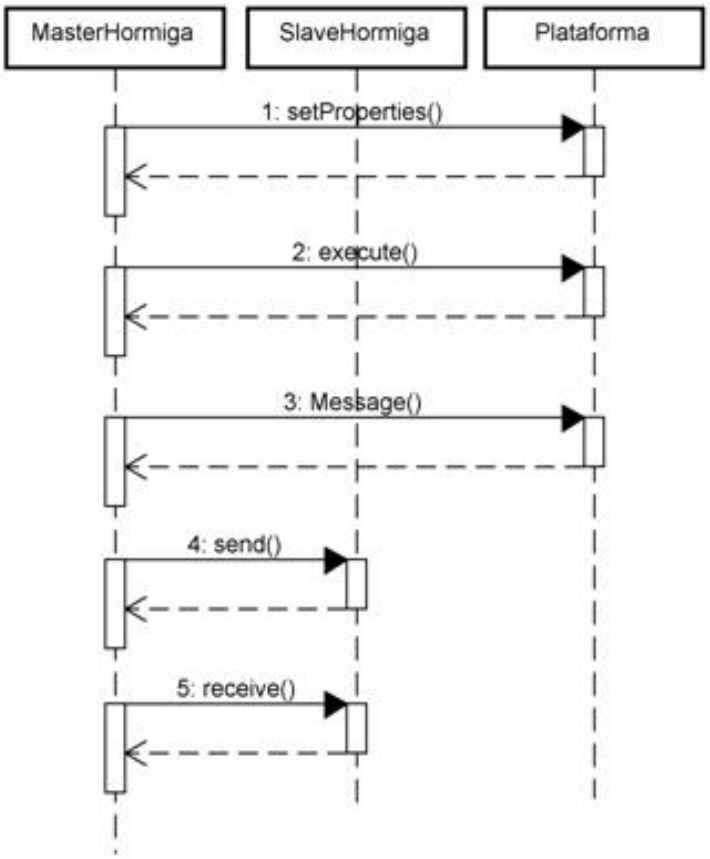


Figura D10. Diagrama de Secuencia donde se muestra la secuencia que sigue la plataforma

En cada procesador habrá un conjunto de hormigas que son asignadas por la clase *ColoniaDHormigas*, además de conservar el tour de cada hormiga con el método *tourH()*, este tour es actualizado cada vez que las hormigas avanzan a otro nodo (ciudad) adyacente a la ciudad anterior, lo cual se verifica con el método *buscaAdy()*.

Como ya se sabe el camino que las hormigas deben elegir es el más corto e ir recordando que ciudades ya han sido visitadas, esto se logra a través de *obtenerTauFeromonas()* que inicializa la matriz de feromonas con un porcentaje que se utiliza como cantidad de feromonas depositadas en el camino por las hormigas.

La elección de distancias cortas depende de la cantidad de feromonas depositadas en el camino por otras hormigas que han pasado, pero cada hormiga puede verificar de dos formas si ese camino es el que mas le conviene, uno es por la cantidad de feromonas sobre el valor del costo del camino, esto es un parámetro que determina si el camino próximo a elegir es el mas corto, siendo el método *obtenerCostoDistancia()* el encargado de esta disertación.

La siguiente figura se muestra la secuencia anterior.

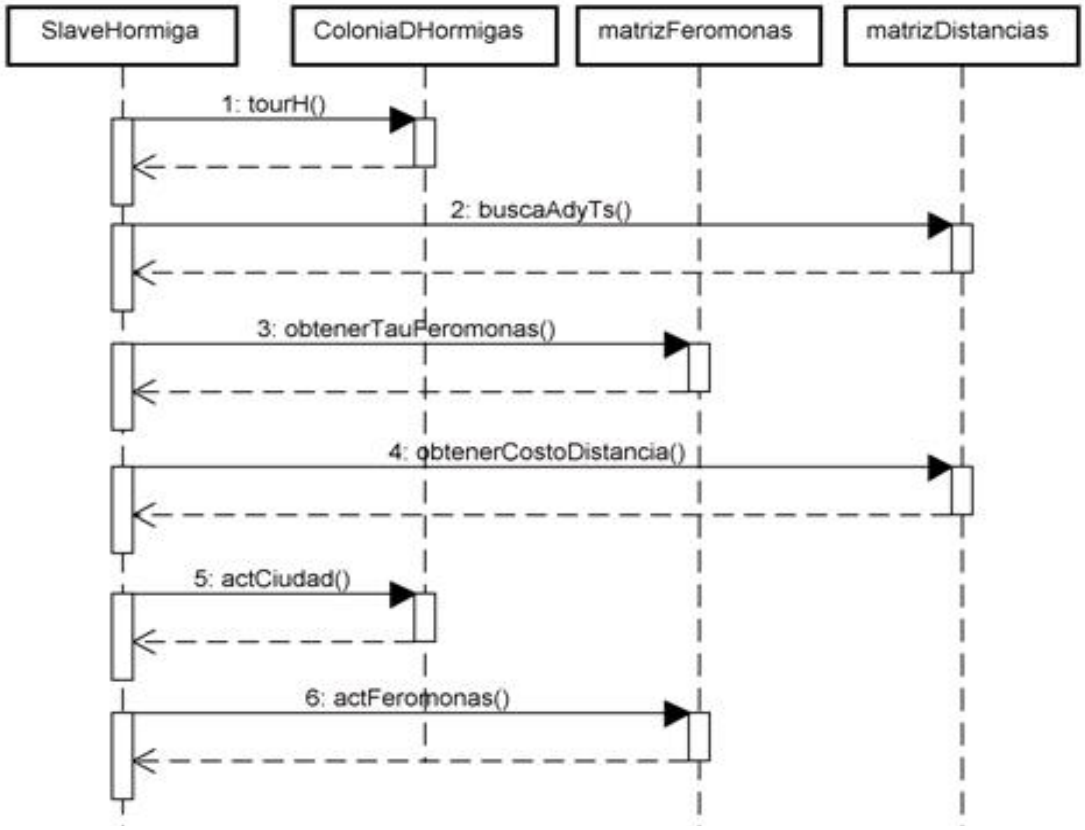


Figura D12. Diagrama de Secuencia donde se muestra el funcionamiento del Algoritmo en paralelo

La comunicación entre las hormigas de cada procesador se da al final, cuando envían la mejor solución encontrada y en la actualización global la matriz de feromonas, esto se hace a través de *receive()*, ya que el master hormiga se quedó en espera de las soluciones de los slave hormiga, que tienen que obtener el tour final con *tourH()* y el costo con *obtenerCostoDistancia()*, el proceso de selección entre todos los costos y toures enviados por los slave hormiga, lo realiza el master y sólo imprime el resultado final, así como el tiempo que demoró el proceso en terminar, esto es con el método final *imprimeMenorCosto()*.

La figura se muestra la última secuencia de la implementación del algoritmo paralelo.

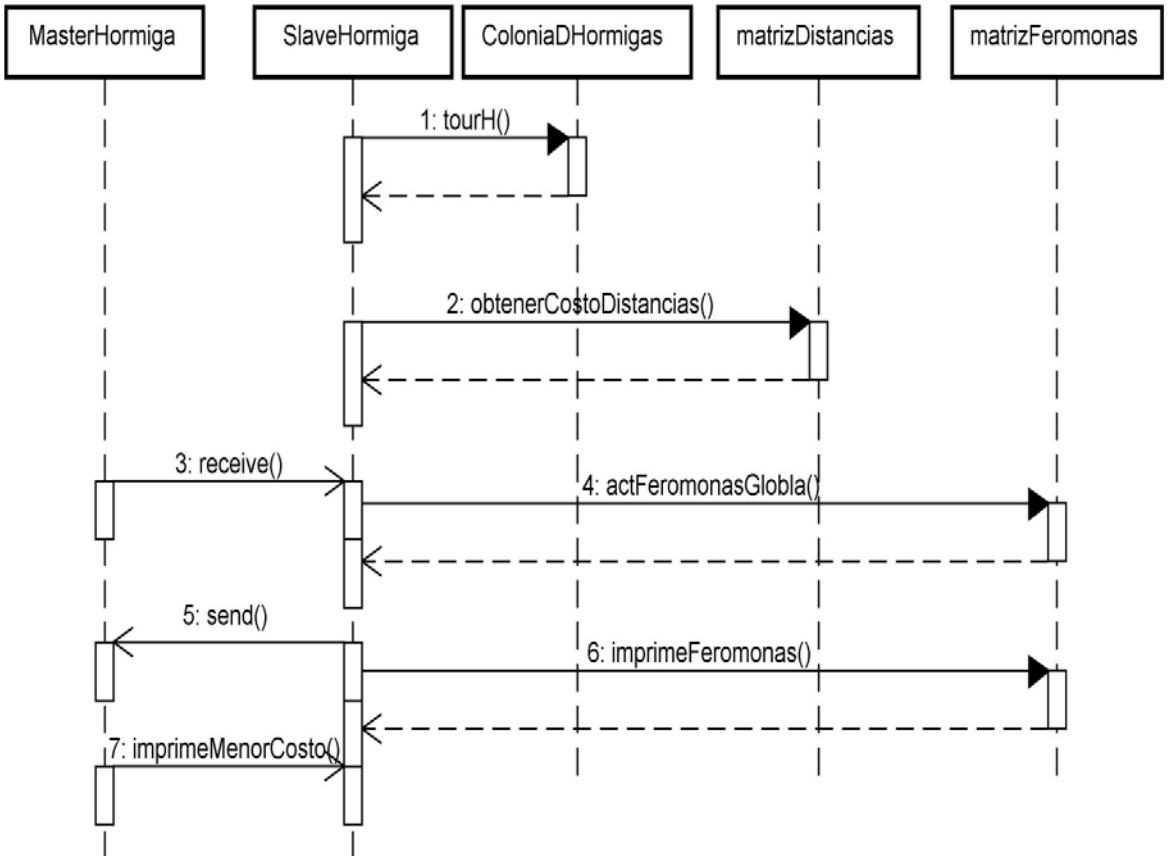


Figura D13. Diagrama de Secuencia donde se muestra la secuencia para elegir el tour mas corto

5. Implementación y resultados

El sistema que se ha desarrollado para este trabajo de investigación es una aplicación en consola que permite ejecutar el algoritmo secuencial y paralelo basado en la Colonia de Hormigas, considerando que el problema a resolver es una grafo el cual debe estar en un documento simple de texto, para poder leerlo desde la aplicación.

5.1 Problemas propuestos

Para probar el desempeño del algoritmo secuencial y paralelo en la Plataforma Paralela [5] se utilizó el problema BR17 para 17 ciudades con el menor costo conocido de 39, el cual es considerado como una de las mejores soluciones para el TSP asimétrico y se encuentra en su forma original en el TSPLIB, se puede visualizar a BR17 la figura 5.1.

Para efectos de este trabajo de investigación se hicieron algunos cambios en su matriz de distancias los cuales se muestran a continuación:

Matriz de Distancias original:

999	3	5	48	48	8	8	5	5	3	3	**0**	3	5	8	8	5
3	**999**	3	48	48	8	8	5	5	**0**	**0**	3	**0**	3	8	8	5
5	3	**999**	72	72	48	48	24	24	3	3	5	3	**0**	48	48	24
48	48	74	**999**	**0**	6	6	12	12	48	48	48	48	74	6	6	12
48	48	74	**0**	**999**	6	6	12	12	48	48	48	48	74	6	6	12
8	8	50	6	6	**999**	**0**	8	8	8	8	8	8	50	**0**	**0**	8
8	8	50	6	6	**0**	**999**	8	8	8	8	8	8	50	**0**	**0**	8
5	5	26	12	12	8	8	**999**	**0**	5	5	5	5	26	8	8	**0**
5	5	26	12	12	8	8	**0**	**999**	5	5	5	5	26	8	8	**0**
3	**0**	3	48	48	8	8	5	5	**999**	**0**	3	**0**	3	8	8	5
3	**0**	3	48	48	8	8	5	5	**0**	**999**	3	**0**	3	8	8	5
0	3	5	48	48	8	8	5	5	3	3	**999**	3	5	8	8	5
3	**0**	3	48	48	8	8	5	5	**0**	**0**	3	**999**	3	8	8	5
5	3	**0**	72	72	48	48	24	24	3	3	5	3	**999**	48	48	24
8	8	50	6	6	**0**	**0**	8	8	8	8	8	8	50	**999**	**0**	8
8	8	50	6	6	**0**	**0**	8	8	8	8	8	8	50	**0**	**999**	8
5	5	26	12	12	8	8	**0**	**0**	5	5	5	5	26	8	8	**999**

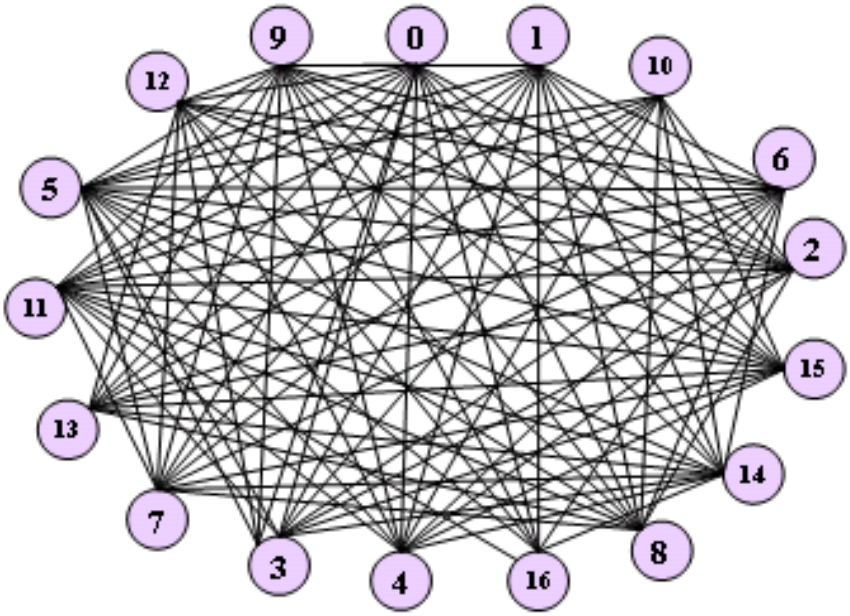


Figura 5.1 Grafo BR17

Los cambios que se realizaron en la Matriz de Distancias de BR17 para este trabajo de investigación fueron los siguientes:

999	3	5	48	48	8	8	5	5	3	3	**0**	3	5	8	8	5
3	**999**	3	48	48	8	8	5	5	**0**	**0**	3	**0**	3	8	8	5
5	3	**999**	72	72	48	48	24	24	3	3	5	3	**0**	48	48	24
48	48	74	**999**	**0**	6	6	12	12	48	48	48	48	74	6	6	12
48	48	74	**0**	**999**	6	6	12	12	48	48	48	48	74	6	6	12
8	8	50	6	6	**999**	**0**	8	8	8	8	8	8	50	**0**	**0**	8
8	8	50	6	6	**0**	**999**	8	8	8	8	8	8	50	**0**	**0**	8
5	5	26	12	12	8	8	**999**	**0**	5	5	5	5	26	8	8	**0**
5	5	26	12	12	8	8	**0**	**999**	5	5	5	5	26	8	8	**0**
3	**0**	3	48	48	8	8	5	5	**999**	**0**	3	**0**	3	8	8	5
3	**0**	3	48	48	8	8	5	5	**0**	**999**	3	**0**	3	8	8	5
0	3	5	48	48	8	8	5	5	3	3	**999**	3	5	8	8	5
3	**0**	3	48	48	8	8	5	5	**0**	**0**	3	**999**	3	8	8	5
5	3	**0**	72	72	48	48	24	24	3	3	5	3	**999**	48	48	24
8	8	50	6	6	**0**	**0**	8	8	8	8	8	8	50	**999**	**0**	8
8	8	50	6	6	**0**	**0**	8	8	8	8	8	8	50	**0**	**999**	8
5	5	26	12	12	8	8	**0**	**0**	5	5	5	5	26	8	8	**999**

Cambio de 0 por 1 y de 999 por 0, ya que la aplicación lee un archivo de texto donde se encuentra la matriz de distancias y tiene como especificación que cuando no exista

adyacencia entre los nodos se representa con 0 y el menor costo para cualquier arista debe ser de 1.

5.2 Comparación de los problemas implementados

La Tabla 5.1 compara los resultados obtenidos para BR17, OLIVER30 y Eilon50, utilizando 6 procesadores, para resolver estos problemas, se encuentra 2 veces la solución óptima, presentando un mejoramiento adicional para el problema Oliver30. En consecuencia, se realizaron experimentos con el problema EILON50 [19], en cuyo caso se encuentra la solución óptima sólo cuando el 10% de las hormigas actualizan la matriz de feromonas.

No. Hormigas-%	1	5-10%	10-20%	15-30%	30-60%
BR17					
Mejor Solución	80	50	**39**	**39**	**39**
Solución Promedio	50	50	39	39	39
Desviación Estándar	4	2	2	1	1
Oliver30					
Mejor Solución	456	**423**	**423**	**423**	**423**
Solución Promedio	435	423	423	423	423
Desviación Estándar	9	4	2	1	1
Eilon50					
Mejor Solución	723	**427**	428	429	436
Solución Promedio	745	429	431	434	444
Desviación Estándar	12	2	2	2	5

Tabla 5.1. Resultado obtenidos de las pruebas en la Plataforma con el Algoritmo paralelo basado en la Colonia de Hormigas

En resumen, de todas las pruebas presentadas, BR17 es la que encontró los mejores resultados experimentales, además, el método aplicado para la paralelización del algoritmo propuesto por mejora la calidad de la solución al aumentar el número de procesadores, esto se debe a que dispone de mayor información del problema enviada por otros procesadores.

5.3 Resultados experimentales

Para comparar el desempeño de los algoritmos secuencial y paralelo, se hicieron pruebas con los problemas BR17, Oliver30 y Eilon50. Se realizaron corridas en una red de 6 computadoras personales con procesadores AMD K6-2 de 350 MHz, y 128 MB de memoria RAM. Estas computadoras personales están conectadas por medio de un swich de 8 puertos, formando una red de área local (LAN) con procesadores corriendo el sistema operativo Windows XP.

Las implementaciones fueron realizadas en la Plataforma Paralela [5]. En la tabla 5.1 se observan los tiempos promedios obtenidos para cada problema y en la Grafica 5.1 se visualiza el tiempo promedio de 10 corridas con uno, dos cuatro y seis procesadores para el problema BR17, sólo a modo de ejemplo.

Problema	10 Ciudades			17 Ciudades		
# Procesadores	Tiempo	Aceleración	Eficiencia	Tiempo	Aceleración	Eficiencia
1	46	Ref 1	Ref. 100	53	Ref 1	Ref. 100
2	34.5	1.94	96.93	47.5	1.94	96.93
4	25.2	3.84	96.11	32.1	3.84	96.11
6	15.3	5.60	93.27	22.3	5.60	93.27

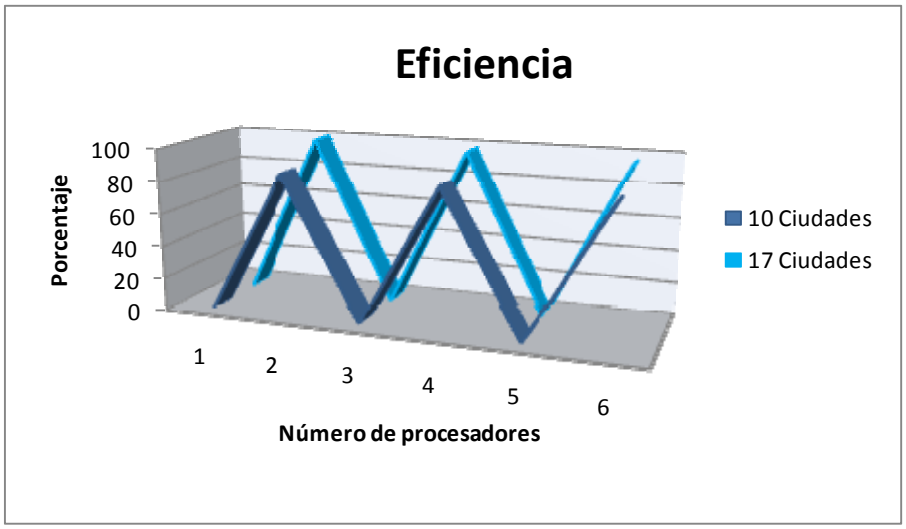
Tabla 5.2. Resultado en tiempos obtenidos de las pruebas en la Plataforma con el Algoritmo paralelo basado en la Colonia de Hormigas para 6 procesadores

Como ejemplo de los resultados experimentales obtenidos utilizando la red de computadoras personales arriba descripta, la Tabla 5.2 muestra la aceleración promedio en 10 corridas utilizando 2, 5 y 6 procesadores, en este ejemplo, para la versión 2. Estos resultados se mantienen muy similares para todas las versiones. Claramente, existe una buena escalabilidad con el número de procesadores.

Grafica 5.1. Grafica donde se muestran los tiempos obtenidos de las pruebas

En lo relativo a la eficiencia, la Grafica 5.2 demuestra que la paralelización del algoritmo es sumamente benéfica y sencilla, logrando en algunos casos, eficiencias del 100%. En

consecuencia, es de esperar que la presente propuesta pueda ser utilizada con un número creciente de procesadores sin mayores complicaciones.



Grafica 5.2. Gráfica donde se muestran la eficiencia de la Plataforma Paralela y el algoritmo paralelo con 6 procesadores

6. Conclusiones

Se ha implementado una novedosa técnica basada en el comportamiento de las hormigas para establecer el camino más corto desde su nido hasta la fuente de alimento y regresar. Simples agentes computacionales, llamados hormigas, colaboran intercambiando información para alcanzar un objetivo común, la optimización del camino. Al moverse de una ciudad a otra, la hormiga deja una señal, marcando el camino recorrido en una matriz de feromonas. Esta información es utilizada por las hormigas que le siguen, con lo cual, los caminos más transitados son más deseables.

Con la técnica, propuesta por Dorigo [4] se resolvió el problema del cajero viajante para 17 ciudades del BR17. Se propuso en el algoritmo original iniciar la matriz de feromonas, realizando evaluaciones entre los costos de las aristas para aprender cuales eran los caminos mas cortos con el método de tabu, asi mejorando, con esto el desempeño del algoritmo.

Al implementar el algoritmo de Optimización de la Colonia de Hormigas, original en la Plataforma Paralela [5] desarrollada en Java, fue posible concluir lo siguiente:

- La solución óptima para un mismo conjunto de parámetros α , β , y ρ , fue hallada en 0.223 segundos en promedio para el método propuesto, frente a 0.815 segundos para el algoritmo de Dorigo et al. (realizando 10 corridas para cada algoritmo).

- Se realizaron 3 experimentos (cada uno con 10 corridas). En 2 de ellos se encontró la solución óptima utilizando el algoritmo de Optimización de la Colonia de Hormigas [4] y la Plataforma Paralela.

En resumen, la propuesta de iniciar la matriz de feromonas y aprender de los recorridos donde las aristas son más cortas, se obtuvo la solución óptima con mayor frecuencia y en menores tiempos de ejecución. Debido a los excelentes resultados experimentales obtenidos, se continuará trabajando en la Plataforma que ya tiene una nueva versión que ofrece total independencia de la librería RMI. Pues el algoritmo de Optimización de la Colonia de Hormigas tiene características que lo hacen especialmente apropiado para su paralelización y distribución entre diversos procesadores. En efecto, cada hormiga es un agente autónomo que construye su propio tour, con la restricción de no viajar a una ciudad ya visitada con anterioridad.

El presente trabajo presentó una alternativa de paralelización asíncrona del algoritmo ACO en la Plataforma Paralela desarrollada en Java y también se demostró que la Plataforma Paralela puede proporcionar a los desarrolladores una herramienta fácil de usar para la implementación de algoritmos complejos como los genéticos.

Puesto que los resultados experimentales demuestran una buena eficiencia, que eventualmente se aproximo al 100% para los problemas mayores, consiguiéndose muy buena escalabilidad, lo que permite postular su utilización eficiente con un número

creciente de procesadores y en problemas de mayor tamaño y complejidad, además de ser un método versátil, aplicable a cualquier variante de ACO o de cualquier algoritmo genético.

En conclusión, el algoritmo de Optimización de la Colonia de Hormigas [4] se presenta como un promisor mecanismo de optimización capaz de resolver satisfactoriamente problemas combinatorios de notable complejidad y gran porte como el problema del TSP, aprovechando al máximo la accesibilidad a un número creciente de computadores personales heterogéneos que gracias a la Plataforma Paralela desarrollada en Java resulta sencillo de implementar.

7. Bibliografía

[1] P. Jog, J.Y. Suh, D. Van Gucht "Parallel Genetic Algorithms Applied to the Traveling Salesman Problem" *SIAM Journal on Optimization* Vol. 1, No. 4, pages 515--529, 1991.

[2] P. P.Mutalik, L. R. Knight, J. L. Blanton y R. L.Wainwright "Solving Combinatorial Optimization Problems Us-ing Parallel Simulated Annealing and Parallel Genetic Algo-rithms", Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing , pp. 1031- 1038, 1992, ACM Press.

[3] R_ L_uling_ B_ Monien_ M_ R_acke_ S_ Tsch_oke_ Solving the TSP with a Distributed Branch_and_Bound Algorithmo a Processor Network_ Technical ReportUniversity of Paderborn

[4] Dorigo M., Maniezzo V. y Colorni A., "An Investigation of some properties of an Ant algorithm". *Proceedings of the Parallel Problem solving from Nature Conference* (PPSN 92), Bruselas – Bélgica, 1992.

[5] Erick P. Rodriguez, *"Una Plataforma para el Desarrollo de Aplicaciones en Paralelo Usando Java"* Tesis de Maestria, Facultad de Ciencias de la Computación, BUAP

[6] Bullnheimer B., Kotsis G. y Strauss C. "Parallelization Strategies for the Ant System", Reporte Técnico POM 9/97, Universidad de Viena, Viena – Austria, 1997.

[7] Stützle T., "Parallelization Strategies for Ant Colony Optimization", *Proceedings of Parallel Problem Solving from Nature – PPSN-V*, Springer Verlag, Vol. 1498, pp. 722-731, 1998.

[8] S.C. Lin, W.F. Punch, E.D.Goodman Coarse-Grain Parallel Genetic Algorithms: Categorization and New Approach. Proceedings of the Sixth IEEE Parallel & Distributed Processing, IEEE Press, pp. 28-37, 1994

[9] D. Abramson, G. Mills, S. Perkins, Parallelization of a genetic algorithm for the computation of efficient train schedules, Proceedings of the 1993 Parallel Computing and Transputers Conference, 1993, pp. 139-149.

[10] J. Sarma, K. De Jong, "An Analysis of Local Selection Algorithms in a Spatially Structured Eevolutionary Algorithms". Proceedings of the Seventh International Conference on Genetic Algorithms, pp. 181-186, 1997.

[11] C. Schwehm, M. Ehrgott and K. Klamroth, "An MCDM approach to portfolio optimization," *European Journal of Operational Research*, vol. 155, no. 3, pp. 752–770, June 2004.

[12] E. Alba and J. M. Troya "A Survey of Parallel Distributed Genetic Algorithms" Complexity, Vol. 4, Nº 4, pp. 31-52. March/April 1999

[13] Filman Robert E.; Daniel P. Friedman(1984), Coordinated Computing: Tools and Techniques for Distributed Software

[14]Almasi, G.S. and Gottlieb, A. 1989. Highly Parallel Computing. Londres, The Benjaming/Cummings Publishing Company

[15] McGraw, J.R. and Axelrod, T.S. 1988. "Exploiting multiprocessors: issues and options." IEEE Software, Vol. 5, (5), pp. 7-25.

[16] Pascal Felber (2003), Semi Automatic Paralellization of Java Applications.

[17] Getov Vladimir, Philippsen Michael(2001), Java Communications for Large-Scale Computing.

[18] Mark Baker. Review: Distributed systems textbook falls short of its full potential. *IEEE Distributed Systems Online*, 2 (4), 2001. ISSN 1541-4922.

[19] Oliver, I., Smith, D. and Holland, J.R., 1987, A study of permutation crossover operators on the traveling salesman problem, in: *Proceedings of the Second International Conference on Genetic Algorithms*, J.J. Grefenstette (ed.) (Lawrence Erlbaum, Hillsdale, New Jersey) pp. 224–230.