Improving a Distributed Agent-Based Ant Colony Optimization for Solving Traveling Salesman Problem

Aleksandar Kaplar*, Milan Vidaković*, Nikola Luburić* and Mirjana Ivanović**

*Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia
**Faculty of Sciences, University of Novi Sad, Novi Sad, Serbia
*{aleksandar.kaplar, minja, nikola.luburic}@uns.ac.rs
**mira@dmi.uns.ac.rs

Abstract - Optimization of a large-scale Traveling Salesman Problem, which is a well-known NP-hard problem in combinatorial optimization, is a time-consuming problem. A modern approach to dealing with such time-consuming problems is with the use of distributed computing, which can significantly improve the speed of the problem-solving algorithm. In this paper, we discuss the design approaches for an agent-based distributed algorithm and their benefits. Based on further analysis and experiments, we have improved our previous agent-based Ant Colony Optimization algorithm for Solving Traveling Salesman Problem using Siebog multiagent middleware.

I. INTRODUCTION

Distributed system can be defined as a network of independent components that communicate and coordinate their actions only by passing messages [1]. The motivation behind the use of distributed systems is the fact that distributed data computing can lower the cost of data processing and increase the system’s robustness with data replication. When used properly, distributed computing can achieve a computational result much more quickly than a single computer can.

One area inherently amenable to parallelization and distributed computing is swarm intelligence. [5]

Ant colony optimization (ACO) [2] is one of the swarm based optimization algorithms used for solving wide range combinatorial optimization problems. It is inspired by the behavior exhibited by ant colonies while searching for a food source. One of the characteristics of ant colonies is that they have the ability to find the shortest path to a food source, by using a large number of independent ants that use only pheromones as a means of communication. In ACO each ant can be seen as an independent processing agent searching through an environment.

In this paper, we describe our intention to improve the execution speed of our previous agent-based distributed ACO algorithm for Traveling Salesman Problem (TSP). The algorithm was initially developed to showcase Siebog’s distributive capabilities [6, 7]. Although successful, the algorithm didn’t fully utilize all the capabilities offered by the Siebog design.

Siebog is our FIPA compliant multiagent middleware built using the Java Platform, Enterprise Edition (Java EE) [8, 9, 10]. Siebog supports both client-side and server-side agents. Client-side agents execute their code on browsers, as JavaScript code, while server-side agents are being executed as EJB (Enterprise Java Beans) components. Siebog utilizes the standards and technologies readily available in Java EE, in order to provide automatic agent load balancing and fault tolerance on the server side [7]. It also uses Java EE built-in features to implement core agent functionalities:

- it uses JMS (Java Message Service) to exchange messages between agents,
- it uses JNDI (Java Naming and Directory Service) as agent discovery service (agent "yellow pages").
- it uses JAAS (Java Authentication and Authorization Service) to implement security concepts.

This paper is organized as follows. Section 2 provides an overview of relevant literature. The formulas of ACO for TSP are presented in Section 3. Section 4 presents our proposed model. The results of the model are presented in Section 4. Finally, Section 5 outlines general conclusion and further work.

II. RELATED WORK

Over the years there have been many attempts at developing optimal parallel and distributed algorithms for ACO. A survey on parallel ant colony optimization was reported by [11]. Authors of that work proposed a new taxonomy for classifying parallel ACO. Their classification includes: master-slave model, cellular model, parallel independent runs model, multi-colony model, and hybrid model. Classification best suited for this work is the multi-colony model.

In a standard multi-colony model configuration, introduced by [12, 13], different colonies of ants work on the same problem independently. Pheromones aren’t shared between the colonies, but after a number of iterations colonies exchange their best solution in order to influence each other via elitist strategy.

The standard multi-colony configuration was successfully applied to various optimization problems,
with improvements over the sequential ACO. Notably, authors of [14], developed a parallel ant colony systems (PACS). Their preliminary test on three datasets showed that PACS outperforms sequential algorithms, especially with larger datasets.

Another notable distributed agent-based algorithm for solving TSP, named ACODA, was developed by [3]. Their experimental results show execution time improvement, and that the system, by partitioning the map between agents, supports scalability.

III. ACO FOR TSP

ACO algorithm is inspired by the behavior exhibited by ants while searching for a food source. Ants, while searching for a food source, secrete pheromones on their way back to the anthill. Other ants can detect paths with secreted pheromones and they may become attracted to them. The paths become more attractive when more pheromone is deposited on them. Another property of the pheromones is that they evaporate over time. Evaporation makes the longer paths less interesting, which decreases the probability that an ant will choose it. However, evaporation will have less influence on shorter paths, due to the fact that they are refreshed more frequently. In time, ants will converge towards the shortest path due to largest concentration of pheromones [2][3].

Mathematical model of ACO for TSP we used in our work is fully described in [2][3], briefly summarized the formulas are the following:

An ant located at a city (node) i chooses to go to city j with the probability:

\[ P_{i,j} = \frac{(\tau_{i,j})^{\alpha} (\eta_{i,j})^{\beta}}{\sum_{k} (\tau_{i,k})^{\alpha} (\eta_{i,k})^{\beta}} \]

where \( \tau_{i,j} \) - is the amount of the pheromone deposited on edge \((i,j)\), \( \eta_{i,j} \) - is the inverse of the weight of edge \((i,j)\), \( \alpha \) - is a parameter to control the influence of parameter \( \tau_{i,j} \), \( \beta \) - is a parameter to control the influence of \( \eta_{i,j} \), \( j \) - represents a city reachable from city \( i \) that was not yet visited by that ant. When an ant determines a new tour of a cost \( L \) it will increase every edge of the tour with the value \( \Delta \tau_{ij} \) which is inversely proportional to the cost of tour. Pheromone update and evaporation, are done while ant traces back it’s steps, with the formula:

\[ \tau_{i,j} = (1-\rho)\tau_{i,j} + \rho \Delta \tau_{i,j} \]

where \( \rho \) is the evaporation rate in the range \( 0 \leq \rho < 1 \). In order to increase the exploration rate an ant can apply local evaporation of to a node with the formula:

\[ \tau_{i,j} = (1-\xi)\tau_{i,j} + \xi \tau_{0} \]

where \( \xi \) is the local evaporation rate in the range \( 0 \leq \xi < 1 \) and \( \tau_{0} = 1/nC \) in which \( n \) represents the number of nodes and \( C \) represents the tour cost approximation (\( C \) usually equals to product of number of nodes and average edge cost between them).

IV. PROPOSED MODEL

When designing an agent based ACO algorithm, we considered two distinct approaches. Approaches are characterized by the representation of the ants. That is, ants can be represented as either an agent in the system or as a message exchanged in the system.

Initially, to display Siebog’s capabilities, an ACO algorithm was developed in which ants were represented as agents, and the problem space was represented by a single map agent. Although not optimized, that algorithm showed that our multi-agent middleware had all the necessary capabilities required for distributed computing, but it had a major flaw brought on by the nature of distributed systems. As mentioned before, distributed systems communicate over a physical network which introduces a delay to the system. That delay combined with a bottleneck which is a single map agent and synchronous access to it resulted in a significant increase in time required for algorithm execution (for a simple map of 16 nodes, the time required was about 1h55m on commodity hardware and about 10m on high-end hardware). In order to improve the execution time we have redesigned the ant itself. In fact, we have implemented it as a message instead of an agent.

In the first approach, ants are agents that search through the problem space. The problem space is represented and maintained by one or more map agents. Ant agents are the algorithm executors, they start the search, use the problem space data to perform calculations, and notify map agents about necessary updates to the problem space. The behavior of map agents can be described as purely responsive because all of their actions are responses to queries generated by ant agents. There are three distinct messaging types, illustrated in Figure 1A, in this approach:

1. Initialization messaging – with which an ant agent request a list of nodes in the problem space,
2. Local move messaging – when choosing the next node to visit an ant queries the map agent for pheromone levels in path from current node to next nodes,
3. Update tour – after finding a possible solution, an ant notifies all map agents about it.

To decrease the number of messages in our distributed system, for the second approach, we represented ants as messages exchanged by map agents. To accommodate such changes, we extended map agents by adding them algorithm execution functionality. Algorithm execution functionality is represented as an infinite loop which takes and processes an ant from the ant queue. After processing (choosing the next node that ants will visit, and updating the problem space), the ant is either sent to another map agent or added to the end of the ant queue, depending on the node to which the ant will go to next. In this approach, there are two distinct messaging types (illustrated in Figure 1B):

1. Send ant – this message happens when an ant needs to visit a node managed by another map agent,
Figure 1. Typical messaging exchange between the agents in: A – ant as an agent approach, B – ant as a message approach.

2. Update tour – when the ant returns to its initial map agent with a possible solution, that map agent notifies other map agents about it.

To fully optimize the algorithm execution in this approach, we have used Siebog's flexibility to separate the agents' message processing from the algorithm execution. The separation is achieved by using a separate thread for the algorithm execution. With that, each map agent is able to process messages, update the problem space, and execute the algorithm steps in parallel.

Inspired by [2], our agent-based algorithm, given in table 1., consists of several steps regardless of the chosen approach.

<table>
<thead>
<tr>
<th>TABLE 1. STEPS OF AN ACO ALGORITHM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEARCH_STEP(ant)</td>
</tr>
<tr>
<td>GET_MAP_DATA()</td>
</tr>
<tr>
<td>if COMPLETED_TOUR(ant):</td>
</tr>
<tr>
<td>NOTIFY_TOUR(ant)</td>
</tr>
<tr>
<td>REINITIALIZE(ant)</td>
</tr>
<tr>
<td>else:</td>
</tr>
<tr>
<td>ANT_MOVE(map_data, ant)</td>
</tr>
</tbody>
</table>

| ANT_MOVE(map_data, ant):           |
| node = perform random choice per equation 1 |
| UPDATE_ANT_PATH(ant,node)         |
| apply local evaporation using equation 4 |
| SEND_ANT(ant, node)               |

| SEND_ANT(ant,node):               |
| if node.agent != current_map_agent: |
| SEARCH_STEP(ant)                   |
| else:                             |
| SEND(ant, node.agent)             |

| NOTIFY_TOUR(ant):                 |
| for edge in ant.tour:             |
| update pheromone levels using equation 3 |
| if STOP_CRITERIA_MET():          |
| STOP_SEARCH()                    |

When the client initiates the search, the problem space is loaded and distributed among a number of map agents. In order to decrease the initial workload of each map agent, ants are evenly assigned to. The client defines the number of ants used in the search. Each ant begins its search of the problem space from a randomly chosen node managed by their assigned map agent. After the initial pick of a start node, ants repeatedly execute the SEARCH_STEP functionality until they return to their start node. In the ant as an agent approach, to complete each step, ant first requests the pheromone levels of each possible path it can take from the current node. When the ant gets a response from a map agent, it executes appropriate calculations to choose the next node it will visit. After choosing the node, it notifies the map agent to apply the calculated local pheromone evaporation and repeats this process again with the next node. On the other hand, in ant as a message approach all calculation are done by map agents. The step of the search process is represented by taking the ant from the ant queue, calculating the next node, and putting the ant at the back of the ant queue (or sending it to another map agent).

During a single ANT_MOVE the ant uses pheromone data, of all the paths it can take, to choose a path that is a city, it will visit next. When the ant chooses the next city, it updates its tour cost by adding the cost of the taken path. With the new data generated by ant, appropriate map agents apply local evaporation to the taken path. In the ant as a message approach, the last step of the ant move is to send it to next map agent (if the next city belongs to another agent), or to add him to the end of the ant queue (if the next city belongs to the map agent). On the other hand, in the ant as an agent approach, the ant agents receive the agent id of the appropriate map agent it needs to query for the map data.

When the ant finishes its tour, it returns to the ant hill (starting city) while depositing the pheromone on the
taken path. Ants return is implemented via NOTIFY_TOUR message that is used to send tour data to all map agents. With the tour data, the map agents update their part of the map and update the best tour if needed. After the updates, map agents check if the stop criteria are met and if it is they send out stop messages to terminate the algorithm execution and return the best tour to the client.

V. EVALUATION AND RESULTS

In order to properly represent the results and compare them with the execution speed of initial implementation, we used a single Siebog node for testing. The machine used for Siebog node had a 4 virtual CPUs and 8 GB of RAM, running a 64-bit version of Windows 7.

Execution speed was tested by measuring the time required for each ant to complete 200 tours of the search space. It happens under requirement that the number of ants is equal to the number of cities on the map. We ran the test on benchmark TSP maps selected from TSPLIB [3], the results are presented in table 2.

The results show that ant as a message approach is significantly faster than ant as an agent approach. When compared to the initial implementation which required 1h35m for ulysses16, both approaches show considerable improvement in the execution speed. The speed discrepancy between two approaches can be attributed to several factors. For the smallest map of 16 nodes, the major factor is the delay introduced by messages subsystem, which can introduce the delay from 5ms to 25ms per message. On the other hand, with larger maps in which more ants are required to properly conduct the search, another major contributing factor to execution time is the number of available threads. Ant represented as an agent requires a separate thread for each agent to fully parallelize the execution of the algorithm. When working with large maps, that number can exceed the number of threads available in the appropriate thread pool (in the conducted test, Siebog had 50 threads available in the thread pool).

From the given results we can conclude that the better approach for this type of problem is the ant as a message approach. Advantage of the ant as a message is twofold. Firstly, it separates message processing from the algorithm execution, which parallelizes the task done by single map agent. Secondly, the number of messages exchanged in this approach is greatly reduced, which in turn improves the execution speed. On the other hand, ant as an agent approach can be considered as an underutilization of processing power. The processing requirements done by ant agents are not complex and require less time than it’s required by message exchange. Benefits of ant as an agent approach could only be achieved with the problems where the performed tasks, done by agents, are computationally intensive.

VI. CONCLUSION

In this paper, we have presented our attempts at improving the execution speed of a distributed agent-based ACO algorithm for TSP, designed for the Siebog multiagent middleware. During the design phase, we considered two distinct approaches in the design of our agent-based algorithm: ant as an agent and ant as a message.

Evaluation of those approaches has shown that optimizing and reducing the number of messages exchanged greatly improves the execution speed of the algorithm. Achieved results show that, with the utilization of the Siebog flexibility, the better solution for this type of problem is the realization of ant as a message.

For the future work, we intend to test and optimize the fully distributed algorithm on a Siebog cluster and compare results with other similar approaches. We believe that with further optimizations of the algorithm, combined with Siebog capabilities, we can produce competitive results.

REFERENCES


